**Program Synthesis by Sketching**

by

Armando Solar-Lezama

B.S. (Texas A&M University) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor in Philosophy

in

Engineering-Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Rastislav Bodik, Chair
Sanjit Seshia
Leo Harrington

Fall 2008

The dissertation of Armando Solar-Lezama is approved:

---

Chair Date

---

Date

---

Date

University of California, Berkeley

Fall 2008

**Program Synthesis by Sketching**

## Abstract

Program Synthesis by Sketching

by

Armando Solar-Lezama

Doctor in Philosophy in Engineering-Electrical Engineering and Computer Science

University of California, Berkeley

Rastislav Bodik, Chair

The goal of software synthesis is to generate programs automatically from high-level specifications. However, efficient implementations for challenging programs require a combination of high-level algorithmic insights and low-level implementation details. Deriving the low-level details is a natural job for a computer, but the synthesizer can not replace the human insight. Therefore, one of the central challenges for software synthesis is to establish a synergy between the programmer and the synthesizer, exploiting the programmer's expertise to reduce the burden on the synthesizer.

This thesis introduces *sketching*, a new style of synthesis that offers a fresh approach to the synergy problem. Previous approaches have relied on meta-programming, or variations of interactive theorem proving to help the synthesizer deduce an efficient implementation. The resulting systems are very powerful, but they require the programmer to master new formalisms far removed from traditional programming models. To make synthesis accessible, programmers must be able to provide their insight effortlessly, using formalisms they already understand.

In Sketching, insight is communicated through a partial program, a *sketch* that expresses the high-level structure of an implementation but leaves holes in place of the low-level details. This form of synthesis is made possible by a new SAT-based inductive synthesis procedure that can efficiently synthesize an implementation from a small number of test cases. This algorithm forms the core of a new counterexample guided inductive synthesis procedure (CEGIS) which combines the inductive synthesizer with a validation procedure to automatically generate test inputs and ensure that the generated program satisfies its

specification. With a few extensions, CEGIS can even use its sequential inductive synthesizer to generate concurrent programs; all the concurrency related reasoning is delegated to an off-the-shelf validation procedure.

The resulting synthesis system scales to real programming problems from a variety of domains ranging from bit-level ciphers to manipulations of linked datastructures. The system was even used to produce a complete optimized implementation of the AES cipher. The concurrency aware synthesizer was also used to synthesize, in a matter of minutes, the details of a fine-locking scheme for a concurrent set, a sense reversing barrier, and even a solution to the dining philosophers problem.

The system was also extended with domain specific knowledge to better handle the problem of implementing stencil computations, an important domain in scientific computing. For this domain, we were able to encode domain specific insight as a problem reduction that converted stencil sketches into simplified sketch problems which CEGIS resolved in a matter of minutes. This specialized synthesizer was used to quickly implement a MultiGrid solver for partial differential equations containing many difficult implementation strategies from the literature.

In short, this thesis shows that sketching is a viable approach to making synthesis practical in a general programming context.

## Acknowledgments

The work described in this thesis has benefited from intense collaboration with a number of individuals both at Berkeley and at IBM Research. First among these is my advisor, whose unwavering optimism and encouragement helped me push the project forward past the inevitable setbacks towards ever more ambitious milestones. His gift for capturing the big picture behind complex ideas was invaluable in moving this project from its humble beginnings as a tool to write bit permutations into a general code synthesis tool. Most importantly, his dedicated mentorship made my graduate school experience a happy and fulfilling one.

This project also owes a debt to IBM Research for its financial, intellectual and moral support. Vijay Saraswat collaborated with us extensively in the early stages of the language design, and he was my main motivation to explore the domain of stencil computations. Vivek Sarkar also provided us with ample support through the early stages of the project. Many of the insights behind the SKETCH language came from the lessons we learned through the user studies he encouraged us to undertake for the StreamBit system. The project also benefited from the collaboration of Kemal Ebcioğlu and Rodric Rabbah in its early stages, and Eran Yahav and Martin Vechev in the context of concurrency. More generally, my summers spent at T.J. Watson were an invaluable complement to my education at Berkeley, not to mention much of my graduate school was supported through IBM Fellowships.

The project also owes a special debt to Chris Jones, Liviu Tancau and Gilad Arnold. The section on concurrent sketching is derived from the joint work that Chris and I did with Ras on sketching concurrent datastructures. Liviu contributed significantly to the implementation of the SKETCH infrastructure and to the work on sketching stencils. Gilad was also my co-author for the work on stencils; moreover, as his officemate, my work benefited greatly from his insights and from his careful attention to details.

The project also benefited greatly from the insights of Sanjit Seshia, and from the support of Robert Brayton and Alan Mishchenko who developed ABC, a circuit analysis tool that we used extensively with great results.

Finally, this work wouldn't have been the same without the support and encouragement from my wife, Sabrina, my parents and my friends. I especially want to thank my friend Alejandro de la Fuente for his help filing this thesis and for his unconditional support.

# Contents

# Part I

# The Sketching Approach to Software Synthesis

# Chapter 1

# Introduction

Modern programming tools and methodologies have proved invaluable in tackling the challenges of scale and emergent complexity in software, but remarkably little help is available for programmers facing more basic programming challenges. Domains as diverse as scientific computing, concurrent datastructures and low-level systems programming all require programmers to produce small but inherently complex routines that demand clever algorithmic insights and careful orchestration of details; the kind of hard problems that separate the winners from the losers in programming competitions.

For these programming problems, pencil and paper remain the most effective programming tools. Modern tools and languages can help ensure that once written, such routines can be reused, managed, and distributed easily, so that only a handful of star programmers ever has to cope with them. But for these programmers, the challenges involved are as big today as they were thirty years ago.

Fortunately, things are changing. Over the last eight years, verification technology has matured to become a practical programming tool. Modern model checking tools, for example, are now able to expose errors in complex routines with relatively little user effort [7,21,37,68]. By automating the validation process, programmers gain the freedom to experiment; they are able to formulate hypothesis and to use the validation tool to expose errors in their thinking. But automated validation is only part of the solution; it would be far better if the programmer could focus on the high-level algorithmic insights, and leave the low-level details to the programming tools. This is the dream of software synthesis; to capture the high-level insights of the programmer and automatically derive efficient implementations.

Software synthesis has a long and fruitful history, but its impact on general programming practice has been limited. This thesis introduces *sketching*, a synthesis technology that blends seamlessly into an imperative programming model, completely redefining the relationship between the synthesizer and the programmer, and potentially bringing synthesis closer to widespread adoption.

## 1.1 The promise and the challenges of software synthesis

Software Synthesis has been one of the Holly Grails of computer science research at least since the late 60s; it was considered by Pnueli to be "one of the most central problems in the theory of programming" [50]. The promise of synthesis is that we should be able to tell the computer *what* to do, and let the synthesizer discover *how* to do it. From a specification, the synthesizer should automatically produce a correct and efficient implementation.

But the pioneers in the field soon realized that fully automatic synthesis was impractical for several reasons. First, the problem was simply too difficult; as Manna and Waldinger pointed out, " programming is among the most demanding human activities, and is among the last tasks computers will do well" [45]. It was unreasonable to expect a synthesizer to rediscover algorithms and implementation techniques whose original discovery had challenged the ingenuity of the brightest minds in the field. Moreover, professional programmers want to have control over their implementations; they want to explore the tradeoffs between different design decisions, and they often have deep knowledge over the problem they are solving and the platform they are targeting. For these reasons, all efforts at synthesis have had to incorporate human insight into the synthesis process. For some systems, this has meant restricting their domain of application and programming the insights for that domain directly into the synthesizer. Other systems have coped with this limitation by providing mechanisms for users to direct the synthesis process, and to provide insights either for an individual problem or for an entire class of programs. Establishing a proper synergy between the human and the synthesizer is fundamental to the success of synthesis.

Domain specific systems take the human insight and build it directly into the synthesizer. This limits the range of programs they can synthesize, but allows them to operate fully automatically for problems that fall into their domain. Some examples of these systems include AutoBayes [28] which produces data analysis programs from statistical models, FFTW [29] which produces fast Fourier transforms optimized for specific architectures, and

StreamIt [64] which can produce very efficient signal processing kernels from a high-level specification. All of these systems are able to generate implementations that often outperform hand-written code. In order to do this, they rely on domain specific analysis and transformation algorithms which encode a lot of accumulated knowledge about how to solve problems in their particular domain. While these systems constitute the most successful instances of synthesis in the field, their specificity has tended to limit the impact they've had on general programming practice.

By contrast, there is a class of synthesis systems which allow the user to provide insight directly into the synthesizer. Most of these systems can trace their roots back to the work on deductive synthesis of Manna and Waldinger [46]. The central idea in deductive synthesis is that a program can be extracted from a constructive proof of the satisfiability of a specification. Today, NuPRL [23] and KIDS [59] are the most successful systems of this kind. They allow the programmer to provide insight about the implementation at a high level, in the form of axioms and theorems about the problem domain, and use these to derive a correct implementation from a high level specification. In the hands of experts, these systems are extremely powerful; for example, KIDS has been used to synthesize very complex programs, including an Airlift Scheduler for the Air Force [26], and a communication protocol for the interoperation of agents [17]. NuPRL, for its part, played a central role in the development of the Ensamble group communication system. It was used, for example, to synthesize an adaptive network protocol from formal specifications [10].

The main drawback of the deductive approach is the level of expertise it demands from its users. It takes a high degree of mathematical maturity to translate insights about an implementation into theorems about the domain, and to guide the interactive theorem prover to produce a derivation of the implementation. To broaden the impact of synthesis technology, we need to lower the expertise barriers that keep most programmers away from these tools.

## 1.2 Rethinking the role of synthesis

To make synthesis more accessible, we had to rethink the way the programmer communicates insight to the synthesizer. Our inspiration came from a system called A-Lisp [4]. This system allows users to write partial lisp programs expressing their high-level knowledge about the desired behavior of an intelligent agent. These partial programs help

the learning algorithm by constraining the set of behaviors that it may consider. *Sketching* is our attempt to use the insights of A-Lisp to redefine the relationship between the programmer and the synthesizer.

In traditional deductive systems, the programmer is provided with a language and a set of formalisms to describe *how* the implementation is to be derived from the specification. Our early experience with the StreamBit [63] synthesis system made us realize it is difficult for programmers to reason in terms of derivations. Instead, programmers often have an idea about the general form of a solution; a high-level strategy that will solve the problem at hand. To turn the strategy into a program, however, they have to orchestrate many low-level details; a process that is difficult and error prone. It therefore made sense to focus the synthesizer on those low-level details, leaving control of the high-level strategy in the hands of the programmer.

The example of A-Lisp made us realize that partial programs offered an ideal way for programmers to define the high-level implementation strategy while leaving the details unspecified. We observed that in many implementations, the implementation strategy is reflected in the overall structure of the code, while the implementation details are encoded in the individual expressions and assignments. Therefore, partial programs offer a very natural way to express the insight about the implementation strategy without resorting to separate formalisms, allowing synthesis to be embedded directly into a standard programming language. The synthesizer becomes a programming assistant in a familiar programming setting.

This was the genesis of sketching, a form of synthesis that uses partial programs as a communication device between the programmer and the synthesizer. The following examples illustrate how this process works in practice for real programming problems.

### 1.2.1 Example: Linked List Reversal

Manipulations of linked data-structures are hard to implement because they require the programmer to visualize how the data structure is going to be transformed by each memory update. On the other hand, this is an ideal problem for sketching: the details are hard to get right, but the programmer usually has a good idea of the overall structure of the solution.

Consider the problem of reversing a linked list. If one does not care about efficiency

at all, a naïve implementation can be written as follows.

```
list reverse(list l){
        if( isEmpty(l) ){
                return l;
        }else{
                node n = popHead(l);
                return append( reverse(l) , n );
        }
}
```

This implementation is very easy to read and understand; it pops the head of the list, reverses the remainder of the list, and then adds the former head to the end of the new list. However, this implementation is very inefficient, as it requires a linear amount of storage to reverse the list. A more efficient implementation would use a loop instead of recursion, and would construct the new list backwards to avoid the linear storage. Without having to think too much about the details of the implementation, we can express these insights in a sketch. First, we know we are going to have to create a new empty list, and we know the implementation is going to need a while loop.

```
list reverseEfficient(list l){
        list nl = new list();
        while( □ ){ □ }
}
```

What we have above is a partial program, but there are an infinite number of ways to complete it. However, we know much more about the solution than what this partial program communicates. For example, we know that the condition for the loop must be a pointer comparison involving some of the memory locations reachable from `l` and `nl`. With the right notation, we can express this information very concisely to the synthesizer. In the SKETCH language, we can define sets of expressions by using regular expression syntax. For example, the set of memory locations which are likely to appear in the comparison can be described as:

```
#define LOC {| (l | nl).(head | tail)(.next)? | null |}
```

Therefore, the set of possible comparison expressions can be defined as

```
#define COMP {| LOC ( == | != ) LOC |}
```

These sets of expressions are called *generators*, and by using them we can begin to bound the set of possible completions for the sketch with little additional effort.

```
list reverseEfficient(list l){
#define LOC {| (l | nl).(head | tail)(.next)? | null |}
#define COMP {| LOC ( == | != ) LOC |}

        list nl = new list();
        while( COMP ){ □ }
}
```

We can do the same thing with the body of the loop. We know that the body will consist of a sequence of assignments to some of the available pointers. We also suspect that not all assignments should happen in all the iterations, so we want to guard them with some condition. We also suspect that a temporary variable will be needed to keep information from one iteration to another, and for good measure, we also allow the synthesizer to use a different iteration condition for the first iteration than for the rest. All of this insight is encoded in the sketch in Figure 1.1. The sketch encodes everything we can easily say about the implementation, and it constraints the search space enough to make it possible for the synthesizer to discover the correct implementation of the list reversal in less than 5 minutes.

In order to solve the sketch, the synthesizer also needs a specification. For this example, the correctness is best defined in terms of the recursive implementation which is easy to check by hand. In sketch, this is done by providing a `main` method like the one shown in Figure 1.2. The main procedure has a set of inputs, and a body containing code and assertions. The synthesizer will try to complete the sketch to ensure that no assertions are violated for any inputs. The type of inputs is limited to integers, bits and fixed length arrays thereof. This is because, as I will show later, the synthesizer relies on an independent validation procedure to establish the correctness of the generated program, and the validation procedures we use can only handle bounded inputs.

The `main` procedure in Figure 1.2 creates two identical lists of length `n` with each node in the list containing a single bit value. It will then ensure that reversing one list with the recursive method and the other one with the resolved sketch results in two identical lists. This is the correctness condition for the sketch, and for large enough values of N, it is enough

```
#define LOC {| (l | nl).(head | tail)(.next)? | null |}
#define LOC2 {| LOC | tmp |}
#define LHS {| (l | nl).(head)(.next)? | nl.tail | tmp |}
#define COMP {| LOC ( == | != ) LOC |}

list reverseEfficient(list l){
        list nl = new list();
        node tmp = null;
        bit c = COMP;
        while(c){
                if( COMP ){ LHS = LOC2; }
                if( COMP ){ LHS = LOC2; }
                if( COMP ){ LHS = LOC2; }
                if( COMP ){ LHS = LOC2; }
                if( COMP ){ LHS = LOC2; }
                c = COMP;
        }
}
```

Figure 1.1: Complete sketch for the linked list reversal problem

```
main(bit[N] elems, int n){
        if( n < N){
        // create an n element list from the input bit-vector.
                list l1 = populate(elems, n);
                list l2 = populate(elems, n);

                l1 = reverse(l1);
                l2 = reverseEfficient(l2);

                assert compare( l1, l2) ;
        }
}
```

Figure 1.2: Specification for lthe linked list reversal

to guarantee the correctness of the resulting program. In the case of our experiment, `N` was set to be equal to 3; values of 4 or 5 work too, but the synthesis process takes longer. The resulting code is shown in Figure 1.3; the code is exactly what the synthesizer produced, except for the fact that I removed a few intermediate temporaries to make the code more readable (for example, the original code had `t1 = l0.head; t2 = t1.next; l_0.tail = t2;` instead of `l_0.tail = l_0.head.next;`). Note that the synthesizer actually discovered a "clever" solution that doesn't use the temporary variable, using the tail pointer of the original list for that purpose instead.

In this example, sketching allowed the programmer to produce an implementation from a clean specification together with a sketch that outlined the programmer's insight about the implementation, but left most of the details unspecified. This was done without resorting to meta-programming or other external formalisms. Moreover, as the next example will show, the technologies developed in this thesis allow the same methodology to be applied to the development of concurrent programs, something that had proved an elusive goal in the field of software synthesis.

```
list reverseSK(list l_0)
{
  list nl_2=new list();
  nl_2.head = null;
  nl_2.tail = null;
  bit c_3=0;
  c_3 = (l_0.head) != (null);
  while(c_3)
  {
    l_0.tail = l_0.head.next;

    l_0.head.next = nl_2.head;

    if((nl_2.tail)!=( l_0.tail)){ nl_2.head = l_0.head; }

    if((null)==( nl_2.tail) ){ nl_2.tail = l_0.head; }

    if((nl_2.head)==( l_0.head)){ l_0.head = l_0.tail; }

    c_3 = (nl_2.tail.next)!=( l_0.head);
  }
  return nl_2;
}
```

Figure 1.3: Solution to listReverse sketch

### 1.2.2 Example: Sketching for concurrency

One of the most compelling applications of sketching is in the domain for concurrent data structures, arguably one of the biggest challenges facing modern systems programming. The challenge comes from the need to maintain the consistency of the data structure in the presence of many simultaneous updates. Moreover, programmers must maintain this consistency while keeping mutual exclusion to a minimum, in order to prevent the data structure from becoming a sequential bottleneck in a highly concurrent application. In order to achieve this, data-structure designers must resort to complex schemes to maintain consistency using only fine grained locking, or even without using locks at all, relying only on atomic primitives provided by the hardware. Additionally, the composition of concurrent objects is far from trivial, so library-based approaches will not shield programmers from the complexities of this domain.

These difficulties make concurrent data structures an ideal domain for sketching. As an example, consider the problem of implementing a concurrent set based on a sorted linked list with sentinel nodes at either end. The basic idea is due to Herlihy [38] and is relatively simple. I show only the remove method. The sequential remove method is quite simple: it traverses the list in search of the element to remove, and when it finds it, it swings the pointer from its predecessor to its successor.

```
bit remove (Set S, int key) {
    bit ret = 0;
    Node prev = null;
    Node cur = S.head;
    find(S, key, prev, cur);
    if (key == cur.key) {
        prev.next = cur.next;
        ret = 1;
    } else {
        ret = 0;
    }
    return ret;
}
```

```
void find (Set S, int key, ref Node prev, ref Node cur) {
        while(cur.key < key){
                prev = cur;
                cur = cur.next;
          }
}
```

A trivial way of making this method concurrent would be to grab a lock upon entry to the remove method and release it on exit. However, this is overly conservative, eliminating any trace of concurrency from the method. Instead, our implementation must use what is called a hand-over-hand locking strategy; the idea is to maintain a sliding window of two locks around some neighborhood of the `cur` and `prev` pointers as the list is traversed. The details of the scheme are tricky; for example, it is not clear how to coordinate the acquiring and releasing of the locks with the update of the pointers. Fortunately, SKETCH can take care of these details for us if we just express the key idea in a sketch.

First, within the `find` method, we know that as the list is traversed, the algorithm should acquire one lock and release one lock, but we do not know how to coordinate the acquisition and release with the pointer updates, so we give the synthesizer the freedom to discover this. Additionally, we also give the synthesizer the freedom to select exactly which lock to acquire and release, and the option to do it conditionally.

```
#define COMP {| ((cur|prev)(.next)? | null) (== | !=) (cur|prev)(.next)?  |}
//            locations in the neighborhood of cur/prev.
#define LOC {| (cur | prev )(.next)? |}
void find (Set S, int key, ref Node prev, ref Node cur) {
    while(cur.key < key){
        reorder{
                prev = cur;
                cur = cur.next;
                if(COMP){ lock(LOC); }// conditionally acquire some lock
                if(COMP){ unlock(LOC); } // conditionally release some lock
        }
    }
}
```

In the sketch, LOC corresponds to a set of locations in the neighborhood of the pointers traversing the list, while COMP is a predicate on those locations. The **reorder** construct gives the synthesizer the freedom to search for the correct strategy for sequencing the update of the pointers with the acquisition and release of locks.

For the **remove** method itself, we know the implementation should keep a sliding window of two locks. The way to enforce this is by telling the synthesizer explicitly that at the end of the remove method, it should release two locks in the neighborhood of the **cur** pointer. As for the beginning of the remove method, we do not know whether we should acquire one lock, both, or none, so we also give this choice to the synthesizer.

```
bit remove (Set S, int key) {
    bit ret = 0;
    Node prev = null;
    Node cur = S.head;
    // let the synthesizer decide whether to acquire locks before find
    // or afterwards. The code is prevented from just acquiring
    // a global lock and releasing it at the end because LOC must be in the
    // neighborhood of cur/prev.
    if(??){ lock (LOC); }
    if(??){ lock (LOC); }
    find(S, key, prev, cur);
    if (key == cur.key) {
        prev.next = cur.next;
        ret = 1;
    } else {
        ret = 0;
    }
    //release the locks.
    unlock (LOC);
    unlock (LOC);

    return ret;
}
```

Writing the sketch was not trivial; the programmer had to know about the hand-over-hand *strategy*. But sketching allows the programmer to communicate the insight to the synthesizer in a natural way, by writing what she already knows about the desired implementation and leaving the rest unspecified. This allows the programmer to view the synthesizer as a programming aid; lowering the cost of adoption, and potentially making synthesis accessible to programmers at large.

Making sketching possible, however, poses new challenges in the design of synthesis algorithms that can make effective use of insight provided as a partial program. The central question for this thesis is whether this approach can be made to work efficiently and reliably enough to have an impact on real world programming problems.

## 1.3 The challenge of sketch based synthesis

The first challenge in supporting a synthesis approach based on partial programs is to define a language that allows programmers to write these partial programs. The language must have clearly defined semantics, so that the set of programs that the sketch can generate and the behavior of such programs can be defined unambiguously. Chapter 2 defines the SKETCH language and its underlying programming model. The semantics of this language are defined in Chapter 3 using a new formalism that allows us to reason about the behavior of a sketch as a function of the choices made for the completion of the holes. The semantics also allows us to reason about the set of solutions to the sketch problem, and how this set is affected by different constructs in the language.

The second challenge is to develop a solution algorithm that implements the semantics defined for the sketching language. This requires a different class of algorithms from those used by deductive synthesis systems. Deductive systems rely on derivation, but this makes it hard to take advantage of partial information about the solution. To see why this is the case, consider the analogy with another well known derivation system: the Rubik cube. In a Rubick cube you have a predefined set of transformations that you must use to transform the cube from an initial state into a final configuration satisfying some constraints (e.g. one color per face). In a Rubick cube, it is very hard to use partial information about the final configuration to speed up the solution. For example, if I know that two red pieces will go together in the final configuration, it is still a lot of work to find a set of transformations that will bring the two pieces together without undoing any partial progress I may

have already achieved. The same is true for synthesis systems based on derivation.

Our solution strategy relies instead on efficient search techniques that can use partial knowledge about the solution to rule out large swaths of the search space and hone into a solution quickly. One of the biggest challenges in turning the synthesis problem into a search problem is the difficulty of establishing the correctness of a candidate solution to the sketch. In deductive synthesis, correctness is often achieved by construction, since the implementation is derived from the specification through semantics preserving derivation rules. By giving up the deductive approach, we lose the correctness by construction, and we are forced to rely on an external validation procedure. The sketch synthesizer we developed uses this apparent limitation to its advantage in a technique we have named counterexample guided inductive synthesis, or CEGIS.

Counterexample guided inductive synthesis is based on an important empirical hypothesis: for most sketches, only a small set of inputs is needed to fully constrain the solution. In other words, it is possible to find a small set of inputs covering all the corner cases in the sketch, such that only a valid solution to the sketch can work correctly for all these inputs. CEGIS uses an efficient SAT based inductive synthesis procedure to produce candidate solutions from small sets of inputs. The crucial observation in the CEGIS algorithms is that the set of corner cases can be discovered automatically by coupling the inductive synthesizer with a validation procedure. Initially, the set of inputs contains only a random input, but once the inductive synthesizer produces its first candidate solution, the solution is checked by the validation procedure. If the candidate is incorrect, the counterexample produced by the validation procedure is fed to the inductive synthesizer, so the next candidate it produces will be guaranteed to work correctly for this corner case. After only a few iterations, the inductive synthesizer will have gathered a representative set of counterexample inputs and will produce a valid candidate which the validation procedure will accept and deliver to the user.

Because of the empirical hypothesis, which we validate in Section 4.3, the CEGIS algorithm is able to converge to a solution after only a handful of iterations, and therefore a handful of calls to the validation procedure. We have observed this even for sketches with candidate and input spaces of astronomical proportions. For example, in one sketch for the AES encryption cipher, shown in Section 6.5, the synthesizer was able to derive the contents of over 1024 32-bit integer constants after analyzing only 600 candidates.

Additionally, the algorithm has the advantage of separating the synthesis and the

validation tasks. The most dramatic consequence of this will be seen in Chapter 8, where we will show that CEGIS can be extended to synthesize concurrent programs *without the need to reason about concurrency in the inductive synthesizer*. I will show that by separating the synthesis and the validation tasks, we are able to combine our sequential inductive synthesizer with SPIN, a bounded model checker that handles concurrency, to efficiently synthesize concurrent programs in a matter of minutes.

The power and generality of the CEGIS algorithm, even allows us to develop specialized synthesizers for important domains through problem reduction. Chapter 11 describes how we developed a sketch synthesizer for the domain of stencil computations, an important domain in scientific computing that finds uses in fluid dynamics, signal processing, and a number of other branches of science and engineering. For this synthesizer, we were able to encode domain specific insight as a problem reduction procedure that was able to take stencil sketches and produce from them simplified sketch problems which CEGIS could handle in a matter of minutes. We were able to use this specialized synthesizer to quickly implement a MultiGrid solver for partial differential equations containing many difficult implementation strategies from the literature.

Ultimately, this thesis will demonstrate that using the above techniques, the SKETCH system can take very clean and general sketches for a variety of challenging programming problems, and produce working implementations in much less time than what the comparable implementation would take by hand.

# Chapter 2

# The Sketching Programming Model

The first major contribution of this thesis is to develop Sketching, a new programming model that relies on synthesis from partial programs. We have implemented this programming model in SKETCH, a simple language intended to serve as a model on how to incorporate synthesis support into existing languages. SKETCH is a simple procedural language extended with a single new construct: a basic integer hole. Together with some simple syntactic sugar, this new construct gives the programmer a robust mechanism to express insight about an implementation while leaving unspecified a lot of the challenging details. The goal is to give the programmer the tools to exploit the synthesizer's ability to derive low level implementation details while maintaining full control over the shape of the resulting code.

In this section we describe the Sketching programming model as implemented in the SKETCH language. The section shows how the basic integer hole can be used in conjunction with standard language features to harness the power of the synthesizer, relieving the programmer from some of the most tedious aspects of programming. The section also gives a formal description of the set of programs that can be synthesized from a sketch. The goal is to provide a complete picture of the sketching programming model.

## 2.1 The Basics

The SKETCH language is built on top of a very small core consisting of a simple procedural language and a single sketching construct: the integer hole denoted by the token **??**. Form the point of view of the programmer, the integer hole is a placeholder that the

synthesizer must replace with a suitable integer constant. The synthesizer ensures that the resulting code will avoid any assertion failures under all possible inputs. For example, the following code snippet can be regarded as the "Hello World" of sketching.

```
void main(int x){
        int y = x * ??;
        assert y == x + x;
}
```

This program illustrates the basic structure of a sketch. It contains three elements you are likely to find in every sketch: (i) a main procedure, (ii) holes, and (iii) assertions. The assertions serve as a specification for the sketch; they express safety and correctness properties which the synthesized program should satisfy under all possible inputs to `main`[1]. The hole is used to define the range of action of the synthesizer. The synthesizer is free to replace the integer hole with a constant, but not to add additional statements or to change the behavior of the program in other ways. For the sketch above, the synthesized code will look like this.

```
void main(int x){
        int y = x * 2;
        assert y == x + x;
}
```

Discovering the correct value of a constant may seem insignificant in the context of the challenges faced by programmers, but for many difficult programming tasks, this is all that is needed to take a high level insight and turn it into an efficient implementation.

As a small but realistic example of this, consider the problem of isolating the rightmost 0-bit in a word `x`. For example, for the word `01010011`, we would like to produce a word containing a `1` in the position of the rightmost `0`; that is `00000100`. There is a trick to do this using only three instructions. You may remember it: the trick takes advantage of the fact that adding a `1` to a string of ones preceded by a zero turns all the ones into zeros and turns the next zero into a one (*i.e.* `000111 + 1 = 001000`) . You may not remember the details, but with sketching you don't have to; you can let the synthesizer discover them. All

[1] As we will describe in further chapters, this guarantee will be weaker for many sketches. The strength of the correctness guarantee is determined by the decision procedure used for validation. Since we restrict ourselves to bounded decision procedures, we are only able to analyze integer inputs falling within a bounded range

you need to remember is the general form of the solution to encode the problem as a sketch. Specifically, you need to remember that the solution involved the addition of a constant to `x`, a negation, and a bitwise `and`. The expression `~(x + ??) & (x + ??)` encodes most of the expressions matching this criteria, and when given a suitable specification, the synthesizer can easily find the right expression.

```
int W = 32;
void main(bit[W] x){
        bit[W] ret = ~(x + ??) & (x + ??);
        //specification: check that ret has the desired property.
        bit found = 0;
        for (int i = 0; i < W; i++)
                if (found || x[i]) { assert ret[i]==0; }
                else{ assert ret[i]==1; found = 1;  }
}
```

In less than a second, the synthesizer is able to discover that the correct expression is `~(x+0) & (x + 1)`. If you think this problem was too easy given the initial hint, consider this question: Can the same trick be used to find the rightmost 1? Without thinking too hard about the problem, we can ask the sketch synthesizer by changing the condition in the check from `(found || x[i])` to `(found || !x[i])`:

```
int W = 32;
void main(bit[W] x){
        bit[W] ret = ~(x + ??) & (x + ??);
        //specification: check that ret has the desired property.
        bit found = 0;
        for (int i = 0; i < W; i++)
                if (found || !x[i]) { assert ret[i]==0; }
                else{ assert ret[i]==1; found = 1;  }
}
```

It again takes less than a second for the synthesizer to tell us that yes, the same basic trick applies, but now the expression is `~(x + 0xFFFFFFFF) & (x+0)`.

In the above example, it was relatively clear that the tricky details in the implementation involved discovering a few constants. In many cases, however, the details in question

do not involve any missing constants. For example, consider the problem of swapping two bit-vectors `x` and `y` without using a temporary register. The insight is that the numbers can be swapped by assigning `x xor y` to `x` and `y` repeatedly in a clever way. The challenge is to find the right sequence of assignments. The insight, therefore, involves no integer constants, but the integer hole can still be used to encode it:

```
int W = 32;
void main(bit[W] x, bit[W] y){
        bit[W] xold = x;
        bit[W] yold = y;
        if(??){ x = x ^ y; }else{ y = x ^ y; }
        if(??){ x = x ^ y; }else{ y = x ^ y; }
        if(??){ x = x ^ y; }else{ y = x ^ y; }
        assert y == xold && x == yold;
}
```

The sketch above uses the integer hole to encode the choice between assigning `x ^ y` to `x` or to `y`. In less than a second, the synthesizer discovers that the three holes should evaluate to **false**, **true** and **false** respectively. After replacing the holes with constants, the synthesizer will perform a small amount of cleanup, eliminating the unnecessary conditionals to produce the code shown below.

```
int W = 32;
void main(bit[W] x, bit[W] y){
        bit[W] xold = x;
        bit[W] yold = y;
        y = x ^ y;
        x = x ^ y;
        y = x ^ y;
        assert y == xold && x == yold;
}
```

The cleanup process performs some constant propagation and eliminates unnecessary conditionals. Its goal is not so much to optimize the code, but to make it cleaner and more readable, and to eliminate structures that were there only for the purpose of giving freedom to the synthesizer. For this reason, the cleanup process is biased towards predictabil-

ity, for example avoiding fixed point computations which may provide higher accuracy, but make the cleanup slower and less predictable.

### 2.1.1 Stating it formally

The preceding section informally described how the synthesizer replaces the integer holes to generate concrete programs. This section describes this process more formally by providing a precise characterization of the set of programs that may be generated by the sketch. This is accomplished by using the formalisms of online partial evaluation [9, 22, 41].

In the notation of Beckman *et al.* [9], online partial evaluation divides the inputs to a program $P(Q, A)$ into static $Q$ and dynamic $A$. The idea is that program $P$ is called repeatedly with different values for $A$ but with identical values for $Q$. Therefore, it's worthwhile to produce a specialized program $P_Q(A)$ such that for all inputs $A$, $P_Q(A) = P(Q, A)$. The specialized program $P_Q$ is generated by statically evaluating any expressions that depend on static data, and generating residual code for those expressions in the program that depend on the dynamic inputs.

In the case of sketching, each integer hole can be though of as a static input to the sketch. The job of the synthesizer is to discover a correct value for these static inputs, and then partially evaluate the sketch with respect to them. More formally, we can define a function $\phi : H \rightarrow \mathbb{Z}$ to denote an assignment of concrete values to holes in the sketch. The set $H$ is the set of integer holes in the program, so $\phi(\texttt{??}_i)$ corresponds to the integer value of the $i^{th}$ hole in the sketch. We call function $\phi$ a *control*, because it controls how the generated program is going to behave. By partially evaluating a sketch $P$ with respect to a control $\phi$, we can produce a *candidate program* $PE(P, \phi) = P_\phi$. Of course, not all candidate programs will be valid solutions to the sketch; the job of the synthesizer is to find a control $\phi$ that leads to a candidate program satisfying the specification.

The partial evaluation procedure is defined by means of rewrite rules that describe how the residual code is generated for each program construct. The partial evaluator maintains an internal state $\hat{\sigma} : L \rightarrow \mathbb{A}$, where $L$ is the set of all variables in the program, and $\mathbb{A} = \mathbb{Z} \cup \{\perp\}$ is a set of abstract values in the spirit of constant propagation: variables found to be constant are mapped to their constant values; dynamic variables are mapped to the special value $\perp$ [2].

[2] For now, we assume all variables to be integers; the boolean values **true** and **false** are represented with the integers 1 and 0 respectively.

For expressions, the partial evaluation of an expression $e$ under state $\hat{\sigma}$ through control $\phi$ results in a residual expression $e'$ as well as a value $v$ which corresponds to the static value of expression $e$, or to $\perp$ if expression $e$ is dynamic.

$$\langle e, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e', v \rangle$$

The rules for expressions are fairly intuitive. For example, variables are evaluated according to the following two rules:

$$\frac{\hat{\sigma}(\mathbf{x}) = \perp}{\langle \mathbf{x}, \hat{\sigma} \rangle \xrightarrow{\phi} \langle \mathbf{x}, \perp \rangle} \qquad \frac{\hat{\sigma}(\mathbf{x}) = n \neq \perp}{\langle \mathbf{x}, \hat{\sigma} \rangle \xrightarrow{\phi} \langle \mathbf{n}, n \rangle} \quad \text{(E-Var)}$$

The rules encode the fact that if the variable has a known value, it will evaluate to that value, and will be replaced in the code with a constant; if it has an unknown value, it will evaluate to $\perp$, and remain unchanged in the code.

Holes are replaced by their concrete value according to the control function $\phi$.

$$\frac{\phi(\texttt{??}_i) \Rightarrow n}{\langle \texttt{??}_i, \hat{\sigma} \rangle \xrightarrow{\phi} \langle \mathbf{n}, n \rangle} \quad \text{(E-Hole)}$$

The rules for arithmetic correspond to standard partial evaluation rules. An integer literal $\mathbf{n}$ is evaluated to a constant $n$ and generates the literal $\mathbf{n}$. In the notation, we use the symbol $\circ$ to stand for an arbitrary binary operator, such as addition. The rules for $\circ$ evaluate the operator if both operands are constants. Otherwise, the residual expression is generated. For operators with short-circuit behavior, rules like E-AndShort below are able to produce a static value even if one of the inputs is not static.

$$\frac{}{\langle \mathbf{n}, \hat{\sigma} \rangle \xrightarrow{\phi} \langle \mathbf{n}, n \rangle} \quad \text{(E-Lit)}$$

$$\frac{\langle e_1, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e_1', n_1 \rangle \quad \langle e_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e_2', n_2 \rangle \quad n_1 \circ n_2 \Rightarrow n}{\langle e_1 \circ e_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle \mathbf{n}, n \rangle} \quad \text{(E-Op1)}$$

$$\frac{\langle e_1, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e_1', \perp \rangle \quad \langle e_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e_2', n_2 \rangle}{\langle e_1 \circ e_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e_1' \circ e_2', \perp \rangle} \quad \text{(E-Op2)}$$

$$\frac{\langle e_1, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e_1', \texttt{false} \rangle \quad \langle e_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e_2', v \rangle}{\langle e_1 \wedge e_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle \texttt{false}, 0 \rangle} \quad \text{(E-AndShort)}$$

Partial evaluation of statements produces a residual statement in addition to modifying the state.

$$\langle c, \hat{\sigma} \rangle \xrightarrow{\phi} \langle c', \hat{\sigma}' \rangle$$

The rule states that, given a state $\hat{\sigma}$, a statement $c$ is partially evaluated to a residual statement $c'$ and a new state $\hat{\sigma}'$ by using the control $\phi$. In future sections, we will sometimes use the notation $PE(c, \phi) = c'$ as a shorthand for $\langle c, \hat{\sigma} \rangle \xrightarrow{\phi} \langle c', \hat{\sigma}' \rangle$ when the final state is not relevant. The partial evaluation rules for statements are shown below.

$$\frac{\langle e, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e', v \rangle}{\langle x := e, \hat{\sigma} \rangle \xrightarrow{\phi} \langle x := e', \hat{\sigma}[x \mapsto v] \rangle} \quad \text{(S-Asgn)}$$

$$\frac{\langle e, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e', true \rangle \;\; \langle c_1, \hat{\sigma} \rangle \xrightarrow{\phi} \langle c_1', \hat{\sigma}_1' \rangle}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle c_1', \hat{\sigma}_1' \rangle} \quad \text{(S-If1)}$$

$$\frac{\begin{array}{c}\langle e, \hat{\sigma} \rangle \xrightarrow{\phi} \langle e', \bot \rangle \\ \langle c_1, \hat{\sigma} \rangle \xrightarrow{\phi} \langle c_1', \hat{\sigma}_1' \rangle \quad \langle c_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle c_2', \hat{\sigma}_2' \rangle \quad \hat{\sigma}_u = \hat{\sigma}_1' \cap \hat{\sigma}_2'\end{array}}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle \textbf{if } e' \textbf{ then } c_1' \textbf{ else } c_2', \hat{\sigma}_u \rangle} \quad \text{(S-If2)}$$

$$\frac{\langle c_1, \hat{\sigma} \rangle \xrightarrow{\phi} \langle c_1', \hat{\sigma}_1' \rangle \;\; \langle c_2, \hat{\sigma}_1' \rangle \xrightarrow{\phi} \langle c_2', \hat{\sigma}_2' \rangle}{\langle c_1; c_2, \hat{\sigma} \rangle \xrightarrow{\phi} \langle c_1'; c_2', \hat{\sigma}_2' \rangle} \quad \text{(S-Seq)}$$

$$\frac{\langle e, \hat{\sigma}_\bot \rangle \xrightarrow{\phi} \langle e', v \rangle \;\; \langle c, \hat{\sigma}_\bot \rangle \xrightarrow{\phi} (c', \hat{\sigma}')}{\langle \textbf{while } e \textbf{ do } c, \hat{\sigma} \rangle \xrightarrow{\phi} \langle \textbf{while } e' \textbf{ do } c', \hat{\sigma} \cap \hat{\sigma}' \rangle} \quad \text{(S-While)}$$

The special state $\sigma_\bot$ maps all variables to $\bot$. The intersection of two states used in the S-If2 and the S-While rule is defined as $\hat{\sigma}_1 \cap \hat{\sigma}_2 = [x \mapsto \hat{\sigma}_1(x) \cap \hat{\sigma}_2(x)]$. The $\cap$ operation on $\mathbb{A}$ is commutative, and is defined by the following rules: given two different values $v_1, v_2 \in \mathbb{A}$, we define $v_1 \cap v_1 = v_1$, $v_1 \cap v_2 = \bot$, and $v_1 \cap \bot = \bot$.

An important property of the above rules is that they are purposely conservative to avoid surprising the programmer. For example, the rule for loops does not attempt to do any fixed point computation, allowing one to reason locally about how the body of the loop will be transformed.

The last rule is the rule that simplifies a procedure in the program. Each procedure will be evaluated independently, and the result will be a concrete candidate for the sketch.

$$\frac{\langle c, \hat{\sigma}_{\perp}\rangle \xrightarrow{\phi} \langle c', \hat{\sigma}'\rangle}{\langle (\textbf{def}\ f(in)\ c), \hat{\sigma}\rangle \xrightarrow{\phi} \langle (\textbf{def}\ f(in)\ c'), \hat{\sigma}\rangle} \qquad \text{(F-Declaration)}$$

These rules summarize the informal description of the language given earlier. While the programmer doesn't need to know these rules in order to use the system, the fact that the synthesizer operates according to them should make it easy for the programmer to develop an intuition for the kind of code that may be generated by the SKETCH synthesizer.

## 2.2 Abstraction in SKETCH

Procedures are one of the most commonly used forms of abstraction in many languages. Procedures allow the programmer to hide the details of a computation behind a simple interface. SKETCH supports procedures exactly as one would expect: holes within them are syntactically replaced with integers that ensure the correctness of the generated program.

```
int linexp(int x, int y){
        return ??*x + ??*y + ??;
}
void main(int x, int y){
        assert linexp(x,y) >= 2*x + y;
        assert linexp(x,y) <= 2*x + y+2;
}
```

For example, for the routines above, there are many different solutions for the holes in `linexp` that will satisfy the first assertion, and there are many that will satisfy the second assertion, but the synthesizer will chose one of the candidates that satisfy them both.

```
int linexp(int x, int y){
        return 2*x + y;
}
void main(int x, int y){
        assert linexp(x,y) >= 2*x + y;
        assert linexp(x,y) <= 2*x + y + 2;
}
```

The procedure `linexp` originally had holes, and therefore corresponded to a set of functions. However, the synthesizer completed the holes to give the procedure a *single* concrete meaning to be used across all calling sites. This gives procedures the same power of abstraction that they would have in the absence of sketching. But, as the following example illustrates, sketching creates the need for a mechanism to abstract *sets* of functions.

```
int[25] transpose5x5(int[25] mat){
        int[25] out;
        for(int i=0; i<5; ++i) for(int j=0; j<5; ++j){
                out[ ??*i + ??*j + ??] = mat[??*i + ??*j + ??];
        }
        return out;
}

void main(int[25] mat, int i, int j){
        int[25] out = transpose5x5(mat);
        assert !(i < 5 && j < 5) || mat[i*5 + j] == out[j*5 + i];
}
```

In the above sketch, `transpose5x5` is a procedure which abstracts the matrix transpose function for $5 \times 5$ matrices. However, within the transpose procedure, the expression `??*i + ??*j + ??` is repeated twice. We would like to abstract this expression to avoid redundancy in the code. However, we can not abstract this expression into a procedure because each use of `??*i + ??*j + ??` has to resolve to a different linear expression. Therefore, we need an abstraction which represents the entire set of functions encoded by `??*i + ??*j + ??`, rather than a single one like the procedure does.

The SKETCH language allows the programmer to abstract a set of functions into a *generator*. For each use of the generator, the synthesizer is free to chose a different function, so generators can serve as an abstraction mechanism for what we call *rich holes*. For the above example, we can abstract `??*i + ??* j + ??` into a generator that represents the set of linear expressions involving `i` and `j`.

```
generator int legen(int i, int j){
        return ??*i + ??*j + ??;
}
int[25] transpose5x5(int[25] mat){
        int[25] out;
        for(int i=0; i<5; ++i) for(int j=0; j<5; ++j){
                out[ legen(i,j) ] = mat[ legen(i,j) ];
        }
        return mat;
}
```

Each call to the generator will resolve to a different expression, resulting in a correct implementation for the $5 \times 5$ transpose.

```
int[25] transpose5x5(int[25] mat){
        int[25] out;
        for(int i=0; i<5; ++i) for(int j=0; j<5; ++j){
                out[ 5*i + j ] = mat[ i + 5*j ];
        }
        return mat;
}
```

Programmers are encouraged to think of generators as procedures which are inlined into their calling context before the sketch is synthesized, so each call to the generator will be independent from every other call. They are similar to a macro, but with the advantage of type safety, and the ability to be recursive.

Recursion greatly enhances the expressive power of generators over simple macros. For example, the generator below uses recursion to describe the set of polynomials in `x` of degree `n`.

```
generator int poly(int n, int x){
        if( n == 0) return ??;
        else return x * poly(n-1, x) + ??
}
```

For recursive generators, the semantics as functions which are inlined into their calling context are a bit more problematic due to the issue of termination. In general, programmers should avoid writing generators where the level of inlining is a function of the inputs to `main`, but they can usually assume that the synthesizer will inline by the correct amount.

### 2.2.1 Stating it formally

The advantage of procedures is that we can reason about them locally; the synthesizer can partially evaluate each procedure independently using the F-Declaration rule. Calls to procedures are handled by partially evaluating their arguments.

$$\frac{(\textbf{def}\ f(in)\ c)\ \ \langle e, \hat{\sigma}\rangle \xrightarrow{\phi} \langle e', v\rangle}{\langle f(e), \hat{\sigma}\rangle \xrightarrow{\phi} \langle f(e'), \hat{\sigma}[\texttt{@} \mapsto \bot]\rangle} \qquad \text{(S-CallProc)}$$

To simplify the exposition, procedure calls are assumed to be statements rather than expressions. They return values by writing to a special variable `@` and have no other side effects besides writing to this variable.

By contrast, Generators require some additional machinery because the value of each hole actually depends on the calling context. A calling context is defined formally as a sequence of generator calling sites $\tau = g_{i_0} \cdot g_{i_1} \cdot \ldots \cdot g_{i_n}$, where the $g_{i_k}$ are the id's of each calling site, and the $\cdot$ operator is used to denote concatenation. We use the symbol $\tau_\emptyset$ to refer to the empty call stack.

The function $\phi$ will now produce values for holes depending on their calling context. We will use the notations $\phi(\texttt{??}_i, \tau) = \phi_\tau(\texttt{??}_i)$ interchangeably to refer to the value of hole $\texttt{??}_i$ under calling context $\tau$.

With this extended notation, F-Declaration rule can be restated as follows.

$$\frac{\langle c, \hat{\sigma}_\bot\rangle \xrightarrow{\phi_{\tau_\emptyset}} \langle c', \hat{\sigma}'\rangle}{\langle(\textbf{def}\ f(in)\ c), \hat{\sigma}\rangle \xrightarrow{\phi_\tau} \langle(\textbf{def}\ f(in)\ c'), \hat{\sigma}\rangle} \qquad \text{(F-Declaration)}$$

This rule makes explicit the fact that the body of a procedure will always be partially evaluated under the empty calling context $\tau_\emptyset$.

By contrast, generator declarations are simply eliminated; this is because the generator doesn't have meaning on its own; its meaning is defined by its calling context.

$$\frac{}{\langle(\textbf{defgen}\ g(in)\ c), \hat{\sigma}\rangle \xrightarrow{\phi_\tau} \langle empty, \hat{\sigma}\rangle} \qquad \text{(G-Declaration)}$$

At each call site, the generator must be partially evaluated under the current calling context and then inlined.

$$\frac{\langle e, \hat{\sigma}\rangle \xrightarrow{\phi_\tau} \langle e', v'\rangle \quad (\textbf{defgen}\ g(in)\ c) \quad \langle c, \hat{\sigma}_\perp[in \mapsto v']\rangle \xrightarrow{\phi_{\tau\cdot g_i}} \langle c', \hat{\sigma}'\rangle}{\langle g(e), \hat{\sigma}\rangle \xrightarrow{\phi_\tau} \langle rename(in := e'; c'), \hat{\sigma}[@ \mapsto \hat{\sigma}'(@)]\rangle} \quad \text{(S-CallGen)}$$

The rule is a little more involved than some of the previous rules. The rule constructs a new calling context $\tau \cdot g_i$ by appending the id of the current call to the current calling context. The partially evaluated body of the generator is inlined into the original call site, and variables are renamed to avoid name conflicts.

The handling of generators shares some similarities with polyvariant partial evaluation [16]. Polyvariant partial evaluation deals with the case when a function may be called with many different values for its static parameters, just like generators can be called under many different calling contexts. One of the main issues in polyvariant partial evaluation is how to avoid an explosion in the number of different versions of a given function. In Section 3.2.1 I will elaborate on how generators deal with the this issue; it will be done primarily by placing bounds on the recursive application of the S-CallGen rule.

The notation is a little intimidating, but in practice, generators are similar enough to macros and inlined procedures that they should be easily assimilated by programmers in the field.

## 2.3 Concurrency in SKETCH

One of the most compelling applications of sketching is the synthesis of concurrent datastructures. In order to express them, we extended the SKETCH language to give users the ability to launch threads and define synchronization primitives. None of these features are specific to sketching; they are just standard concurrency primitives. The insights involved in concurrent programs can be very different from those involved in sequential programs, but the same sketching constructs which proved useful in the sequential case can be used to express insight about synchronization and mutual exclusion in concurrent sketches. The application of sketching to concurrent data structures did inspire us to create a number of high-level sketching constructs which are particularly useful when writing concurrent sketches. However, none of these constructs are specific to concurrent datastructures; in fact, they've proven extremely useful for sequential sketches as well.

Sketch supports a fork-join concurrency model. Threads are created with the construct **fork (int i, N)** $c$ which spawns $N$ threads and blocks until all $N$ threads terminate. Each thread executes the statement $c$. Each thread has a unique thread id ranging from 0 to $N-1$ that can be read from the index variable `i`. The current version of the SKETCH system limits the structure of sketches to contain a single **fork**. This is only for engineering reasons, as the algorithms used to handle concurrency generalize relatively easily to multiple and even nested **fork** statements.

All variables declared inside $c$ are thread-local. All other variables, together with the heap, constitute sequentially consistent shared memory. In other words, all reads and writes to shared variables occur atomically, and the result of the computation will always correspond to some sequential ordering of the operations of all the threads. As we shall see in Chapter 8, the synthesis algorithm will exploit this property extensively.

The SKETCH language also includes support for synchronization primitives. Deciding on a set of primitives to support was a challenge because of the variety of synchronization primitives supported by various platforms. Moreover, we also wanted to support atomic primitives such as compare-and-swap (`CAS`), which are supported in some platforms, and which are very useful in defining lock-free data structures.

To address this problem while keeping the language small, the SKETCH language has native support for a single synchronization construct: the conditional atomic section [40]. A conditional atomic is an atomic section that blocks until its condition holds. Conditional atomics are a very high level synchronization construct which is very difficult to implement efficiently in a general setting. However, it can be used to define a wide range of synchronization primitives, from basic locks, to semaphores, to atomic primitives such as compare-and-swap. The idea is that implementations of the language can be customized to support the synchronization primitives available in the target platform, but the primitives can be can be described at the language level using the conditional atomic section.

As an example of this, consider the basic lock. A lock can be easily expressed in terms of conditional atomics as shown below.

```
struct Lock {
   int owner = -1;
}                              unlock(Lock lk) {
lock(Lock lk) {                  assert lk.owner == pid;
  atomic(lk.owner == -1){        lk.owner = -1;
     lk.owner = pid;           }
  }
}
```

It is worth noting that SKETCH does not support spin-locks, so even they must be modeled using conditional atomics. We discuss this limitation in more general terms in Chapter 7.

## 2.4 Syntactic Sugar

We conjecture that the language described so far contains the basic ingredients needed to describe arbitrary sets of programs. Even if this is so, however, the language is relatively low level, which can sometimes make it difficult to express programmer's insight in a sketch. To simplify the sketching process, SKETCH offers a rich set of higher level sketching constructs which are implemented as syntactic sugar over the base language, and which allow users to more naturally express intent without getting bogged down in the details.

### 2.4.1 Higher level constructs

We currently support three high level sketching constructs which have shown to be very effective in writing sketches: (i) a **repeat** construct for repeated patterns (ii) regular-expression expression generators, (iii) a **reorder** block.

**Repetition** Often in a sketch, we want to express the fact that a block is composed of $N$ statements, all of which can be generated from the same sketched statement. The construct `repeat(e){ body; }` allows us to express this concept in a clean and concise way as illustrated by the following example.

Consider the problem of computing the logarithm base two of an integer represented as a vector of bits. A strategy for this can be succinctly described as follows:

> Computing the logarithm base two is equivalent to finding the index of the most significant 1 in the word. We can compute this logarithmically through a divide and conquare approach. Use a bitmask to focus on one half of the current word at a time. The value of the most significant bit of the answer will be one if the first

> half of the word contains a `1`, and zero otherwise. Follow this logic recursively to compute the subsequent bits of the answer; i.e. take the half which contained the most significant 1 and divide it in half to find the second most significant bit of the answer, and so forth.

In the above algorithm, the step that has to be repeated involves checking whether a fragment of the word is equal to zero, and if it isn't, then setting the correct bit in the answer to `1` and shifting the current value so that on the next iteration, the fragment of interest will be in the same position regardless of whether the `1` was in the first half or the second half. We need to do this a few times; it's not immediately obvious how many, but we can use the **repeat** construct to leave this decision up to the synthesizer.

```
bit[W] sklog2(bit[W] in){
        bit[W] ret = 0;
        repeat (??){
                if (!iszero(in & ??)) { //Is the chosen fragment equal to zero?
                    ret = ret | ??;  // If not, set a bit in the output to one
                    in = in >> ??; // and realign
                }
        }
        return ret;
}
```

Note that **repeat** is different from a **for** loop because the body of repeat will actually be replicated, so the holes in each iteration may contain different values.

In general, writing a statement of the form

```
...
repeat(e){ body; }
...
```

is equivalent to declaring a generator like the one below, and then calling it as `repeatGen(vars, e);`, where `vars` is the set of variables referenced by the body of the loop.

```
generator void repeatGen(ref vars, int i){
        if( i > 0){
                body;
                repeatGen(vars, i-1);
        }
}
```

**Regular-expression expression generators** Regular-expression generators (hereafter RE-generators) allow the programmer to sketch both r-value and l-value expressions with a restricted regular grammar.

The RE-generator construct has the form **{|**$e$**|}**, where $e$ is a regular expression literal. The semantics of the construct is that the synthesizer substitutes the syntactic occurrence of the construct with a string from $L(e)$ such that the substitution resolves the sketch. RE-generators are not simply expanded as macros, however; for programmability, we require that each component regular expression $e$ be well typed.

RE-generators are typically used to enumerate symbolic memory locations or values. For example, the following PSKETCH fragment shows how we sketched the use of a *compare-and-swap (CAS)* instruction in a concurrent doubly-linked data structure.

```
CAS({| head(.next)? (.next)? |},
    {| newNode(.next|.prev)? |},
    {| newNode(.next|.prev)? |})
```

The programmer's insight is that the CAS should compare a node in the neighborhood of `head` with some node in the neighborhood of `newNode`, and if they match, replace the first node with a node in the neighborhood of `newNode`. The RE-generators are used to specify these sets of nodes, so the code above effectively specifies 27 different possible uses of CAS that match the programmer's intuition.

We made a design decision to support only two regular expression operators: choices $e_1$ `|` $e_2$ and optional expressions $e$`?`. The exclusion of Kleene closure might seem arbitrary at first sight, but keep in mind that RE-generators are used to generate bounded program text. In real code, it is unusual to find chains of pointer dereferences of the form `{|p(.next)*|}` with more than two or three levels of dereferencing, so adding Kleene closure would increase the search space without any significant programmability benefit.

```
#define NODE {| (tprev|cur|prev)(.next)? |}
#define COMP {| (!)? ((null|cur|prev)(.next)? ==
                      (null|cur|prev)(.next)?) |}
while(cur.key < key){
  Node tprev = prev;
  reorder {
    if (COMP) { lock (NODE); }
    if (COMP) { unlock (NODE); }
    prev = cur;
    cur = cur.next;
  }
}
```

Figure 2.1: A sketch of hand-over-hand locking.

**Reorder block** Concurrent data structures often depend on careful statement ordering to satisfy desired invariants. For this reason, we extended PSketch with a **reorder** construct that leaves the synthesizer in charge of determining the correct order for the statements in a block of code. The synthesizer considers all possible orders of these statements and selects one that, together with other choices made by the synthesizer, turns the sketch into a program that meets the specification.

One of the most compelling uses of **reorder** is to describe a "soup" of operations which, when ordered in the right sequence, can produce a correct program. We saw an example of this in the introduction, where we used **reorder** in the hand-over-hand locking sketch. In this example we knew that we had to acquire and release some locks, and also walk some pointers, so we placed this soup of operations in a **reorder** block and allowed the synthesizer to discover an ordering that would guarantee mutual exclusion. The sketch is shown in Figure 2.1, and the synthesized code is shown in Figure 2.2. Note how the synthesizer used the freedom to reorder statements to discover the correct strategy for acquiring and releasing the locks. [3]

[3] Sketch does not necessarily resolve **reorder** so that it minimizes mutual exclussion. If optimality is desired, we believe the best way to achieve it is to synthesize many correct candidates and select the best one by measuring the performance of each, as is done in autotuning [12]. Still, the programmer can use

```
while(cur.key < key){
    if (prev != null)
      unlock (prev);
    lock (cur.next);
    prev = cur;
    cur = cur.next;
}
```

Figure 2.2: The sketch from Figure 2.1, resolved.

### 2.4.2 Reference Implementations as Specifications

There is a large class of problems for which it is easy to write a reference implementation if one doesn't care much for performance. It's only when one tries to write an *efficient* implementation that the programming task becomes difficult. When sketching such programs, it makes sense to us a reference implementation as a specification by asserting that, for all inputs, the output of the sketched implementation must match the output of the reference one. For example, in Section 2.4.1, we showed a sketch `sklog2` for an efficient implementation of a procedure to compute the logarithm base two of an integer. To ensure that the implementation produced by the synthesizer is correct, we can require that `sklog2` be equivalent to a simpler but inefficient implementation of log2.

```
bit[W] log2(bit[W] in) {
        bit[W] i = castInt(W);
        bit[W] minusone = 0; minusone = !minusone;
        for(int t=0; t<W; ++t){
                i = i + minusone;
                if (in[(int)i]){
                        return i;
                }
        }
}
```

assert statements to constrain solutions to only those with mutexes that are, *e.g.*, separated by at most two statements.

```
void main(bit[W] in){
        assert sklog2(in) == log2(in);
}
```

The SKETCH language provides some syntactic sugar to make it easier to assert these equivalences. In SKETCH, we use the keyword **implements** to declare that one procedure should be equivalent to another procedure. Therefore, rather than writing the main procedure above, we can simply add **implements** log2 to the declaration of sklog2.

```
bit[W] sklog2(bit[W] in) implements log2{
        ...
}
```

The implements keyword can be understood as syntactic sugar. Declaring that one procedure p1(in) implements another procedure p2(in) is equivalent to adding an extra input in to main, and adding **assert** p1(in) == p2(in); at the beginning of main. However, the SKETCH synthesizer also exploits the additional semantic information provided by **implements** regarding the equivalence of the two procedures. For example, if the sketch is recursive, the synthesizer will replace recursive calls to the sketch with calls to the spec.

The sketch language together with these higher-level constructs and syntactic sugar provides an expressive language to communicate insight through partial programs. The challenge is to develop a synthesis algorithm that is able to use this insight to efficiently produce the implementation desired by the programmer. In the next section I describe the synthesis algorithms I developed as part of this thesis which make synthesis from partial programs possible.

# Part II

# Solution of Sequential Sketches

# Chapter 3

# Synthesis Semantics of SKETCH

The SKETCH synthesizer resolves sketches by formulating the synthesis problem as a constraint satisfaction problem. In order to do this, it needs a precise definition of the semantics of a sketch that allows it to reason about the correctness of the sketch as a function of the holes. In principle, the semantics of a sketch can be trivially defined in terms of the semantics of all the programs which the sketch can generate. But in practice, it is difficult to answer semantic questions about a sketch if these have to be reduced to semantic questions about each of its candidate solutions. Instead, we need a formalism which allows us to: (i) describe the semantics of all the candidates simultaneously, (ii) reason about how the value of each hole affects the meaning of the synthesized program, and (iii) reason about which values for the holes are valid and which ones are not.

This section introduces the *synthesis semantics* of the SKETCH language, described using the formalisms of denotational semantics [55]. The semantics satisfies our three desired attributes. It describes the semantics of all candidates simultaneously by tracking the value of each variable as a function of the control $\phi$, the concrete values assigned to each hole. This allows us to understand how changes in the values of holes affect the state of the program and therefore its meaning. Additionally, the synthesis semantics tracks how each statement in the program affects the set of valid controls, and therefore the set of valid candidates. The semantics will enable a simple constructive definition for the set of valid solutions to a sketch which Chapter 4 will use to frame the synthesis problem as a constraint satisfaction problem.

| | |
|---|---|
| $L$ | The set of variables in the program. $x$ is used to refer to an arbitrary variable. |
| $H$ | The set of holes in the program. All holes are assumed to be uniquely labeled as $??_i$. |
| $\mathcal{T}$ | The set of calling contexts. A calling context $\tau = g_{i_0} \cdot g_{i_1} \cdot \ldots \cdot g_{i_n}$ is a sequence of generator call sites. $\tau_\emptyset$ is the empty context. |
| $\mathbf{\Phi} = H \times \mathcal{T} \rightarrow \mathbb{Z}$ | The set of control functions. A control $\phi$ corresponds to a particular assignment of integers to holes, where each hole is identified by its label $??_i$ together with its calling context $\tau$. |
| $\mathcal{P}(\mathbf{\Phi})$ | The powerset of $\mathbf{\Phi}$. $\Phi$ is used to refer to a given subset of $\mathbf{\Phi}$. |
| $\Psi = \mathbf{\Phi} \rightarrow \mathbb{Z}$ | The set of parameterized values. A parameterized $\psi$ value produces an integer for each control. |
| $\Sigma = L \rightarrow \Psi$ | The set of all states. A state $\sigma$ maps each variable to a parameterized value. |

Figure 3.1: Notation for the synthesis semantics of SKETCH

## 3.1 Preliminaries

The synthesis semantics tracks the relationships between the values of the holes and the values of variables in the program for an arbitrary candidate. To better illustrate how this is done, I will rely on two very simple examples. The first one is the "Hello World" program introduced in Section 2.1; it is shown below with the hole numbered to make it easier to identify later.

```
void main(int x){
        int y = x * ??0;
        assert y == x + x;
}
```

The second one is a small variation of the "Hello World" program which uses generators to help illustrate how the semantic rules behave in their presence. The call sites for the generators have been labeled with an identifier to help us to refer to them in the text.

```
generator int linexp(int t){
        return t*??0 + ??1;
}
```

```
generator int linexp2(int t1, int t2){
        return linexp_g1(t1) + linexp_g2 (t2);
}

void main(int x, int z){
        int y = linexp2_g0(x, z);
        assert y == x + x + z;
}
```

The first important concept underlying the synthesis semantics is the concept of a *control function* $\phi : H \times \mathcal{T} \rightarrow \mathbb{Z}$ introduced earlier in Section 2.1.1. The control function describes the value of each hole in the program, and therefore fully characterizes a candidate solution to the sketch. The control also takes as a parameter a calling context $\tau$ because the value of holes in generators depends on the calling context. In the simple `HelloWorld` example, there is only one hole, and because there are no generators, the only calling context is the empty calling context $\tau_\emptyset$, so the domain of any $\phi$ for this sketch is just the singleton set $\{(\texttt{??}_0, \tau_\emptyset)\}$. For the second example, there are two holes, and two different valid calling contexts for these holes: $\tau_1 = g_0 \cdot g_1$ and $\tau_2 = g_0 \cdot g_2$. Therefore, any $\phi$ for this sketch must define a value for each of the following four pairs of holes and calling contexts: $(\texttt{??}_0, \tau_1)$, $(\texttt{??}_0, \tau_2)$, $(\texttt{??}_1, \tau_1)$, $(\texttt{??}_1, \tau_2)$.

The second important concept is the *parameterized value*. In order to track the semantics of all candidate solutions simultaneously, the synthesis semantics must track the value of each variable as a function of the control $\phi$. We refer to such a function as a parameterized value because it is parameterized by $\phi$. The greek letter $\Psi$ designates the set of all parameterized values. For example, for the first sketch, `y` has the parameterized value $\lambda\phi.x * \phi(\texttt{??}_0, \tau_\emptyset)$, where $x$ is the value of the input `x`. In the case of the second sketch, the one with generators, `y` has the parameterized value $\lambda\phi.x * \phi(\texttt{??}_0, \tau_1) + \phi(\texttt{??}_1, \tau_1) + z * \phi(\texttt{??}_0, \tau_2) + \phi(\texttt{??}_1, \tau_2)$.

The extensive use of parameterized values wherever standard denotational semantics would have used integers is what allows the synthesis semantics to track the relationship between the values of holes and the values of variables in the program. The notation is summarized in Figure 3.1.

$$
\begin{array}{llll}
\text{expressions} & e & ::= & n \mid \mathrm{x} \mid \mathrm{x}[e] \mid e \circ e \mid \texttt{??} \\
\text{statements} & c & ::= & \mathrm{x} := e \mid \mathrm{x}[e] := e \mid \textbf{skip} \mid f(e) \mid c\,;c \mid \\
 & & & \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c \mid \textbf{assert}\ e \mid \\
 & & & \textbf{while}\ e\ \textbf{do}\ c \\
\text{functions} & f & ::= & \textbf{def}\ \mathrm{f}(\mathrm{x})\ c \\
\text{generators} & g & ::= & \textbf{defgen}\ \mathrm{g}(\mathrm{x})\ c \\
\text{programs} & p & ::= & p\ f \mid p\ g \mid \epsilon
\end{array}
$$

Figure 3.2: Abstract syntax for a simplified subset of the SKETCH language

## 3.2 The Semantics

The synthesis semantics are described formally through a very simple model language described by the abstract syntax in Figure 3.2. The language has been simplified in a few cosmetic ways to make the presentation simpler. For example, the operator $\circ$ is used to denote an arbitrary binary operator. Expressions are assumed to have no side effects; expressions that might lead to an error, such as out of bounds array accesses or division by zero, are assumed to be preceded by an appropriate assertion so the expressions themselves can be modeled as being side effect free. Procedure calls are assumed to be statements rather than expressions; they return values by writing to a special variable `@`, and they have no other side effects besides writing to this variable and possibly causing assertion failures.

Parameterized values allow us to define the synthesis semantics following many of the formalisms of standard denotational semantics. As in denotational semantics, the meaning of an expression is defined recursively through a *denotation function*.

$$\mathcal{A}[\![\circ]\!]^{\tau} : Aexp \rightarrow (\Sigma \rightarrow \Psi)$$

The denotation function defines the meaning of any expression as a function from a state to a parameterized value. The state $\sigma : L \rightarrow \Psi$ of the program is a mapping from the set of variable names $L$ to parameterized values. The $\tau$ in the denotation function indicates the calling context under which the interpretation is taking place.

The denotation function is defined recursively for various types of expressions, quite similar to the way these functions are defined in denotational semantics. The only new rule is the rule for evaluating a hole, which produces a function that takes in a control $\phi$, and

produces the value of the hole on that control under the current calling context $\tau$.

$$\begin{aligned}
\mathcal{A}[\![x]\!]^\tau\sigma &= \sigma(x) \\
\mathcal{A}[\![??_i]\!]^\tau\sigma &= \lambda\phi.\phi(??_i, \tau) \\
\mathcal{A}[\![e_1 \circ e_2]\!]^\tau\sigma &= \lambda\phi.\ \mathcal{A}[\![e_1]\!]^\tau\sigma\phi \circ \mathcal{A}[\![e_2]\!]^\tau\sigma\phi
\end{aligned}$$

For example, for the generator `linexp` in the second example, the denotation function for the return expression is defined as follows:

$$\begin{aligned}
\mathcal{A}[\![\mathtt{t} * \mathtt{??}_0 + \mathtt{??}_1]\!]^\tau\sigma &= \lambda\phi.\ \mathcal{A}[\![\mathtt{t} * \mathtt{??}_0]\!]^\tau\sigma\phi + \mathcal{A}[\![\mathtt{??}_1]\!]^\tau\sigma\phi \\
&= \lambda\phi.\ \mathcal{A}[\![\mathtt{t}]\!]^\tau\sigma\phi * \mathcal{A}[\![\mathtt{??}_0]\!]^\tau\sigma\phi + \phi(\mathtt{??}_1, \tau) \\
&= \lambda\phi.\sigma(\mathtt{t}) * \phi(\mathtt{??}_0, \tau) + \phi(\mathtt{??}_1, \tau)
\end{aligned}$$

Unlike expressions, commands have side effects. To model these, the denotation function defines the meaning of a command as a transformation on a state and a set of candidate controls. From the initial state and control set, the command produces an updated state and a subset of the original control set containing only those controls which are valid for that command, *i.e.* those that do not cause assertion failures. Expressions do not need to track these sets because, as was said earlier, we have assumed that evaluation of expressions will never lead to errors.

$$\mathcal{C}[\![\circ]\!]^\tau : \mathit{Command} \rightarrow (\langle \Sigma, \mathcal{P}(\mathbf{\Phi})\rangle \rightarrow \langle \Sigma, \mathcal{P}(\mathbf{\Phi})\rangle)$$

The two most basic rules are those for assertions and assignments.

$$\begin{aligned}
\mathcal{C}[\![x := e]\!]^\tau\langle\sigma, \Phi\rangle &= \langle\sigma[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma], \Phi\rangle \\
\mathcal{C}[\![\mathbf{assert}\ e]\!]^\tau\langle\sigma, \Phi\rangle &= \langle\sigma, \{\phi \in \Phi : \mathcal{A}[\![e]\!]^\tau\sigma\phi = 1\}\rangle
\end{aligned}$$

Assignments modify only the state while leaving the set of candidate controls unmodified. Assertions, on the other hand, narrow the set of valid controls, to include only those that will cause the assertion to succeed.

Sequencing of commands is easy to define; it is just a composition of two functions.

$$\mathcal{C}[\![c_1; c_2]\!]^\tau\langle\sigma, \Phi\rangle = \mathcal{C}[\![c_2]\!]^\tau(\ \mathcal{C}[\![c_1]\!]^\tau\langle\sigma, \Phi\rangle)$$

**Example.** To illustrate how these rules operate, consider the denotation function for the body of the `HelloWorld` example. For this example, the initial state will just map the input variable `x` to a symbolic input value $x$.

$$\mathcal{C}[\![\texttt{y = x * ??}_0\texttt{; assert y == x + x;}]\!]^\tau\langle[\texttt{x} \mapsto x], \mathbf{\Phi}\rangle$$
$$= \mathcal{C}[\![\texttt{assert y == x + x}]\!]^\tau\ \mathcal{C}[\![\texttt{y = x * ??}_0]\!]^\tau\langle[\texttt{x} \mapsto x], \mathbf{\Phi}\rangle$$

In that equation, $\mathcal{C}[\![\texttt{y = x * ??}_0]\!]^\tau\langle[\texttt{x} \mapsto x], \mathbf{\Phi}\rangle$ evaluates to the following pair.

$$\begin{aligned}\mathcal{C}[\![\texttt{y = x * ??}_0]\!]^\tau\langle[\texttt{x} \mapsto x], \mathbf{\Phi}\rangle &= \langle[\texttt{x} \mapsto x, \texttt{y} \mapsto (\ \mathcal{A}[\![\texttt{x * ??}_0]\!]^\tau[\texttt{x} \mapsto x])], \mathbf{\Phi}\rangle \\ &= \langle[\texttt{x} \mapsto x, \texttt{y} \mapsto \lambda\phi.x * \phi(\texttt{??}_0, \tau_\emptyset)], \mathbf{\Phi}\rangle\end{aligned}$$

Therefore,

$$\mathcal{C}[\![\texttt{assert y == x + x}]\!]^\tau\ \mathcal{C}[\![\texttt{y = x * ??}_0]\!]^\tau\langle[\texttt{x} \mapsto x], \mathbf{\Phi}\rangle$$
$$= \mathcal{C}[\![\texttt{assert y == x + x}]\!]^\tau\langle[\texttt{x} \mapsto x, \texttt{y} \mapsto \lambda\phi.x * \phi(\texttt{??}_0, \tau_\emptyset)], \mathbf{\Phi}\rangle$$
$$= \langle[\texttt{x} \mapsto x, \texttt{y} \mapsto \lambda\phi.x * \phi(\texttt{??}_0, \tau_\emptyset)], \{\phi \in \mathbf{\Phi} : x * \phi(\texttt{??}_0, \tau_\emptyset) == x + x\}\rangle$$

The resulting pair tells us what we needed to know about the semantics of the `HelloWorld` program. On the one hand, it shows the exact relationship between the state and the choice of value for the hole. On the other hand, it constrains the set of valid control functions to those satisfying the relationship $x * \phi(\texttt{??}_0, \tau_\emptyset) == x + x$.

The **if** rule, is a little more involved due to the handling of $\Phi$. In an **if** statement, each branch is evaluated under the subset of $\Phi$ that would cause the program to take that branch, and the resulting sets of controls are combined through set union. In the rules below we use the notational shortcut $a?b:c$ to represent the function that returns $b$ if $a$ is true and $c$ otherwise.

$$\mathcal{C}[\![\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2]\!]^\tau\langle\sigma, \Phi\rangle = \langle\sigma', \Phi'\rangle$$

where $\sigma'$ and $\Phi'$ are defined through the following equations:

$$\Phi_t = \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau\sigma\phi = true\}$$
$$\Phi_f = \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau\sigma\phi = false\}$$
$$\langle\sigma_1, \Phi_1\rangle = \mathcal{C}[\![c_1]\!]^\tau\langle\sigma, \Phi_t\rangle$$
$$\langle\sigma_2, \Phi_2\rangle = \mathcal{C}[\![c_2]\!]^\tau\langle\sigma, \Phi_f\rangle$$
$$\Phi' = (\Phi_1) \cup (\Phi_2)$$
$$\sigma' = \lambda x.\lambda\phi.\ \mathcal{A}[\![e]\!]^\tau\sigma\phi\ ?\ \sigma_1 x\phi\ :\ \sigma_2 x\phi$$

**while** loops are handled in a similar way as they are handled in regular denotational semantics, by defining their denotation function recursively.

$$\begin{aligned} W(\langle\sigma\,,\Phi\rangle) = \;& \mathcal{C}[\![\textbf{while}\ e\ \textbf{do}\ c]\!]^\tau\langle\sigma\,,\Phi\rangle = \langle\sigma'\,,\Phi'\rangle \\ & \Phi_t = \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau\sigma\phi = 1\} \\ & \Phi_f = \{\phi \in \Phi :\ \mathcal{A}[\![e]\!]^\tau\sigma\phi = 0\} \\ & \langle\sigma_1\,,\Phi_1\rangle = W(\ \mathcal{C}[\![c]\!]^\tau\langle\sigma\,,\Phi_t\rangle) \\ & \Phi' = (\Phi_1) \cup (\Phi_f) \\ & \sigma' = \lambda x.\lambda\phi.\ \mathcal{A}[\![e]\!]^\tau\sigma\phi\ \texttt{?}\ \sigma_1 x\phi\ \texttt{:}\ \sigma x\phi \end{aligned}$$

For some loops, it is possible to solve the equation above to derive a closed form expression for $W$. For example, consider the loop below.

```
wihile i < N do
        assert ??0 > i
        i = i+1
```

For this loop, the closed form solution for $W$ is

$$W(\sigma,\Phi) = \begin{cases} \sigma(i) < N & \langle\sigma[i \mapsto N]\,,\Phi \cap \{\phi : \phi(\texttt{??}_0) > N-1)\}\rangle \\ else & \langle\sigma\,,\Phi\rangle \end{cases} \tag{3.2.1}$$

One can check that this function satisfies the recursive constraints for $W$. First, let $i_0 = \sigma(i)$ and $c_0 = \phi(\texttt{??}_0)$. Then, $\Phi_t$ and $\Phi_f$ are defined as follows.

$$\Phi_t = \begin{cases} i_0 < N & \Phi \\ else & \emptyset \end{cases} \quad and \quad \Phi_f = \begin{cases} i_0 < N & \emptyset \\ else & \Phi \end{cases}$$

Then, evaluating the body of the loop under $\langle\sigma\,,\Phi_t\rangle$ becomes

$$\mathcal{C}[\![c]\!]^\tau\langle\sigma\,,\Phi_t\rangle = \begin{cases} i_0 < N & \langle\sigma[i \mapsto i_0+1]\,,\Phi \cap \{\phi : \phi(\texttt{??}_0) > i_0\}\rangle \\ else & \langle\sigma[i \mapsto i_0+1]\,,\emptyset\rangle \end{cases} \tag{3.2.2}$$

Thus, if we let $\langle\sigma_1\,,\Phi_1\rangle = W(\ \mathcal{C}[\![c]\!]^\tau\langle\sigma\,,\Phi_t\rangle)$, then we have the following definitions

for $\sigma_1$ and $\Phi_1$ by equation Equation (3.2.1).

$$\sigma_1 = \begin{cases} i_0 + 1 < N & \sigma[i \mapsto N] \\ else & \sigma[i \mapsto i_0 + 1] \end{cases}$$

$$\Phi_1 = \begin{cases} i_0 + 1 < N & (\Phi \cap \{\phi : \phi(\texttt{??}_0) > i_0\}) \cap \{\phi : \phi(\texttt{??}_0) > N - 1\} \\ i_0 + 1 \geq N \wedge i_0 < N & (\Phi \cap \{\phi : \phi(\texttt{??}_0) > i_0\}) \\ else & \emptyset \end{cases}$$

From this, it is easy to see that both $\Phi' = \Phi_1 \cup \Phi_f$ satisfies the definition of $W$.

$$\Phi_1 \cup \Phi_f = \begin{cases} i_0 + 1 < N \wedge i_0 < N & (\Phi \cap \{\phi : \phi(\texttt{??}_0) > i_0\}) \cap \{\phi : \phi(\texttt{??}_0) > N - 1\} \\ & = \Phi \cap \{\phi : \phi(\texttt{??}_0) > N - 1\} \\ i_0 + 1 \geq N \wedge i_0 < N & \Phi \cap \{\phi : \phi(\texttt{??}_0) > i_0\} = \Phi \cap \{\phi : \phi(\texttt{??}_0) > N - 1\} \\ else & \emptyset \cup \Phi = \Phi \end{cases}$$

Similarly, $\sigma' = \lambda x.\lambda\phi.\ \mathcal{A}[\![e]\!]^\tau \sigma\phi\ ?\ \sigma_1 x\phi\ :\ \sigma x\phi$, also satisfies the definition for $W$, since $\sigma'(i)$ now has the following value.

$$\sigma'(i) = \begin{cases} i_0 + 1 < N \wedge i_0 < N & N \\ i_0 + 1 \geq N \wedge i_0 < N & i_0 + 1 = N \\ else & i_0 \end{cases}$$

Unfortunately, the problem of finding a closed form for the $W$ function of a loop is undecidable. In our synthesizer, we will get around this problem by bounding the number of iterations of loops. For states $\sigma$ that cause the loop to iterate more than the allowed number of times, we define $W(\sigma, \Phi) = (\sigma', \emptyset)$. In practice this will mean that our synthesizer may fail to find a solution to a sketch when one actually exists, or more commonly, that the user will have to make sure that the bounds in the number of iterations are enough to handle all the inputs that the synthesizer may consider.

There are still some semantics left to describe, namely the semantics of procedures and generators, and the current limited support for language features such as arrays and heap allocated objects. However, I have covered the major ideas behind the synthesis semantics, which will allow us to define the set of valid solutions to a sketch in Section 3.3, and subsequently to reason formally about our novel counterexample guide inductive synthesis algorithm (Chapter 4).

### 3.2.1 Procedures and Generators

Procedure calls behave as we would expect from standard denotational semantics. For a function defined as **def** f(x) $c$, the semantics of a call to $\texttt{f}(e)$ are defined by evaluating the body of the function under the empty calling context $\tau_\emptyset$ , and the initial state $\sigma_\perp[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma]$, where $\sigma_\perp$ is the empty state.

$$\langle \sigma', \Phi' \rangle = \mathcal{C}[\![c]\!]^{\tau_\emptyset}\langle \sigma_\perp[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma], \Phi \rangle \tag{3.2.3}$$

$$\mathcal{C}[\![f(e)]\!]^\tau\langle \sigma, \Phi \rangle = \langle \sigma[\texttt{@} \mapsto \sigma'(\texttt{@})], \Phi' \rangle \tag{3.2.4}$$

In the definition, the return value of `f` is stored in the special variable `@` as explained before.

The evaluation of generators is only slightly different. Instead of evaluating the body under the empty calling context, the body is evaluated under the calling context $\tau \cdot g_i$, where $g_i$ identifies the current call site for the generator. Therefore, the semantics for a call to a generator defined as **defgen** g(x) $c$ from a call site $g_i$ are defined by the formulas below.

$$\langle \sigma', \Phi' \rangle = \mathcal{C}[\![c]\!]^{\tau \cdot g_i}\langle \sigma_\perp[x \mapsto \mathcal{A}[\![e]\!]^\tau\sigma], \Phi \rangle \tag{3.2.5}$$

$$\mathcal{C}[\![g(e)]\!]^\tau\langle \sigma, \Phi \rangle = \langle \sigma[\texttt{@} \mapsto \sigma'(\texttt{@})], \Phi' \rangle \tag{3.2.6}$$

An interesting observation is that procedure calls have the effect of forgetting the calling context, so generators called from a procedure will behave the same regardless of the calling context of the procedure.

#### Bounded Semantics for Generators

As we saw in Section 2.2, a generator represents a set of functions, and the synthesizer is free to select any of these functions to replace a call to the generator. However, there is a problem with the way we defined the synthesis semantics for generators: they make generators *too* powerful. So powerful, in fact, that they can represents sets which include functions that are not even computable, a clear problem if we expect to synthesize code from them.

The problem is that the semantics defined so far allow programmers to write sketches which can only be resolved with an infinite $\phi$. A trivial example of this would be the universal generator:

```
generator int univ(int x){
        if( abs(x) > 0 ){ return univ(abs(x)-1); }
        else { return ??; }
}
```

According to the synthesis semantics, the generator above can be made to represent any function in the set $\mathbb{N} \rightarrow \mathbb{Z}$, even though we know some functions in this set are not computable.

This problem is closely related to the problem of non-termination in the partial evaluation procedure from Section 2.2.1. Any control $\phi$ that causes a generator to resolve to a non-computable function will lead to non-termination in the partial evaluation of the generator (the converse is not true).

To address this problem, we provide a slight modification to the semantics which we call *bounded generator semantics*. Bounded generator semantics bounds the recursion of generators by specifying a bounded set of call stacks $\bar{\boldsymbol{\tau}}$. Thus, for a generator defined as **defgen** g(x) $c$, the semantics of a call to g at call site $g_i$ now involve a check of whether $\tau \in \bar{\boldsymbol{\tau}}$.

$$\langle \sigma', \Phi' \rangle = \mathcal{C}[\![c]\!]^{\tau \cdot g_i} \langle \sigma_\perp[x \mapsto \mathcal{A}[\![e]\!]^\tau \sigma], \Phi \rangle \tag{3.2.7}$$

$$\mathcal{C}[\![g(e)]\!]^\tau \langle \sigma, \Phi \rangle = \begin{cases} \langle \sigma[@ \mapsto \sigma'(@)], \Phi' \rangle & \text{if } \tau \in \bar{\boldsymbol{\tau}} \\ \langle \sigma, \emptyset \rangle \end{cases} \tag{3.2.8}$$

One can see from the formulas that trying to evaluate a generator when the current stack doesn't belong to $\bar{\boldsymbol{\tau}}$ has the same effect as an assertion failure.

In order to keep the synthesis semantics consistent with the partial evaluation procedure, we need to make a corresponding change to the rules for partial evaluation of generators.

$$\frac{(\textbf{defgen}\ g(in)\ c)\ \ \langle e, \hat{\sigma} \rangle \xrightarrow{\phi_\tau} \langle e', v' \rangle \quad \tau \in \bar{\boldsymbol{\tau}} \quad \langle c, \hat{\sigma}_\perp[in \mapsto v'] \rangle \xrightarrow{\phi_{\tau \cdot g_i}} \langle c', \hat{\sigma}' \rangle}{\langle g(e), \hat{\sigma} \rangle \xrightarrow{\phi_\tau} \langle rename(in := e'; c'), \hat{\sigma}[@ \mapsto \hat{\sigma}'(@)] \rangle} \quad \text{(S-CallGen)}$$

$$\frac{\tau \notin \bar{\boldsymbol{\tau}}}{\langle g(e), \hat{\sigma} \rangle \xrightarrow{\phi_\tau} \langle \textbf{assert false}, \hat{\sigma}[@ \mapsto \perp] \rangle} \quad \text{(S-CallGen2)}$$

With this change it becomes possible to prove a correspondence between the partial evaluation procedure and the synthesis semantics of SKETCH. Bounding the recursion of

generators also brings us one step closer towards making the synthesis semantics decidable.

### 3.2.2 Additional Constructs

The sketch language also supports bounded arrays. It wouldn't be too hard to define the semantics for unbounded arrays, but bounded arrays are easier to describe, and they are the only ones our synthesizer supports so far. Bounded arrays are easier to describe because we simply represent them as a set of individual scalar variables, one for each element. Thus, for example, an array $x$ of size $n$ will be treated as $n$ independent variables $x_1$ to $x_n$. This makes the semantics of array reading and writing straightforward. Reading element $e$ from an array is merely a switch statement.

$$\mathcal{A}[\![x[e]]\!]^\tau\sigma = \lambda\phi.switch(\ \mathcal{A}[\![e]\!]^\tau\sigma\phi)(case\ 1 : \sigma(x_1)\phi; \ldots case\ n : \sigma(x_n)\phi) \tag{3.2.9}$$

Similarly, an assignment to element $e$ updates every single element in the array with a conditional expression.

$$\mathcal{C}[\![x[e] := u]\!]^\tau\langle\sigma\,,\,\Phi\rangle = \langle\sigma[x_i \mapsto \lambda\phi.\ \mathcal{A}[\![e]\!]^\tau\sigma\phi = i?\ \mathcal{A}[\![u]\!]^\tau\sigma\phi{:}\sigma(x_i)\phi]\,,\,\Phi\rangle \tag{3.2.10}$$

The SKETCH language also offers support for pointers and heap allocated objects, but instead of defining them as part of the core language, we implement them as syntactic sugar by modeling the heap with a set of arrays, one for each field for each type of object. A pointer is an index that is used to access these arrays, making field dereference a common array access. The language also supports reference parameters to functions; these are currently implemented with copy-in copy-out semantics; in other words, the values of the actual parameters are copied into the formal parameters before the call. After the call, the last value of the formal parameters is copied back to the actual parameters.

## 3.3 The Sketch Resolution equation

In SKETCH, it is required that all sketches have a `main` procedure which constitutes the entry point to the sketch. The semantics of a program $P$ are thus defined in terms of the effect of calling its `main` procedure.

$$\mathcal{C}[\![P]\!]^\tau\langle\sigma\,,\,\Phi\rangle = \ \mathcal{C}[\![\texttt{main}(in)]\!]^{\tau_\emptyset}\langle\sigma\,,\,\Phi\rangle \tag{3.3.1}$$

From this definition, we can define a set of valid controls $\Phi$ to be one which satisfies the sketch resolution equation.

**Equation 1 (Sketch Resolution)** *The set of controls $\Phi$ is said to be valid if it is invariant under $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$ for any initial state, as expressed in the equation below. The projection function $\pi_\Phi$ extracts the set $\Phi$ from a pair $\langle \sigma, \Phi \rangle$.*

$$\forall\, \sigma\ \pi_\Phi(\ \mathcal{C}[\![P]\!]^{\tau_\emptyset}\langle \sigma, \Phi\rangle) = \Phi \tag{3.3.2}$$

*I will use $\Phi^*$ to denote the maximal set of legal controls, or maximal solution to the sketch. This maximal set can be defined constructively as the greatest fixed point of a function that processes an input set of controls with a non-deterministically selected initial state.*

$$F(\Phi_{in}) := \textbf{let}\ \sigma \in \Sigma\ \textbf{then}\ (\ \mathcal{C}[\![P]\!]^{\tau_\emptyset}\langle \sigma, \Phi_{in}\rangle = \langle \sigma', \Phi_{out}\rangle; \textbf{return}\ \Phi_{out})$$

Chapter 4 will explain how to use the synthesis semantics to symbolically approximate $\Phi^*$.

## 3.4 Important properties of the semantics

This section presents a handful of theorems and lemmas that follow from the definitions of the synthesis semantics. Section 3.4.1 shows that the result of partially evaluating a program with a control $\phi$ is consistent with the behavior of the program as described by the synthesis semantics. Section 3.4.2 presents a handful of algebraic properties of the synthesis semantics rules which will enable some optimizations in our sketch synthesis algorithm.

### 3.4.1 Soundness of the Partial Evaluation Rules

An important feature of the synthesis semantics is that it should be consistent with the programmer's view of sketching defined in Chapter 2 through partial evaluation. Consistency means that partially evaluating a sketch $P$ under any control $\phi \in \Phi^*$ in the maximal solution to the sketch resolution equation, should produce a concrete program that will satisfy all its assertions under all possible inputs. Additionally, the behavior of this concrete program should be consistent with the behavior that the synthesis semantics defines for the sketch under control $\phi$, so if a variable $\mathbf{x}$ has value $v$ in the concrete program,

$\sigma(x)\phi$ in the synthesis semantics should also equal $v$. These two requirements are stated formally in the following theorem.

**Theorem 3.4.1 (Soundness of Programmer's View)** *Let $\Phi^*$ be a maximal solution to sketch $P$ as defined in Section 3.3, and let $\phi \in \Phi^*$ and $P_\phi = PE(P, \phi)$. Then, the following three equations must hold for all input states $\sigma$ whenever the synthesis semantics converge (i.e. the programs terminate).*

$$\mathcal{C}[\![P]\!]^{\tau_\emptyset}\langle \sigma\,, \Phi^*\rangle = \langle \sigma_s\,, \Phi^*\rangle \tag{3.4.1}$$

$$\mathcal{C}[\![P_\phi]\!]^{\tau_\emptyset}\langle \sigma\,, \mathbf{\Phi}\rangle = \langle \sigma_\phi\,, \mathbf{\Phi}\rangle \tag{3.4.2}$$

$$\forall x \in L.\ \sigma_s(x)\phi = \sigma_\phi(x)\phi \tag{3.4.3}$$

*Note the equations above express two separate properties: 1) The equivalence of the final states. 2) The fact that $P_\phi$ will not fail any assertion.*

**Proof Outline** I will only show an outline of the proof, as the actual proof is rather mechanical. The proof strategy for both claims in the theorem is to do induction on the structure of the derivation; similar in spirit to the way Gomard proved the correctness of partial evaluation for lambda calculus [34].

**Equivalence of states** The proof for the equivalence of states involves three derivations at once: a derivation to evaluate the semantics on $P$, a derivation to evaluate the semantics on $P_\phi$, and the derivation that computes $P_\phi$ from $P$. The idea is to show the equivalence of statements in $P$ and $P_\phi$ by assuming the equivalence of their subcomponents, and showing how the equivalence is preserved by the partial evaluation rules. This inductive argument will be well founded as long as the derivation doesn't recur forever.

I begin by defining an inductive hypothesis. Let $s$ be an arbitrary statement in $P$, and let $s_\phi$ be the corresponding statement in $P_\phi$, where the two are related by the partial evaluation rule.

$$\langle s, \hat{\sigma}^i\rangle \xrightarrow{\phi} \langle s_\phi, \hat{\sigma}^{i+1}\rangle \tag{3.4.4}$$

Now, consider an evaluation of $s$ within the evaluation of $P$, and the corresponding evaluation of $s_\phi$ within the evaluation of $P_\phi$.

$$\mathcal{C}[\![s]\!]^{\tau}\langle \sigma_s^i, \Phi_s^i\rangle = \langle \sigma_s^{i+1}, \Phi_s^{i+1}\rangle \tag{3.4.5}$$

$$\mathcal{C}[\![s_\phi]\!]^{\tau}\langle \sigma_\phi^i, \Phi_\phi^i\rangle = \langle \sigma_\phi^{i+1}, \Phi_\phi^{i+1}\rangle \tag{3.4.6}$$

These two evaluations will preserve the invariant that the value of any variable $x$ in the state of the original program evaluated under control $\phi$ must equal the value of $x$ in the state of the partially evaluated program. An additional invariant is that if any variable $x$ has a constant value in $\hat{\sigma}^i$, then its parameterized value in $\sigma_s^i$ will be equal to that constant when evaluated with control $\phi$.

$$\forall x \in L.\ \hat{\sigma}^i(x) \neq \perp \quad \Rightarrow \quad \hat{\sigma}^i(x) = \sigma_s^i(x)\phi \tag{3.4.7}$$

$$\forall x \in L.\ \sigma_s^i(x)\phi \quad = \quad \sigma_\phi^i(x)\phi \tag{3.4.8}$$

The invariant establishes a relationship between the three states: the state $\hat{\sigma}^i$ used to partially evaluate $s$, and the states $\sigma_s^i$ and $\sigma_\phi^i$ used to evaluate the semantics for both the original statement and the partially evaluated one. Note that $\sigma_\phi^i(x)$ will actually be a constant independent of $\phi$, since $s_\phi$ has already been partially evaluated. Also note that this invariant is vacuously satisfied by the initial states of the partial evaluation and the two programs in question.

From the inductive hypothesis, it is easy to prove that the partial evaluation of expressions preserves their semantics under control $\phi$.

**Lemma 1** *Applying partial evaluation on an expression $e$ preserves its semantics under control $\phi$. Let $e$ be an expression in $P$, and let $e_\phi$ be the corresponding expression in $P_\phi$.*

$$\langle e, \hat{\sigma}^i \rangle \xrightarrow{\phi} \langle e_\phi, v \rangle$$

*Then their semantics are equivalent under $\phi$, and if $v$ is a constant, then their parametrized value evaluates to $v$ under control $\phi$.*

$$\mathcal{A}[\![e]\!]^\tau \sigma_s^i \phi = \mathcal{A}[\![e_\phi]\!]^\tau \sigma_\phi^i \phi \quad \textit{and} \quad v \neq \perp \ \Rightarrow \ v = \mathcal{A}[\![e]\!]^\tau \sigma_s^i \phi$$

The lemma can be proved by induction on a case by case basis on the type of $e$. For example, if $e$ is a hole $\texttt{??}_i$, then $e_\phi = \phi_\tau(\texttt{??}_i) = n$ according to the E-Hole rule.

$$\frac{\phi_\tau(\texttt{??}_i) \Rightarrow n}{\langle \texttt{??}_i, \hat{\sigma} \rangle \xrightarrow{\phi_\tau} \langle \mathrm{n}, n \rangle}$$

Then, $\mathcal{A}[\![e]\!]^\tau \sigma_1 \phi = \mathcal{A}[\![e_\phi]\!]^\tau \sigma_2 \phi = n$ according to the synthesis semantics, satisfying the lemma. The same logic can be used for variables and arithmetic expressions. □

Using the lemma, we can show that the inductive hypothesis is preserved by Equations 3.4.4, 3.4.5 and 3.4.6. This must be proved on a case by case basis for each of the

different types of statements. For example, in the case of assignment, partial evaluation updates the state and replaces the right-hand side with a partially evaluated expression.

$$\frac{\langle e, \hat{\sigma}^i\rangle \xrightarrow{\phi} \langle e_\phi, v\rangle}{\langle x := e, \hat{\sigma}^i\rangle \xrightarrow{\phi} \langle x := e_\phi, \hat{\sigma}^i[x \mapsto v] = \hat{\sigma}^{i+1}\rangle}$$

From the lemma on evaluation of expressions, we have that

$$\mathcal{A}[\![e]\!]^\tau \sigma_s^i \phi = \mathcal{A}[\![e_\phi]\!]^\tau \sigma_\phi^i \phi$$

Which means that $\sigma_s^{i+1}(x)\phi = \sigma_\phi^{i+1}(x)\phi$ by the assignment rule. Additionally, if $\hat{\sigma}^{i+1}(x) \neq \perp$ then it means that $v \neq \perp$, which means by our lemma on expression evaluation that $\sigma_s^{i+1}(x)\phi = v = \hat{\sigma}^{i+1}(x)$, as Equation (3.4.7) requires.

The argument becomes somewhat more complicated in the case of loops, because of the recursive definition of the denotation function. Consider a loop of the form **while** $e$ **do** $c$ , which is partially evaluated to **while** $e_\phi$ **do** $c_\phi$ according to the partial evaluation rule from Section 2.1.1.

$$\frac{\langle e, \hat{\sigma}_\perp\rangle \xrightarrow{\phi} \langle e_\phi, v\rangle \;\; \langle c, \hat{\sigma}_\perp\rangle \xrightarrow{\phi} (c_\phi, \hat{\sigma}')}{\langle \textbf{while}\ e\ \textbf{do}\ c, \hat{\sigma}^i\rangle \xrightarrow{\phi} \langle \textbf{while}\ e_\phi\ \textbf{do}\ c_\phi, \hat{\sigma}^{i+1} = \hat{\sigma}^i \cap \hat{\sigma}'\rangle}$$

Now, let $W_s(\langle\sigma, \Phi\rangle) = \mathcal{C}[\![\textbf{while}\ e\ \textbf{do}\ c]\!]^\tau$, and $W_\phi(\langle\sigma, \Phi\rangle) = \mathcal{C}[\![\textbf{while}\ e_\phi\ \textbf{do}\ c_\phi]\!]^\tau$; then, the equations below define the semantics for both the original and the partially evaluated loops.

$$\Phi_{ts} = \{\phi \in \Phi_s : \mathcal{A}[\![e]\!]^\tau \sigma_s^i \phi = 1\} \tag{3.4.9}$$

$$\Phi_{t\phi} = \{\phi \in \Phi_\phi : \mathcal{A}[\![e_\phi]\!]^\tau \sigma_\phi^i \phi = 1\} \tag{3.4.10}$$

$$\langle \sigma_s^{i+1}, \Phi_s^1\rangle = \mathcal{C}[\![c]\!]^\tau \langle \sigma_s^i, \Phi_{ts}\rangle \tag{3.4.11}$$

$$\langle \sigma_\phi^{i+1}, \Phi_\phi^1\rangle = \mathcal{C}[\![c_\phi]\!]^\tau \langle \sigma_\phi^i, \Phi_{t\phi}\rangle \tag{3.4.12}$$

$$\langle \sigma_s^{i+2}, \Phi_s^2\rangle = W_s(\sigma_s^{i+1}, \Phi_s') \tag{3.4.13}$$

$$\langle \sigma_\phi^{i+2}, \Phi_\phi^2\rangle = W_\phi(\sigma_\phi^{i+1}, \Phi_\phi') \tag{3.4.14}$$

$$\sigma_s^{i+3} = \pi_\sigma(W_s(\langle\sigma_s^i, \Phi\rangle)) = \lambda x.\lambda\phi.\ \mathcal{A}[\![e]\!]^\tau \sigma_s^i \phi\ ?\ \sigma_s^{i+2} x\phi\ :\ \sigma_s^i x\phi \tag{3.4.15}$$

$$\sigma_\phi^{i+3} = \pi_\sigma(W_\phi(\langle\sigma_\phi^i, \Phi\rangle)) = \lambda x.\lambda\phi.\ \mathcal{A}[\![e_\phi]\!]^\tau \sigma_\phi^i \phi\ ?\ \sigma_\phi^{i+2} x\phi\ :\ \sigma_\phi^i x\phi \tag{3.4.16}$$

Note that the states $\hat{\sigma}_\perp$, $\sigma_s^i$ and $\sigma_\phi^i$ satisfy the invariants from Equations 3.4.7 and 3.4.8, so by the induction hypothesis, we have that $\hat{\sigma}'$, $\sigma_s^{i+1}$ and $\sigma_\phi^{i+1}$ also satisfy the invariants from Equations 3.4.7 and 3.4.8. Now, we want to claim that $\hat{\sigma}^{i+1}$, $\sigma_s^{i+2}$ and $\sigma_\phi^{i+2}$

satisfy the invariant by an inductive argument on Equations 3.4.13 and 3.4.14. However, we can not make an inductive argument because we do not know that $\hat{\sigma}^i$, $\sigma_s^{i+1}$ and $\sigma_\phi^{i+1}$ satisfy the invariant. The way around this problem is to notice that partially evaluating the loop under state $\hat{\sigma}^i$ is completely equivalent to evaluating under state $\hat{\sigma}'$ as a consequence of the way the partial evaluation rule is defined. Therefore, the two states are equivalent for the purpose of partially evaluating the loop, even if they are actually different. So the inductive step works, and we can conclude that $\hat{\sigma}^{i+1}$, $\sigma_s^{i+2}$ and $\sigma_\phi^{i+2}$ satisfy the invariants. Additionally, by the lemma for evaluation of expressions, we have that $\mathcal{A}[\![e_\phi]\!]^\tau \sigma_\phi^i \phi = \mathcal{A}[\![e]\!]^\tau \sigma_s^i \phi$, so from this it follows that $\hat{\sigma}^{i+1}$, $\sigma_s^{i+3}$ and $\sigma_\phi^{i+3}$ satisfy the invariants, and therefore that the invariants are preserved by partial evaluation of the loop.

The only hole in this argument is that in order for the induction for the loop to be well founded, the evaluation of the loop must terminate. In the statement of the theorem, we made an explicit assumption that loops terminate. For a concrete program, this means that for any input $\sigma$, there exists a $k$ such that the loop will always iterate fewer than $k$ times. In standard denotational semantics, it also means that the denotation function will reach a fixed point after $k$ recursive evaluations. For the synthesis semantics, we stated earlier that we could put a $k$ bound on loops by establishing that for the $k^{th}$ recursive evaluation of $W$, $W(\langle \sigma, \Phi \rangle) = \langle \sigma, \emptyset \rangle$. It can be proved that any control $\phi$ that is a solution to the sketch under this bounded definition of the semantics will have the property that the loop in question will not iterate more than $k$ times for any input. It is also possible to show that the invariant is preserved by the last recursive evaluation due to the fact that the loop in the concrete program will terminate before $k$ iterations. Therefore, the induction is well founded and the proof works as promised.

This is the basic argument, and it works the same for all other constructs in the language, completing this outline of the proof of the equivalence of the states. All that is left to prove now is the second claim of the theorem, regarding the fact that $P_\phi$ will not fail any assertion.

**Absence of errors in $P_\phi$** The proof for the absence of assertion failures in $P_\phi$ is also a proof by induction on the structure of the derivation, using again the notation from Equations 3.4.4, 3.4.5 and 3.4.6. In this case, the inductive hypothesis is that if $\phi \in \Phi_s^i$,

then $\Phi^i_\phi = \mathbf{\Phi}$, and otherwise, $\Phi^i_\phi = \emptyset$.

$$\phi \in \Phi^i_s \quad \Rightarrow \quad \Phi^i_\phi = \mathbf{\Phi} \tag{3.4.17}$$

$$\phi \notin \Phi^i_s \quad \Rightarrow \quad \Phi^i_\phi = \emptyset \tag{3.4.18}$$

The intuition for this invariant is that the concrete program $P_\phi$ under input $\sigma$ may not execute all statements. Those statements that it does execute will have $\Phi^i_\phi = \mathbf{\Phi}$, and those statements that are not executed will have $\Phi^i_\phi = \emptyset$. An assertion failure means that the assertion executes, but the statement after it does not, so $\Phi^i_\phi = \mathbf{\Phi}$ but $\Phi^{i+1}_\phi = \emptyset$. In the case of the sketch $P$, those statements that would execute if $P$ were to be partially evaluated with $\phi$ are those that have $\phi \in \Phi^i_s$. What the induction proof does is formalize this intuition.

The first step is to show that the invariant holds at the beginning of the derivation, which is true because $\phi \in \Phi^* = \Phi^0_s$ and $\Phi^0_\phi = \mathbf{\Phi}$. The next step is to show that the invariant is maintained by all the different classes of statements. The proof is rather mechanical, so I will only illustrate the case of assertions.

Consider an assertion of the form **assert** $e$, and the partially evaluated version of the assertion **assert** $e_\phi$.

$$\mathcal{C}[\![\textbf{assert } e]\!]^\tau \langle \sigma^i_s, \Phi^i_s \rangle = \langle \sigma^i_s = \sigma^{i+1}_s, \Phi^{i+1}_s = \{\phi_o \in \Phi^i_s : \ \mathcal{A}[\![e]\!]^\tau \sigma^i_s \phi_o = 1\} \rangle \tag{3.4.19}$$

$$\mathcal{C}[\![\textbf{assert } e_\phi]\!]^\tau \langle \sigma^i_\phi, \Phi^i_\phi \rangle = \langle \sigma^i_\phi = \sigma^{i+1}_\phi, \Phi^{i+1}_\phi = \{\phi_o \in \Phi^i_\phi : \ \mathcal{A}[\![e_\phi]\!]^\tau \sigma^i_\phi \phi_o = 1\} \rangle \tag{3.4.20}$$

Recall that we have established that $\mathcal{A}[\![e_\phi]\!]^\tau \sigma^i_\phi \phi = \mathcal{A}[\![e]\!]^\tau \sigma^i_s \phi$. Additionally, $e_\phi$ is independent of the control, so we have a stronger equality $\forall \phi_o.\ \mathcal{A}[\![e_\phi]\!]^\tau \sigma^i_\phi \phi_o = \mathcal{A}[\![e]\!]^\tau \sigma^i_s \phi$. From these equalities, we can verify the inductive hypothesis through case analysis.

- **case 1:** $\phi \in \Phi^i_s$ and $\mathcal{A}[\![e]\!]^\tau \sigma^i_s \phi = 1$. For this case, $\Phi^i_\phi = \mathbf{\Phi}$ by the inductive hypothesis, and $\forall \phi_o.\ \mathcal{A}[\![e_\phi]\!]^\tau \sigma^i_\phi \phi_0 = \mathcal{A}[\![e]\!]^\tau \sigma^i_s \phi = 1$, so $\phi \in \Phi^{i+1}_s$ and $\Phi^{i+1}_\phi = \mathbf{\Phi}$, preserving the invariant.

- **case 2:** $\phi \in \Phi^i_s$ and $\mathcal{A}[\![e]\!]^\tau \sigma^i_s \phi = 0$. For this case, $\Phi^i_\phi = \mathbf{\Phi}$ by the inductive hypothesis, and $\forall \phi_o.\ \mathcal{A}[\![e_\phi]\!]^\tau \sigma^i_\phi \phi_0 = \mathcal{A}[\![e]\!]^\tau \sigma^i_s \phi = 0$, so $\phi \notin \Phi^{i+1}_s$ and $\Phi^{i+1}_\phi = \emptyset$, preserving the invariant.

- **case 3:** $\phi \notin \Phi^i_s$. For this case, $\Phi^i_\phi = \emptyset$ by the inductive hypothesis, and because assertion is monotonic, then $\phi \notin \Phi^{i+1}_s$ and $\Phi^i_\phi = \emptyset$, again preserving the invariant.

The proof for other cases follows a similar logic, and this completes the proof. The theorem makes formal the relationship between the synthesis semantics and the concrete programs that the sketch may generate.

### 3.4.2 Some Algebraic Properties of the Semantic Rules.

In this section, I give some useful lemmas that follow directly from the synthesis semantics. These lemmas will prove useful in supporting some optimizations in our solution algorithm.

The first lemma shows distributivity of the denotation function over the $\cap$ operator.

**Lemma 2**

$$\begin{array}{l}\mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_x) = (\sigma'_a, \Phi'_x) \\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_y) = (\sigma'_a, \Phi'_y)\end{array} \rightarrow \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_x \cap \Phi_y) = \begin{array}{l}(\sigma'_a, \Phi'_x \cap \Phi'_y)\ or \\ (\sigma'_a, \Phi'_x \cap \Phi_y)\ or \\ (\sigma'_a, \Phi_x \cap \Phi'_y)\end{array}$$

**Proof:** This lemma can be proved by induction on the structure of the derivation. The only interesting cases are the proofs for the **assert** and **if** rules; all other cases are rather mechanical.

In the case of **assert**, this is easy to check. Notice that if we define $\Phi_e = \{\phi : \mathcal{A}[\![e]\!]^\tau \sigma\phi = 1\}$, then

$$\mathcal{C}[\![\textbf{assert } e]\!]^\tau\langle\sigma\,, \Phi_x\rangle = \langle\sigma\,, \Phi_e \cap \Phi_x\rangle$$
$$\mathcal{C}[\![\textbf{assert } e]\!]^\tau\langle\sigma\,, \Phi_y\rangle = \langle\sigma\,, \Phi_e \cap \Phi_y\rangle$$
$$\mathcal{C}[\![\textbf{assert } e]\!]^\tau\langle\sigma\,, \Phi_x \cap \Phi_y\rangle = \langle\sigma\,, \Phi_e \cap (\Phi_x \cap \Phi_y)\rangle$$

And the property in the lemma follows from this last equation.

In the case of **if**, a similar argument holds. First, consider the original rule for evaluation of an **if** statement.

$$\Phi_t = \{\phi \in \Phi_x : \mathcal{A}[\![e]\!]^\tau \sigma\phi = true\}$$
$$\Phi_f = \{\phi \in \Phi_x : \mathcal{A}[\![e]\!]^\tau \sigma\phi = false\}$$
$$\langle\sigma_1\,, \Phi_1\rangle = \mathcal{C}[\![c_1]\!]^\tau\langle\sigma\,, \Phi_t\rangle$$
$$\langle\sigma_2\,, \Phi_2\rangle = \mathcal{C}[\![c_2]\!]^\tau\langle\sigma\,, \Phi_f\rangle$$
$$\Phi'_x = (\Phi_1) \cup (\Phi_2)$$
$$\mathcal{C}[\![\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]\!]^\tau\langle\sigma\,, \Phi_x\rangle = \langle\sigma'\,, \Phi'_x\rangle$$

It is possible to rewrite the definition of $\Phi'_x$ using the inductive hypothesis. First, lets define $\Phi_e = \{\phi : \mathcal{A}[\![e]\!]^\tau \sigma\phi = true\}$, then we have that

$$\Phi_t = \Phi_e \cap \Phi_x$$
$$\Phi_f = (\neg\Phi_e) \cap \Phi_x$$

Now, let

$$\langle \sigma_{1e}, \Phi_{1e} \rangle = \mathcal{C}[\![c_1]\!]^\tau \langle \sigma, \Phi_e \rangle$$
$$\langle \sigma_{2e}, \Phi_{2e} \rangle = \mathcal{C}[\![c_2]\!]^\tau \langle \sigma, (\neg\Phi_e) \rangle$$

Then, by the inductive hypothesis and the original **if** rule, we have that

$$\begin{aligned} \Phi'_x &= (\Phi_{1e} \cap \Phi_x) \cup (\Phi_{2e} \cap \Phi_x) \\ &= (\Phi_{1e} \cup \Phi_{2e}) \cap \Phi_x \end{aligned}$$

And, by a similar argument,

$$\begin{aligned} \Phi'_y &= (\Phi_{1e} \cap \Phi_y) \cup (\Phi_{2e} \cap \Phi_y) \\ &= (\Phi_{1e} \cup \Phi_2) \cap \Phi_y \end{aligned}$$

And also

$$\pi_\Phi(\ \mathcal{C}[\![\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2]\!]^\tau \langle \sigma, \Phi_x \cap \Phi_y \rangle) = (\Phi_{1e} \cup \Phi_2) \cap (\Phi_y \cap \Phi_x)$$

And again, the properties in the lemma follow directly from this last equation. □

From the distributivity lemma, monotonicity is easy to prove as a corollary. Monotonicity will allow us to make convergence arguments about iterative algorithms.

**Corollary 1**

$$\pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_x)) = \Phi'_x$$
$$\pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_y)) = \Phi'_y$$
$$\Phi_x \subset \Phi_y \rightarrow \Phi'_x \subset \Phi'_y$$

**Proof:** If $\Phi_x \subset \Phi_y$, then $\Phi_x = \Phi_x \cap \Phi_y$, so

$$\begin{aligned}\pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_x)) &= \pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_x \cap \Phi_y)) \\ &= \Phi'_x \cap \Phi'_y\end{aligned}$$

So $\Phi'_x = \Phi'_x \cap \Phi'_y \subset \Phi'_y$. $\square$

This completes the description of the synthesis semantics of the SKETCH language. The section has shown how the semantics model the behavior of all candidates simultaneously, and how they can be used to prove the safety of program transformations, such as those involved in the partial evaluation. The formal machinery developed in this section will also enable a formal description of the solution algorithm presented in the next section, and even its extension to concurrent programs presented in Chapter 8.

# Chapter 4

# Counterexample Guided Inductive Synthesis

## 4.1 Overview

In sketching, user insight is provided in the form of a partial program that needs to be completed, rather than as a set of derivation strategies the way it is in traditional deductive synthesizers. Therefore, the algorithms used for deductive synthesis are not as well suited to the problem of solving sketches, which requires algorithms that can search for the missing code fragments efficiently and without additional user input.

The problem has been made simpler by the way the SKETCH language defines all the sketching constructs in terms of the basic integer hole. This reduces the synthesis problem to a search for constant values to assign to each hole in the sketch. The synthesis semantics from the previous section allow this search to be framed as a constraint satisfaction problem. Specifically, the synthesis semantics define the meaning of a program $P$ through a denotation function $\mathcal{C}[\![P]\!]^{\tau_\emptyset}\langle\sigma_{in}, \Phi\rangle \rightarrow \langle\sigma_{out}, \Phi'\rangle$. The function describes how the program transforms an input state $\sigma_{in}$ into an output state $\sigma_{out}$, and how an initial set $\Phi$ of candidate solutions is constrained down to a subset $\Phi'$ containing only those solutions which are correct for input $\sigma_{in}$. Therefore, a valid candidate is one that satisfies the constraints imposed by each of the possible inputs to the sketch.

$$\forall\, \sigma \in \Sigma \;\; \mathcal{C}[\![P]\!]^{\tau_\emptyset}\langle\sigma, \{\phi\}\rangle = \langle\sigma', \{\phi\}\rangle \tag{4.1.1}$$

The synthesis semantics allow us to derive a set of constraints on $\phi$ in terms of the

input state $\sigma$. If we use the predicate $Q(\phi, \sigma)$ to represent these constraints, the synthesis problem becomes a doubly quantified constraint system.

$$\exists \phi \, \forall \sigma \; Q(\phi, \sigma) \tag{4.1.2}$$

Unfortunately, solving constraint systems involving such universally quantified variables is difficult, and many existing approaches do not scale to the size and complexity of the sketches we want to solve. For example, symbolically eliminating $\sigma$ from the constraint system $Q$ works for simple cases like the `HelloWorld` example, but is infeasible when the constraints involve tens of thousands of arithmetic and boolean operations. Similarly, we decided early on to bound the set of possible inputs. This makes it theoretically possible to expand the equation above to a conjunction of $Q(\phi, \sigma_i)$ for all the individual inputs $\sigma_i$. But while the domain of the input variables is bounded, for realistic sketches the space of possible inputs is still huge, on the order of $2^{128}$ elements for some sketches we've solved. In short, none of the obvious strategies will work.

In fact, there is a competition every year for solvers capable of solving boolean constraint systems with multiple quantifiers, such as Equation (4.1.2). But Section 6.4 will show that even the solver that won the competition last year is incapable of solving the constraint systems generated by relatively simple sketches. In short, none of the generic approaches to solving such constraint systems will allow us to solve the sketch synthesis problem. However, sketches are not arbitrary constraint systems; they are partial programs written to convey the high level structure of a solution while leaving the details unspecified. Therefore, a decision procedure that takes advantage of the structure embodied in sketches can succeed where the general solution strategies failed.

### 4.1.1 Solving Sketches with Inductive Synthesis

The crucial observation that makes sketch synthesis possible is that for many sketches, an implementation that works correctly for the common case and for all the different corner cases is likely to work correctly for all inputs. For example, consider the sketch of a remove method for a doubly linked list.

```
generator cond(list l, node n){
        node n1 = {| l.head | l.tail | n | n.next | n.prev | null |}
        node n2 = {| l.head | l.tail | n | n.next | n.prev | null |}
        return {|  n1 ( == | != ) n2 | ?? |}
}
generator assign(list l, node n){
        {| l.head | l.tail | n.prev | n.next | n.prev.next | n.next.prev |} =
                        {| l.head | l.tail | n | n.next | n.prev | null |} ;
}
static void remove(list l, node n){
        loop(4){
                if(cond(l,n)){assign(l, n);}
        }
}
```

The space of candidate solutions for this sketch is enormous, and so is the space of possible inputs. However, in addition to the common case where an element is removed from the middle of the list, there are only a handful of corner cases that can cause problems, such as the cases involving removal of the head, the tail, and removal from a list of size one. Therefore, the synthesis problem can be simplified enormously by focusing only on a handful of inputs that are representative of the common case and of the problematic corner cases. This insight can be made more formal through the following empirical hypothesis.

**Hypothesis 1 (Bounded Observation Hypothesis)** *For a given sketch $P$, it is possible to find a small set of inputs $E$ that fully represents the entire domain of inputs $\Sigma$ such that any set of controls $\Phi$ satisfying Equation* (4.1.3) *will also be a solution to the sketch resolution equation of Section 3.3.*

$$\forall\, \sigma \in E\ \pi_{\Phi}(\ \mathcal{C}[\![P]\!]^{\tau_\emptyset}\langle \sigma\,,\, \Phi\rangle) = \Phi \tag{4.1.3}$$

*The function $\pi_\Phi$ is the projection operator that selects the candidate set $\Phi$ from a pair $\langle \sigma'\,,\, \Phi\rangle$.*

The hypothesis implies that we can frame the sketch synthesis problem as an inductive synthesis problem. Inductive synthesis is the process of generating a program from concrete observations of its behavior, where an observation describes the expected behavior

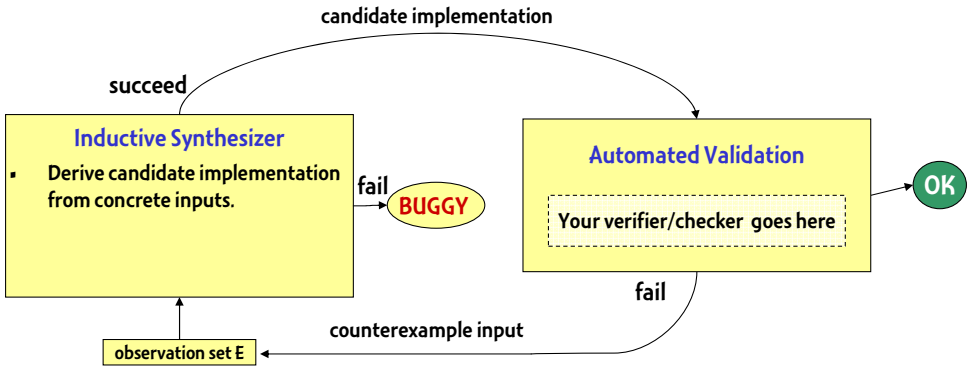



Figure 4.1: Counterexample driven synthesis algorithm.

of the program on a specific input [5]. The inductive synthesizer uses each new observation to refine its hypothesis about what the correct program should be until it converges to a solution. Inductive synthesis had its origin in the work by Gold [33] on language learning, and the pioneering work by Shapiro [57] on inductive synthesis and its application to algorithmic debugging among others.

Three important problems must be resolved in order to apply inductive synthesis to the problem of sketch resolution. First, it is necessary to have a mechanism to generate observations to drive the inductive synthesis. This mechanism should be able to generate inputs that exercise the corner cases in the implementation so the inductive synthesis quickly converges to a correct candidate. Second, the system needs a mechanism to determine convergence, *i.e.* to decide when the candidate derived from the set of observations actually generalizes to work correctly for all inputs. And finally, the system needs an inductive synthesis procedure capable of efficiently solving Equation (4.1.3) for realistic sketches.

To address the first two problems, we designed a counterexample guided inductive synthesis algorithm (CEGIS). This algorithm handles convergence checking and observation generation by coupling the inductive synthesizer with a validation procedure as illustrated in Figure 4.1. In the algorithm, a validation procedure checks the candidate implementation produced by the inductive synthesizer. If the validation succeeds, the candidate is considered correct, and is returned to the user. If validation fails, then the validation procedure is expected to produce a concrete input which exhibits the bug in the candidate program. The witness to the bug can then be used as an observation for the inductive synthesizer.

The CEGIS algorithm owes an intellectual debt to the idea of counterexample guided abstraction refinement (CEGAR) introduced by Clarke *et al.* [18] to cope with the state explosion problem in model checking. CEGAR exploits the observation that a counterexample is much easier to find in an abstract model, but abstract models can produce spurious counterexamples which are infeasible in the concrete model. This drawback can be alleviated by combining the abstract model checker with a validation procedure that can check whether a counterexample is indeed feasible for the original model. If it isn't, the validation procedure can refine the abstraction to disallow the spurious counterexample, and the cycle can be repeated. If we view the input set $E$ as an abstraction of the original input domain, the CEGIS algorithm can be seen as an application of the CEGAR idea to the problem of program synthesis.

## 4.2 Formalization of Algorithm and Termination Issues

The algorithm illustrated in Figure 4.1 can be succinctly expressed in terms of the synthesis semantics. In the algorithm below, $\Phi_i$ is the set of all controls which satisfy the specification for the input states $E = \{\sigma_0, \ldots, \sigma_{i-1}\}$. The control $\phi_i$ is a candidate selected non-deterministically from $\Phi_i$, and it constitutes the result of the inductive synthesis, as it is guaranteed to be correct for all inputs in $E$. The state $\sigma_i$ is an input which exposes an error in the candidate program represented by $\phi_i$. The initial control set $\Phi_0$ is initialized to $\mathbf{\Phi}$, the set of all controls, while $\sigma_0$ is initialized to a random initial state.

**Algorithm 1 (CEGIS Algorithm)** .

$\sigma_0 := \sigma_{\texttt{random}}$
$\Phi_0 = \mathbf{\Phi}$
$i := 0$
**do**
  $i = i + 1$

  **def** $\Phi_i$ `s.t.` $\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_{i-1}, \Phi_{i-1}) = (\sigma', \Phi_i)$
  **if** $\Phi_i = \emptyset$ **then return** `UNSAT_SKETCH`
  **def** $\phi_i \in \Phi_i$

  } `Inductive Synthesis Phase`

  **def** $\sigma_i$ `s.t.` $\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_i, \{\phi_i\}) = (\sigma', \emptyset)$ } `Validation Phase`

**while** $\sigma_i \neq$ **null**
**return** $PE(P, \phi_i)$

Each iteration of the CEGIS loop starts with the inductive synthesis phase. In this phase, a new set $\Phi_i$ is computed by removing from $\Phi_{i-1}$ those controls which cause the specification to be violated for the input $\sigma_{i-1}$. As will be described in detail in Chapter 5, $\Phi_i$ is represented symbolically, and it is derived by applying $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$ to $\sigma_{i-1}$ and to the symbolic representation of $\Phi_{i-1}$. The symbolic representation is then queried for an element $\phi_i \in \Phi_i$ which is the result of the inductive synthesis phase.

The validation phase of the algorithm checks whether the candidate solution associated with $\phi_i$ satisfies the specification for all possible inputs. If it does, then the candidate generated from control $\phi_i$ is the solution that the algorithm was looking for; if it does not, then the process is repeated until either a solution is found or $\Phi_i$ becomes empty. In the latter case, we can assert that the sketch has no valid solutions.

The sets $\Phi_i$ generated by the CEGIS algorithm form a series that approaches $\Phi^*$, the maximal solution of the sketch equation, in strictly monotonic fashion. To see this, first note that each of the $\Phi_i$ are going to be a superset of $\Phi^*$ by Corollary 1 in Section 3.4.2. Additionally, for each $i > 1$, $\Phi_i \subset \Phi_{i-1}$. This is clear from the fact that for $i > 1$, $\phi_{i-1} \in \Phi_{i-1}$, but $\phi_{i-1} \notin \Phi_i$. This means that if $\mathbf{\Phi}$ is bounded, then the procedure above is guaranteed to terminate, and $\Phi_i$ will converge towards $\Phi^*$. In fact, because the algorithm is only looking for a single $\phi \in \Phi^*$, it can actually terminate before $\Phi_i$ has converged to $\Phi^*$ if the $\phi_i$ selected from $\Phi_i$ also happens to be in $\Phi^*$.

The theoretical convergence properties of the algorithm are not great. The number of iterations is bounded by the maximum of the size of the control space and the size of the input space. Even for bounded sketches, these sizes can be astronomical. Moreover, if we don't bound $\mathbf{\Phi}$, the CEGIS algorithm can easily iterate forever. A curious example of this is the sketch below, which requires that the $i^{th}$ bit of $\texttt{??}_0$ be equal to $i \; mod \; 2$.

```
void main(int i){

  int z = (??0 / pow(2, i) ) % 2;

  assert z == i % 2;
}
```

If we didn't bound the set of possible values for $\texttt{??}_0$, then the CEGIS algorithm would iterate forever on this sketch which actually has no solution according to the synthesis semantics.

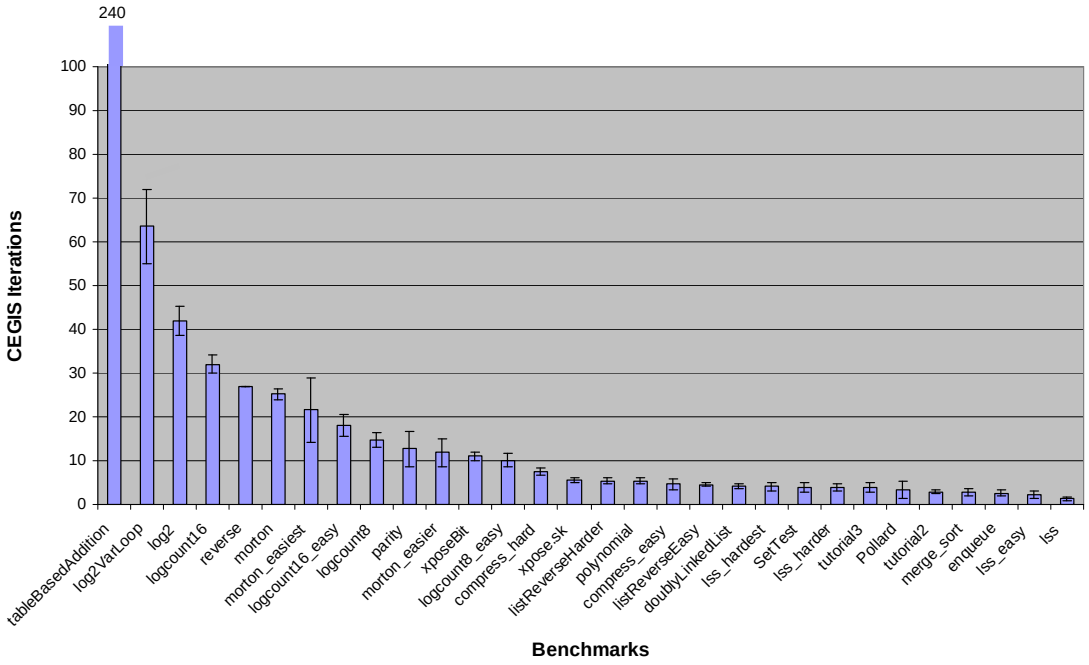



Figure 4.2: Iterations per Benchmark.

However, the CEGIS algorithm was design to exploit our intuition that a few inputs covering all the relevant corner cases should allow us to infer the correct solution to the sketch. As the next section will show, the number of iterations of the CEGIS loop for real programming problems is surprisingly small; on average, each iteration will cut the size of the candidate set by a factor of about $2^{15}$, so sketches with candidate spaces with $2^{600}$ elements take on the order of 40 iterations, and sketches with $2^{80}$ candidates can converge after about five iterations.

## 4.3 Empirical Validation of Bounded Observation Hypothesis

The CEGIS algorithm was designed to exploit the observation that a small number of carefully selected inputs should be sufficient to allow an inductive synthesizer to derive a correct solution to the sketch. In this section, we seek to validate this empirical hypothesis by analyzing the behavior of the solver on 30 different sketches of various degrees of complexity, ranging from small bit manipulation routines to complex datastructure manipulations like the list reversal example from the introduction. The problems are listed in Appendix A,

and are also described in more detail in Section 6.1. Each benchmark was run an average of 10 times to get an average number of iterations of the CEGIS loop, as well as the standard deviation on the number of iterations.

The actual solution times for all these benchmarks are described in detail in Section 6.1; they range from a few seconds to about 12 minutes for the hardest benchmark. For this section, however, we want to focus on the number of iterations because this is a measure that is fundamental to the CEGIS algorithm. The actual solution time will vary widely depending on the technology used for the inductive synthesizer and the validation procedure, but the number of iterations is more a function of the characteristics of the individual benchmarks.

For all the benchmarks, the number of iterations was very small given the size of the input and the candidate spaces. The largest number of iterations was for the `tableBasedAddition` benchmark, which implements an addition of two 4-bit numbers as a single table lookup, where all the entries in the table are left empty for the synthesizer to discover. For this benchmark, the number of iterations was, as we would expect, equal to the size of the input space, since each input provides information about only one entry in the table. For less contrived benchmarks, however, the CEGIS algorithm was very good at abstracting the entire input space into a few representative inputs. For example, for the `listReverseHarder` example from the introduction, the synthesizer found a solution to the sketch from between 4 and 6 representative input lists.

Figure 4.2 also shows error bars of one standard deviation for each benchmark. The variability in the number of iterations for a given benchmark comes from the nondeterministic choice the CEGIS algorithm makes in selecting a $\phi_i \in \Phi_i$, and from the choice of the $\sigma_i$ that the validation phase decides to produce. However, you may notice that the number of iterations was fairly stable for each of the benchmarks. Of the 30 problems we tested, only 5 had a standard deviation of more than 2 iterations, and only 3 had a standard deviation larger than 4 iterations. This consistency suggests that there is something intrinsic to each benchmark that determines the number of observations needed for inductive synthesis to converge, irrespective of the nondeterministic choices made by the synthesizer.

I was interested in gaining a better understanding of the properties of a sketch that determine the number of observations needed for convergence, in order to better predict the scalability of the approach for more complex sketches. In order to do this, I analyzed the correlation between the number of iterations and a small number of benchmark parameters.

Counte

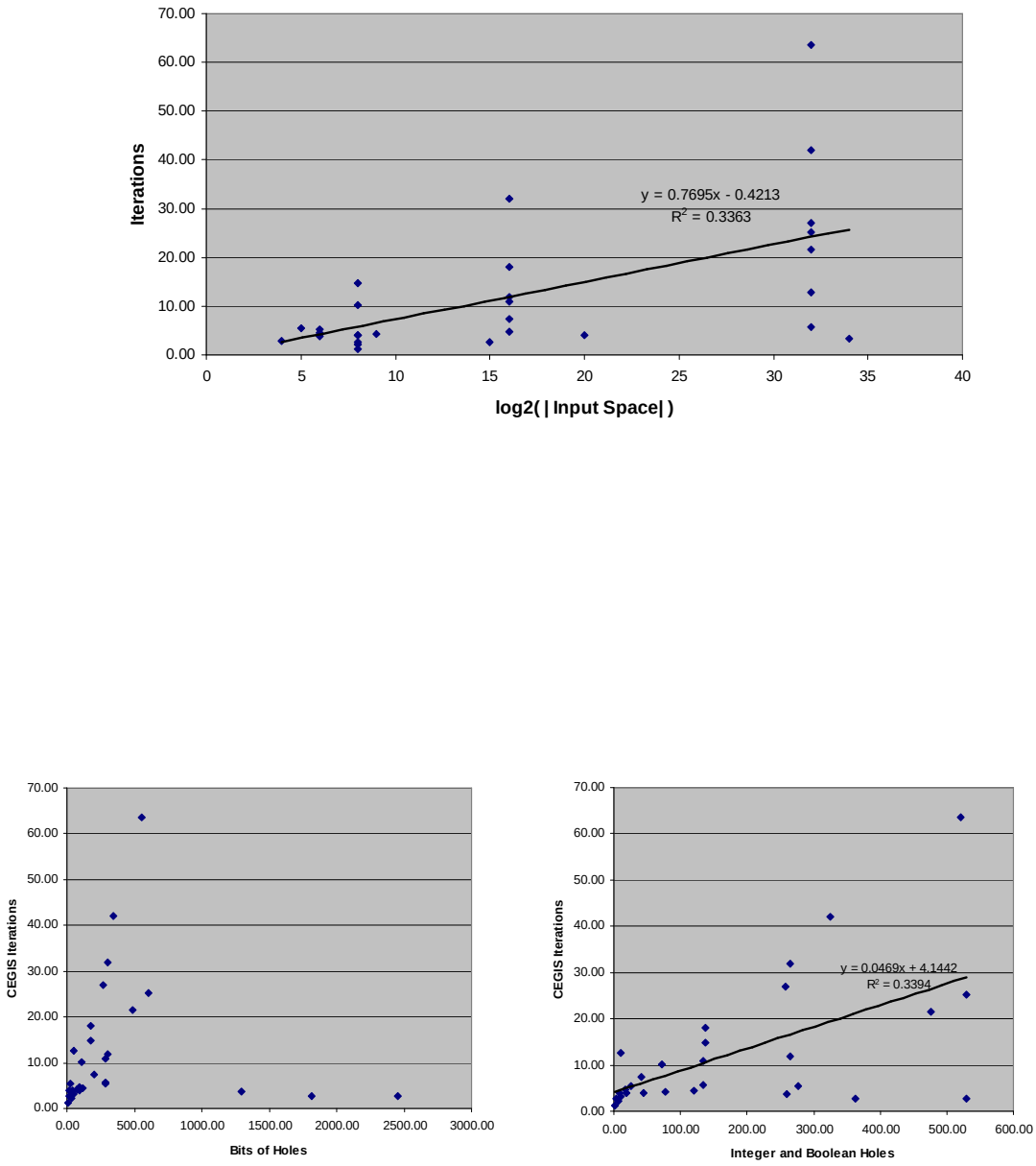



Figure 4.4: Iterations vs. number of holes.

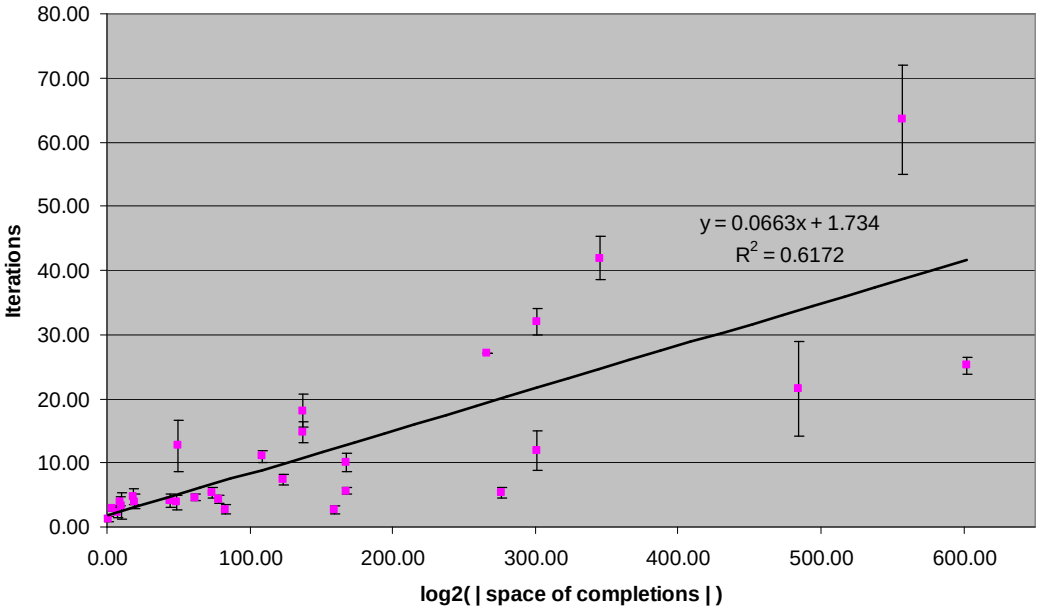



Figure 4.5: Iterations vs. distinct candidates.

I started with three plausible hypothesis regarding the number of iterations of the CEGIS loop.

- **Hypothesis 1**: The number of iterations is proportional to the log of the size of the input space. After all, it seems plausible that more concrete inputs would be needed to represent a bigger input space, while the small number of iterations for the huge sizes of the input space suggested that if any relationship existed, it should be logarithmic.

- **Hypothesis 2**: The number of iterations is proportional to the number of holes in the program. This sounds plausible; after all, we know that for polynomials, $N$ independent observations are enough to discover the coefficients in a polynomial of degree $N$. The same is true for linear equations, so it's plausible that sketches could behave the same way.

- **Hypothesis 3**: The number of iterations is proportional to the log of the size of the candidate space. This hypothesis is similar to the previous one, but it acknowledges the fact that for some skethes, many different controls may produce the exact same candidate program.

Figure 4.3 shows the relationship between the number of iterations and the logarithm of the input size to the benchmark. As we can see, there is a slight correlation, but it's very weak; $R^2$ is about 0.33, and it's easy to see that the number of iterations may vary widely, even for benchmarks with the same number of inputs.

Figure 4.4 on the left shows the relationship between the number of iterations and the number of bits used to represent all the integer and boolean holes in the program. By default, most integer holes are represented using 5 bits, but some of the higher level constructs may translate into integer holes that use more or fewer bits. The graph on the right shows the relationship between the number of iterations and the number of independent integer holes. When considering the number of bits, the relationship is very weak, although this is mostly due to a handful of benchmarks that have huge numbers of bits for the holes, but take very few iterations. When we consider the number of independent integer holes, the relationship improves, but it's still very weak.

Finally, Figure 4.5 shows the relationship between the number of iterations and the logarithm of the size of the candidate space of the benchmark. Now, we can see the correlation is much stronger; $R^2$ is now over 0.61. The reason why the log of the candidate space is very different from the number of bits of holes is that some sketches exhibit a lot of redundancy: there are many combinations of hole values that after partial evaluation produce the same program, and there are some combinations which are simply illegal. For example, if an integer hole is used to select among three different choices, the sketch may have a statement of the form `int t = ??; assert t<3;`. In this case, the synthesizer may be using 5 bits to represent the hole being assigned to `t`, but there are only three distinct candidates. Recall that when we compared the number of iterations against the number of bits for the holes, there was a set of benchmark which took few iterations even though they had a huge number of unknowns. These benchmarks were the `hardSort`, the `enqueue` and the `tutorial3` benchmarks. All of these benchmarks involved complex generators to describe large classes of expressions. The generators, however, had a lot of redundancy. For example, for `tutorial3`, there was a single generator which represented a family of $4 \times 10^{14} \approx 2^{49}$ syntactically distinct expressions, but used 259 integer holes, each represented with 5-bits. The fact that the number of iterations is better predicted by the number of unique candidates than the number of holes points to one of the strengths of the CEGIS approach: the ability to eliminate large classes of equivalent candidates with a single representative input.

These experiments largely validate the bounded observation hypothesis. They have

demonstrated that for many real problems, the number of observations needed to find a valid control is quite small. Moreover, they show that the each new iteration is able to eliminate a fraction of the remaining candidate space, including huge numbers of equivalent solutions. Having shown this, it remains to be shown how effectively the inductive synthesis procedure is able to generate candidate solutions from sets of observations; this will be the subject of the next two sections.

# Chapter 5

# SAT Based Inductive Synthesis and Validation

The CEGIS procedure depends on an inductive synthesizer to generate candidate implementations from a small set of inputs, and a validation procedure to produce counterexample inputs exposing problems in invalid candidates. In the previous section, I showed how the inductive synthesizer and the validator could be expressed in terms of the synthesis semantics. This section will start by showing how the bounded synthesis semantics can be used to derive constraint systems for both inductive synthesis and validation (Section 5.1). It will also show how these constraint systems are symbolically manipulated to make them smaller and easier to solve (Section 5.2), and how the resulting systems are converted to SAT for efficient solution (Section 5.3).

## 5.1 Symbolic Evaluation of Synthesis Semantics

In the formalization of the CEGIS algorithm, I showed how the inductive synthesis and the validation problems could be expressed in terms of the synthesis semantics.

$$\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_{i-1}, \Phi_{i-1}) = (\sigma', \Phi_i) \tag{5.1.1}$$

$$\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_i, \{\phi_i\}) = (\sigma', \emptyset) \tag{5.1.2}$$

Inductive synthesis is defined using Equation (5.1.1), which describes how a set of candidate controls $\Phi_{i-1}$ is constrained to a set $\Phi_i$ by removing from it those controls that are invalid for input $\sigma_{i-1}$. The inductive synthesizer must then select a control $\phi_i \in \Phi_i$ which represents the

| | | | |
|---|---|---|---|
| Base expressions | $e$ | $::=$ | $c \mid h_{i,\tau} \mid in_i$ |
| Arithmetic expressions | $e$ | $::=$ | $+(e_1, e_2) \mid -(e) \mid *(e_1, e_2) \mid$ $div(e_1, e_2) \mid mod(e_1, e_2)$ |
| Comparisson expressions | $e$ | $::=$ | $<(e_1, e_2) \mid >(e_1, e_2) \mid$ $\geq(e_1, e_2) \mid \leq(e_1, e_2) \mid =(e_1, e_2)$ |
| Boolean expressions | $e$ | $::=$ | $\vee(e_1, e_2) \mid \wedge(e_1, e_2) \mid \oplus(e_1, e_2) \mid \neg(e)$ |
| Selection expressions | $e$ | $::=$ | $mux_n(e_{idx}, e_1, \ldots, e_n) := e_{e_{idx}}$ $if_c(e_{ind}, e_{\neq}, e_{=}) := (e_{ind} = c)\ ?\ e_{=} : e_{\neq}$ |

Table 5.1: Intermediate language used to represent parameterized values

solution to the inductive synthesis problem. Validation is defined through Equation (5.1.2); it requires the synthesizer to select an input $\sigma_i$ that shows that control $\phi_i$ can not be in the set of solutions to the sketch synthesis equation, *i.e.* input $\sigma_i$ causes an assertion failure on the candidate represented by control $\phi_i$.

These two equations describe inductive synthesis and validation respectively, but they are not algorithmic; the semantic rules describe manipulations on sets and functions in the abstract, but they don't tell us how these objects should be represented, or how the manipulations should be implemented. This section describes an implementation of the synthesis semantics that turns the inductive synthesis and validation problems into constraint satisfaction problems. The implementation is based on an important idea: that sets can be represented symbolically as systems of constraints. Specifically, a set $\Phi$ of controls can be represented as constraints that must be satisfied by all the controls in $\Phi$. For example, the constraint $(\phi(\texttt{??}_0) = 5 \wedge \phi(\texttt{??}_1) < 3)$ represents the set of all controls that assign 5 to the hole $\texttt{??}_0$ and a value less than 3 to hole $\texttt{??}_1$. By representing sets of controls symbolically as systems of constraints, we will be able to derive systems of constraints for $\Phi_i$ by manipulating the constraints representing $\Phi_{i-1}$ according to the rules of the synthesis semantics. Extracting a control $\phi \in \Phi_i$ then becomes a constraint satisfaction problem.

To show how the constraint systems are constructed, I begin by describing the representations of controls. A control $\phi$ describes the value of each hole under every possible calling context. The number of holes in the program is bounded, and because we are restricting ourselves to bounded semantics, so is the number of calling contexts. Therefore, if we assume there are $k$ distinct pairs of holes and calling contexts, we can represent $\phi$ as a *control vector*, $\langle h_0, \ldots, h_k \rangle$, where each control value $h_i$ corresponds to the value of a specific hole under a specific calling context. I will sometimes use the notation $h_{i,\tau}$ when I want to make explicit the exact hole and calling context for a given control value. The constraint system will describe constraints on the values $h_i$ that make up the control vector.

Now, recall that the values of expressions and variables are represented in the semantics as parameterized values, which are functions mapping controls to concrete values $\psi : \mathbf{\Phi} \rightarrow \mathbb{Z}$. The synthesizer represents parameterized values symbolically as expressions in the language described in Table 5.1. These expressions are represented in the synthesizer as DAGs, rather than trees, to allow sharing of common subexpressions. The base expressions in this language can be of three types:

- Controls $h_{i,\tau}$ indicating a specific component in the control vector.
- Integer constants.
- Input nodes $in_i$, which serve as place holders for concrete inputs.

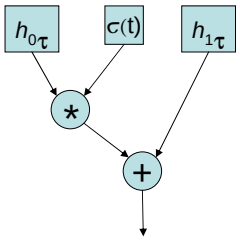
The state $\sigma$ is a mapping of variable names to parameterized values. Through the derivation process, the state is read and updated according to the synthesis semantics. For example, consider the various rules for evaluating expressions. Those rules are easily adapted to construct expressions in the intermediate language of Table 5.1.

$$\begin{aligned} \mathcal{A}[\![x]\!]^\tau\sigma &= \sigma(x) \\ \mathcal{A}[\![??_i]\!]^\tau\sigma &= h_{i,\tau} \\ \mathcal{A}[\![e_1 \ + \ e_2]\!]^\tau\sigma &= +(\ \mathcal{A}[\![e_1]\!]^\tau\sigma,\ \mathcal{A}[\![e_2]\!]^\tau\sigma) \end{aligned}$$

For example, the expression $\texttt{t} \ * \ \texttt{??}_0 + \ \texttt{??}_1$ is translated into an expression in the intermediate language through the following syntax directed translation.

$$\begin{aligned} \mathcal{A}[\![\texttt{t} * \texttt{??}_0 + \texttt{??}_1]\!]^\tau\sigma &= +(\ \mathcal{A}[\![\texttt{t} * \texttt{??}_0]\!]^\tau\sigma,\ \mathcal{A}[\![\texttt{??}_1]\!]^\tau\sigma) \\ &= +(*(\ \mathcal{A}[\![\texttt{t}]\!]^\tau\sigma,\ \mathcal{A}[\![\texttt{??}_0]\!]^\tau\sigma), h_{1,\tau}) \\ &= +(*(\sigma(\texttt{t}), h_{0,\tau}), h_{1,\tau}) \end{aligned}$$

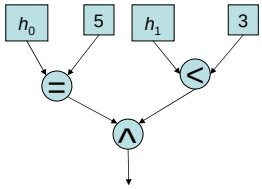
The final expression is a function of the control values $h_{0,\tau}$ and $h_{1,\tau}$, and the input value of variable `t`, and is represented graphically by the tree shown below.




Sets of controls are represented as systems of constraints on the control values. To represent and manipulate these constraints, we exploit the intermediate language used to represent parameterized values by associating with each set $\Phi$ a characteristic function $\psi_\Phi$ related to $\Phi$ through the following equation.

$$\Phi = \{\phi : \psi_\Phi(\phi) \neq 0\} \tag{5.1.3}$$

In other words, a control $\phi$ belongs to $\Phi$ if and only if it satisfies the constraint $\psi_\Phi(\phi) \neq 0$. For example, if $\Phi$ is the set of controls satisfying $\phi(\texttt{??}_0) = 5$ and $\phi(\texttt{??}_1) < 3$, this set will be represented with the characteristic function $\wedge(=(h_0, 5), <(h_1, 3))$, shown graphically below.




Most standard set operations are easy to perform on the symbolic representation. For example, if $\psi_{\Phi_1}$ and $\psi_{\Phi_2}$ are the characteristic functions for the sets $\Phi_1$ and $\Phi_2$ respectively, then the characteristic functions for the complement, intersection, and union of these sets are easy to construct from $\psi_{\Phi_1}$ and $\psi_{\Phi_2}$ as illustrated below.

$$\text{Complement} \quad \neg\Phi_1 = \quad \neg(\psi_{\Phi_1}) \tag{5.1.4}$$

$$\text{Intersection} \quad \Phi_1 \cap \Phi_2 = \quad \wedge(\psi_{\Phi_1}, \psi_{\Phi_2}) \tag{5.1.5}$$

$$\text{Union} \quad \Phi_1 \cup \Phi_2 = \quad \vee(\psi_{\Phi_1}, \psi_{\Phi_2}) \tag{5.1.6}$$

$$\tag{5.1.7}$$

To make the presentation easier to follow, I give special names to some sets of interest.

- $\psi_{\mathbf{\Phi}} := 1$ represents the universal set $\mathbf{\Phi}$.
- $\psi_{\{\phi\}}$ represents the singleton set containing only $\phi$.
- $\psi_{\emptyset} := 0$ represents the empty set.

The rules of the synthesis semantics are used to construct the characteristic functions through syntax directed translation. For example, the basic statements of assignment and assertion manipulate the state and the set of valid controls according to the following rules.

$$\mathcal{C}[\![x := e]\!]^{\tau}\langle\sigma\,,\,\psi_{\Phi}\rangle = \langle\sigma[x \mapsto\ \mathcal{A}[\![e]\!]^{\tau}\sigma]\,,\,\psi_{\Phi}\rangle \tag{5.1.8}$$

$$\mathcal{C}[\![\textbf{assert}\ e]\!]^{\tau_{\emptyset}}\langle\sigma\,,\,\psi_{\Phi}\rangle = \langle\sigma\,,\,\wedge(\ \mathcal{A}[\![\sigma]\!]^{e}, \psi_{\Phi})\rangle \tag{5.1.9}$$

The same is true of the **if** statement; for the statement **if** $e$ **then** $c_1$ **else** $c_2$, we can follow the synthesis semantics to evaluate the two branches of the conditional.

$$\begin{aligned}
\psi_e &= \ \mathcal{A}[\![e]\!]^{\tau}\sigma \\
\psi_{\Phi_t} &= \wedge(\psi_{\Phi}, \psi_e) \\
\psi_{\Phi_f} &= \wedge(\psi_{\Phi}, \neg(\psi_e)) \\
\langle\sigma_1\,,\,\psi_{\Phi_1}\rangle &= \ \mathcal{C}[\![c_1]\!]^{\tau}\langle\sigma\,,\,\psi_{\Phi_t}\rangle \\
\langle\sigma_2\,,\,\psi_{\Phi_2}\rangle &= \ \mathcal{C}[\![c_2]\!]^{\tau}\langle\sigma\,,\,\psi_{\Phi_f}\rangle
\end{aligned}$$

Then, the rule for the **if** statement becomes

$$\begin{aligned}
&\mathcal{C}[\![\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2]\!]^{\tau}\langle\sigma\,,\,\psi_{\Phi}\rangle \\
&\qquad = \langle\lambda x.mux_2(\psi_e, \sigma_2(x), \sigma_1(x))\,,\,\vee(\psi_{\Phi_1},\ \psi_{\Phi_2})\rangle
\end{aligned}$$

The rules for loops and procedure calls follow the same logic; the symbolic representations are manipulated according to the synthesis semantics, replacing set operations with operations on the characteristic functions. Because we are using bounded semantics, we do not have to worry about termination of loops or recursion.

An important advantage of representing sets as a constraint on the value of a characteristic function is that it is possible to query for a control in the set through a constraint satisfaction procedure. Any solution $\phi$ to the constraint $\psi_{\Phi}(\phi) \neq 0$ is guaranteed to belong to $\Phi$; if the constraints are unsatisfiable, then it means that $\Phi$ is empty.

The idea of representing sets as systems of constraints is not new. In fact, it was one of the major advances behind symbolic model checking [48]. However, this is the first time this idea has been used for the purpose of software synthesis.

### 5.1.1 Inductive Synthesis

Inductive synthesis requires the synthesizer to compute a set of candidate solutions $\Phi_i$ from the input state $\sigma_{i-1}$ and the previous set of candidates. This can be done directly by evaluating $\mathcal{C}[\![P]\!]^\tau(\sigma_{i-1}, \Phi_{i-1})$ as was described earlier in the section, but the SKETCH synthesizer actually takes a shortcut. To make the process more efficient, and avoid having to evaluate $\mathcal{C}[\![P]\!]^\tau$ on each iteration of the CEGIS loop, the synthesizer takes advantage of the distributivity lemma from Section 3.4.2.

$$\begin{array}{l} \pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_x)) = \Phi'_x \\ \pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_y)) = \Phi'_y \end{array} \Rightarrow \pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_x \cap \Phi_y)) \begin{array}{l} = \Phi'_x \cap \Phi'_y \\ = \Phi'_x \cap \Phi_y \\ = \Phi_x \cap \Phi'_y \end{array}$$

From this lemma, we can derive the following useful corollary.

**Corollary 2** *The set $\Phi_i$ can be computed without having to evaluate $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$ on $\Phi_{i-1}$ through the following formula.*

$$\pi_\Phi(\ \mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_{i-1}, \mathbf{\Phi})) = \Phi'_i$$
$$\Phi_i = \Phi'_i \cap \Phi_{i-1}$$

**Proof:** The corollary follows directly from distributivity,

$$\begin{array}{l} \pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_{i-1}, \mathbf{\Phi})) = \Phi'_i \\ \pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_{i-1}, \Phi_{i-1})) = \Phi_i \end{array} \Rightarrow \pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \mathbf{\Phi} \cap \Phi_{i-1})) = \Phi'_i \cap \Phi_{i-1}$$

Since $\pi_\Phi(\ \mathcal{C}[\![P]\!]^\tau(\sigma_a, \Phi_{i-1})) = \Phi_i$, this completes the proof. □

By applying the corollary to the characteristic functions of the sets, we have that we can construct $\psi_{\Phi_i}$, the characteristic function of the set $\Phi_i$, by anding together the functions $\psi_{\Phi_{i-1}}$ and $\psi_{\Phi'_i}$. Thus, if we could construct the function $\psi_{\Phi'_i}$ without having to evaluate $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$, we would be able to construct $\Phi_i$ from $\Phi_{i-1}$ without having to do syntax directed translation for every CEGIS iteration.

To do this, the synthesizer evaluates $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$ on an input state $\sigma_{sym}$ that maps each input variable $in_k$ to an input node.

$$\pi_\Phi(\ \mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_{sym}, \psi_{\mathbf{\Phi}})) = \psi_{sym}$$

Then, $\Phi'_i$ can be constructed easily by replacing all the input nodes $in_k$ in $\psi_{sym}$ with constant nodes with value equal to $\sigma_{i-1}(in_k)$.

$$\Phi'_i = substitute(\psi_{sym}, \sigma_{i-1})$$

Consequently, $\psi_{\Phi_i}$ can be constructed from $\psi_{\Phi_{i-1}}$ without having to perform syntax directed translation on each iteration.

$$\psi_{\Phi_i} = \wedge(\psi_{\Phi_{i-1}}, substitute(\psi_{sym}, \sigma_{i-1}))$$

The resulting function $\psi_{\Phi_i}$ is then queried for a concrete control vector by searching for a $\phi$ that satisfies the constraint $\psi_{\Phi_i}(\phi) = 1$.

In our synthesizer, computing $\psi_{\Phi_i}$ in this way brings a number of advantages, both from the point of view of performance and from the point of view of software engineering. First, the denotation function $\mathcal{C}[\![P]\!]^{\tau}$ only has to be evaluated once, rather than on every iteration. Second, as I will demonstrate shortly, the function $\psi_{sym}$ can also be used to solve the validation problem without requiring the denotation function to be symbolically evaluated again; this allows the CEGIS loop to run with very little overhead. In fact, in the SKETCH synthesizer, the evaluation of $\mathcal{C}[\![P]\!]^{\tau}$ is performed by a clean and modular frontend written in Java, while the CEGIS loop is implemented very efficiently in a separate backend written in C++.

### 5.1.2 Validation

The result of the inductive synthesis phase is a control $\phi_i$, from which a concrete candidate program $P_{\phi_i} = PE(P, \phi_i)$ can be generated through partial evaluation. The goal of the validation phase is to find an input that causes an assertion failure in this candidate program. This can also be framed directly in terms of the synthesis semantics as the problem of finding an input $\sigma_i$ that shows that the control $\phi_i$ is not a solution to the sketch equation, so

$$\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_i, \{\phi_i\}) = (\sigma', \emptyset) \tag{5.1.10}$$

Just like in the case of inductive synthesis, the synthesizer uses the distributiviy lemma to frame this problem as a constraint satisfaction problem that does not require the evaluation of $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$ on each iteration of the CEGIS loop. With the distributivity lemma, it is possible to restate Equation (5.1.10) as follows.

$$\pi_\Phi(\ \mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_i, \mathbf{\Phi})) = \Phi'$$
$$\emptyset = \Phi' \cap \{\phi_i\}$$

Thus, the validation becomes a search for a $\sigma_i$ such that

$$\psi_\emptyset = \wedge(\psi_{\{\phi_i\}}, \mathit{substitute}(\psi_{sym}, \sigma_i)) \tag{5.1.11}$$

Because $\psi_{\{\phi_i\}}(\phi) = 1$ iff $\phi = \phi_i$, Equation (5.1.11) above is equivalent to the equation below.

$$\exists\, \sigma_i.\ \mathit{substitute}(\psi_{sym}, \sigma_i)(\phi_i) = 0 \tag{5.1.12}$$

The SKETCH synthesizer searches for an input state $\sigma_i$ satisfying the equation above using the exact same satisfyability procedure used for inductive synthesis.

Interestingly, the validation procedure that results from this symbolic manipulation is equivalent to the SAT-based bounded model checker developed by Clarke, Kroening and Yorav [19]. Their tool, called CBMC, also translates a program into a set of Boolean constraints and uses SAT to solve the system for a counterexample. Some of their encodings are different from outs, but the high-level ideas are the same.

Saturn [71] is another SAT based validation tool that operates through similar principles. One of the key features of Saturn is that it's able to abstract procedures into summaries, allowing for modular verification, which our system does not support. As we will describe in the following sections, our system uses some of the encoding techniques created for Saturn. What is most interesting about the similarity between our procedure with Saturn and CBMC is that the same techniques that proved successful for bug finding can be effective for inductive synthesis.

## 5.2 Preprocessing of Symbolic Representations

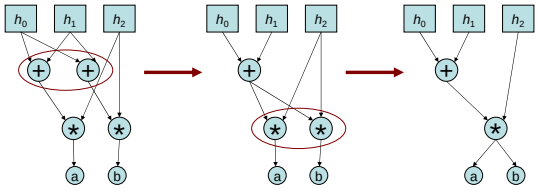
Earlier it was pointed our that parameterized values are represented as DAGS rather than trees so that common subexpressions can be shared. This is only one of a number of

optimizations that the SKETCH synthesizer applies to the characteristic functions to reduce their size and make the resulting constraint system easier to solve.

The synthesizer applies four classes of optimizations, most of which are specialized versions of well known optimizations from the literature. The common theme among all of these optimizations is that they exploit the structure embodied in the DAG representation.

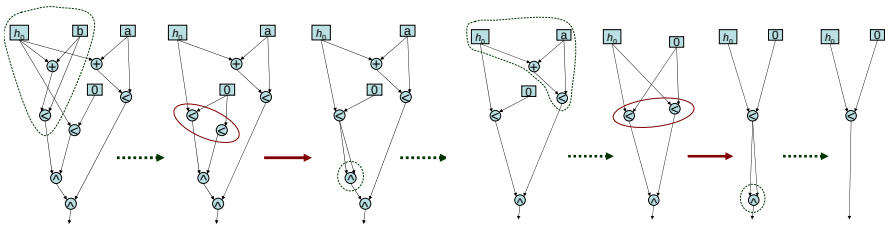
- structural hashing,
- algebraic symplification through pattern matching,
- forward and backward valueset analysis,
- canonicalization of ands and ors.

**Structural Hashing.** This is an old idea used widely in many circuit level solvers such as ABC [49] and SMT solvers such as UCLID [15] and STP [31]. The optimization performs common subexpression elimination similar to the way it is done by compilers. A hash value is computed for each subexpression based on its operator and operands, and if two subexpressions are found to be equivalent, one of them is eliminated. This ensures that the final DAG will not have many copies of structurally identical nodes.




**Simplification through pattern matching.** The DAGs constructed through direct application of the semantic rules often contain a lot of inefficiency which can be easily eliminated through simple pattern matching. The simplification is performed by applying the rewrite rules in Table 5.2 in tandem with structural hashing. The DAG is traversed in topological order; each node is first matched with the patterns on the left hand side of the rules; if a match is found, the node is replaced with the right hand side of the rule, and then the resulting node is hashed to see if any equivalent nodes are already present in the DAG. The figure below illustrates this process on a small circuit; red solid arrows correspond to structural hashing steps, while green dotted arrows correspond to pattern matching steps. The

rewrite rules allow the structural hashing to identify common subexpressions which would otherwise appear to be structurally different.



**Forward Valueset Analysis.** The pattern matching rules achieve constant propagation, so if the value of an expression is independent of the input and the control, the pattern matching rules will replace this expression with a constant. However, in a typical program there are many expressions that do depend on the input, but whose value ranges over a small set of values. Knowing this set of values will make some optimizations possible; for example, suppose an expression $e$ can have values 1, 3, or 5; then we know that the expression $e < 6$ will always evaluate to true, even though simple constant propagation would conclude that the value of this inequality is dependent on the input.

The forward valueset analysis uses a dataflow analysis to approximate the set of possible values that all the nodes in the DAG may take. These sets are then used to replace some expressions in the DAG with constants.

To define the analysis, let $V(e) \subset \mathbb{Z}$ be the approximation to the set of possible values that an expression $e$ may take under different inputs. The set $V(e)$ is defined recursively from the sets of values of the subexpressions that make up $e$ according to the following equations.

$$
\begin{aligned}
V(h_i) &= \mathbb{Z} \\
V(in_i) &= \mathbb{Z} \\
V(c) &= \{c\} \\
V(\circ(x,y)) &= \begin{cases} \mathbb{Z} & \textbf{if } V(x) = \mathbb{Z} \vee V(y) = \mathbb{Z} \\ \{a \circ b : a \in V(x) \wedge b \in V(y)\} & \text{otherwise} \end{cases} \\
V(if_c(t,x,y)) &= V(x) \cup V(y) \\
V(mux_n(t,x_1,\ldots,x_n)) &= \bigcup_{1 \le s \le n} V(x_s)
\end{aligned}
$$

The equations use the convention that the letter $c$ corresponds to a constant node and the operator $\circ$ stands for an arbitrary operator.

In the actual implementation, the sets $V(e)$ are represent explicitly as lists of values, so we must give a bound on the maximum length of the set to keep the representation from getting too big; once the list of values reaches this maximum, the set is widened to the symbolic value $\mathbb{Z}$.

Once the set of values for each node has been derived, all inequalities are checked to see if they will hold constant for all their possible inputs. For example, for an expression of the form $>(x, y)$, if for all $t \in V(x)$ and $v \in V(y)$, the inequality $t > v$ is true, the node $>(x, y)$ can be replaced with the constant node with value 1, and if the inequality is false for all $t$ and $v$, then it can be replaced with a constant node 0. The same substitution is performed for the operators $>, \geq, <$ and $\leq$,. Similarly, for expressions of the form $if_c(a, x, y)$, if $c$ is not in the set $V(a)$, then $a$ can never be equal to $c$, so the expression can be replaced with the subexpression $y$ according to the definition of $if_c(a, x, y)$.

The combination of structural hashing, simplification through pattern matching and forward value flow is applied as the circuit is constructed, preventing redundant nodes from fragmenting the heap before being optimized away.

**Backward Valueset Analysis** The motivation for this analysis is that in some cases, the value of a node is only relevant under certain conditions. For example, if there is a node $mux_2(a, x, y)$, and this node is the only successor of $y$, then we could simplify $y$ using the assumption that $a = 1$, since $y$ is only relevant in this case.

For this analysis, we define for each node $e$ in the DAG a set $\kappa(e)$ of facts which can be assumed to be true when optimizing this node. The facts are of the form $(a = n)$, where $a$ is a node in the DAG, and $n$ is an integer. The algorithm computes the set of facts for each node and simplifies nodes based on their collected facts, all in a single backward traversal of the DAG.

The algorithm begins by initializing the set of facts for each node $e$ to $\kappa(e) = U$, where $U$ is the set of all facts. The set of facts of the output node *out* is initialized to the empt set $\kappa(out) = \{\}$. The algorithm traverses the DAG backwards, updating $\kappa$ and replacing nodes according to the rules in Table 5.3.

$$\frac{\text{preconditions}}{\langle e, \kappa \rangle \rightarrow \langle e', \kappa_{updated} \rangle}$$

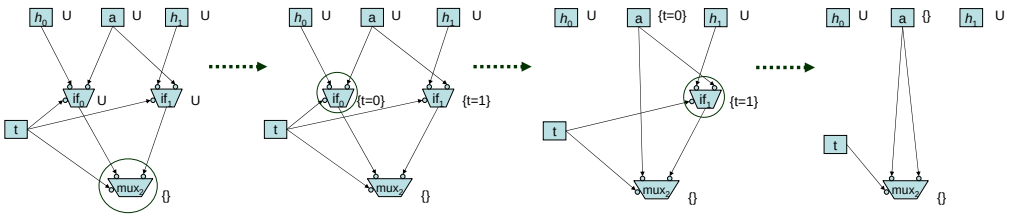
Each rule has a set of preconditions, and describes how the node $e$ is transformed into a node $e'$, and how $\kappa$ is updated with new information.

The intuition behind these rules is that for a node $e$, $\kappa(e)$ is the intersection of the facts provided by each of $e$'s children. Most expressions are processed by the default rule, which says that the facts that the expression contributes to its parents are simply its own facts, and the expression itself remains unchanged. Some expressions can contribute additional information to their parents; for example, an expression $e = \wedge(x, y)$ can contribute to its parent $x$ the fact that $y = 1$, because if $y = 0$, the value of $x$ does not matter to the node $\wedge(x, y)$. This is expressed in the rules by updating $\kappa(x)$ to $\kappa(x) \cap (\kappa(e) \cup \{y = 1\})$. Additionally, the nodes may be simplified or eliminated based on their facts. For example, if $e = \wedge(x, y)$, and $\kappa(e)$ contains the fact $\{y = 0\}$, then the node can be replaced with the constant 0.

The sequence below illustrates this process on a simple DAG. The DAG is what the system will generate for the following sequence of array operations.

```
int[2] x = ??;
x[t] = a;
out = x[t];
```

Notice that the assignment to `x[t]` is handled as a conditional assignment to each element of `x` by using the $if_0$ and $if_1$ nodes. However, the optimization discovers that these nodes are redundant because the value of `x[0]` is only relevant if $t$ is 0, so it is fine assume that `x[0] = a` regardless of the value of $t$.




If the DAG is traversed backwards form the root as specified, the algorithm will be well defined; for each node, only one rule will match, and by the time node $e$ is reached, $\kappa(e)$ will not contain any contradicting facts, such as $y = t$ and $y = t'$ for $t \neq t'$. Because this analysis runs backwards, it can not be applied in the same pass as the previous analysis;

however, its running time is quite small in the context of the overall synthesis process. Both this optimization and the previous one are applying what amounts to abstract interpretation to the problem of simplifying systems of constraints; something that, to my knowledge, isn't done by either UCLID or STP.

**Canonicalization of Associative-Commutative Operations.** Structural hashing can eliminate common sub-expressions, but it handles associativity and commutativity very poorly. For example, structural hashing can recognize that $a \vee b$ and $b \vee a$ are equivalent, but it can not recognize the equivalence of $(a \vee b) \vee c$ and $(a \vee c) \vee b$. This optimization canonicalizes complex expressions involving the operators $\vee$ and $\wedge$, making it possible to recognize common subexpressions involving these operators.

This analysis can be applied to any operator that is associative, commutative, and idempotent (i.e. $x \vee x = x$), which is true only of $\vee$ and $\wedge$. The reason is that if an operation has these three characteristics, a complex expression involving this operation can be fully characterized by the set of its unique operands. For example, any expression involving only the operator $\vee$ and the variables $a$, $b$ and $c$ is equivalent to the expression $a \vee b \vee c$. For the rest of this explanation, I will assume we are dealing with $\vee$ nodes, but the exact same procedure is used to canonicalize $\wedge$ nodes.

As a first step, the algorithm computes for each node $\vee(x, y)$ a set $S(\vee(x, y))$, containing those nodes of type different from $\vee$ that flow to $\vee(x, y)$ through a sequence of $\vee$ nodes. This computation is straightforward; we can define it recursively through the following formulas: $S(\vee(x, y)) = ts_x \cup ts_y$ where

$$ts_x = \begin{cases} s(x) & \text{if } x = \vee(a, b) \text{ for some } a, b \\ \{x\} & \text{otherwise} \end{cases}$$

and same for $ts_y$

Because $\vee$ is idempotent, commutative and associative, nodes whose sets $s$ are equal can be considered equivalent and merged.

Once the algorithm has computed the $S$ set for all the $\vee$ nodes, and merged all nodes with identical $S$ sets, the algorithm proceeds by using dynamic programming to find shared subsets among the $S$ sets, and create $\vee$ expressions that maximize the amount of sharing of common subexpressions.

| Original Pattern | | Replacement |
|---|---|---|
| $\wedge(x,x)$ | $\rightarrow$ | $x$ |
| $\wedge(x,\neg x)$ | $\rightarrow$ | $0$ |
| $\wedge(c1,c2)$ | $\rightarrow$ | $c1 \wedge c2$ |
| $\wedge(0,x)$ | $\rightarrow$ | $0$ |
| $\wedge(x,\wedge(y,\neg x))$ | $\rightarrow$ | $0$ |
| $\vee(x,x)$ | $\rightarrow$ | $x$ |
| $\vee(x,\neg x)$ | $\rightarrow$ | $1$ |
| $\vee(c1,c2)$ | $\rightarrow$ | $c1 \vee c2$ |
| $\vee(1,x)$ | $\rightarrow$ | $1$ |
| $\vee(x,\vee(y,\neg x))$ | $\rightarrow$ | $1$ |
| $\oplus(x,x)$ | $\rightarrow$ | $0$ |
| $\oplus(x,\neg x)$ | $\rightarrow$ | $1$ |
| $\oplus(c1,c2)$ | $\rightarrow$ | $c1 \oplus c2$ |
| $\neg c1$ | $\rightarrow$ | $1 - c1$ |
| $\neg\neg x$ | $\rightarrow$ | $x$ |
| $+(c1,c2)$ | $\rightarrow$ | $c1 + c2$ |
| $+(x,0)$ | $\rightarrow$ | $x$ |
| $+(+(x,c1),c2)$ | $\rightarrow$ | $+(x, c1 + c2)$ |
| $+(x,-x)$ | $\rightarrow$ | $0$ |
| $+(+(x,y),-x)$ | $\rightarrow$ | $y$ |
| $*(c1,c2)$ | $\rightarrow$ | $c1 * c2$ |
| $*(x,0)$ | $\rightarrow$ | $0$ |
| $*(x,1)$ | $\rightarrow$ | $x$ |
| $div(c1,c2)$ | $\rightarrow$ | $c1/c2$ |
| $div(mod(x,y),y)$ | $\rightarrow$ | $0$ |
| $mod(c1,c2)$ | $\rightarrow$ | $c1\ mod\ c2$ |
| $mod(mod(x,y),y)$ | $\rightarrow$ | $mod(x,y)$ |
| $-(c1)$ | $\rightarrow$ | $-c1$ |
| $-(-(x))$ | $\rightarrow$ | $x$ |
| $>(c1,c2)$ | $\rightarrow$ | $c1 > c2$ |
| $>(x,x)$ | $\rightarrow$ | $0$ |
| $>(+(x,y),+(x,z))$ | $\rightarrow$ | $>(y,z)$ |
| $>(+(x,y),x)$ | $\rightarrow$ | $>(y,0)$ |

| Original Pattern | | Replacement |
|---|---|---|
| $\geq(c1,c2)$ | $\rightarrow$ | $c1 \geq c2$ |
| $\geq(x,x)$ | $\rightarrow$ | $1$ |
| $\geq(+(x,y),+(x,z))$ | $\rightarrow$ | $\geq(y,z)$ |
| $\geq(+(x,y),x)$ | $\rightarrow$ | $\geq(y,0)$ |
| $<(c1,c2)$ | $\rightarrow$ | $c1 < c2$ |
| $<(x,x)$ | $\rightarrow$ | $0$ |
| $<(+(x,y),+(x,z))$ | $\rightarrow$ | $<(y,z)$ |
| $<(+(x,y),x)$ | $\rightarrow$ | $<(y,0)$ |
| $\leq(c1,c2)$ | $\rightarrow$ | $c1 \leq c2$ |
| $\leq(x,x)$ | $\rightarrow$ | $1$ |
| $\leq(+(x,y),+(x,z))$ | $\rightarrow$ | $\leq(y,z)$ |
| $\leq(+(x,y),x)$ | $\rightarrow$ | $\leq(y,0)$ |
| $=(c1,c2)$ | $\rightarrow$ | $c1 = c2$ |
| $=(x,x)$ | $\rightarrow$ | $1$ |
| $=(+(x,y),+(x,z))$ | $\rightarrow$ | $=(y,z)$ |
| $=(+(x,y),x)$ | $\rightarrow$ | $=(y,0)$ |
| $=(x,y),x,y:\mathbb{B}$ | $\rightarrow$ | $\oplus(x,y)$ |
| $mux_N(c1,x_1,\ldots,x_N)$ | $\rightarrow$ | $x_{c1}$ |
| $mux_N(y,x,x,\ldots,x)$ | $\rightarrow$ | $mux_2(<(y,N),0,x)$ |
| $mux_2(x,1,0),x:\mathbb{B}$ | $\rightarrow$ | $x$ |
| $mux_2(x,0,1),x:\mathbb{B}$ | $\rightarrow$ | $\neg x$ |
| $mux_2(x,0,y),x,y:\mathbb{B}$ | $\rightarrow$ | $\wedge(x,y)$ |
| $mux_2(x,y,1),x,y:\mathbb{B}$ | $\rightarrow$ | $\vee(x,y)$ |
| $mux_2(x,x,y),x,y:\mathbb{B}$ | $\rightarrow$ | $\wedge(x,y)$ |
| $mux_2(x,\neg x,y),x,y:\mathbb{B}$ | $\rightarrow$ | $\vee(\neg x,y)$ |
| $mux_2(=(x,c1),y,z)$ | $\rightarrow$ | $if_{c1}(x,z,y)$ |
| $mux_2(x,y,mux_2(a,y,c))$ | $\rightarrow$ | $mux_2(\wedge(x,a),y,c)$ |
| $mux_2(x,y,mux_2(x,b,c))$ | $\rightarrow$ | $mux_2(x,y,c)$ |
| $if_{c1}(c1,x,y)$ | $\rightarrow$ | $x$ |
| $if_{c1}(c2,x,y)c2 \neq c1$ | $\rightarrow$ | $y$ |
| $if_{c1}(x,y,y)$ | $\rightarrow$ | $y$ |

The rules use $c1$ and $c2$ to refer to constant nodes. On the right hand side, an operation on two constants, such as $c1 + c2$, corresponds to the integer node with the resulting value.

Table 5.2: Rewrite rules for circuit optimization

Multiplexer rules:

$$\frac{e = mux_N(a, x_1, \ldots, x_N) \;\; (a = t) \in \kappa(e)}{\langle mux_N(a, x_1, \ldots, x_N), \kappa\rangle \rightarrow \langle x_t, \kappa[x_t \mapsto \kappa(x_t) \cap \kappa(e)]\rangle}$$

$$\frac{e = mux_N(a, x_1, \ldots, x_N) \;\; \forall_t (a = t) \notin \kappa(e)}{\langle mux_N(a, x_1, \ldots, x_N), \kappa\rangle \rightarrow \langle e, \kappa[a \mapsto \kappa(a) \cap \kappa(e), \forall_{1 \leq i \leq N} x_i \mapsto \kappa(x_i) \cap (\kappa(e) \cup \{a = i\}]\rangle}$$

$if_c$ rules:

$$\frac{e = if_c(a, x, y) \quad (a = t) \in \kappa(e) \quad t \neq c}{\langle if_c(a, x, y), \kappa\rangle \rightarrow \langle x, \kappa[x \mapsto \kappa(x) \cap \kappa(e)]\rangle} \qquad \frac{e = if_c(a, x, y) \;\; (a = c) \in \kappa(e)}{\langle if_c(a, x, y), \kappa\rangle \rightarrow \langle y, \kappa[y \mapsto \kappa(y) \cap \kappa(e)]\rangle}$$

$$\frac{e = if_c(a, x, y) \;\; \forall_t (a = t) \notin \kappa(e)}{\langle if_c(a, x, y), \kappa\rangle \rightarrow \langle e, \kappa[a \mapsto \kappa(a) \cap \kappa(e), y \mapsto \kappa(y) \cap (\kappa(e) \cup \{a = c\}), x \mapsto \kappa(x) \cap \kappa(e)]\rangle}$$

Disjunction rules:

$$\frac{e = \vee(x, y) \;\; (x = 1) \in \kappa(e) \vee (y = 1) \in \kappa(e)}{\langle \vee(x, y), \kappa\rangle \rightarrow \langle 1, \kappa\rangle} \qquad \frac{e = \vee(x, y) \;\; (x = 0) \in \kappa(e) \wedge (y = 0) \in \kappa(e)}{\langle \vee(x, y), \kappa\rangle \rightarrow \langle 0, \kappa\rangle}$$

$$\frac{e = \vee(x, y) \;\; (y = 0) \in \kappa(e) \wedge (x = t) \notin \kappa(e)}{\langle \vee(x, y), \kappa\rangle \rightarrow \langle x, \kappa[x \mapsto \kappa(x) \cap \kappa(e)]\rangle}$$

$$\frac{e = \vee(x, y) \;\; \forall_{t_1, t_2} (y = t_1), (x = t_2) \notin \kappa(e)}{\langle \vee(x, y), \kappa\rangle \rightarrow \langle \vee(x, y), \kappa[y \mapsto \kappa(y) \cap (\kappa(e) \cup \{x = 0\}), x \mapsto \kappa(x) \cap (\kappa(e) \cup \{y = 0\})]\rangle}$$

Conjunction rules:

$$\frac{e = \wedge(x, y) \;\; (x = 0) \in \kappa(e) \vee (y = 0) \in \kappa(e)}{\langle \wedge(x, y), \kappa\rangle \rightarrow \langle 0, \kappa\rangle} \qquad \frac{e = \wedge(x, y) \;\; (x = 1) \in \kappa(e) \wedge (y = 1) \in \kappa(e)}{\langle \wedge(x, y), \kappa\rangle \rightarrow \langle 1, \kappa\rangle}$$

$$\frac{e = \wedge(x, y) \;\; (y = 1) \in \kappa(e) \wedge (x = t) \notin \kappa(e)}{\langle \wedge(x, y), \kappa\rangle \rightarrow \langle x, \kappa[x \mapsto \kappa(x) \cap \kappa(e)]\rangle}$$

$$\frac{e = \wedge(x, y) \;\; \forall_{t_1, t_2} (y = t_1), (x = t_2) \notin \kappa(e)}{\langle \wedge(x, y), \kappa\rangle \rightarrow \langle \wedge(x, y), \kappa[y \mapsto \kappa(y) \cap (\kappa(e) \cup \{x = 1\}), x \mapsto \kappa(x) \cap (\kappa(e) \cup \{y = 1\})]\rangle}$$

default rule:

$$\frac{e = \circ(x, y)}{\langle \circ(x, y), \kappa\rangle \rightarrow \langle \circ(x, y), \kappa[x \mapsto \kappa(x) \cap \kappa(e), y \mapsto \kappa(y) \cap \kappa(e)]\rangle}$$

Table 5.3: Update rules for backward analysis

## 5.3 Translation to SAT

The sketch synthesizer solves the constraint systems generated through the above process by translation to SAT. The translation proceeds in two steps: the DAG is first expanded into a boolean circuit, which is then translated into a SAT problem. I will first describe the translation into a boolean circuit, and then discuss the translation from circuit to a SAT problem.

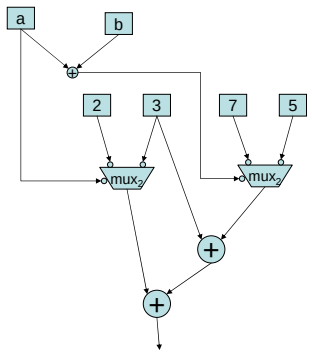
### 5.3.1 From DAG to boolean circuit

The translation into a circuit involves primarily the expansion of all arithmetic, comparison and selection expressions into boolean expressions. A common approach to this problem, employed by CMBC and Saturn is to use some variation of bit-blasting by representing integers as bit-vectors and encoding each non-boolean function through its boolean representation as a circuit that implements the operation.

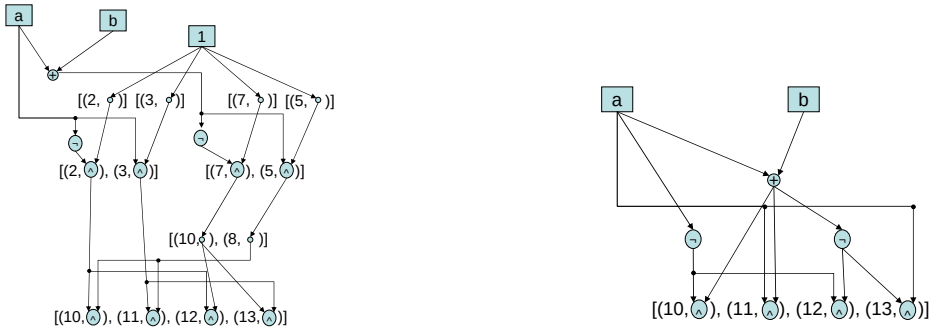
For SKETCH we opted instead for a sparse encoding which represents integers as a set of guarded values of the form $(v, b)$, where $v$ is an integer constant and the *guard* $b$ is a boolean function. If $b$ is true, then the integer has value $v$. We maintain an invariant that for all inputs at most one guard will be true. Similarly, we maintain uniqueness of the guarded values by taking a disjunction of terms guarding the same value, *i.e.* if after some operation the representation for a number becomes $[(4, b_1), (3, b_2), (4, b_3)]$, this will be represented as $[(4, b_1 \vee b_3), (3, b_2)]$. This encoding is very similar to the one used by Saturn [71] to encode pointers, but we use it to encode all integers.

Before describing formally how the sparse representation is used to generate boolean circuits from arithmetic expressions, I will illustrate the logic behind this representation with an example.

```
if(a ⊕ b)
        x = 5;
else
        x = 7;
if(a)
        y = 3;
else
        y = 2;
t = x + 3;
v = t + y;
```




For this fragment, the sparse representation for x in terms of the input variables $a$ and $b$ is $x = \{(5, a \oplus b), (7, \neg(a \oplus b))\}$. Adding 3 to $x$ produces the new value $t = \{(8, a \oplus b), (10, \neg(a \oplus b))\}$; note that in the sparse representation, the boolean formulas are unaffected by addition or subtraction of integers. The figure below shows the resulting boolean circuit for the fragment of code above before and after constant propagation.




A bit-vector representation for this snippet of code would require each bit of $x$ to be described in terms of $a$ and $b$, and the representation of $x + 3$ would require a bit-vector addition. This is not to say that the sparse representation is always better than the bit-vector representation; it is not. In particular, the representation is inefficient for integers that may range over a large number of values. This may happen either when we want to consider a very large range of values for inputs, or when the sketch contains a lot of multiplications, which cause a severe growth in the size of the sparse representation. Beyond the efficiency considerations, however, the most important quality of this representation is the ease of implementation. Adding support for complex operations such as multiplication, division and reminder is trivial compared with having to encode the bit-vector representations of these operations.

The circuits for the various integer operations are constructed using the sparse representation through a syntax directed translation. For a constant expression $n$, the sparse representation uses a single guarded value with a guard equal to 1:

$$\frac{}{n : \texttt{spar} \rightarrow [(n, 1)]}$$

Arithmetic expressions are handled by applying the arithmetic operation on pairs of values from the two operands and guarding them with the conjunction of their respective conditions.

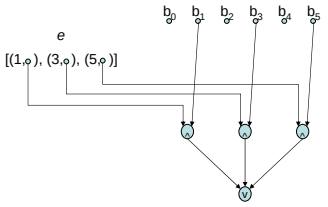
$$\frac{\begin{array}{c} e_1 : \texttt{spar} \rightarrow [(v_1^1, b_1^1), \ldots (v_k^1, b_k^1)] \\ e_2 : \texttt{spar} \rightarrow [(v_1^2, b_1^2), \ldots (v_l^2, b_l^2)] \end{array}}{\circ(e_1, e_2) : \texttt{spar} \rightarrow [(v_i^1 \circ v_j^2, b_i^1 \wedge b_j^2) \mid 1 \leq i \leq k, 1 \leq j \leq l]}$$

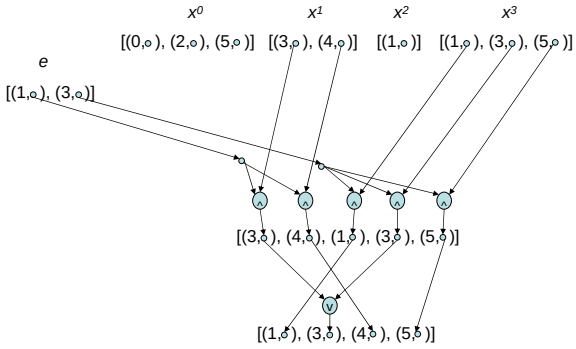
The multiplexer function may take integer or bit inputs, but its selector input must always be an integer, so it will be converted according to one of the following formulas.

$$\frac{\langle e : \texttt{spar} \rangle \rightarrow [(x_0, f_0), \ldots (x_s, f_s)] \;\; b_i : \texttt{bit}}{\langle mux_n(e, b_0, \ldots, b_k) : \texttt{bit} \rangle \rightarrow (\bigvee_{0 \leq j \leq s} (b_{x_j} \wedge f_j))}$$

$$\frac{\langle e : \texttt{spar} \rangle \rightarrow [(e_0, f_0), \ldots (e_s, f_s)] \;\; \langle x^i : \texttt{spar} \rangle \rightarrow [(v_0^i, b_0^i), \ldots, (v_{l_i}^i, b_{l_i}^i)]}{\langle mux_n(e, x^0, \ldots, x^k) : \texttt{spar} \rangle \rightarrow [(v_i^{e_j}, b_i^{e_j} \wedge f_j) \mid 0 \leq j \leq s, 0 \leq i \leq l_{e_j}]}$$

Below are two sample circuits generated by the above rules; the one on the left corresponds to bit inputs, while the one on the right corresponds to integer inputs.




As the boolean circuit for the constraints is derived, the synthesizer performs constant propagation before converting the resulting boolean constraint into a SAT problem.

### 5.3.2 From boolean circuit to SAT

The SKETCH synthesizer has two different options for translating the boolean representation into a SAT problem. The first method is through direct conversion of the boolean circuit into CNF clauses. The second method actually constructs the boolean circuit and feeds it to a boolean circuit analysis tool called ABC [49]. The first approach is to convert each node in the boolean circuit into a small set of CNF clauses by using standard conversion formulas. When following this approach, the synthesizer never builds a representation of the boolean circuit as such. Instead, it directly generates the CNF clauses as it traverses the original DAG to perform the conversion described in the previous section. This has the advantage of simplicity and avoiding additional overhead of an extra translation step.

In the second approach, the synthesizer explicitly builds a graph representation of the boolean circuit, and feed it to a circuit analysis tool called ABC [49]. ABC uses an and-inverter-graph (AIG) representation of the circuit to efficiently analyze and optimize it. Furthermore, for large circuits it is sometimes able to break the equivalence checking problem into smaller SAT problems that it's able to solve more quickly. The main disadvantage of this approach is the additional memory overhead of ABC compared with direct translation to SAT, which can be significant for larger benchmarks.

This translation completes the process of expressing formulas in the synthesis semantics as boolean constraint systems suitable for solution with a SAT solver. The process is very general; it is able to handle complex sketches ranging from low-level bit twiddling to difficult data-structure manipulations, as the following section will illustrate. The price paid the power and generality of this approach is the boundedness constraint; the need to statically unroll loops, and the requirement that all integers be constrained to a small range of values. This is a major tradeoff, but if the history of model checking over the last ten years is any guide, this looks like a tradeoff worth making.

# Chapter 6

# Empirical Evaluation

The primary goal of this chapter is to provide some insight into the scope of sketching problems the synthesizer is currently able to solve, and the time and memory resources it uses in order to solve them. The chapter also compares the performance of the synthesizer with off-the-shelf solvers for quantified boolean formulas, as well as the effect on the solution time of changes to a few properties of the sketch, such as the number of holes, or the size of the space of inputs. Finally, the section analyzes the benefits of the optimizations described in Section 5.2 and Section 5.3.

The experiments in this chapter were all performed on ThinkPad laptop with a single core Intel T1300 at 1.66GHz with 2MB of L2 cache and 1GB of memory. All the performance numbers in this section are averages from 4 to 7 different executions using different random seeds for the initial counterexample and the random restart in the SAT solver. The experiments used the 2004 version of MiniSat and version 60513 of ABC. The following are the highlights of the evaluation.

- **SKETCH Scales to real problems** The performance of the synthesizer is shown for several variations of 18 representative benchmarks from a variety of domains, all of which resolve in less than 15 minutes and use less than 250 MB of memory. A complete implementation of AES was also produced with SKETCH; it had a candidate space of the order of 32K elements but took only about an hour to resolve.

- **The SKETCH language can naturally express programmer insight** The section uses examples from different domains to show how the insight behind a tricky algorithm can be succinctly expressed in a sketch.

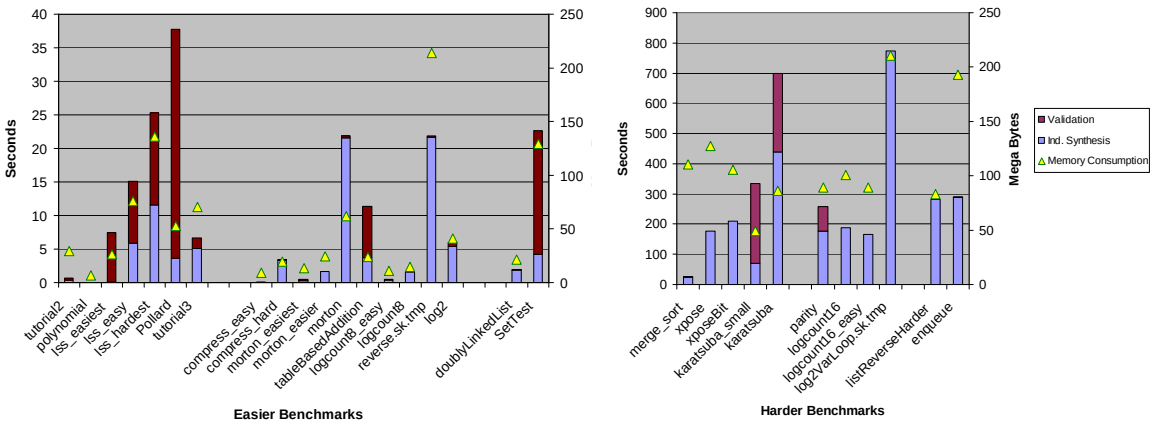



Figure 6.1: Solution time and memory consumption for selected benchmarks.

- **Additional input from the programmer makes synthesis faster** The section shows how on different benchmarks, small amounts of extra information from the programmer can improve the solution time by orders of magnitude.

- **Optimizations can greatly improve performance** A couple of benchmarks saw performance improvements of 10x and 33x from using ABC as the boolean solver, but the improvement was not uniform. Some saw big performance degradation, so it is important to pick the correct solver. The high-level optimization had smaller effects, but still were able improve the solution time by an additional factor of 6 for some benchmarks.

- **For sketching problems, CEGIS is better than general QBF solvers** Even the best QBF solver was unable to solve problems which the SKETCH synthesizer solves in a couple of minutes.

## 6.1 Performance of Selected Benchmarks

Figure 6.1 shows the solution times for several variations of 18 representative benchmarks. The benchmarks are mostly real programming problems which the synthesizer can solve in less than 15 minutes. Many of these benchmarks were written by our group, but a handful of them were developed by students from a graduate introductory programming languages class held at UC Berkeley in the fall of 2007. The benchmarks can be roughly categorized into three groups: bit manipulations, integer manipulations and linked data structures.


Figure 6.2: Examples of log-shifting for scattering and gathering bits.

**Bit manipulations** What characterizes these benchmarks is that they treat machine words as bit-vectors. All these benchmarks use the **`implements`** directive (Section 2.4.2) to provide specifications in the form of reference implementations that manipulate each bit individually. The sketches contain the necessary insight to take advantage of bit-level parallelism. These benchmarks were our first application of sketching because the low-level details almost always involve discovering bit-masks and precise shift amounts, so we could write sketches for them using only the integer hole, even before we had any of the higher level sketching constructs.

**Example** A typical benchmark of this category is the `morton` benchmark, written by graduate student Jacob Burnim. A 32-bit morton number is computed by interleaving the bits of two 16 bit integers $x$ and $y$, so that bit $r_{2*i}$ of the result equals bit $x_i$ of $x$, and bit $r_{2*i+1}$ corresponds to bit $y_i$ of $y$. According to Anderson, "Morton numbers are useful for linearizing 2D integer coordinates, so x and y are combined into a single number that can be compared easily and has the property that a number is usually close to another if their x and y values are close" [3]. It's easy to interleave the bits of two 16-bit integers by selecting the bits one by one, but it's possible to do it more efficiently by taking advantage of the ability to shift all the bits in a word with a single instruction; while the bit-by-bit approach takes $O(W)$ operations for a word of size $W$, the task can be achieved with $O(log(W))$ operations by using bit-vector parallelism. The high-level insight can be stated as follows.

> First, scatter the 16 bits of each of the two inputs across the even bits of a 32 bit word. Then, `or` together the resulting two words, shifting one of the words by one to align its bits with the gaps in the other word. The scatter can be done with log-shifting, a technique for efficiently scattering or gathering bits by shifting many bits at a time as illustrated in Figure 6.2. A logshifter can be implemented by repeatedly shifting some bits, `or`ing them with the original word, and then masking the result.

The insight can be expressed succinctly in a sketch. The `logshift` generator encapsulates the basics of logshifting, but leaves unspecified the tricky details of exactly what bits to mask and how much to shift on each step; this means that the generator could actually be reused to implement other scattering patterns different from the one required for this problem. The sketch also leaves unspecified the number of steps required for the logshifter; this is a potential problem because it gives the synthesizer the freedom to include more steps than necessary. The student produced a second version of this benchmark (`morton_easiest`) that specifies that on each iteration the shift amount should be reduced by half. This version of the benchmark is guaranteed to produce the desired answer, and resolves much faster because of the added information.

```
int W = 16;
generator bit[2*W] logshift(bit[2*W] in){
   int pt = 4*W;
  repeat(??) {
    // Shift some of the bits, and mask out their original positions.
    in = (in | (in << ??)) & ??;
  }
}

bit[2*W] morton(bit[W] x, bit[W] y) implements mortonSpec{
  bit[2*W] x2 = logshift(x);
  bit[2*W] y2 = logshift(y);
  return x2 | (y2 << 1);
}
```

□

All the benchmarks in the second group in Figure 6.1 are bit manipulation benchmarks. These benchmarks are difficult to solve despite their relatively small size (compress is the largest one of these sketches and it's only 47 lines of code). There are two reasons for this. First, their candidate spaces are often huge; a single 32-bit mask will have billions of possible solutions. Moreover, the holes are often very tightly coupled, in the sense that every bit in the output potentially depends on the value of every single hole, as was the case in the `morton` example. This makes these benchmarks very challenging for the solver. On the other hand, they are a great domain for sketching because it is very challenging to program by hand, and there is often a very good match between the insight and the sketch. Moreover, because these benchmarks are inherently bounded, the SAT-based validation procedure can provide absolute correctness guarantees.

**Integer manipulations.** These benchmarks manipulate integers or arrays of integers; with the manipulations typically involving some arithmetic. Their specifications also consist of reference implementations, while the sketches often must take advantage of some mathematical principle to achieve better performance at the expense of clarity. All the benchmarks in the first group in Figure 6.1 are integer manipulation benchmarks.

**Example** A great example from this domain is the Karatsuba multiplication algorithm for large integers (`karatsuba`). The algorithm is a building block of many public key cipher implementations. It uses a divide and conquer approach to multiply integers with $N$ digits in $O(N^{1.585})$, as opposed to the standard $O(N^2)$ from the grade school multiplication algorithm.

The algorithm starts by decomposing two $N$-digit numbers $x$ and $y$ into two halfs: $x = x_1 b^{N/2} + x_0$, $y = y_1 b^{N/2} + y_0$, where $b$ is the base. The standard multiplication can be defined recursively in terms of the two halfs.

$$x * y = b^N x_1 * y_1 + b^{N/2}(x_1 * y_0 + x_0 * y_1) + x_0 * y_0$$

The expensive (big-integer) multiplication is denoted with the $*$ operator. The multiplication with the base terms is implemented with shifts, so it is not an expensive operation.

Let us illustrate how Karatsuba might have been able to invent (and implement) his algorithm with the assistance of sketching. He would first observe that it may be possible to replace the four expensive multiplications with three expensive multiplications. He would guess that one cannot avoid computing terms $x_0 * y_0$ and $x_1 * y_1$, so he would focus on replacing the term $x_1 * y_0 + x_0 * y_1$ with a one-multiplication term. This optimization would be performed at the expense of adding big-integer additions or subtractions, a good trade-off since their complexity is linear rather than quadratic. In mathematical notation, the idea can be expressed in the following sketch, where the generator $poly(n, x_1, \ldots, x_k)$ produces a polynomial in $k$ variables of degree $n$.

$$\begin{aligned} x * y = {} & poly(??, b) * (x_1 * y_1) \\ & + poly(??, b) * (poly(1, x_1, x_0, y_1, y_0) * poly(1, x_1, x_0, y_1, y_0)) \\ & + poly(??, b) * (x_0 * y_0) \end{aligned}$$

```
int[N*2] k (bit[N] x, bit[N] y) implements mult {
   if (N<=1) return x*y;
        int[N/2] x1, x2, y1, y2;
        int[N]  a=0, b=0, c=0;
        int[N*2] out = 0;

        x1=x[0::N/2];  x2=x[N/2::N/2];
        y1=y[0::N/2];  y2=y[N/2::N/2];

        a = multHalf(x1, y1); //perform recursive multiplications
        b = multHalf(x2, y2);
        c = multHalf(poly1(x1,x2,y1,y2), poly1(x1,x2,y1,y2));

        out = a;
        out = plus(out, shift(b, No2));
        loop(??){
            int[N] t = {| a | b |  c |};
            // shift is equivalent to multiplication by a power of the base.
            out = plus( out, shift( {| t | minus(t) |}, {| N | N/2 | 0 |} )  );
        }
        out = normalize(out);
        return out;
}

int[No4] poly1(int[No4] a, int[No4] b, int[No4] c, int [No4] d){
        int[No4] out = 0;
        if(??)  out = plus(out, {| a | minus(a)|} );
        if(??)  out = plus(out, {| b | minus(b)|} );
        if(??)  out = plus(out, {| c | minus(c)|} );
        if(??)  out = plus(out, {| d | minus(d)|} );

        return out;
}
```

Figure 6.3: Sketch for Karatsuba's multiplication.

It turns out that the idea for this optimization is correct; the correct formula is shown below.

$$\begin{aligned} x * y = & (b^2 + b) * (x_1 * y_1) \\ & + b * ((x_1 - x_0) * (y_1 - y_0)) \\ & + (b + 1) * (x_0 * y_0) \end{aligned}$$

Creating an implementation using the SKETCH system is just as simple. The sketch in Figure 6.3 contains the same insight expressed above in more mathematical notation, but it also addresses the representation issues for the integers and their operations. Integers are represented as $N$ element arrays of **int**s; addition, complement and shifting are all provided through separate routines. The half ranges are read from the original input array using special array notation available in the language, where `A[a::b]` correspond to a range of `b` elements in `A` starting with the element at position `a`. Multiplications by the base term are encoded through a shift operation.

Ideally, we would like for the routine to be parametrized by $N$, and the solver to guarantee the result for all $N$. Unfortunately, the SKETCH solver can not reason about unbounded operations, so the correct answer was derived by setting $N$ to 4, and limiting the range of integer values to two bits. □

The `karatsuba` benchmark illustrates many relevant aspects of integer benchmarks. First, because we use bounded model checking as our correctness criteria, we can not provide strong correctness guarantees. For most of these benchmarks, validation was performed for all integer inputs in the range $[0, 8]$. Only the tutorial benchmarks were validated for inputs in the range $[0, 32]$. For these benchmarks these ranges happened to be sufficient in the sense that the programs that were correct for all inputs between 0 and 8 turned out to be correct programs, but this could only be ascertained through hand examination of the result.

The ranges of inputs are fairly small even by the standards of bounded model checking. This is partly a consequence of the use of the guarded value representation of integers described in Section 5.3. This representation is very efficient when representing integers ranging over a small set of values, but it grows very quickly, making it impractical to validate sketches over a wide range of input values. It is very likely that the growing power and availability of SMT solvers capable of reasoning about integers will have a big impact on these benchmarks. In spite of this, the synthesizer is able to quickly produce correct implementations from sketches with a lot of freedom for many interesting kernels.

**Linked data structures** These benchmarks involve manipulation of data structures in the heap. The linked list reversal from the introduction is an example of this class of benchmark.

**Example** Another interesting benchmark in this category is the `SetTest` benchmark. This benchmark implements a tree-based set using a hash table as a reference implementation. One of the problems that make tree manipulation tricky is symmetry: the code for the different cases is very similar except some cases have to use the left child and some have to use the right child, and it is easy to get confused about which child should be used where. Sketching allowed us to eliminate this redundancy by using generators. The fragment of the `SetTest` sketch shown below uses a generator to produce the code that decides whether to add a new node as a child of the current node or to continue traversing.

```
bit add(Tree t, int v){
        TreeNode n =t.root;
        if(n == null){
                t.root = newTreeNode(v);
                return ??;
        }
        while(n != null){
                if( n.val == v){ return ??; }
                if(n.val < v){
                        if(choice(n, v)){
                                return ??;
                        }
                }else{
                        if(choice(n, v)){
                                return ??;
                        }
                }
        }
        return ??;
}
```

The generator will produce the correct code both for the case when `n.val` is less than `v` and when it is not.

```
generator bit choice(ref TreeNode n, int v){
        if({| n(.left | .right) |} == null){
                {| n(.left | .right) |} = newTreeNode(v);
                return ??;
        }else{
                n = {| n(.left | .right) |} ;
                return ??;
        }
}
```

□

Like the integer manipulation benchmarks, datastructure benchmarks also have to cope with the limitations of the validation procedure. Our validation procedure can not guarantee the absolute correctness of the synthesized implementation, only its correctness against a bounded test harness. For example, the test harness for the **enqueue** benchmark, shown below, enforces the equivalence of the sketched queue with an array implementation on an input directed sequence of operations.

```
void main(int[N] in, bit[N] ctrl) {
        int [N] qarray=0;
        int head = 0;
        int tail = 0;
        Queue q = new Queue();
        init(q);
        for(int i=0; i<N; ++i){
                if(ctrl[i]){ // decide to enqueue or dequeue based on the input.
                        enqueue(q, in[i]);
                        qarray[tail] = in[i]; //The same operation is applied
                        tail = tail+1;       //to the queue and to the array.
                }else{
                        bit tmp;
                        if(q.head != null){
                            tmp = dequeue(q);
                        }else{
                            tmp = -1;
                        }
                        if(head != tail){
                            assert tmp == qarray[head];
                            head = head + 1;
                        }else{ // Array queue is empty, sketched queue
                            assert tmp == -1; // should be empty too.
}       }       }       }
```

This type of test harness is often referred to in the verification literature as a "most general client" [2] because verifying that it works correctly for all possible inputs will prove

that the queue works correctly for all sequences of up to `N` operations, which is a very good, but it's not the same as verifying that the queue is correct.

Another interesting feature of the datastructure benchmarks, especially when compared with the bit manipulations, is that one can leave a great amount of code unspecified while keeping the search space relatively small. For example, in the `listReverseHard` benchmark, the assignments in the body of the loop specified remarkably little, leaving a lot of freedom to the synthesizer, but the synthesizer only had 60 different possibilities to search through for each assignment. By contrast, a single bit-mask in the `morton` benchmark can have $2^{32}$ different possible values. This means that sketches can be allowed to have a lot of freedom without overwhelming the synthesizer. At the same time, we can see that the solution times for these benchmarks can be quite large given their small input and candidate spaces, which seems to suggest that our very naïve representation of the heap as a set of arrays (Section 3.2.2) may have a lot of room for improvement.

## 6.2 Factors affecting performance of the SKETCH synthesizer

This section analyzes how the performance of the synthesizer is affected by the sketch. In particular, it looks at how the solution time is affected by features such as the number of holes, and bounds on the sizes of datastructures. The goal is to give the programmer some insight into how to cope with sketches that take too long to resolve.

### 6.2.1 Synthesis time Vs. Holes

We have argued before that sketching offers the programmer a very natural way to help the synthesizer. If the synthesizer is taking too long to resolve a sketch, the programmer can help by simply writing more of the code. The number of holes in a sketch has a strong impact on its solution time, so writing even a little more code can help the synthesizer find a solution much faster. Figure 6.4 illustrates precisely how much faster on a handful of benchmarks.

Overall, we observe that, reducing the number of holes reduces the synthesis time by orders of magnitude. This is a very intuitive result; reducing the number of holes reduces the search space that the synthesizer has to consider, and therefore the time it takes to search through it. Thus, if a sketch is taking too long in the synthesis phase, it may benefit from additional information that reduces the number of holes.

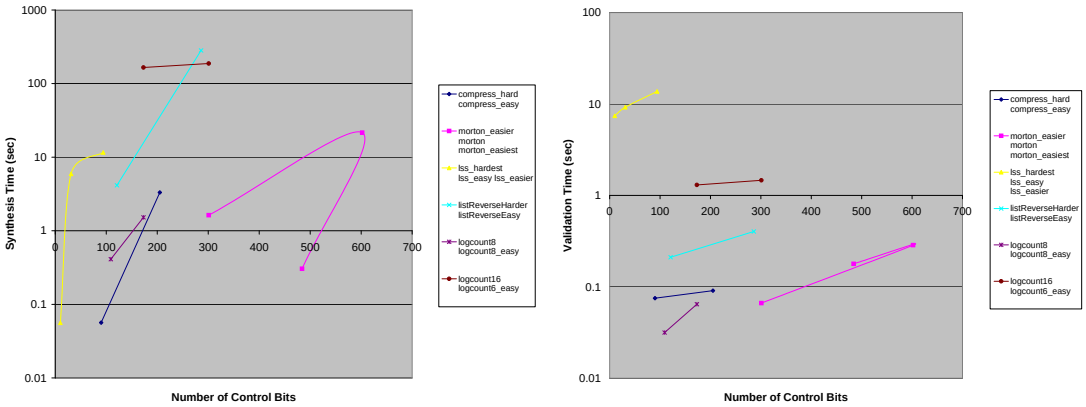



Figure 6.4: `parity`: Synthesis and Validation against number of control bits for specific sketches.

More surprisingly, reducing the number of holes also has a small effect on the validation time. This may appear surprising because the validation only sees concrete candidates, so it should not be affected by the size of the candidate space. However, recall that Section 4.3 observed that bigger candidate spaces often lead to more iterations of the CEGIS loop, which implies more calls to the validation procedure, so validation time increases even though the average time per validation stays roughly the same.

The differences in synthesis and validation times are more pronounced for some benchmarks than for others. For example, consider the two versions of `listReverse`. The harder one corresponds to the sketch from Section 1.2.1; the simpler one fully specifies the left hand side of each assignment while leaving everything else unchanged. The performance difference is dramatic. The more detailed sketch resolves in a matter of seconds, while the freer one takes several minutes.

The story for the `morton` benchmark is similar. The hardest version of `morton` corresponds to the sketch shown in the previous section. The intermediate version takes advantage of the fact that the `logshift` should synthesize to the same function for both `x` and `y`, so it uses a procedure instead of a generator, and asserts the correctness of the logshift procedure independently. The third version of `morton` uses a generator like the first one, but exposes in the sketch the insight that the shift amounts for the log shifter should decrease by half for every step. Just as before, additional information significantly reduces the solution time.

The `morton` benchmark is also interesting because it shows the effect of providing

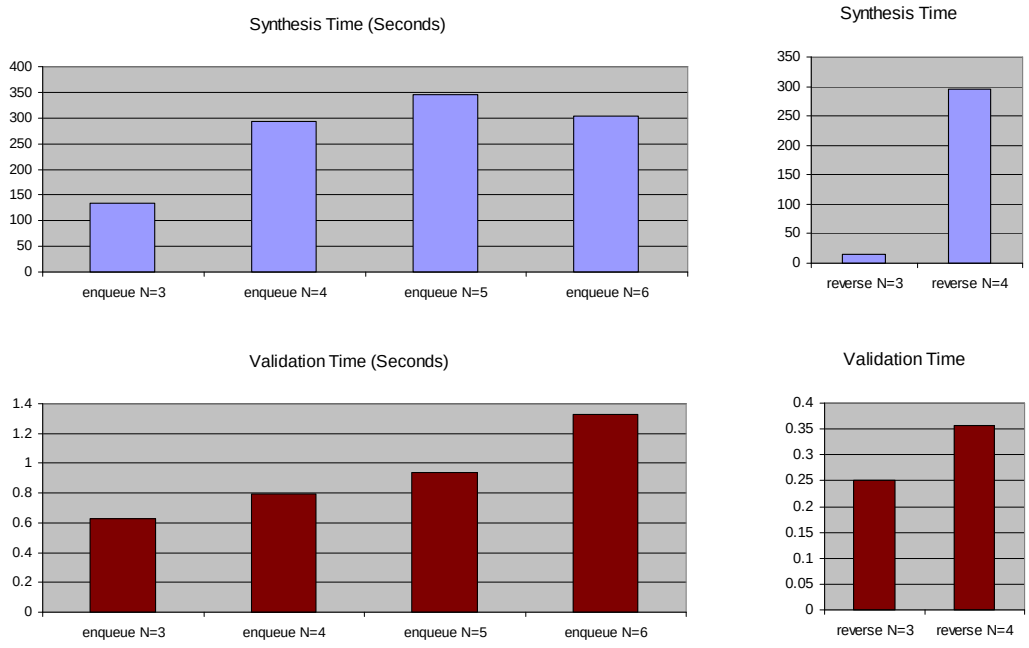



Figure 6.5: Solution time with varying test sizes.

different kinds of information. The first optimization reduced the number of bits of holes from 600 to 300, and this reduced the synthesis time from 21.5 to 1.6. On the other hand, the second optimization reduced the number of holes by less, from 600 to 484, but the effect on the synthesis time was much bigger, from 21.5 to 0.3 seconds. Apparently, knowing the shift amount was much more useful to the synthesizer than knowing that `logshift` is the same for both cases. This was probably due to the fact that knowing the shift amounts allowed it to infer the number of steps in the logshifter, which meant it had fewer bit-masks to search through. Overall, the lesson from these experiments is that providing a little more information in a sketch can usually have a big impact on the time it takes to resolve it.

### 6.2.2 Synthesis Time Vs. Test Size

Sketches of linked data structures use a bounded test harness to define correctness, so let us now explore the effect that the bounds in the test harness can have on the solution time for a sketch. Figure 6.5 shows the effect of the bound sizes on the solution time for a couple of the data structure sketches. As expected, changing these sizes has a big impact on the validation time. After all, the validator has to work harder to prove the correctness

of the solution over a bigger input space. As we can see, the effect is very clear; for the **enqueue** benchmark, for example, the validation time increases linearly with `N`, the number of operations performed by the test harness.

More surprisingly, changing the bounds of the test harness has a big impact on the synthesis time as well. This is surprising because the synthesizer works only with concrete inputs, so it should not be affected by the fact that these inputs came from a bigger pool of possible inputs. However, bigger bounds allow the validator to produce bigger counterexamples; in turn, bigger counterexamples mean it took more steps for the incorrect control values to cause an assertion failure, and therefore the constraint system for the synthesis problem is bigger and harder to solve. We have observed a similar effect with integer benchmarks; increasing the range of the integers doesn't just make validation harder, it makes synthesis harder.

In short, users should start by trying to synthesize their code with small bounds for their test harnesses; the generated code can always be checked with bigger bounds to get better assurance about its correctness. Doing this automatically in the synthesizer should help greatly with the performance of datastructure problems.

## 6.3 Analysis of the Optimizations

This section quantifies the performance benefit of the optimizations described in Section 5.2 and Section 5.3. In Section 5.3, we described how the SKETCH synthesizer can use the circuit analysis tool ABC to solve the boolean constraints, while Section 5.2 described a series of higher-level optimizations that are applied to the DAG representation of the constraints. The combination of boolean and higher-level optimizations raises three important questions which this section will try to answer.

1. How much performance is gained by the optimizations?
2. Do the optimizations benefit the benchmarks uniformly? And if not, is it easy to decide a priori what optimizations should be applied to a given benchmark?
3. How redundant are the higher-level optimizations with the boolean optimizations performed by ABC?

The first two questions will be answered independently for the two classes of optimizations in Section 6.3.1 and Section 6.3.2 respectively. Section 6.3.2 will address the third question by

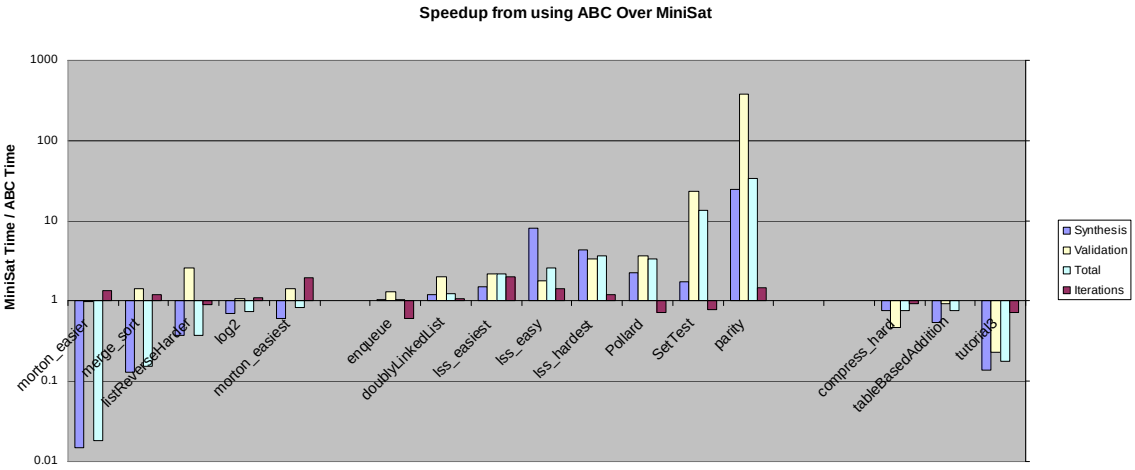



Figure 6.6: Effect of ABC on solver performance.

showing that the higher-level optimizations improve the solution time even when the boolean constraints are solved with ABC, suggesting that higher-level optimizations complement the boolean optimizations.

### 6.3.1 Effect of ABC

First, I will analyze the performance benefits of using ABC to solve the boolean satisfiability problem, as opposed to using MiniSat directly (see Section 5.3). The conclusion from this analysis is that ABC can offer dramatic performance improvements on some benchmarks (a factor of 33 for one benchmark), but it can hurt the performance of others, so it's very important to use the right solver. As a default rule for all our experiments so far, we used ABC for the linked datastructure benchmarks, and MiniSat for all the others. As this section will show, this is a fairly good heuristic, although it misses some optimization opportunities.

Figure 6.6 shows the effect on the solution time of using ABC instead of MiniSat. The $y$ axis shows the solution time for MiniSat divided by the solution time for ABC ($t_{mini}/t_{abc}$); a value greater than one means ABC improved the solution time, while a value less than one implies that using ABC made the synthesizer slower.

We can see in the graph that benchmarks fall into three categories. For a handful of benchmarks, ABC was uniformly good; in some cases amazingly so. For example, for the `parity` benchmark, the solution time went from 257 seconds to 7 seconds. `SetTest` similarly

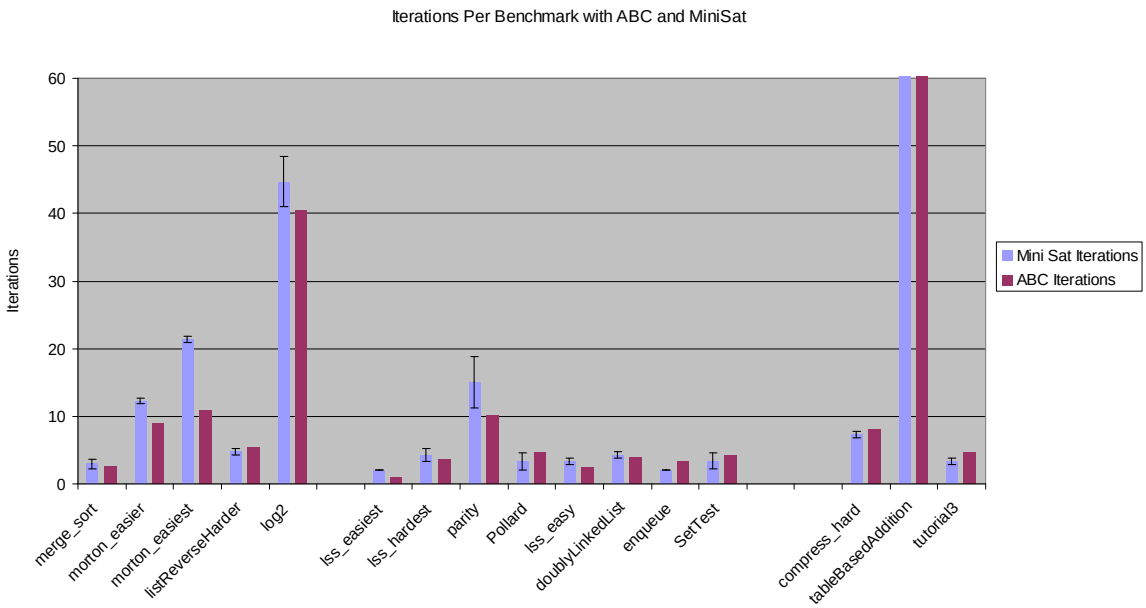



Figure 6.7: Effect of ABC on the number of iterations.

exhibited a 96% reduction in the solution time, going from 450 seconds to 33 seconds.

A second group of benchmarks, on the other hand, exhibited uniform performance degradation in the hands of ABC. The table shows a few of them, but overlooks another group of benchmarks which timed out or exhausted the available memory when using ABC. This group included mostly bit and integer manipulation benchmarks, including the `morton`, `xpose`, `karatsuba` and `logcount` benchmarks.

The most interesting is a third group, which exhibited enormous performance improvement on the validation phase, which was completely offset by a dramatic performance degradation on the synthesis phase. At first, it appears surprising that the synthesis and validation problems for the same benchmark could be so different that one benefits from ABC while the other is impacted so negatively.

The explanation for this anomaly can be found by noticing the statistically significant difference between the number of CEGIS iterations when using ABC vs. MiniSat, illustrated in Figure 6.7. The figure shows the average number of iterations of the CEGIS loop under both ABC and MiniSat. In the case of MiniSat, it also shows error bars of 1 standard deviation around the average. What jumps out of this graph is that for many

benchmarks, ABC caused a statistically significant drop in the number of iterations of the loop. This was particularly marked for those benchmarks that showed improvements in validation time but degradation in the synthesis time. As we shall see, this difference in the number of iterations will allow us to explain the anomaly.

The random choices made by the SAT solver can sometimes lead to variability in the number of iterations. For example, sometimes, the synthesizer "gets lucky" and chooses a correct candidate even when the inductive synthesis problem was not yet fully constrained and it could have picked an incorrect one. Sometimes, the validator choses counterexample inputs that expose more than one bug in a candidate; causing the CEGIS loop to converge faster. However, the systematic difference between the number of iterations for MiniSat and ABC is probably due to the way the two solvers work. MiniSat first tries to set a variables to zero, and only sets them to one if the zero value lead to a conflict. As a consequence, counterexamples tend to change little from one iteration to the next; they only change enough to expose a new bug. ABC, on the other hand, tends to produce counterexamples that look more random due to the extensive manipulations it performs on the boolean circuit. Therefore, counterexamples from ABC are more likely to expose more than one bug, and therefore lead to fewer CEGIS iterations. This, in turn is the key to why ABC causes some benchmarks to improve their validation time but degrade their synthesis time.

For a given benchmark, fewer CEGIS iterations usually translate to shorter solution times. For example, Figure 6.8 shows nine individual runs of the solver on the `logcount16_easy` benchmark using MiniSat. The $x$ axis is the iteration number, and the $y$ axis is the sum of the synthesis or verification time up to iteration $x$. Notice how the total validation time is nearly constant regardless of the number of iterations; moreover, the last validation usually takes longer than all the previous validation steps combined. The synthesis time, in turn, grows exponentially with the number of iterations; more iterations lead to longer solution times. In fact, most of the variability in the solution time for this benchmark can be attributed to variation in the number of CEGIS iterations. Therefore, we expect the difference in iterations between ABC and MiniSat to have an effect on the solution time as well.

Figure 6.9 shows detailed plots with aggregate times per iteration for `morton` and `log2`, two benchmarks where the synthesis time suffered as a result of using ABC. In the case of `morton`, we can see that on every iteration ABC is simply slower than MiniSat for both synthesis and validation. Validation only looks faster because the performance degradation

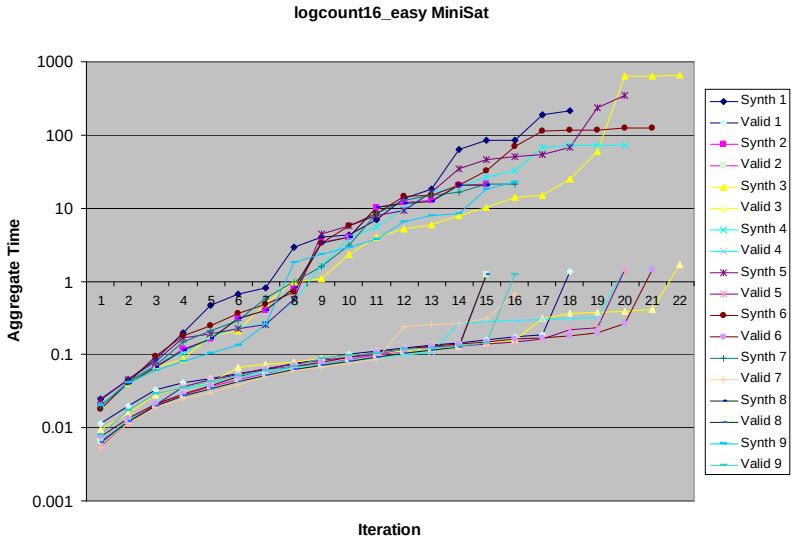



Figure 6.8: `logcount16_easy`: Synthesis and Validation time accross iterations.

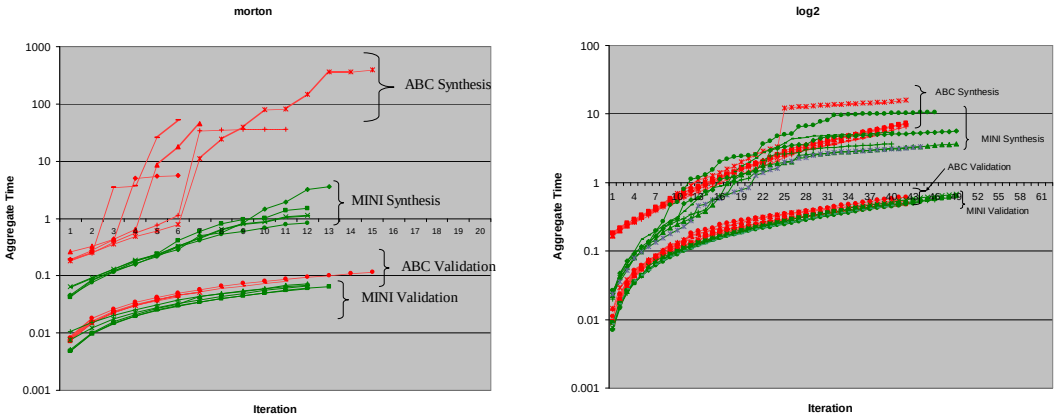



Figure 6.9: Synthesis and Validation time accross iterations for `morton` and `log2`.

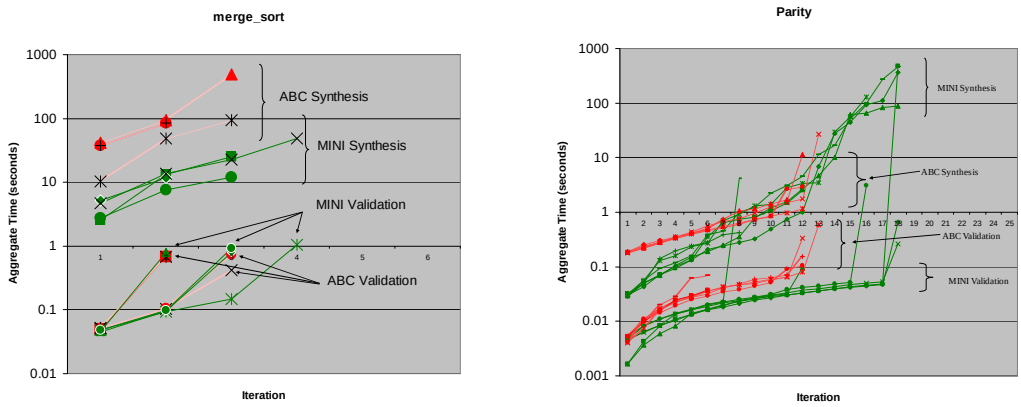



Figure 6.10: Synthesis and Validation time accross iterations for `merge_sort` and `parity`.

is offset by the smaller number of iterations. For `log2`, we can see something similar: ABC is mostly slower than MiniSat; the smaller validation time is merely an artifact of the smaller number of iterations. The `merge_sort` benchmark, shown in Figure 6.10 had fewer iterations, so the effect was less noticeable. In this benchmark, the validation time is virtually identical for both ABC and MiniSat, while the synthesis time shows a clear degradation when using ABC.

A similar plot was created to see the effect of ABC on `parity`, one of the benchmarks that shows enormous performance improvements with ABC. On the validation side, we can see that the biggest improvement comes from ABC's far superior performance on the final validation step, which is very hard on MiniSat. On the synthesis side, we can see that ABC starts slower, but scales much better than MiniSat; this combined with the smaller number of iterations leads to a dramatic reduction in the overall solution time. For the other benchmarks that benefit significantly from ABC, the results are more straightforward. For most of the linked datastructure benchmarks, ABC is simply much faster across the board on every iteration.

In summary, ABC produced significant performance improvements on many benchmarks, but the performance improvements were not uniform. Most linked datastructure benchmarks benefited from ABC, but a few integer and bit manipulation benchmarks suffered large performance degradations; this observation justifies the rule of using ABC for datastructure benchmarks and MiniSat for all others. Overall, though, ABC is a very useful tool to significantly reduce the solution time of some of the hardest benchmarks.

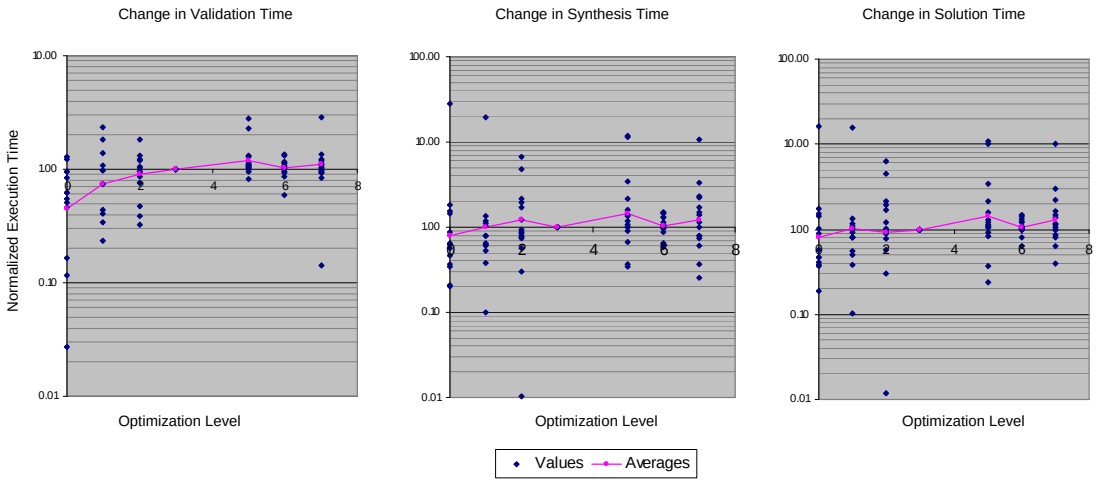



Figure 6.11: Effect of optimizations on the solution time for the benchmarks `morton`, `merge_sort`, `morton_easier`, `listReverseHarder`, `log2`, `lss_hardest`, `parity` and `SetTest`.

### 6.3.2 Effect of High-Level Optimizations

Section 5.2 described a series of optimizations that the synthesizer performs on the DAG representation of the problem. We refer to these as high-level optimizations to distinguish them from the low-level circuit optimizations performed by ABC. For this section we are again interested in how much performance is gained through these optimizations, and whether the benefits are distributed uniformly. Additionally, some of the optimizations performed at this stage, such as structural hashing, are very similar to optimizations applied by ABC, so it is important to know if there is any benefit in applying the two sets of optimizations, or if ABC just repeats the work done by the high-level optimizations.

To gauge the effect of the high-level optimizations from Section 5.2, I defined 7 levels of optimization[1].

- **Level 0** performs no optimization at all. The circuit generated by evaluation of the completion semantics is fed directly to the SAT solver without further optimization.
- **Level 1** performs only structural hashing. Common subexpressions are eliminated, but nothing more. Moreover, if the **`implements`** keyword is used, the structural hashing is performed independently in the sketch and the specification.

[1]You may note that Level 4 is missing; This level used to apply another pass of forward optimizations on the final circuit, but we found this second pass to be completely redundant, so it was omitted from the results

- **Level 2** performs pattern matching and forward valueflow in addition to structural hashing; I refer to this combination as the forward optimizations. Again, if the **`implements`** keyword is used, optimizations are performed independently on the spec and sketch, and not on the final constraint system.
- **Level 3** performs a second pass of forward optimizations on the final constraint system.
- **Level 5** performs backwards value flow analysis in addition to the previous optimizations.
- **Level 6** includes the level 5 optimizations and also canonicalizes ands and ors.
- **Level 7** in addition to the level 6 optimizations, applies the forward optimizations to each individual inductive synthesis problem, so the constraints are specialized for the specific concrete inputs before being converted to SAT.

For these experiments, we selected 8 representative benchmarks, and ran them under all optimization levels with both ABC and MiniSat. Figure 6.11 shows the effect of performing the optimizations. The graphs, from left to right, show the effect of the optimizations for the synthesis phase, the validation phase and the overall solution time; the $y$ axis in the graphs is the solution speed normalized by the speed with level 3 optimizations ($t_{L3}/t_{Li}$). The line in the chart corresponds to the geometric mean for all the benchmarks in our sample and for both solvers for each optimization level. All the charts show the expected trend; as the optimization level goes up, the solution time improves; the only anomalies are a slight performance degradation in the synthesis time going from level 2 to level 3, and a slight decrease going from level 5 to level 6. The trends become more pronounced if we remove the bit manipulation benchmarks from consideration as shown in Figure 6.12. In this chart, we also show separate average lines for ABC and MiniSat, which show that the optimizations have an effect in performance even on top of ABC.

Averages, however, tell only part of the story. If we look at individual benchmarks, we can see some clearer trends going up the optimization levels. Figure 6.13 shows the synthesis and validation time for a few selected benchmarks. For `merge_sort` and `SetTest`, the effect of the optimizations is clear and significant. The biggest effect comes from structural hashing, but the effect of pattern matching is very significant too. Backwards value flow is significant also. Surprisingly, the effect of optimizing commutative operations is almost

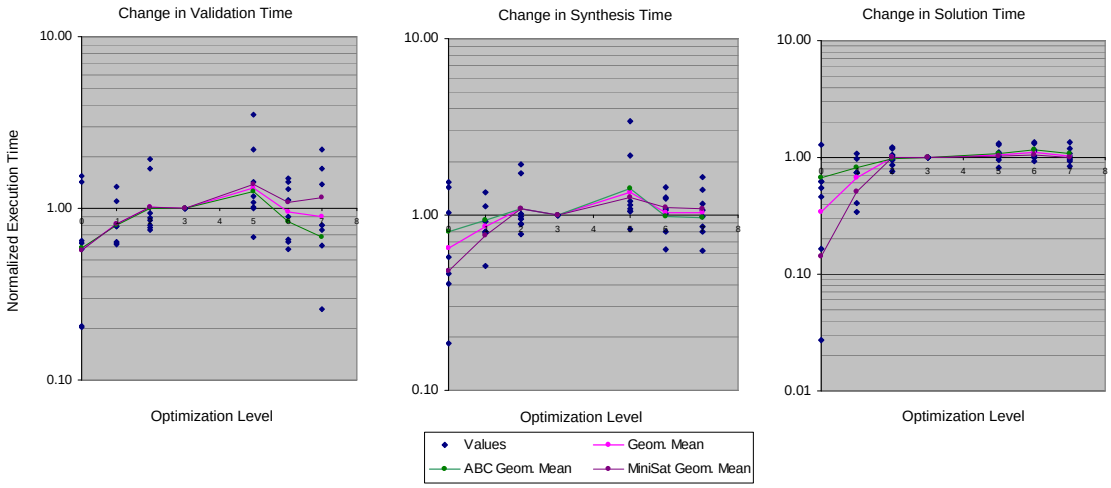



Figure 6.12: Effect of optimizations on the solution time for the benchmarks the benchmarks `merge_sort`, `listReverseHarder`, `lss_hardest` and `SetTest`.

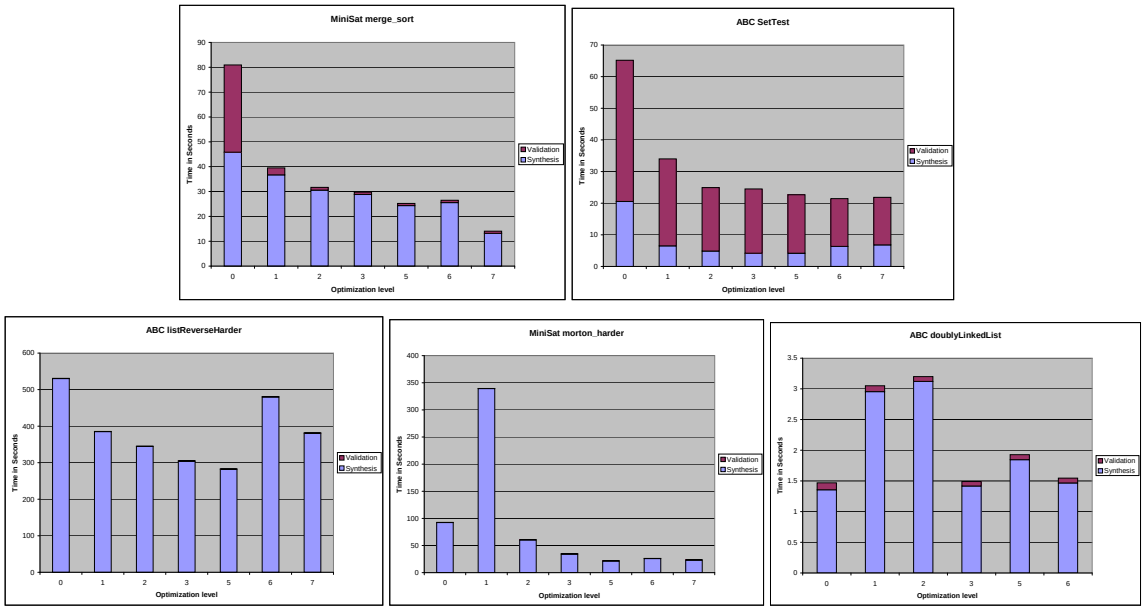



Figure 6.13: Effect of optimizations on solution time for individual benchmarks.

negligible. This is surprising because this optimization causes a significant reduction in the size of the circuit for most benchmarks, but this is almost never reflected in the solution time.

For `listReversal`, `morton` and `doublyLinkedList`, the results are more mixed. `listReverse` responds very well to optimization levels 1 through 5, but the optimization for commutative operations actually causes a performance degradation which is only partially offset by reoptimizing the circuit at each CEGIS iteration. The `morton_harder` benchmark responds well to most of the optimizations, except for a mysterious performance degradation under optimization level 1. The degradation was fairly consistent, and remained even after averaging over seven different runs. Finally, the `doublyLinkedList` benchmark also showed an inconsistent response to the optimizations. However, the solution time for this benchmark is relatively small, so the penalty for selecting the wrong optimization level is not very significant.

The individual performance results also show that the high-level optimizations compose well with the optimizations performed by ABC. This is somewhat surprising; for example, ABC performs structural hashing on the boolean problem, and this optimization is idempotent, *i.e.* applying it once is the same as applying it twice. Therefore, it is interesting that applying structural hashing on the DAG causes a performance gain even when solving the boolean problem with ABC. There are two explanations for this; first, structural hashing is able to discover more common subexpressions in the high-level circuit. For example, at the high level, it is easy to discover that $a + b$ and $b + a$ are equivalent subexpressions; once the two expressions have been expanded to boolean circuits, structural hashing may no longer be able to establish their equivalence. The second explanation is that applying the optimization on the high level circuit prevents many Boolean nodes from being created only to be eliminated later, thus preventing fragmentation of the heap.

The results can be summarized in the following four observations.

1. Optimization levels 1, 2 and 3 had a positive effect that was almost universal. Nevertheless, some benchmarks exhibited some hard-to-explain performance degradations moving up these optimization levels.

2. The results were mixed for Optimization levels 6 and 7. While a few benchmarks benefited from these, some exhibited significant performance degradations. For this reason, the SKETCH solver uses optimization level 5 by default, allowing you to override this with a command line flag.

3. Optimizations had their worst effects on bit-level benchmarks. A few of these exhibited performance degradation with higher levels of optimization.

4. None of the benchmarks showed as much performance improvement as they showed with ABC; however, the optimizations compose well with ABC, making for big overall performance improvements when the optimizations are used together.

## 6.4 Comparison with QBF

One of the original motivations for the CEGIS algorithm was the difficulty of solving constraint systems with multiple quantifiers. Using the strategies in Section 5.1, we can define the sketch synthesis problem as the problem of finding a control $\phi$ satisfying the following equation, where the predicate $Q$ is the constraint $substitute(\psi_{sym}, \sigma)(\phi) = 1$, as defined in Section 5.1.1.

$$\exists\phi \; \forall\sigma \; Q(\phi, \sigma) \tag{6.4.1}$$

The predicate $Q$ is a boolean formula, so the equation above is a satisfiability problem on a quantified boolean formula (QBF). Over the last few years, there has been a lot of interest in QBF solvers. Every year, there is even a QBF competition held side by side with the annual SAT competition at the International Conference on Theory and Applications of Satisfiability Testing. Therefore, an important question is: How does the CEGIS algorithm compare with general QBF solvers?

To answer this question, I generated QBF problems for four representative benchmarks of varying degrees of difficulty: `polynomial`, `doublyLinkedList`, `lss_hardest` and `parity`. The QBF problems were generated from the optimized constraint system, so the QBF solver could benefit from all the high level optimizations available to the SKETCH synthesizer. It is worth noting that even though there are only two quantifiers in Equation (6.4.1), this is actually a 3-QBF problem, because converting $Q$ to conjunctive normal form requires the introduction of temporary variables which are existentially quantified.

$$\exists\phi \; \forall\sigma \; \exists t \; Q_{cnf}(\phi, \sigma, t) \tag{6.4.2}$$

The QBF formulas from the four benchmarks were fed to `2clsQ`, the winner of the 2006 QBF competition [54], and `quantor` version 3.0, the winner of the 2008 competition [11]. In both cases the results support the CEGIS approach to resolving sketches.

In the case of `2clsQ`, the performance difference was overwhelming. Of the four benchmarks, `2clsQ` was only able to resolve `polynomial`, the easiest one. For this benchmark, `2clsQ` took 94 seconds to find a solution, compared to 0.1 seconds it took SKETCH with MiniSat. `2clsQ` was unable to solve any of the other three benchmarks in the 20 minutes of allotted time, while SKETCH was able to solve `parity`, the hardest of these benchmarks, in 257 seconds using MiniSat, and in only 11 seconds using ABC.

| Benchmark | SKETCH sol. time | CEGIS Iters. | `2clsQ` sol. time |
|---|---|---|---|
| `polynomial` | 0.1 sec | 5.3 | 94 sec. |
| `doublyLinkedList` | 2.6 sec | 4 | $>$ 20 min. |
| `lss_hardest` | 25 sec | 4.3 | Ran out of memory. |
| `parity` | 257 sec | 15 | $>$20 min. |

`quantor` did much better on the easier benchmarks, but it was still unable to compete with CEGIS on the harder problems. For `polynomial` and `doublyLinkedList`, `quantor` finished in about the same time as SKETCH. For both `parity` and `lss_hard`, however, `quantor` exhausted all available memory after the first two minutes of execution. After this, the system started thrashing and became unresponsive, so the execution had to be stopped. By contrast, SKETCH was able to solve both of these benchmarks using less than 150MB of memory.

| Benchmark | SKETCH sol. time | SKETCH Mem. | `Quantor` sol. time |
|---|---|---|---|
| `polynomial` | 0.1 sec | 7 MB | 0.15 sec. |
| `doublyLinkedList` | 2.6 sec | 16MB | 3.4 sec. |
| `lss_harder` | 25 sec | 136MB | Ran out of memory. |
| `parity` | 257 sec | 89 MB | Ran out of memory. |

There is an alternative encoding into a QBF problem with avoids the third quantifier; the idea is to negate Equation (6.4.1) before converting the predicate into CNF.

$$\forall\phi\ \exists\sigma\ \bar{Q}(\phi,\sigma) \tag{6.4.3}$$

Then, $\bar{Q}$, the negation of $Q$, can be converted to CNF without introducing an additional quantifier alternation.

$$\forall\phi\ \exists\sigma\ \exists t\ \bar{Q}_{cnf}(\phi,\sigma,t) \tag{6.4.4}$$

Now, the QBF solver must find a $\phi$ that falsifies the equation above. However, this encoding proved to be even worse than the previous one; with this encoding, `quantor` was unable to solve even the `polynomial` problem without running out of memory.

The CEGIS approach is unlikely to beat the QBF solvers on arbitrary QBF problems. However, on sketching problems, the CEGIS algorithm is able to exploit the bounded

observation hypothesis and efficiently synthesize a correct candidate from only a small set of inputs. Moreover, the CEGIS approach has the useful property of separating synthesis and validation, allowing the best techniques to be used for each of these two functions. The benefit of this will become most apparent in Chapter 8, when I discuss the generalization of CEGIS to concurrent programs.

## 6.5 Case Study: Sketching AES

As a case study, we used the SKETCH synthesizer to create a full implementation of the AES cipher [27] by synthesizing many of its most difficult fragments. The core of the cipher consists of 14 rounds which take a 128-bit input block and a round key and processes it, followed by a final round.

```
bit[W] round(bit[W] in, bit[W] rkey){
  bit [W] t1 = ByteSub(in);
  bit [W] t2 = ShiftRows(t1);
  bit [W] t3 = MixColumns(t2);
  return t3 ^ rkey;
}
```

The `ByteSub` transformation performs a set of table lookups to do a substitution on each byte; `ShiftRows` permutes the bytes in the block; and `MixColumns` transforms each word by treating it as a 4 element vector in the Galois field $GF(2^8)$, then multiplying it with a matrix whose elements are also in $GF(2^8)$. The final round is like the other rounds but without the `MixColumns` transformation.

In the optimized version, all the operations in the round are folded into a set of table lookups. A programmer implementing AES by traditional means would have to derive the formula for generating the table entries; this may be difficult if one is not familiar with the algebra involved. The programmer would then have to write an ad-hoc code generator to produce the table from the specification through some algebraic manipulation, and then would have to incorporate the generated table into the code and check the correctness of the cipher using known input/output pairs.

By contrast, SKETCH is able to synthesize the tables automatically and verify their correctness. Figure 6.14 shows the sketch for the regular round. The sketch for the final round is similar, except that it uses only one table instead of four, and it combines outputs from the tables using masks—which are left unspecified—instead of xors.

```
int[4] roundSK(bit[32][4] in, bit[32][4] rkey)
    implements round{
  bit[32][4][256] T = ??; // synthesize 32768 bits in the table
  bit[32][4] output = 0;
  bit[32] mask = 0x000000FF;
  int[4][4] ch = {{0,1,2,3},{1,2,3,0},
                  {2,3,0,1},{3,0,1,2}};
  for(int i=0; i<4; ++i){
    int i0 = (int) in[ch[i][0]] &  mask; // 1st 8 bits of in[ch[i][0]]
    int i1 = (int)(in[ch[i][1]] >> 8) & mask; // 2nd 8 bits
    int i2 = (int)(in[ch[i][2]] >> 16) & mask; // 3rd 8 bits
    int i3 = (int)(in[ch[i][3]] >> 24) & mask; // last 8 bits
    output[i] = T[0][i0] ^ T[1][i1]
              ^ T[2][i2] ^ T[3][i3];
    output[i] = output[i] ^ rkey[i];
  }
  return output;
}
```

Figure 6.14: Sketch for one round of AES.

| | | |
|---|---|---|
| Total Synth: | 13.183 min | |
| Total Verify: | 65.7 min | |
| Synth MiniSat: | 1.17 sec | avg time per SAT problem |
| Synth ABC: | 3.4 sec | avg time per SAT problem |
| Verify MiniSat: | 5.33 sec | avg time per SAT problem |
| Verify ABC: | 50 sec | avg time per SAT problem |

Table 6.1: Solution time for roundSK in AES benchmark. The times for Synth and Verify ABC correspond to the times for the last 10 iterations which were solved with ABC.

The `roundSK` sketch places a lot of stress on the solver since there are 32,768 bits in the table that have to be generated. Furthermore, each input considered by the solver helps complete only a small number of table entries, so the synthesize/verify loop has to iterate 655 times. Nonetheless, the solver is able to complete the sketch in about an hour. Table 6.1 shows the exact times spent by the two SAT solvers involved. All instances of synthesis were solved using MiniSat. For verification, we used MiniSat for the first 645 iterations. For the last 10 iterations we switched our SAT solver to ABC [49] because it provides much better performance for hard SAT problems.

**Performance of generated code.** The resulting code was run against a hand optimized AES implementation from open SSL. The runtime for 50000 encryptions was as follows:

| | |
|---|---|
| OpenSSL AES | 19.652 ms |
| Sketch | 21.307 ms |
| Spec | 19936.100 ms |

The difference between the hand coded AES and the sketched version is less than 10%. We can also see that the original specification, which is very close to the specification of AES [27], is over 1000 times slower.

## 6.6 Conclusions

The chapter has showcased the power of the SKETCH system to synthesize the low-level details for small but complex programs in a variety of domains. In addition to that, it has shown some potential avenues for significant improvement in the synthesizer's performance.

For example, the chapter documented the enormous impact that choosing the best combination of solvers and optimizations can have on performance. Having a detailed model

to predict the best combination of optimizations, or better yet, running many of these in parallel, could make the system much more efficient and much easier to use. Programmers would get an answer faster, and would not have to waste time trying different solver combinations. Similarly, the chapter illustrated the effect of the model sizes and test harnesses on the synthesizer performance. This suggests that an intelligent strategy that gradually increments the size of the models or the complexity of the test harness could have a huge effect on the solution time, especially for problems involving linked datastructures.

There are a number of other strategies that could significantly improve the performance of the solver. For example, using constraint solvers that can reason about integers could make a big difference for integer problems. Similarly, improved search strategies that take advantage of more semantic information about the sketch could make the search more efficient. Additionally, there is great room for improvement in the encoding of higher-level sketching constructs and advanced language features; the current encoding is very simple and naïve.

Beyond performance, however, it is important to keep in mind that a synthesizer is a productivity tool. The ultimate test for any optimization is the extent to which it is able to improve programmer productivity. A detailed analysis of how improvements in performance affect programmer productivity is one of the great omissions in this chapter. Quantifying this impact requires user studies and analysis of the use of sketching in the field; these are beyond the scope of this thesis. However, from my own experience working with different versions of the synthesizer, I have observed that performance improvements do have a strong impact in the way the programmer interacts with the tool. A more powerful synthesizer expands the scope of programs that the programmer can tackle, and allows for sketches with more freedom and less detail.

# Part III

# Sketching for Concurrent Programs

# Chapter 7

# Semantics for Concurrent Sketches

Concurrent algorithms and data structures present an especially appealing target for sketching. First, they constitute a very challenging domain for programmers; a domain where translating a high-level intuition into a correct implementation requires detailed low-level reasoning about all possible interactions among threads. Moreover, sketching can help programmers cope with this complexity by helping with the low-level details involved in these programs, as was illustrated by the concurrent set example in the introduction.

Unfortunately, the synthesis algorithm presented so far was designed to handle only sequential sketches. Specifically, the synthesis semantics, which we used to derive the synthesis algorithm, are inherently sequential. In order to solve concurrent sketches, we need a mechanism to reason formally about their semantics and their sets of valid solutions. This is achieved in this section by exploiting the concept of a *trace*.

A trace of a concurrent program is a sequential ordering of the operations executed by all the threads. Our language assumes sequential consistency, which means that any concurrent execution of the program is equivalent to some sequential interleaving of the threads, and therefore has a corresponding trace. Because a trace is just a sequential program, we can reason about the semantics of a concurrent program in terms of the sequential semantics of all its traces. This idea is illustrated by our generalization of the sketch resolution equation.

## 7.1 The Concurrent Sketch Resolution Equation

Let $tr(P)$ be the set of all possible traces for a given sketch $P$. For now, I define a trace informally as some interleaving of the operations in all the threads. The semantics of a

sketch can thus be defined in terms of the sequential semantics for each of the traces in $tr(P)$. In particular, the sketch resolution equation can be restated to account for concurrency as follows.

**Equation 2 (Concurrent Sketch Resolution Equation)**

$$\forall t \in tr(P) \forall\, \sigma\ \pi_{\Phi}(\ \mathcal{C}[\![t]\!]^{\tau_\emptyset}\langle \sigma\,,\, \Phi\rangle) = \Phi \tag{7.1.1}$$

The equation is very similar to its sequential counterpart; the key difference is that now the set of valid controls must be invariant not just under all inputs, but also under all traces.

The equation above is incomplete without a precise characterization of the set of all traces of a sketch. Section 7.2 will describe this set formally, in a way that ensures that solutions to the sketch equation actually correspond to deadlock free programs that satisfy all their assertions.

One drawback in our approach is that candidates can only be eliminated based on violations of safety properties on a trace. This is unfortunate because liveness properties can be very important for concurrent programs. The synthesizer gets around this problem through the boundedness assumption; after all, if an execution is bounded, any liveness property can be expressed as a safety property which must hold after a bounded number of steps. For example, given a bound $N$, the synthesizer enforces termination by requiring that candidates terminate after $N$ execution steps for the bounded inputs it considers.

## 7.2 Tracing Semantics

The idea of reasoning about concurrent programs in terms of their possible traces is not new; in his seminal work, Mazurkiewicz used traces to define the concurrent semantics of Petri nets [47]. However, applying this idea to sketching introduces some unique challenges. The most notable of these comes from the fact that sketches can not be executed in the traditional sense because they are incomplete; therefore, one can not talk about traces in terms of an execution. This requires us to introduce a more abstract concept of traces suitable for sketch synthesis, one that allows us to apply sequential inductive synthesis techniques to the problem of synthesizing concurrent programs.

### 7.2.1 Traces of Sketches

For a concrete concurrent program, a trace is some interleaving of the operations executed by all the threads. A sketch, however, has holes, and therefore can't be executed like a concrete program can. Thus, a sketch requires a more abstract definition of a trace that is independent of any execution. To illustrate the issues involved, consider the following simple example.

```
void main(){
        fork(int i; 2){
                if(??){
                        x = 10;
                }else{
                        x = 5;
                }
                t = x;
                x = t + 3;
        }
        assert x < 15;
}
```

The sketch above can be resolved to two different concrete programs, shown in Figure 7.1 together with their set of possible traces. The symbol $\rangle\!\rangle$ is used to separate the individual atomic steps in a trace, and the underlined steps correspond to steps executed by thread 0. In the figure one can see that when $\phi_1(\texttt{??}) = 1$, some of the traces cause assertion failures, whereas all traces satisfy the assertion when $\phi_0(\texttt{??}) = 0$, so it is clear that the latter candidate is the one we expect from the synthesizer. The problem at hand is to produce a set of traces for the sketch, rather than for the individual candidates, such that $\{\phi_0\}$ is the solution to the sketch synthesis equation.

To address this problem, we define a set of traces for the sketch through a non-deterministic scheduling function *sched* that produces an ordering of all the atomic operations in a sketch. The set $tr(P)$ of traces for a sketch $P$ is then defined as the set of all the possible traces that can be generated by the non-deterministic function *sched*.

$$sched(P) \rightarrow s_1 \rangle\!\rangle s_2 \rangle\!\rangle \ldots \rangle\!\rangle s_n$$

$\phi_1(??) = 1$

```
void main(){
        fork(int i; 2){
                x = 10;
                t = x;
                x = t + 3;
        }
        assert x < 15;
}
```

x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ t0=x ⟫ x=10 ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=16*
x=10 ⟫ t0=x ⟫ x=10 ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ t0=x ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=16*
x=10 ⟫ x=10 ⟫ t0=x ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t0=x ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t1=x ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t1=x ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=16*
x=10 ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=16*
x=10 ⟫ x=10 ⟫ t0=x ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t0=x ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t1=x ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t1=x ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=16*
x=10 ⟫ t1=x ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ t1=x ⟫ x=10 ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=13*
x=10 ⟫ t1=x ⟫ x=10 ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=16*
x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=13*

$\phi_0(??) = 0$

```
void main(){
        fork(int i; 2){
                x = 5;
                t = x;
                x = t + 3;
        }
        assert x < 15;
}
```

x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ t0=x ⟫ x=5 ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=11*
x=5 ⟫ t0=x ⟫ x=5 ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ t0=x ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=11*
x=5 ⟫ x=5 ⟫ t0=x ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t0=x ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t1=x ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t1=x ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=11*
x=5 ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=11*
x=5 ⟫ x=5 ⟫ t0=x ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t0=x ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t1=x ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t1=x ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=11*
x=5 ⟫ t1=x ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ t1=x ⟫ x=5 ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=8*
x=5 ⟫ t1=x ⟫ x=5 ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=11*
x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ **assert** x < 15; *//x=8*

Figure 7.1: Two possible completions for a sketch with their respective traces.

The symbol $\rangle\!\rangle$ is used to separate each of the atomic steps $s_i$ that make up the trace. The individual atomic statements $s_i$ can be of any of the following four types.

- Assignments.
- Atomic Blocks.
- Assumptions.
- Assertions.

Assignments and atomic blocks are interpreted according to their sequential semantics. We didn't define assumptions in the core language, but they are easy to define as syntactic sugar by introducing a variable $\triangle$ that becomes false when an assumption is violated.

$$\mathcal{C}[\![\texttt{assume}\ e]\!]^\tau\langle\sigma\,,\,\Phi\rangle = \langle\sigma[\triangle \mapsto \sigma(\triangle) \wedge\ \mathcal{A}[\![e]\!]^\tau]\,,\,\Phi\rangle \tag{7.2.1}$$

This forces us to alter slightly the semantics of assertions, so that they become conditional on the value of $\triangle$; in other words, **assert** $e$ will now be interpreted as **if** $(\triangle)$ **then assert** $e$. The net effect is that an assumption failure causes the reminder of the trace to be ignored for the purpose of eliminating candidates.

The *sched* function works by generating a non-deterministic interleaving of the statements in the sketch, in much the same way as a scheduler would for a concrete program. The main difference between *sched* and a traditional scheduler is that in generating the trace, *sched* must make assumptions about the direction of branches. These assumptions are documented by adding `assume` statements into the trace. Figure 7.2 shows how these traces look like for our running example. Notice how the set of traces contains the information about the traces for each of the individual candidates, but the `assume`s document the direction of branches, which in this case are determined exclusively by the value of the hole. Thanks to these `assume`s, it is possible to determine from this set of traces that $\{\phi_0\}$ will satisfy the sketch equation.

The *sched* function is defined recursively in terms of non-deterministic rewrite rules. For the sake of simplicity, the rules below assume that each expression and each assignment statement is atomic, so they contain at most one read or write to a global variable. It is possible to rewrite the rules to eliminate this requirement, but that would just obscure the presentation.

```
void main(){
 fork(int i; 2){
  if(??){
   x = 10;
  }else{
   x = 5;
  }
  t = x;
  x = t + 3;
 }
 assert x < 15;
}
```

```
assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ assume ??=1 ⟫ x=5 ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ assume ??=1 ⟫ x=5 ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=1 ⟫ x=5 ⟫ t1=x ⟫ x=t1+3 ⟫ assume ??=1 ⟫ x=5 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ assume ??=0 ⟫ x=10 ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ t1=x ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ t1=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ t1=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ t0=x ⟫ x=t1+3 ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ x=t1+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ assume ??=0 ⟫ x=10 ⟫ x=t0+3 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ assume ??=0 ⟫ x=10 ⟫ x=t1+3 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
assume ??=0 ⟫ x=10 ⟫ t1=x ⟫ x=t1+3 ⟫ assume ??=0 ⟫ x=10 ⟫ t0=x ⟫ x=t0+3 ⟫ assert x < 15;
```

Figure 7.2: A sample of the traces for the sketch generated by the *sched* function.

For sequential statements, the definition simply separates the statements that make up a compound statement.

$$\begin{aligned} sched(c1; c2) &\rightarrow (sched(c1) \between sched(c2)) \\ sched(x = e) &\rightarrow (x = e) \\ sched(\textbf{assert}\ e) &\rightarrow (\textbf{assert}\ e) \end{aligned}$$

For conditional statements, the *sched* function must make a non-deterministic decision regarding which branch to traverse.

$$sched(\textbf{if}\ e\ \textbf{then}\ c1\ \textbf{else}\ c2) \rightarrow \begin{cases} assume\ e \between sched(c1) \\ assume\ \neg e \between sched(c2) \end{cases}$$

Similarly for loops, the *sched* function must decide for each iteration whether to stop or to continue iterating.

$$sched(\textbf{while}\ e\ \textbf{do}\ c1) \rightarrow \begin{cases} assume\ \neg e \\ assume\ e \between sched(c1) \between sched(\textbf{while}\ e\ \textbf{do}\ c1) \end{cases}$$

For atomic statements, the *sched* function does not have to do much; the body of the atomic is simply scheduled as a single atomic step in the trace.

$$sched(\textbf{atomic}\ c1) \rightarrow (c1)$$

Finally, the rule for **fork** must take the traces from each thread and interleave them together into the single trace.

$$sched(\textbf{fork(t, N)}\ c) \rightarrow mix_N(sched(ren(c, 0)), sched(ren(c, 1)), \ldots, sched(ren(c, N-1)))$$

The function $ren(c, i)$ renames all local variables $x$ in $c$ to $x_i$, and replaces variable `t` with the integer $i$. The function $mix_N$ takes $N$ traces and nondeterministically interleaves their atomic statements to produce a single trace. The traces resulting trace will be able to eliminate any candidate control that would cause an assertion failure while satisfying all the assumptions.

```
void main(){
        int cnt  =0, t = 0;
        fork(int i; 2){
                atomic(cnt == i){
                        cnt++;
                        t = i*(cnt+1);
                }
        }
        assert t == ??;
}
```

(cnt = 0; t=0) ⟫ (assume cnt = 0; cnt++; t = 0*(cnt+1)) ⟫ (assume cnt = 1; cnt++; t = 1*(cnt+1)) ⟫ **assert** t =??
(cnt = 0; t=0) ⟫ (assume cnt = 1; cnt++; t = 1*(cnt+1)) ⟫ (assume cnt = 0; cnt++; t = 0*(cnt+1)) ⟫ **assert** t = ??

Figure 7.3: Handling conditional atomics with assumptions

### 7.2.2 Conditional atomics and deadlock

Conditional atomics involve more subtlety than some of the other constructs. This is because the *sched* function must enforce the conditional execution requirement; *i.e.* the conditional atomic must not execute until its condition is satisfied. Additionally, conditional atomics can cause deadlock because of their blocking semantics, so we would like to have traces that eliminate any deadlock-prone candidates.

To cope with the conditional execution requirement, we want to consider only those traces that execute the conditional atomic when its condition is satisfied. This can be achieved by assuming the execution condition before executing the body of the conditional atomic. Once a trace fails an assumption, it gets ignored because it ceases to have an effect on the set of valid configurations. Figure 7.3 illustrates this approach. For this sketch there are two different traces, but the second one fails the assumption, so the value of the hole is defined by the first trace.

The problem with this seemingly simple solution is that a conditional with an unsatisfiable condition would cause all traces to fail their assumptions, and consequently would not be able to eliminate any candidates even when they are clearly erroneous. For example, consider the sketch in Figure 7.4. For this sketch, any $\phi$ satisfying $\phi(??) \geq 2$ will be a valid configuration for both traces, but only because setting the value of the hole to a constant greater than one would make the condition unsatisfiable under any schedule; in other words, it would cause a deadlock.

```
void main(){
        int cnt  =0, t = 0;
        fork(int i; 2){
                atomic(cnt == i+??){
                        cnt++;
                        t = i*(cnt+1);
                }
        }
        assert t == 2;
}
```

---

```
(cnt = 0; t=0) ⟫ (assume cnt = ??; cnt++; t = 0*(cnt+1)) ⟫ (assume cnt = 1+??; cnt++; t = 1*(cnt+1)) ⟫ assert t =2
(cnt = 0; t=0) ⟫ (assume cnt = 1+??; cnt++; t = 1*(cnt+1)) ⟫ (assume cnt = ??; cnt++; t = 0*(cnt+1)) ⟫ assert t = 2
```

Figure 7.4: Handling conditional atomics with assumptions

The way around this problem is to combine the conditional execution requirement with deadlock detection. This can be done through the following definition of the *sched* function.

$$sched(\textbf{atomic}\ e \Rightarrow c1) \rightarrow \left(\begin{array}{l} \textbf{if}\ e \\ \textbf{then}\ (c1;\ t = 1) \\ \textbf{else}\ (\texttt{dl} = \texttt{dl} + 1;\ \textbf{assert}\ \texttt{dl} \neq N;\ t = 0) \end{array}\right) \rangle\!\rangle (\texttt{assume}\ t)$$

The entire right hand side of this formula translates into two atomic steps. It first checks for the condition to hold; if the condition doesn't hold, then it checks for the presence of deadlock; if there is no deadlock, then the assumption in the next atomic step will fail. In order for the scheme to work correctly, there must be a prologue at the beginning of the **fork** that initializes $N$ to the number of threads and $dl$ to zero. There is also a need for an epilogue at the end of each thread incrementing $dl$ by one, so that the scheme can detect deadlocks involving only some of the threads. With these modifications, the formula for **fork** is redefined as shown below.

$$sched(\textbf{fork(t, M)}\ c) \rightarrow (dl = 0; N = M) \rangle\!\rangle$$
$$mix_N(sched(ren(c,0)) \rangle\!\rangle (dl = dl + 1), \ldots, sched(ren(c, N-1)) \rangle\!\rangle (dl = dl + 1))$$

The above strategy begs the question of why separate the assumption check into a separate atomic step. The reason for this is that in order to detect a deadlock, the deadlock detection for each of the threads must be scheduled before all the **assume**s. This is because

the deadlock will only trigger an assertion failure when the last thread to arrive at the deadlock executes its deadlock detection code, so it is important that none of the previous threads has issued an `assume`. This is best illustrated with a couple of traces from the example in Figure 7.4. The traces are shown in Figure 7.5; the trace on the left has the conditional atomics in the right order; notice how this trace will reject any $\phi$ with $\phi(\texttt{??}) \neq 0$ thanks to the deadlock detection. The trace on the right shows how the `assume`s prevent any valid candidate from being eliminated when the conditional atomics are scheduled in an invalid order.

With this definition of the set of traces for a sketch, the sketch resolution equation will properly define the set of valid candidates for a sketch. Candidates that cause assertion failures or deadlocks for some thread interleaving will be eliminated, resulting in a set of correct candidates. As in the sequential case, however, solving the sketch equation directly will be impossible for all but the most trivial sketches because the set of possible traces is enormous. Fortunately, a generalization of the CEGIS algorithm will allow us to solve sketches like the one in the introduction in a couple of minutes.

## 7.3 Effect of program transformations

In this section, I focus on program transformations that affect the set of traces the program may produce, but without affecting the set of solutions to the sketch equation. An especially useful class of such transformations involves transformations affecting conditionals. For example, consider the transformations illustrated by the rewrite rules below. In the transformation rules, $t_i$ is used to indicate a fresh local variable, $e_i^l$ is an expression involving only local variables, and $s_i^{atom}$ is a statement involving a single atomic operation, either an assignment, an assertion, an assumption or an atomic statement.

$$\textbf{if } e \textbf{ then } (s_1; s_2) \rightarrow t_i = e; (\textbf{if } t_i \textbf{ then } s_1); (\textbf{if } t_i \textbf{ then } s_2) \tag{7.3.1}$$

$$\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \rightarrow t_i = e; (\textbf{if } t_i \textbf{ then } s_1); (\textbf{if } \neg t_i \textbf{ then } s_2) \tag{7.3.2}$$

$$\textbf{if } e_1^l \textbf{ then if } e_2^l \textbf{ then } s_1 \rightarrow \textbf{if}(\ e_1^l \wedge e_2^l) \textbf{ then } s_1 \tag{7.3.3}$$

$$\textbf{if } e_1^l \textbf{ then } s_1^{atom} \rightarrow \textbf{atomic}\{\ \textbf{if } e_1^l \textbf{ then } s_1^{atom}\} \tag{7.3.4}$$

These transformations affect the set of traces generated for a program, but do not affect the set of solutions to the sketch equation. For example, consider the transformation

Valid Order

| thread | step |
|---|---|
| s: | (cnt = 0; t=0; dl=0) ⟫ |
| | **if**(cnt = **??**){ |
| | cnt++; |
| | t = 0∗(cnt+1); |
| | $t_0$=1; |
| 0: | }**else**{ ⟫ |
| | dl = dl + 1; |
| | **assert** dl != 2; |
| | $t_0$=0; |
| | } |
| | **if**(cnt = 1+**??**){ |
| | cnt++; |
| | t = 0∗(cnt+1); |
| | $t_1$=1; |
| 1: | }**else**{ ⟫ |
| | dl = dl + 1; |
| | **assert** dl != 2; |
| | $t_1$=0; |
| | } |
| 0: | assume $t_0$ ⟫ |
| 1: | assume $t_1$ ⟫ |
| 0: | dl = dl + 1 ⟫ |
| 1: | dl = dl + 1 ⟫ |
| s: | **assert** t==2; |

Invalid Order

| thread | step |
|---|---|
| s: | (cnt = 0; t=0; dl=0) ⟫ |
| | **if**(cnt = 1+**??**){ |
| | cnt++; |
| | t = 0∗(cnt+1); |
| | $t_1$=1; |
| 1: | }**else**{ ⟫ |
| | dl = dl + 1; |
| | **assert** dl != 2; |
| | $t_1$=0; |
| | } |
| | **if**(cnt = **??**){ |
| | cnt++; |
| | t = 0∗(cnt+1); |
| | $t_0$=1; |
| 0: | }**else**{ ⟫ |
| | dl = dl + 1; |
| | **assert** dl != 2; |
| | $t_0$=0; |
| | } |
| 0: | assume $t_0$ ⟫ |
| 1: | assume $t_1$ ⟫ |
| 0: | dl = dl + 1 ⟫ |
| 1: | dl = dl + 1 ⟫ |
| s: | **assert** t==2; |

Figure 7.5: Correct traces for the example in Figure 7.4.

expressed in Equation (7.3.2). For the left hand side, the *sched* function can produce two different traces.

$$sched(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2) \rightarrow \begin{cases} assume\ e \rangle\!\rangle sched(s_1) \\ assume\ \neg e \rangle\!\rangle sched(s_2) \end{cases}$$

For the right hand side, there are four possible traces produced by the *trace* function.

$$sched(t_i = e; (\textbf{if } t_i \textbf{ then } s_1); (\textbf{if } \neg t_i \textbf{ then } s_2)) \rightarrow \begin{cases} t_i = e \rangle\!\rangle assume\ t_i \rangle\!\rangle sched(s_1) \rangle\!\rangle assume\ \neg t_i \rangle\!\rangle sched(s_2) \\ t_i = e \rangle\!\rangle assume\ t_i \rangle\!\rangle sched(s_1) \rangle\!\rangle assume\ \neg\neg t_i \\ t_i = e \rangle\!\rangle assume\ \neg t_i \rangle\!\rangle assume\ \neg t_i \rangle\!\rangle sched(s_2) \\ t_i = e \rangle\!\rangle assume\ \neg t_i \rangle\!\rangle sched(s_1) \rangle\!\rangle assume\ \neg\neg t_i \rangle\!\rangle sched(s_2) \end{cases}$$

These traces will be interleaved with the traces from the other threads, so the transformation greatly increases the number traces in the set $tr(P)$. However, the set of solutions to the sketch equation will remain unaffected by the transformation. To see why this is the case, note that the second and third traces for the transformed statement are fully equivalent to the two traces for the original statement. This can be proved by resorting to partial order reduction [32], since all the **assume**s in the new version involve only local variables. As for the other two traces, note that they contain contradictory assumptions, so at most, only their prefixes can affect the set of solutions to the sketch; however, these prefixes are equivalent to the prefixes of the second and third traces, so overall, the set of solutions will be unaffected by the transformation.

By themselves, the first two transformations are not very useful, but they can be used to enable the second two transformations, which have the effect of removing assumptions from traces, allowing more information to be extracted from a single trace. For example, the transformation in Equation (7.3.4) completely eliminates from the trace any assumptions associated with the **if** statement. These transformations will be extremely valuable in generalizing the CEGIS algorithm to deal with concurrency.

# Chapter 8

# Concurrent CEGIS

The previous section defined the concurrent semantics of sketches in terms of the sequential semantics of their traces, where each trace captures the semantics of a particular interleaving of the threads in the sketch. Using this formalism, the sketch resolution equation can be generalized to the concurrent case by requiring a valid control $\phi$ to satisfy all assertions under all possible inputs $\sigma$ *and* for all possible traces $t$:

$$\forall\, \sigma \in \Sigma\, \forall\, t \in tr(P)\ \pi_{\Phi}(\ \mathcal{C}[\![t]\!]^{\tau_\emptyset}(\sigma, \{\phi\})) = \{\phi\}. \tag{8.0.1}$$

The need to reason about all possible traces makes the synthesis problem harder than it was for the sequential case. Just as in the sequential case, however, the synthesizer can exploit a bounded observation hypothesis to make the synthesis problem tractable.

**Hypothesis 2** *For a given sketch* `P`*, it is possible to find a small set of inputs* $E$ *and and a small set of traces* $T$ *that fully represent the entire universe of possible inputs and traces* $\Sigma \times tr(P)$*, in the sense that any control* $\phi$ *satisfying*

$$\forall t \in T\, \forall\, \sigma \in E\ \ \mathcal{C}[\![t]\!]^{\tau_\emptyset}\langle \sigma\,, \{\phi\}\rangle = \langle \sigma', \{\phi\}\rangle \tag{8.0.2}$$

*will also satisfy Equation* (8.0.1)*.*

## 8.1 The Algorithm

The bounded observation hypothesis allows us to solve the concurrent synthesis problem using counterexample guided inductive synthesis, just like in the sequential case

(see Section 4.1). The algorithm uses an inductive synthesis procedure to generate candidates that are valid for a small set of inputs *and traces*. The candidates are then checked by a validation procedure; if the validation finds no errors, the candidate is determined to be a solution to the synthesis problem. If the validation procedure finds an error, it produces a trace and an input to expose the error, and these are fed to the inductive synthesizer, which is then responsible for producing a new candidate solution. The complete algorithm is shown below; the notation is the same as before; $PE(P, \phi)$ is the concrete program that results from partially evaluating program $P$ with control $\phi$; $\Phi_i$ is the $i^{th}$ approximation to the set of valid controls, and control $\phi_i \in \Phi_i$ is the $i^{th}$ candidate solution for the sketch; $\sigma_i$ and $t_i$ are the counterexample input and trace respectively. The expression $\pi_\Phi(\ \mathcal{C}[\![t_{i-1}]\!]^{\tau_\emptyset}(\sigma_{i-1}, \Phi_{i-1}))$ evaluates the trace $t_{i-1}$ under the synthesis semantics on input state $\sigma_{i-1}$ and control set $\Phi_{i-1}$, and uses the projection function $\pi_\Phi$ to select the new control set from the result.

**Algorithm 2 (Concurrency Aware CEGIS Algorithm)** .

```
Input: P
φ_0 := φ_random
Φ_0 = Φ
i := 0;
do

    def (t_i, σ_i) = validate(PE(P, φ_i))          } Validation Phase
    if (t_i = null) then break                     }

    def Φ_i = π_Φ( C[[t_{i-1}]]^{τ_∅}(σ_{i-1}, Φ_{i-1}))   }
    if Φ_i = ∅ then return ERROR                           } Inductive Synthesis Phase
    def φ_i ∈ Φ_i                                          }

    i = i + 1
while true
return φ_i
```

The algorithm operates by the same principles as the sequential CEGIS, so the arguments we made in Section 4.1 regarding convergence of the control sets $\Phi_i$ apply in this context as well; however, there are some important differences. The first important difference with the sequential algorithm is that the inductive synthesizer now operates on traces, as opposed to programs. However, a trace is just a sequential interleaving of the statements in the original program, so it can be treated just like a sequential program: it can be partially evaluated with respect to a control, and it can be analyzed using the synthesis semantics.

This is very important because it means the inductive synthesizer does not have to reason about concurrency; as a consequence, it is not affected by astronomical number of possible interleavings in the original program. The tasks of reasoning about all possible interleavings is delegated to the validation procedure, and this is the second important difference with sequential CEGIS.

The validation procedure is now in charge of all the concurrency related reasoning. It must be able to either validate the current candidate, or produce a counterexample input and trace to expose a bug. Because the validator must reason about concurrency, it can no longer be described in terms of the synthesis semantics. Fortunately, most off-the-shelf validation procedures for concurrent programs are able to produce counterexample traces for buggy inputs. This allows the current synthesizer to delegate all the concurrency related reasoning to an off-the-shelf component; we use the SPIN model checker, but many other validation procedures could easily fit the bill.

There are other more superficial differences between the algorithm above and the sequential CEGIS from Section 4.1. For example, the synthesis and verification phases are swapped in the loop only because it's easier to generate a random control than it is to generate a random trace. Another difference is that the new algorithm must perform partial evaluation on each iteration of the CEGIS loop; this is because the validation procedure can no longer be expressed in terms of the synthesis semantics, so on each iteration, the algorithm must produce a concrete program to pass to the external validation procedure. Similarly, recall that the sequential CEGIS was able to avoid evaluating the synthesis semantics on each iteration of the CEGIS loop thanks to the manipulations from Section 5.1.1. In the concurrent CEGIS algorithm, however, each iteration must evaluate the semantics on a different trace, so the overhead of evaluating the semantics on each iteration can not be avoided. These differences are superficial, but they have an impact on the performance of the implementation.

There is one important aspect of the algorithm which the discussion so far has ignored. The validation procedure validates the candidate $P_{\phi_i} = PE(P, \phi_i)$. If the candidate has an error, the validation procedure will produce a trace $t_{P_{\phi i}}$ for program $P_{\phi_i}$ exposing the error; however, the inductive synthesis procedure needs a trace $t_P \in tr(P)$ of the sketch; therefore, we need a procedure for converting a trace acquired from one program into a trace for a sketch; we call this process *projection*, and it is crucial to the effectiveness of the above algorithm.

## 8.2 Trace Projection

In sequential CEGIS, the validator produced an counterexample input $\sigma_i$, which could be directly used by the inductive synthesizer to eliminate an entire class of candidates; ideally, $\sigma_i$ would eliminate all the candidates exhibiting the same bug that $\sigma_i$ exhibited in candidate $P_{\phi_i}$. In the concurrent case, on the other hand, the validation procedure generates a trace that is intimately tied to the specific candidate $P_{\phi_i}$. The trace describes the exact execution order of statements in $P_{\phi_i}$, and encodes all the control decisions in a particular execution of this program. The challenge, is to generate from this trace a new trace for the sketch $P$ that allows the inductive synthesizer to eliminate not just $P_{\phi_i}$, but also other candidates sharing the same bug, even if these other candidates differ considerably from $P_{\phi_i}$. This is the goal of trace projection.

A trace projection, denoted $t_{P_\phi} \triangleright P$, is an operation that takes a sketch $P$ and a trace $t_{P_\phi}$ generated from a candidate $P_\phi$, and produces a new trace $t_P$ for the sketch $P$. In order for the CEGIS algorithm to converge, the projected trace must at least allow the inductive synthesizer to eliminate the candidate $P_\phi$ that produced the original trace. Additionally, the effectiveness of the CEGIS algorithm will be greatly enhanced if the projected trace exhibits *preservation of errors*. Preservation implies that if the program $P_{\phi j}$ exhibits the same bug which the trace $t_{P_{\phi i}}$ exposed in $P_{\phi i}$, then $t_P = t_{P_{\phi i}} \triangleright P$ should eliminate $\phi_j$ in addition to $\phi_i$. This implies that $PE(t_P, \phi_j)$ exhibits an assertion failure on input $\sigma_i$, just like $t_{P_{\phi i}}$.

This informal notion of preservation is inherently vague due to the varied nature of bugs and the difficulty of establishing the root cause of a failure by analyzing a trace. A bug in a candidate may be caused by lack of enough synchronization leading to a race, or too much of it leading to a deadlock, among many other pathologies. Moreover, the root cause of the error may lie far from the point of failure in the program. So instead of full preservation of errors, we settle for a simpler property that can serve as a proxy; one that can be enforced cheaply, and that at the very least guarantees that the projected trace will eliminate the original buggy candidate.

To help us define a suitable proxy for preservation of errors, I introduce the notion of *step-ordering preservation*. Section 7.2 defined a trace of a sketch as an interleaving of atomic steps from all the threads in the program. We say that a trace $t'$ *preserves* a trace $t$ ($t' \cong_p t$) if all steps common to $t'$ and $t$ are executed in the same order in both traces. Thus, in order for $t'$ to preserve $t$, we require that if step $s_1$ precedes step $s_2$ in $t$, and both $s_1$ and

$s_2$ are present in $t'$, then $s_1$ precedes $s_2$ in $t'$. The preservation relation $\cong_p$ is reflexive and commutative.

This notion of step-ordering preservation is practically relevant because preserving step ordering is likely to preserve the conditions that lead to an error. To see why this is the case, consider two traces $t_1$ and $t_2$, which share a set of statements $\{s_1, \ldots, s_n\}$ that lead trace $t_1$ to fail an assertion. If $t_2$ preserves the order of these statements, then it will likely preserve the dataflow relationship among these statements that lead to the error in $t_1$. I use the word likely, because it is possible for $t_2$ to preserve the ordering of the shared statements without preserving the dataflow; this would happen if $t_2$ contains some additional statement not present in $t_1$ which sits between the shared statements and alters the dataflow. However, step-ordering preservation is simple to enforce and we have observed that it usually succeeds in preserving errors.

Therefore, we can achieve some level of preservation of errors by preserving step ordering. Unlike preservation of errors, however, step-ordering preservation is a well defined property that can be used to state a set of formal requirements for the projection algorithm.

**Definition 1 (Requirements of trace projection)** *A projection $t_P = t_{P_\phi} \triangleright P$ from a trace $t_{P_\phi}$ generated from a candidate $P_\phi$ to a sketch $P$ must satisfy the following two properties.*

- *For all controls $\phi_o$, the trace $t_{\phi_o} = PE(t_P, \phi_o)$, obtained by partially evaluating the projected trace $t_P$ with control $\phi_o$, must preserve the step ordering of the original trace; i.e. $t_{\phi_o} \cong_p t_{P_\phi}$.*
- *For all controls $\phi_o$, the trace $t_{\phi_o} = PE(t_P, \phi_o)$ must be semantically equivalent to a valid trace for $P_{\phi_o}$; i.e. $t_{\phi_o} \in tr(P_{\phi_o})$.*

*The desired relationship between the traces $t_{P_\phi}$, $t_{\phi_o}$ and $t_P$ is illustrated by the diagram in Figure 8.1.*

The first apparent obstacle to satisfying this definition stems from the blocking behavior of synchronization primitives. To illustrate the problem, consider the sketch shown below; it contains two statements `s1` and `s2`, and uses a boolean hole to leave unspecified the synchronization around them.

```
bool c = ??;
```

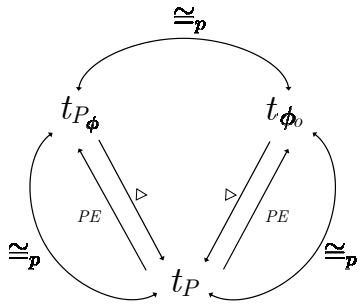



Figure 8.1: Relationship between partial evaluation, projection and the preservation relation

```
thrd1: { sa; if (c) wait; s1; if (!c) signal }
thrd2: { sb; if (!c) wait; s2; if (c) signal }
```

This sketch corresponds to two candidate programs, selected based on the value of `c`:

$P_t$: `thrd1: { sa; wait; s1; } thrd2: { sb; s2; signal; }`

$P_f$: `thrd1: { sa; s1; signal; } thrd2: { sb; wait; s2; }`

Because of synchronization, the program $P_t$ must execute $s2$ before $s1$, while $P_f$ must execute $s1$ before $s2$. How then is it possible for the projected trace to be specialized into a valid trace for both $P_t$ and $P_f$? The answer lies in the way the *sched* procedure described in Section 7.2 models blocking behavior. All blocking primitives are described in terms of conditional atomic sections. The *sched* procedure models the conditional atomic by using an `assume` to check whether its execution condition is satisfied when the conditional atomic is scheduled to run. If it is not, the assume fails and the rest of the trace is ignored. In the above example, a trace from $P_f$ will be projected into a trace for the original program having $s1$ before $s2$, so if the trace is specialized with $c = true$, then we will get a trace for $P_t$ that tries to execute $s1$ before $s2$. The trace will be a valid trace in $tr(P_t)$, although it will be guaranteed to fail an assumption after it tries to execute the `wait` before having executed the `signal`. The net result is that the only part of the trace that matters is the part of the trace corresponding to the longest prefix of the execution for which it is possible to preserve step ordering.

More problematic are control decisions made by the *sched* procedure which prevent the trace $PE(t_P, \phi)$ from being equivalent to a valid trace for program $P_\phi = PE(P, \phi)$. For example, if program $P$ contains an **if** statement of the form:

```
if(??_i){
        x = a;
}else{
        x = b;
}
```

then a trace produced by $sched(P)$ will either include the statements `assume ??`$_i$ and `x=a` or the statements `assume`$\neg$`??`$_i$ and `x=b`. In the first case, specializing the resulting trace with $\phi_f($`??`$_i) = 0$ will lead to a trace that is invalid for the program $P_{\phi_f}$ because program $P_{\phi_f}$ executes `x=b` unconditionally, but the specialized trace does not include `x=b`. Similarly, specializing the trace containing `assume`$\neg$`??`$_i$ and `x=b` with a control $\phi_t($`??`$_i) = 1$ will lead to a trace that is invalid for $P_{\phi_t}$, *i.e.* $PE(t_P, \phi_t) \notin tr(P_{\phi_t})$.

The way around this problem can be found in the program transformations from Section 7.3. That section showed that it was possible to eliminate all the control decisions from traces without affecting the semantics of the program by performing if-conversion on the sketch. For the code fragment above, if-conversion results in a code fragment where all the control flow has been moved inside atomic sections.

```
t = ??_i;
atomic{ if( t) x = a; }
atomic{ if(!t) x = b; }
```

A trace for this new code fragment no longer has to make an arbitrary decision about control flow, and therefore, specializing any trace with a control $\phi$ will lead to a valid trace for program $P_\phi$.

By eliminating the need for traces to make arbitrary decisions about control flow, if-conversion makes it possible to define a projection mechanism that satisfies the two requirements in the definition.

### 8.2.1 Mechanics of Trace Projection

The trace projection algorithm presented in this section is able to satisfy the properties established by the diagram in Figure 8.1 by transforming the original sketch $P$ into a semantically equivalent sketch $P'$ through if-conversion. After unrolling any loops in the program and applying the transformation rules from Section 7.3, the sketch is now a

straight-line sequence of atomic statements. This implies that when defining a trace for this sketch, the only non-deterministic decisions left for the *sched* procedure are those involved in interleaving the threads; all the non-determinism surrounding control flow decisions has been eliminated. Therefore, the trace projection algorithm only has to concern itself with finding a suitable ordering for the atomic steps from each of the threads.

The first input to the algorithm is a trace $t$ consisting of a list of steps, where each step is an atomic statement $t^i_n$ identified by an id $i$ and the thread $n$ which executed it. The second input is an if-converted sketch $P'$ containing a single **fork** statement.

```
void main(){
        pre;
        fork(int i; N){
                S;
        }
        post;
}
```

Now, because $P'$ has been if-converted, the expression $sched(S)$ is now fully deterministic; it simply breaks $S$ into a sequence of atomic steps $s_0 \rangle\!\rangle s_1 \rangle\!\rangle \ldots \rangle\!\rangle s_k$. Now, from Section 7.2.1, we know that a valid trace for $P'$ is defined by interleaving the steps from $sched(S)$ according to the equation below.

$$sched(\texttt{fork(t, N)}\ S) \rightarrow mix_N(sched(ren(S,0)), sched(ren(S,1)), \ldots, sched(ren(S,N-1)))$$

Any interleaving will lead to a valid trace, but if we want a trace that is a projection of the input trace $t$, the interleaving of the atomic steps must satisfy the partial order imposed by $t$. This can be expressed as a set of constraints on the interleaving. I use the notation $s^i_n$ to refer to refer to the $i^{th}$ atomic step in $sched(ren(S,n))$, and $t^i_n$ to refer to the $i^{th}$ step executed by the thread $n$ in trace $t$. Additionally, I define a function $map(t^i_m) = s^j_m$ that maps steps $t^i_m$ in $t$ to steps $s^j_m$ in $sched(ren(S,m))$. For the most part, each statement in $t$ has exactly one corresponding step in $sched(ren(S,m))$; however, recall that conditional atomics are modeled with two atomic steps: one to check the condition and possibly execute the action, and another one to check the assumption. Thus, if $t^i_m$ is a conditional atomic statement, then $s^j_m = map(t^i_m)$ is the step that contains the action, and $s^{j+1}_m$ is the step that checks the assumption. I also use the notation $s^i_n < s^j_m$ to indicate that step $s^i_n$ is

scheduled before step $s_n^j$ in the trace. With this notation, I define three constraints on the interleaving.

1. If step $t_n^i$ precedes step $t_m^j$ in the input trace $t$, then $map(t_n^i) < map(t_m^j)$ in $sched(P')$.
2. If $n = m \ \wedge \ i < j$ then $s_n^i < s_m^j$ in $sched(P')$.
3. If the trace $t_{\phi i}$ exposes a deadlock involving a set of conditional atomic steps $D = \{t_m^j, \ldots\}$, then if $t_m^j \in D$ and $t_n^j \in D$ corresponds to two steps in the deadlock set, then

$$s_n^{i'} = map(t_n^i) \ \wedge \ s_m^{j'} = map(t_m^j) \quad \Rightarrow \quad s_m^{i'} < s_m^{j'+1}$$

The first constraint ensures that the trace will preserve the partial order required by the input trace $t$. The second constraint is a constraint of the $mix_N$ function to ensure that the trace will preserve the program order for each of the threads.

The third constraint is more complicated; it is there to ensure that deadlock detection works properly, and is essential to guarantee that the projected trace will be able to eliminate the candidate that produced the original trace $t$. To see why this is the case, consider a sketch $P$ with a conditional atomic $\textbf{atomic}\ e \ \Rightarrow \ s$, and suppose that the validation procedure returns a trace that runs two threads and deadlocks on this conditional atomic. Now, let $t_0^i$ and $t_1^j$ correspond to the two steps involved in the deadlock, so $t_0^i = (\textbf{atomic}\ e \ \Rightarrow \ s)$ for thread 0, and $t_1^j = (\textbf{atomic}\ e \ \Rightarrow \ s)$ for thread 1.

The *sched* function for the conditional atomic produces two atomic steps as shown below; the first one corresponds to $s_n^{i'} = map(t_n^i)$ and involves the body of the conditional atomic, while the second one corresponds to $s_n^{i'+1}$ and uses an assumption to stop the evaluation of the trace if the conditional atomic is scheduled when it wasn't ready to run.

$$sched(\textbf{atomic}\ e \Rightarrow c1) \rightarrow \left( \begin{array}{l} \textbf{if}\ e \\ \textbf{then}\ (c1;\ t = 1) \\ \textbf{else}\ (\texttt{dl} = \texttt{dl} + 1;\ \textbf{assert}\ \texttt{dl} \neq N;\ t = 0) \end{array} \right) \rangle\!\rangle (\textsf{assume}\ t)$$

The reason for the third constraint on the ordering of the steps is that the deadlock detection in the steps $s_n^{i'}$ will cause an assertion failure only after step $s_n^{i'}$ has been executed by all threads $n$ involved in the deadlock. Therefore, if any of the `assume`s in steps $s_n^{i'+1}$ executes before all the threads in the deadlock have executed their deadlock detection code,

the deadlock will not be detected, and the projected trace will fail to eliminate the buggy candidate.

The trace that results from this process will have all the desired properties; partially evaluating the trace for the sketch with a control $\phi$ will lead to a valid trace for candidate $P_\phi$ that preserves the step ordering of the original candidate. This allows the projected trace to eliminate not just the buggy candidate that produced it, but a whole family of candidates. This in turn allows the CEGIS algorithm to converge in a small number of iterations, even for sketches with billions of unique candidates.

## 8.3 Related Work

The model checking community has had an interest in concurrent synthesis from the very beginning. In fact, Clarke and Emerson's seminal paper on model checking looked at the problem of synthesizing the synchronization logic in a program from temporal specifications [20]. Pnueli and Rosner [51] also studied the problem of concurrent synthesis from the point of view of synthesizing distributed reactive synthesis from a temporal specification. Compared with the resounding success of model checking, however, the work on synthesis from temporal specifications has had limited practical impact due to the computational complexity of the algorithms involved.

While computer-aided verification of concurrent programs has gained significant momentum in recent years, the automated *synthesis* of concurrent algorithms has had a slower start; most recent work in the field is designed for synthesis within in a specific domain of algorithms (*e.g.* [8]). Notable in this context is the work on synthesis of concurrent garbage collectors by Vechev *et al.* [66,67] In their earlier work [66], the authors apply an automated transformation-based space exploration to derive provably correct variants from a basic (correct) concurrent GC implementation. In a more recent work [67] an exhaustive exploration procedure is applied to a space of implementations on variants with varying degrees of atomicity and instruction reordering, and combined with effective pruning of vacuously incorrect implementation sub-spaces. In this approach the authors deploy a separate verification procedure based on the SPIN model checker [40] to check the absence of concurrency bugs in each of the generated candidate implementations.

More recently, they have generalized their approach to the synthesis of concurrent datastructures [65]. Their framework, unlike ours, is capable of verifying concurrent im-

plementations that manipulate arbitrary unbounded data structures, thanks to the use of abstraction in the verification procedure. This, however, is not an inherent limitation of our approach. Also, the generation method used in their approach heavily depends on tailored semantic rules to prune the search space effectively, and is restricted to a predefined set of concurrency-related transformations and synchronization primitives. In contrast, our projection procedure is able to achieve pruning without the need for domain specific knowledge, reducing the problem instead to a constraint solving problem and delegating the effort of conducting an effective search to an efficient, general purpose SAT-based solver.

# Chapter 9

# Empirical Evaluation of the SKETCH System

This section presents an empirical evaluation of the concurrent SKETCH language and the concurrent CEGIS algorithm from Chapter 8. Specifically, it evaluates the performance of the concurrency support in the SKETCH compiler and the expressiveness of the SKETCH language on a suite of benchmarks. For the benchmarks, we selected problems that are regarded as difficult to implement by practitioners in the field of concurrent programming. This section mirrors the evaluation section from our PLDI08 paper [62], but all the performance data has been updated to reflect the current state of the synthesizer. The results of our evaluation are encouraging.

- SKETCH successfully searched spaces of about $10^9$ syntactically unique candidates in about 10 minutes, consuming less than 800 MB of memory.
- Our CEGIS algorithm required only a few observations (meaning only a few calls to the verifier) to resolve a sketch, or determine that it could not be resolved. In our benchmarks, SKETCH required 10 iterations to find a correct implementation from a space of about $10^8$ possibilities. SKETCH was also able to show after only 7 observations that one of our benchmark sketches could not be resolved.
- The trace projection through if-conversion is shown to be crucial in generalizing CEGIS to concurrent programs. Without it, counterexample traces preserve too little information to make the approach effective.

The expressiveness of the SKETCH language is harder to evaluate, but we show example sketches of our benchmarks below and argue that they capture the insight behind a solution, with a minimum of unnecessary detail. These results suggest that programmers facing concurrency challenges might find SKETCH useful.

## 9.1 Overview of Experiments

For the evaluation, we ran several variations of five representative benchmarks on an IBM T60 laptop (the same one described in Chapter 6). The goal of the experiments was to validate the overall effectiveness of the sketching approach to concurrent data structures, as well as some of the individual design decisions in the concurrency extensions to SKETCH. Specifically, the experiments attempt to validate the following hypothesis.

*Synthesis scales to realistic synthesis problems.* This hypothesis is validated by using the synthesizer to solve sketches for difficult programming problems. The sketches leave unspecified most of the challenging details in these benchmarks, leaving only what is necessary to express the clever insights behind the desired implementations. For a few sketches, we show different versions of the sketch to explore how the number of holes in the sketch affects the solution time.

*The bounded observation hypothesis holds.* The bounded observation hypothesis is central to the CEGIS algorithm. This section shows that the number of observations required to resolve a sketch (or show that it cannot be resolved) is not dramatically larger than in the sequential case.

*Our trace projection algorithm is effective at preserving error information.* Section 8.2 established a set of requirements on the projection algorithm that were meant to ensure preservation of errors by the projected trace. We evaluate the effectiveness of the resulting projected algorithm by comparing it with a simpler alternative, one that does not perform if-conversion on the sketch and therefore produces traces with many `assume` statements. We show that the resulting traces preserve too little information, causing the CEGIS algorithm to exhaust the available resources before converging.

*The* SKETCH *language is expressive for this domain.* We do not attempt to measure this quantitatively; instead, we show how we expressed the insights behind our benchmarks using SKETCH.

| Sketch | Description | $\|C\|$ |
|---|---|---|
| `queueE1` | Lock-free queue: restricted `Enqueue()` | 4 |
| `queueE2` | Lock-free queue, full `Enqueue()` | $10^6$ |
| `queueDE1` | `queueE1`, plus sketched `Dequeue()` | $10^3$ |
| `queueDE2` | `queueE2`, plus sketched `Dequeue()` | $10^8$ |
| `barrier1` | Sense-reversing barrier, restricted | $10^4$ |
| `barrier2` | Sense-reversing barrier, full | $10^7$ |
| `fineset1` | Fine-locked list: `find()`, `add()` and `remove()` | $10^9$ |
| `lazyset` | Lazy list, singly-locked `remove()` | $10^3$ |
| `dinphilo` | Approximation of dining philosophers problem | $10^6$ |

Table 9.1: Summary of benchmark sketches. $C$ is the set of syntactically distinct candidate programs encoded by each sketch.

### 9.1.1 Benchmarks

The benchmarks are intended to represent various sketching scenarios across different problems. Table 9.1 summarizes the more detailed descriptions of the benchmarks that follow.

#### Lock-free queue

This benchmark came from an undergraduate exam in an Operating Systems class held at Berkeley in 2005. The problem looks simple, but less than 30% of the students solved it correctly, even with additional hints from the ones described here. The exam described the problem as follows:

> An object such as a queue is considered "lock-free" if multiple processes can operate on this object simultaneously without requiring the use of locks, busy-waiting, or sleeping. We will construct a lock-free FIFO queue using an atomic "swap" operation. This queue needs both an `Enqueue` and a `Dequeue` method.
>
> Instead of the traditional `Head` and `Tail` pointers, we will have `PrevHead` and `Tail` pointers. `PrevHead` will point at the last object returned from the queue, so `PrevHead.next` will point to the head of the queue. Here are the basic class definitions, under the assumption that only one thread accesses the queue at a time.

```
// Holding cell for an entry
class QueueEntry {
  QueueEntry next = null;
  Object stored;
  int taken = 0;
```

```
  QueueEntry(Object newobject) { stored = newobject; }
}
// The actual Queue (not yet lock-free!)
class Queue {
  QueueEntry prevHead = new QueueEntry(null);
  QueueEntry tail = prevHead;
  void Enqueue(Object newobject) {
    QueueEntry newEntry = new QueueEntry(newobject);
    tail.next = newEntry;
    tail = newEntry;
  }
  Object Dequeue() {
    QueueEntry nextEntry = prevHead.next;
    while (nextEntry != null && nextEntry.taken == 1)
      nextEntry = nextEntry.next;
    if (nextEntry == null)
      return null;
    else {
      nextEntry.taken = 1;
      prevHead = nextEntry;
      return nextEntry.stored;
} } }
```

Suppose that we have an atomic swap instruction that takes a local variable (register) and a memory location and swaps their contents. In a relaxed dialect of Java that allows pointers, it can be described as follows.

```
Object AtomicSwap(variable addr, Object newValue) {
  Object result = *addr;  // Get old value (object)
  *addr = newValue;       // Store new object
  return result;          // Return old contents
}
```

**Problem (a).** Using the `AtomicSwap()` operation, rewrite code for `Enqueue()` such that it will work for any number of simultaneous `Enqueue` and `Dequeue` operations. You should never need to busy wait. Do not use locking (*e.g.*, test-and-set lock). Although tricky, it can be done in a few lines.

**Problem (b).** Rewrite code for `Dequeue()` such that it will work for any number of simultaneous threads working at once. Again, do not use locking. You should never need to busy-wait. □

For the enqueue method in this queue, we wrote a sketch, shown below, that embodies the limited information about the implementation provided in the problem. Lines 1 and 2 in the sketch express our knowledge that the solution must first allocate a new object.

Lines 5 and 7 express our knowledge that the solution will involve one or possibly two pointer assignments to a handful of locations including `tail`, `tail.next` and possibly `newEntry.next`. Line 6 expresses the requirement that the solution must use the `AtomicSwap` construct. The **`reorder`** and the regular expression generators are used to express our ignorance about the precise use of the `AtomicSwap`, and the exact way in which the assignments must take place.

```
#define aLocation {| tail(.next)? | (tmp|newEntry).next |}
#define aValue    {| (tail|tmp|newEntry)(.next)? | null |}
#define anExpr(x,y)  {| x==y | x!=y | false |}

void Enqueue(Object newobject) {

1   QueueEntry tmp = null;
2   QueueEntry newEntry = new QueueEntry(newobject);

4   reorder {
5     aLocation = aValue;
6     tmp = AtomicSwap(aLocation, aValue);
7     if (anExpr(tmp, aValue)) aLocation = aValue;
8   }

}
```

We also wrote an alternative sketch for enqueue containing only 4 choices to analyze how the algorithm scales with the number of unknowns in the sketch.

For `Dequeue`, the sketch attempts to implement the method with a single **`while`** loop. The sketch was written in a few minutes, and it is very simple. The sketch places in a reorder block all the statements that one could reasonably expect to be necessary for the solution and leaves the rest to the synthesizer. For example, we know we will need the atomic swap to read the `taken` bit and possibly set it (line 9); we know `prevHead` must be updated (line 8), we know we will probably need a temporary pointer which must be updated in each iteration (line 6), and we know that in some cases there will be nothing to dequeue and we will have to just return null (line 7). The reorder block is essential to allow the synthesizer to discover the right order for these actions.

```
1  Object Dequeue() {
2    QueueEntry tmp = null;
3    boolean taken = 1;
4    while (taken) {
5      reorder {
6        tmp = {| prevHead(.next)?(.next)? |};
7        if (tmp == null)  return null;
8        prevHead = {| (tmp|prevHead)(.next)? |};
9        if (!tmp.taken)  taken = AtomicSwap(tmp.taken, 1);
10     }
11   }
12   return tmp.stored;
13 }
```

The `queueE1` and `queueE2` versions of this benchmark use the simple and hard `Enqueue` sketches respectively, and they both use a full implementation of `Dequeue`. The `queueDE1` and `queueDE2` combine the `Dequeue` sketch shown above with the simple and hard enqueue sketches respectively.

The queue benchmarks were resolved with respect to the conjunction of the following correctness conditions:

- *Sequential consistency of each individual operation* [44]. If a thread $A$ enqueues $a_1$ and $a_2$, then $a_1$ must be come before $a_2$ in the queue. Note that the queue is *not* sequentially consistent with respect to both enqueue and dequeue; *i.e.* it is possible for a thread to perform an enqueue followed by a dequeue and have the queue behave as if the dequeue was executed first. Sequential consistency does not compose, so an object may be sequentially consistent with respect to enqueue and with respect to dequeue, but that does not mean it is sequentially consistent with respect to enqueue and dequeue together. This is one of the reasons why sequential consistency is a weaker condition than linearizability [39].

- *Structural integrity.* The queue is not corrupted by concurrent operations. Specifically: (1) the head and tail are not `null`; (2) `prevHead.taken == 1`; (3) the tail is reachable from the head; (3) `tail.next == null`; (4) there are no cycles in the queue; (5) no "untaken" nodes precede "taken" ones.

SKETCH also enforces memory safety by default: no `null` pointers may be dereferenced, and array accesses must be within bounds. It is worth noting that for `queueE2` and `queueDE2`, we found that we had to use more than one operation per thread or more than two threads for verification in order to get solutions that generalized to more threads and more operations per thread.

### Sense-reversing barrier

Barriers allow multiple threads to synchronize at the same program point before continuing. A correct implementation must allow the last thread to reach the barrier to realize that it is the last thread and to release the other threads waiting at the barrier. However, once awaken, the other threads should only be allowed to pass through the barrier once; they should be stopped upon reaching the barrier again.

This is more difficult than it may appear at first sight. It is tempting to use a counter to count how many threads have reached the barrier, forcing the threads to wait until the count equals the number of threads, and then releasing the threads and resetting the counter. However, this scheme does not work because once released, the threads could run past the barrier an arbitrary number of times before the counter is reset.

The insight to solving this problem is to separate consecutive barrier points into two phases: even and odd. The phase is called the barrier's "sense," and reverses after each barrier point [36]. The barrier object keeps the global boolean sense, and each thread has a local sense. Having even and odd barriers for consecutive points allows us to give a thread permission to pass through the even sense while preventing it form going past the odd sense.

However, this insight is far from an implementation. The barrier code requires subtle reasoning about interleaved threads and intermediate barrier states. We claim that the SKETCH language is well suited to capturing the insight behind a sense-reversing barrier. Below, we sketch the barrier's `next()` method. The sketch encodes `next()` as a "soup" of operations, to be executed (or not) under some conditions on the barrier state. The synthesizer is left to find an implementation that avoids harmful races, deadlocks, and other intricate details.

The first step is to write the `Barrier` datastructure; it must cointain (1) a `sense` to indicate the current phase; (2) `senses`, an array with the local sense of each thread; and (3) `count`, the number of threads yet to reach the barrier. The soup of operations

comprising the insight behind `next()` included the following actions:

1. Update the thread's own sense of the barrier.
2. Atomically decrement the count of threads yet to arrive.
3. Under some condition, wait until the barrier sense changes to some predicate of the thread's own sense.
4. Under some condition, set the barrier's sense and yet-to-arrive count so as to wake up the other threads, and prepare the barrier for the next phase.

Before finishing the sketch, we define "under some condition" as a SKETCH generator function that returns a boolean expression of its arguments:

```
boolean predicate (a, b, c, d) {
  return {| (!)? (a==b | (a|b)==?? | c | d) |};
}
```

Now, translating the operations above into a sketch is straightforward. We make them into a "soup" by placing them in a **reorder** block:

```
void next (Barrier b, Thread th) {
  boolean s = b.senses[th];
  s = predicate (0, 0, s, s);
  int cv = 0;
  boolean tmp = 0;
  reorder {
    // (1) Update t's local sense
    b.senses[th] = s;
    // (2) Decr. count of yet-to-arrive threads
    cv = AtomicReadAndDecr (b.count);
    // (3) Wake up other threads, reset barrier
    tmp = predicate (b.count, cv, s, tmp);
    if (tmp) {
      reorder { // The order of these operations is important
        b.count = N;
```

```
          b.sense = predicate (b.count, cv, s, s);
        }
      }
      // (4) Wait at barrier
      tmp = predicate (b.count, cv, s, tmp);
      if (tmp) {
        boolean t = predicate (0, 0, s, s);
        atomic (b.sense == t);
      }
    }
  }
```

The benchmark `barrier2` is the sketch shown above. The companion `barrier1` is a reduced version with a smaller candidate program space. The barrier's correctness was established by a client program that ensured that threads always joined properly at each barrier point, together with the implicit deadlock check performed by SKETCH. This client program launched $N$ threads that reached a barrier $B$ times. Before waiting at the $b^{th}$ invocation of `next()`, each thread $t$ set a bit `reached[t][b]`. After passing through the $b^{th}$ call to `next()`, each thread ensured that its two neighbors $t - l$ and $t + 1$ also reached the $b^{th}$ barrier by asserting `reached[t-l][b] && reached[t+l][b]`.

### Finely locked, list-based set

This is the benchmark presented in Section 1.2.2 in the introduction. It is our largest concurrent benchmark in terms of the number of unknowns. The benchmark implements a set as a sorted linked list. The implementation uses a hand-over-hand locking strategy, where the program maintains a sliding window of locked nodes as it traverses the list, allowing concurrent modifications to disjoint areas of the list.

It is important to note that the version of this benchmark evaluated in the thesis differs significantly from the version evaluated in the PLDI 08 paper [62]. Thanks to the improvements in the quality of the implementation, I was able to write a sketch that is much less constrained compared to the one in [62].

### Singly-locked remove() method of lazy list

This is a problem proposed by [38]. Its basis is a lazily-updated, list-based set data structure due to [35]. The `add()` and `remove()` methods of this set are optimistic, in that they traverse the data structure without locking. Only when the list is to be modified do they check that the their view of the list is still valid. Both `add()` and `remove()` acquire two locks before modifying the list.

This problem asks whether the list's `remove()` method can be modified to take only one lock, instead of two (the answer is "no"). We translated this problem into a sketch for SKETCH to solve by first removing the **`lock`** statements from the original `remove()` method. Next, we gave SKETCH the freedom to **`lock`** any one of a set of nodes at any point in the body of the stripped-down `remove()`, and likewise for **`unlock`**. The correctness criteria for this sketch were the same as for the `fineset*` benchmarks.

When we ran this benchmark with two threads performing both `add` and `remove`, the synthesizer returned "NO", as expected. Surprisingly, SKETCH *was* actually able to find a solution that worked for the case where one thread performs only adds and another thread performs only removes.

### Dining philosophers

This problem has $P$ philosophers at a circular table, with a plate of spaghetti in the center. A philosopher needs two chopsticks to eat. Each philosopher has chopsticks at his left and right, but because the table is circular, there are only $P$ total chopsticks. The problem is to find a chopstick-acquisition policy which avoids deadlock, in which no philosopher can eat; and starvation, in which particular philosophers cannot eat. Thus, we want a resource policy that satisfies the properties (1) some philosopher can always eat; and (2) every philosopher will always eventually eat.

We modeled the problem in SKETCH as follows: there are $P$ philosophers encoded as a **`fork(int p; P)`** block, each contending for its left and right of $P$ locks. The philosophers attempt to eat $T$ times, blocking if they cannot acquire their left and right chopsticks. The resource acquisition policy was sketched as an expression of $t$, $p$, $P$, which indicated whether the right or left chopstick should be acquired first. The order in which the chopsticks were released was also left unspecified. As to correctness, SKETCH implicitly enforces property (1) above by ensuring that the execution is deadlock free. As we described earlier, we can

only enforce livenes properties by approximating them as a safety property in a bounded execution. Our sketch approximates property (2) by ensuring that all philosophers are able to eat $T$ times in the $P * T$ steps of the execution. With this sketch and this correctness conditions, the synthesizer was able to produce a correct implementation of the protocol; a minor variant over the standard solution presented in textbooks [58].

## 9.2 Overall Performance of the SKETCH Synthesizer

We ran the synthesizer on each benchmark using test harnesses with various numbers of threads and operations, and with different patterns of operations when possible. The particular tests of the `queue*`, `fineset*`, and `lazyset` benchmarks are labeled with the following scheme: a test named $ed(ed|ed)$ means that first a sequential enqueue $e$ was performed , next a sequential dequeue $d$, and finally two threads were forked to each perform an enqueue then dequeue $(ed|ed)$. The set tests use the same scheme, with $a$ and $r$ standing for "add" and "remove", respectively. For each test, we gathered the following data:

- *Itns* – the number of observations required for CEGIS to terminate.
- $S_{total}$, $V_{total}$ – total amount of time spent in synthesis and validation, respectively.
- $\%S_{solve}$, $\%V_{solve}$ – for synthesis, the percentage of time spent in the SAT solver as opposed to the preprocessing phases. For validation it is the percentage of time spent in SPIN.
- *Time:Total* – total elapsed time between invoking SKETCH and it returning an answer. This time does not equal $S_{total}+V_{total}$ because part of the time is spent in our compiler frontend.
- $S_{mem}$, $V_{mem}$ – the amount of memory used by the inductive synthesizer and the validator respectively.
- *Memory:Total* – the maximum memory used by the synthesizer, verifier, and SKETCH. The maximum total memory includes memory used by our Java frontend.

The performance results are tabulated in Table 9.2. The results are an improvement compared to the results we reported in [62], even though the current results were run on a

| | Test | Itns | Time (s) | | | | | Maximum Memory (MB) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Total | $S_{total}$ | $\%S_{solve}$ | $V_{total}$ | $\%V_{solve}$ | Total | $S_{mem}$ | $V_{mem}$ |
| barrier1 | $N=3, B=2$ | 8 | 33.0 | 10.13 | 23.3% | 11.38 | 1.8% | 39.02 | 22.51 | 3.09 |
| | $N=3, B=3$ | 10 | 47.6 | 17.08 | 21.3% | 14.56 | 4.1% | 51.67 | 28.44 | 5.43 |
| | $N=4, B=3$ | 13 | 112.2 | 43.09 | 18.4% | 37.09 | 49.5% | 114.23 | 48.53 | 77.10 |
| barrier2 | $N=3, B=3$ | 35.2 | 1948.8 | 1558.21 | 66.7% | 52.43 | 1.3% | 347.23 | 248.84 | 4.30 |
| | $N=4, B=3$ | 23 | 496.8 | 313.42 | 32.6% | 42.33 | 29.0% | 186.27 | 131.61 | 56.21 |
| dinphilo | $N=3, T=5$ | 4.1 | 33.9 | 17.55 | 41.4% | 5.81 | 8.8% | 70.71 | 43.62 | 4.26 |
| dinphilo | $N=4, T=3$ | 3.8 | 29.7 | 12.50 | 31.3% | 8.96 | 46.8% | 58.98 | 38.88 | 14.22 |
| dinphilo | $N=5, T=3$ | 6 | 231.5 | 46.88 | 27.6% | 158.87 | 95.2% | 314.73 | 65.35 | 268.57 |
| fineset1 | $ar(aaaa\|rrrr)$ | 4 | 535.30 | 371.64 | 10.6% | 68.29 | 2.3% | 991.75 | 638.86 | 7.58 |
| | $ar(ar\|ar)$ | 2 | 130.5 | 88.46 | 26.6% | 13.20 | 1.3% | 582.18 | 502.14 | 3.09 |
| | $ar(arar\|arar)$ | 5 | 420.4 | 306.33 | 10.5% | 37.46 | 5.4% | 776.33 | 637.84 | 8.56 |
| | $ar(ar\|ar\|ar)$ | 4 | 308.5 | 214.54 | 14.4% | 36.59 | 29.4% | 706.44 | 567.34 | 25.94 |
| | $ar(a\|r\|a\|r)$ | 2 | 138.0 | 88.11 | 22.9% | 20.45 | 37.9% | 629.56 | 515.46 | 15.59 |
| fineset2 | $araa(r\|r)$ | 6 | 486.0 | 392.72 | 11.7% | 22.98 | 0.5% | 787.80 | 727.77 | 2.70 |
| | $ar(a\|r)$ | 1 | 46.9 | 18.51 | 31.3% | 9.83 | 0.4% | 185.16 | 150.28 | 2.89 |
| lazyset | $ar(aa\|rr)$ | 16.2 | 169.6 | 36.91 | 13.8% | 53.66 | 3.2% | 261.44 | 37.66 | 4.06 |
| buggy lazyset | $ar(ar\|ar)$ | 9 | 95.04 | 19.93 | 22.3% | 28.97 | 2.4% | 200.71 | 33.31 | 3.28 |
| queueDE1 | $ed(ed\|ed)$ | 2 | 14.3 | 3.18 | 37.6% | 7.01 | 10.2% | 43.56 | 28.04 | 4.84 |
| | $ed(ee\|dd)$ | 2 | 10.0 | 1.03 | 31.7% | 5.70 | 1.0% | 21.05 | 11.59 | 2.89 |
| queueDE2 | $ed(ed\|ed)$ | 13 | 3602.7 | 3448.89 | 94.0% | 39.26 | 5.0% | 438.30 | 288.51 | 6.60 |
| | $ed(ee\|dd)$ | 4 | 39.5 | 20.81 | 65.8% | 10.14 | 0.7% | 72.83 | 43.25 | 2.89 |
| queueE1 | $ed(ed\|ed)$ | 1 | 8.1 | 0.10 | 25.1% | 5.94 | 10.4% | 11.63 | 6.70 | 4.26 |
| | $ed(ee\|dd)$ | 1 | 5.7 | 0.10 | 25.0% | 3.83 | 0.8% | 11.63 | 6.70 | 2.89 |
| | $ed(e\|e\|e)ddd$ | 1 | 8.2 | 0.25 | 22.7% | 6.09 | 46.6% | 20.13 | 7.87 | 15.20 |
| queueE2 | $ed(ed\|ed)$ | 3 | 16.7 | 2.61 | 19.1% | 9.85 | 12.4% | 23.00 | 15.03 | 6.60 |
| | $ed(ee\|dd)$ | 2 | 10.8 | 1.62 | 25.3% | 5.81 | 0.8% | 24.17 | 14.87 | 2.89 |
| | $ed(e\|e\|e)ddd$ | 10.5 | 192.0 | 130.08 | 57.0% | 28.30 | 25.8% | 111.23 | 79.16 | 27.74 |

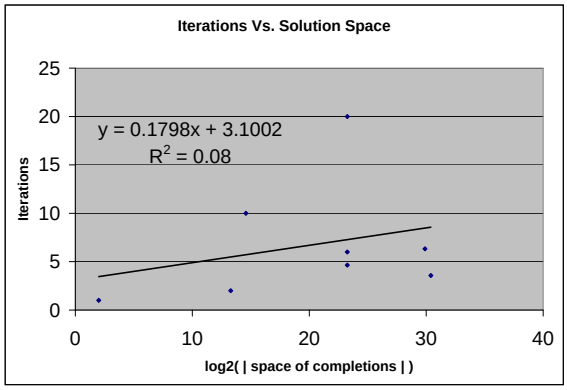
Table 9.2: Performance results.

much more limited machine. While the PLDI08 results were run on a 2GHz Core 2 Duo with 2 GB of RAM, the results in this thesis were all run on a 1.66 GHz laptop with only 1 GB of RAM.

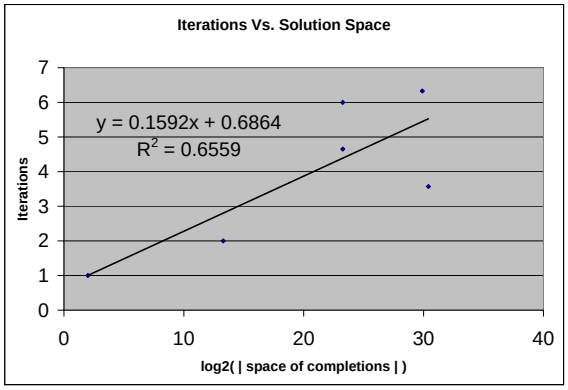
A few important features can be noticed in the data. First, for most benchmarks, the synthesis time is clearly dominant, in many cases by a large factor. This contrasts with the PLDI08 results where the validation time clearly dominated. The reason for this difference is that in the PLDI08 implementation, the validation phase would replace all the holes in the sketch with integer values, and then pass the resulting code to SPIN without any further preprocessing. In the new implementation, the synthesizer partially evaluates the sketch with respect to the candidate control $\phi$, and performs elimination of transitive assignments and dead code elimination on the candidate implementation before passing it to SPIN. This significantly reduced the time required to build the SPIN model, as well as the actual solution time in the validation phase.

A second important feature in the data is the impact that the test client has on the performance of the solver. It makes sense that a more elaborate client that tests more operations with more threads will cause the validation time to increase significantly. However, more elaborate clients cause an important increase not just in the validation time, but also in the synthesis time. We saw a similar effect in Section 6.2.2, where increasing the bounds in the test harness caused the synthesis time to go up. We hypothesize that the reasons for this are very similar. More elaborate test harnesses lead to more elaborate counterexamples which make the synthesis problem harder.

A final observation has to do with the quality of the current implementation. One can see that most benchmarks spend only a small fraction of their time in the solvers. Instead, a large fraction of the time is spent in the frontend, and in the loading and preprocessing of circuit representations by the synthesizer. The problem stems from the way the concurrent solver was built from the sequential SKETCH solver. In the sequential solver, the evaluation of the denotation function $\mathcal{C}[\![P]\!]^{\tau_\emptyset}$ could afford to be inefficient because it ran only once. But the way the concurrent SKETCH solver was constructed, the denotation function is evaluated on each iteration. Moreover, the implementation currently can not maintain the symbolic set of candidates from one CEGIS iteration to the next because the inductive synthesizer is launched as a new process on each iteration of the CEGIS loop, so the denotation function is actually evaluated on a concatenation of all the traces seen so far, not just the last one. This makes the process very inefficient and leads to a lot of




(a) For all benchmarks




(b) Excluding `barrier1` and `2`

Figure 9.1: Candidates Vs. Observations

wasted time and memory. We have actually found this to be one of the biggest obstacles to solving bigger sketches: all the overhead exhausts the available memory. Surprisingly, even with all these inefficiencies, the synthesizer was still powerful enough to synthesize correct implementations from the sketches shown earlier.

## 9.3 Trace projection through if-conversion

One of the central questions to evaluate is the effectiveness of the CEGIS approach in finding a small number of representative traces that will allow the inductive synthesizer to generate a correct implementation. Specifically, we want to answer two concrete questions. First, we want to evaluate how the number of iterations scales with the size of the candidate space and compare it with the scaling in the sequential setting. Second, we want to determine to what extent this scaling was enhanced by the way we defined trace projection.

For the sequential CEGIS, we found that the number of iterations was approximately proportional to the *log* of the solution space. For concurrent CEGIS, this continues to be the case, but the correlation is now much weaker. This is clear already in Table 9.2 from the fact that using a different test harness on the same sketch leads to very different iteration counts. Graph (a) in Figure 9.1 compares the number of iterations to the *log* of the solution space for the sketch. The graph shows that the correlation is very weak compared to what it was in the sequential case. Part of the problem is in the barrier sketches, which take many iterations given the size of their candidate spaces. Graph (b) shows the correla-

tion excluding the barrier sketches; the correlation has improved, but the scaling factor is much worse than for the sequential case. Whereas in the sequential case each iteration cut the solution space by an average factor of $2^{14}$, in the concurrent case each iteration cuts the solution space only by an average factor of about $2^{6.3}$. This is still a big factor, but it shows that each counterexample is not providing nearly as much information as they provided in the sequential case.

The projected traces may not have the elimination power of counterexample inputs in the sequential setting, but they are a significant improvement over more naïve projection strategies. I tried to compare the projection scheme from Section 8.2 with a more naïve projection scheme that does not involve if-conversion. Instead, the counterexample trace is projected into a trace for the sketch that uses assumes to track the direction of each branch taken by the trace. This makes traces very brittle: any change to the control vector that causes a change in the control flow makes the trace invalid from the point of the change onward.

I implemented this naïve projection scheme and the resulting algorithm proved to be extremely inefficient, just as I expected. With the naïve projection, only the dining philosophers benchmark could be resolved, and it took on average twice as many iterations and twice as much time. All the other benchmarks would either run out of memory, or in the case of the `barrier1` benchmark, exceed their time limit after 30 iterations.

## 9.4 Conclusions

The current results on the concurrent SKETCH synthesis system are very encouraging. We have been able to use the synthesizer to write difficult implementations for well-known programming problems. We have also shown that a handful of carefully selected traces can provide enough information about a program to allow you to synthesize many of the low-level details. This is an interesting result, and could have potential applications for other forms of analysis; for example, it is easier to reason about a concurrent object in the context of a larger program if we know that only a small number of interleavings are likely to produce distinct behaviors. That said, the current implementation is still far from achieving its full potential.

The relatively poor performance of the concurrent SKETCH synthesizer compared to its sequential counterpart suggests that there is a lot of room for improvement. As we

have seen, a big problem is the implementation itself. For the problems we ran, only a small fraction of the time was spent on the SAT solver or the model checker; instead, a lot of time was spent performing repeated work. For example, on every iteration of the loop, the denotation function is applied to a concatenation of all the traces produced so far in order to compute the control set $\Phi_i$ from scratch, instead of computing it incrementally from $\Phi_{i-1}$ as the formal description of the algorithm suggests.

In addition to the implementation issues, there is a lot of room for improvement in the projection algorithm to allow it to preserve even more information from one candidate to another. The ordering constraints from Section 8.2 that define the projection algorithm are very lose; many different total orders will satisfy these ordering constraints, and some of these orders have more elimination power than others. I believe part of the variability in the number of iterations comes from the arbitrary decisions that the synthesizer has to make about the total order of statements in the projected trace.

That said, our early results appear to validate the Sketching approach to synthesis. Moreover, the theory we have develop provides a solid foundation to move this research forward.

# Part IV

# Domain Specific Sketching for Stencils

# Chapter 10

# Motivation for Domain Specific Sketching

Sketches provide a novel mechanism for communicating user insight while leaving implementation mechanics unspecified. One of they key features of sketching is its generality. The solution algorithm doesn't rely on any form of domain knowledge, and is thus able to handle arbitrary sketches with remarkable efficiency. Nevertheless, there are classes of problems for which a lot of domain knowledge is available. For these problems, we would like to have synthesizers that can exploit this domain knowledge to synthesize solutions faster, but without compromising the advantages of the sketching approach.

I decided to test this idea on the domain of *stencil computations*. Stencils constitute a class of scientific computations with broad application in areas as diverse as signal processing, fluid dynamics and economics. They are very easy to specify but can be devilishly difficult to implement, as efficient implementations often require the orchestration of many low level details. This alone makes them a very good candidate for the sketching approach.

Moreover, they are a great candidate for domain specific sketching. On the one hand, good sketches for many benchmarks in this domain are too complex for the SKETCH synthesizer to handle, even under very small bounds on the input sizes. On the other hand, even the complex implementations exhibit a very regular structure which can be exploited by domain specific synthesis procedures.

This section gives a precise characterization of the stencil domain, and shows some

common implementation techniques which are difficult to program because of the many details they involve. The section shows how sketching can be used to easily implement many of these strategies by leaving many details unspecified. Finally, the section also provides some high level arguments for why these sketches are hard to solve by the standard SKETCH synthesizer.

## 10.1 Characterization of the Stencil Domain

In the scientific computing literature, a stencil is a nearest-neighbor computation on a grid, where the new value of a grid entry is computed as a function of the old values of some of its neighbors. Stencils are generally classified by the number of neighbors they consider and the dimensions of the grid. For example, a typical four-point stencil in two-dimensions computes a value $a^{\text{new}}_{i,j}$ as a linear combination of the values $a^{\text{old}}_{i+1,j}$, $a^{\text{old}}_{i,j+1}$, $a^{\text{old}}_{i-1,j}$ and $a^{\text{old}}_{i,j-1}$. This paper uses a broader definition of stencils: we define a stencil to be a function that computes each element of an output grid by performing constant time operations on a bounded number of input grid elements. Therefore, our definition includes operations like transposition, which are not normally considered stencils.

Stencils form the core of many scientific applications; in particular, most PDE solvers work through repeated applications of different stencils. Stencils are also important in signal processing, image analysis and even compression; for example, the wavelet transform that forms the basis of the JPEG2000 image compression standard is implemented as a sequence of stencil computations. Our broader definition of stencils also covers data permutations, such as matrix transpositions, and even data-dependent permutations such as scatter and gather operations.

As befits such an important class of problems, great efforts have been expended in automatically identifying and optimizing stencils, particularly in languages for high-performance computing, such as HPF [53] and ZPL [60]. Fully automatic optimization approaches, however, are constrained by the set of transformations built into the compiler, as well as the analysis and heuristics used to decide if it is possible and convenient to apply a given one. These constraints are relevant for production codes because stencil optimization is still an area of active research [30,42,43,56]. For this reason, many production-level stencil codes are still hand-tuned in Fortran and C++.

Hand-optimized stencil implementations can be fiendishly complicated despite their

relatively small size. As one of the authors can attest from personal experience, one can easily spend several days hunting for a bug in two hundred lines of this low-level code. The complexity in these implementations arises from a large number of low-level expressions controlling nested looping and multidimensional indexing. These expressions have little intuitive meaning for the programmer because they do not resemble the specification. To make matters worse, programming errors may have subtle effects which are hard to spot. For example, it is possible for iterative algorithms that work by repeated applications of a stencil to produce the correct answer even when coded with a buggy stencil; they algorithm may just take much longer to converge.

For these reasons, the domain of stencils is well suited for sketching: stencil specifications can usually be stated cleanly and concisely, in a few dozen lines of code, and the low-level expressions which complicate the implementations can be efficiently synthesized by the compiler. As the following section will argue, many of the complexities involved in implementing efficient stencil implementations are nicely localized in the code, and thus easy to leave unspecified for the synthesizer to discover.

## 10.2 The Complexity of Stencil Implementations

The easiest way to implement a stencil is with a set of nested loops that compute an output grid from an input grid point by point. As a simple example, consider a 2-point 1-dimenstional stencil computed according to the equation $X_i^t = X_{i-1}^{t-1} + X_{i+1}^{t-1}$. A trivial implementation is shown below.

```
void sten1d(float[N] in, float[M,N] X) {
  for (int i = 0; i < N; ++i)
    X[0, i] = in[i];
  for (int t = 1; t < M; ++t)
    for (int i = 1; i < N-1; ++i)
      X[t, i] = X[t-1, i-1] + X[t-1, i+1] ;
}
```

The first step in a more efficient implementation is to divide the computation into blocks, so rows are computed $T$ at a time instead of one by one.

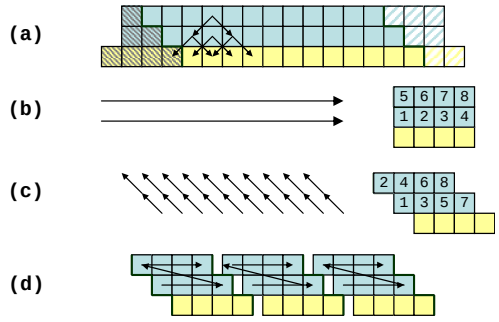



Figure 10.1: (a) Grid with some of the data dependencies. The regions in the two ends correspond to the corner cases which the time-skewed implementation will have to handle differently. (b) Iteration direction for the spec. (c) Iteration direction for the time-skewed implementation. (d) Iteration direction for the base case of the cache-oblivious scheme.

```
int T = 3;
void sten1d(float[N] in, float[M,N] X) {
  for (int i = 0; i < N; ++i)
    X[0, i] = in[i];
  for (int t = 1; t < M; t=t+T)
    innerSten(X[t::T, 0::N]);
}
void innerSten(float[T,N] X){
  for (int t = 0; t < T; ++t)
    for (int i = 1; i < N-1; ++i)
      X[t, i] = X[t-1, i-1] + X[t-1, i+1] ;
}
```

This alone doesn't improve performance, and is also not that hard to implement, especially if we are free to make the block size a multiple of the input size. However, this enables an optimization known as time-skewing [70], which greatly improve the cache efficiency of `innerSten`.

The time-skewing transformation requires a change int the iteration pattern from the one in Figure 10.1(b) to the one in Figure 10.1(c). Note that in order to preserve the dependencies, the traversal now takes place along diagonals, which forces the cells close to the boundary to be treated as special cases, as shown in Figure 10.1(a). It is easy to see that one must write three separate loop nests, two for the corner cases and one for the steady state. However, determining the exact expressions to use for the loop iteration bounds is a challenging task, especially for the corner cases, where the iteration bound for the inner

loop depends on the outer loop. We can express the implementation idea in a sketch that leaves all these details unspecified.

To understand the sketch shown below, recall that in SKETCH, generators are inlined into their call site before the holes are replaced with constants. In effect, we can think of generators as macros (see Section 2.2). In the sketch below, `linexpG` is a generator that produces expressions involving sums and differences of its arguments, and the generator `loopNest` produces loop-nests with arbitrary loop conditions, but which follow the diagonal pattern we desire.

```
generator int linexpG(int a, b, c = 0) {
  rv = ??;
  if (??) rv = {| rv (+ | - ) a |};
  if (??) rv = {| rv (+ | - ) b |};
  if (??) rv = {| rv (+ | - ) c |};
  return rv;
}
generator void loopNest(float[T,N] X) {
  for (int i = linexpG(N, T); i < linexpG(N, T); ++i)
    for (int t = linexpG(N, T, i); t < linexpG(N, T, i); ++t)
      X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
}
void innerStenSK(float[T,N] X) implements innerSten{
  if (N >= 3) {
    loopNest(X);  // generate left corner case
    loopNest(X);  // generate steady-state loop
    loopNest(X);  // generate right corner case
  } else
    sten1d(in, X);  // optimization inapplicable
}
```

The key idea of the implementation is expressed in the sketch by stating that $X^t_{i-t} = X^{t-1}_{i-t-1} + X^{t-1}_{i-t+1}$, but the low-level details of the loop iteration bounds are left for the compiler to discover. As an added benefit, the sketch spares the programmer from the error-prone task of having to code the three cases separately. Instead, all three loops are synthesized from the generator `loopNest`.

This sketch resolves to the correct implementation in less than 4 minutes, and produces the code shown below. The synthesized expressions are underlined.

```
void innerStenSK(float[T,N] X) implements innerSten{
  if (N >= 3) {
    for (int i = 0; i < T; ++i)
      for (int t = 1; t < i; ++t)
        X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
    for (int i = T; i < N; ++i)
      for (int t = 1; t < T; ++t)
        X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
    for (int i = N; i < N+T; ++i)
      for (int t = i-N+2; t < T; ++t)
        X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
  } else
    sten1d(in, X);
}
```

Another strategy that has been proposed for optimizing stencils is the use of recursive partitioning to achieve cache-oblivious behavior [30]. The idea is to recursively subdivide the iteration space to ensure locality at every level of granularity. We can easily implement a 1-dimensional recursive partitioning on top of the `innerStenSK` function by using sketching to derive the details.

First, we need a specification. For that we will slightly modify our `innerSten` function to take as parameters the iteration bounds for one of the loops.

```
void mainLoop(float[T,N] X, int n1, int n2) {
  for (int i = n1; i < n2; ++i)
    for (int t = 1; t < T; ++t)
      X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
}
```

Now, the implementation is recursive, so it has a base case and a recursive case. For the base case we are going to reverse the loops, since the block is small enough that we believe we can get better cache behavior this way: it will result in consecutive reads instead of interleaved ones. Since we are not sure how the reversal is going to affect the index expressions, we just replace them with the `linexpG` generator. On the other hand, we wish to retain control of the recursive partitioning and the size of the base case, so that portion of the sketch is fully specified. The sketch is shown below.

```
void mainLoopSK(float[T,N] X, int n1, int n2)
                        implements mainLoop {
  if (n2-n1 < 4)
    for (int t = 1; t < T; ++t)
      for (int i = n1; i < n2; ++i)
        X[t, i-t] = X[t-1, linexpG(i,t)]
                    + X[t-1, linexpG(i,t)];
  else {
    int m = (n1 + n2) / 2;
    X = mainLoopSK(X, n1, m);
    X = mainLoopSK(X, m, n2);
  }
  return X;
}
```

Cache-oblivious implementations are known to be difficult to implement by hand, especially for higher dimensions. But using the algorithms presented in this paper, the sketch compiler easily synthesizes the low-level details for this complex implementation. (Note that for a 3-D stencil, we would have to deal with 16 different corner cases, making an optimization by hand extremely difficult [42].) Furthermore, no compiler we know of will generate a recursive cache-oblivious implementation automatically.

This example has illustrated the potential of sketching to simplify the development of efficient implementations for the domain of stencil kernels. The basic SKETCH synthesizer, however, is not powerful enough to handle sketches of stencils of even moderate complexity. Fortunately, stencils have a lot of structure that the synthesizer can exploit to make the synthesis problem easier; this is exactly what the following chapter will describe.

# Chapter 11

# Specializing the Synthesizer

In the previous section, we saw the challenge that stencils pose to the programmer, and the potential of sketching to make stencil programs much easier to write. Unfortunately, the standard SKETCH compiler is unable to provide satisfactory solutions to even relatively simple stencil sketches. The root of the problem lies in the unbounded nature of stencils. Programmers want their stencil kernels to work for grids of arbitrary sizes, but the synthesizer is only able to cope with bounded arrays. Moreover, even under a bounded model assumption, the minimum grid size required to exercise all the corner cases is usually larger than what the synthesizer is able to handle.

At the same time, we have defined our domain with a very strong assumption: each element in the output is a function of a bounded number of elements in the input, and is produced through only a bounded amount of computation. This is a huge assumption, and if we can exploit it, it will allow us to resolve sketches not just for very small bounded grids, but for arbitrary grid sizes.

## 11.1 Algorithm Overview

The key insight to exploit the bounded computation assumption is that proving the equivalence of two stencils is equivalent to proving that they compute the same value for any arbitrary grid element. The solver exploits this insight by deriving, for the spec and the sketch, a function that describes how to compute an arbitrary element in the output grid from a small number of elements in the input. For example, given a 2-point 1-D stencil `s`, the synthesizer will derive the function `reduced_s`:

```
float[N] s(float[N] in);
float    reduced_s(float[4] v, int N, int idx);
```

The reduced stencil `reduced_s` computes `s(in)[idx]`, *i.e.*, the value of the `idx`-th element of the grid computed by the original stencil `s`. The finite array `v` represents the elements from the input grid `in` that are needed to compute `s(in)[idx]`. (This section will explain why four elements are needed for a 2-point stencil.)

Unlike the original stencil, the reduced function takes only a bounded number of inputs, and performs a bounded amount of computation. In particular, the value of `N` no longer determines the amount of computation; it is only passed because it may be needed to compute the output value, *e.g.* to identify if the desired output element is close to the grid boundary.

Therefore, the synthesizer can now guarantee the equivalence between the sketched implementation of a stencil and a simpler reference implementation by enforcing the equivalence of their reduced functions. This helps the synthesizer is because it turns corner cases in the iteration space into corner cases in the input space. As we saw before, the CEGIS algorithm is very good at identifying corner case inputs needed to fully constraint the sketch, so this transformation spares the synthesizer from wasting time analyzing highly repetitive computations.

The reduction from a function like `s` to a function like `reduced_s` works in three steps; the same steps are applied to both the spec and the sketch. First, the reduction bounds the size of the program output by focusing on a single element of the output grid. Second, it uses symbolic manipulation to bound the number of computations performed by the stencil. Finally, it uses abstraction to bound the size of the input. To illustrate the process, we use the 2-point 1-D stencil, together with a somewhat contrived sketch which nonetheless hints at how more complex sketches are reduced. For the sake of simplicity, the sketch differs from the specification only slightly: it uses holes to adjust its index expressions to compensate for its loop bounds, which differ from those in the spec. The functions `f`, `g`, and `h` stand for arbitrary index expressions (these functions do not have side effects):

```
float[N] spec(float[N] in) {
  foreach i ∈ [1, N-2]
    out[f(i)] = in[g(i)] + in[h(i)];
}
```

```
float[N] sketch(float[N] in) implements spec {
  foreach k ∈ [0, N-??]
    out[f(k+??)] = in[g(k+??)] + in[h(k+??)];
}
```

**Bounding the output.** We replace the unbounded output grid with a single scalar by making the program return the value of a single element of that output grid. Specifically, a stencil function `s` is transformed to a scalar function `scalar_s` such that `s(in)[idx]=scalar_s(in, idx)`. The transformation does not lose any information, and the behavior of the original program can be obtained by invoking the *scalar function* of `s` as shown below for both the spec and the sketch.

```
float[N] spec(int[N] in) {
  foreach j ∈ [0, N-1] out[j] = scalar_spec(in, j);
}
float[N] sketch(int[N] in) implements spec {
  foreach j ∈ [0, N-1] out[j] = scalar_sketch(in, j);
}
```

Note that aside from calling a different scalar function, the spec and the sketch are identical (both compute all elements of the output grid `out`). Hence, we reduced the stencil synthesis problem to the problem of making the scalar functions `scalar_sketch` and `scalar_spec` behave identically.

The two scalar functions are shown below. Their goal is to express `out[idx]` in terms of the input grid. Both functions achieve this with a two-step symbolic back-substitution process. First, they compute the iteration in which the output element `out[idx]` was most recently assigned. (In our example, each output element was assigned at most once, but as we shall see in the next section this is not required by the algorithm.) The number of this iteration is stored in variable `last`. Next, the value of `out[idx]` is computed by evaluating the right-hand-side expression of the most recent assignment. If the right-hand-side expression refers to values other than the input, the two-step process is repeated (not needed in our example).

```
float scalar_spec1(float[N] in, int idx) {
  int last = UNDEFINED;
  foreach i ∈ [1, N-2]
    if (idx == f(i)) last = i;
  if (last == UNDEFINED) return 0;
  return in[g(last)] + in[h(last)];
}
float scalar_sketch1(float[N] in, int idx) {
  int last = UNDEFINED;
  foreach k ∈ [0, N-??]
    if (idx == f(k+??)) last = k;
  if (last == UNDEFINED) return 0;
  return in[g(last+??)] + in[h(last+??)];
}
```

**Bounding the number of computations.** The scalar functions execute an unbounded number of computations because they contain loops controlled by the free variable `N`. However, the only purpose of the loops is to identify the latest iteration that wrote to `out[idx]`. This operation can be expressed declaratively as `last = max(`$\{[1,N-2] \cap f^{-1}(idx)\}$`)`, where $f^{-1}$ is the inverse of the index expression `f`.

The advantage of this formulation is that it allows us to use a symbolic solver to reduce the evaluation of the latest iteration expression into a finite sequence of operations. Our current solver is quite rudimentary, but it has already been able to handle all the problems we have tried so far, including several from real-world benchmarks.

For our running example, the declarative formulation allows us to write the scalar functions as follows.

```
float scalar_spec2(float[N] in, int idx) {
  int last = max({[1,N-2] ∩ f^-1(idx)});
  if (last == UNDEFINED) return 0;
  return in[g(last)] + in[h(last)];
}
```

```
float scalar_sketch2(float[N] in, int idx) {
  int last = max({k | k ∈ [0,N−??]
                    ∧ k+?? ∈ f^{-1}(idx)});
  if (last == UNDEFINED) return 0;
  return in[g(last+??)] + in[h(last+??)];
}
```

If `f` is the identity function, for example, the symbolic solver will replace the last-iteration computation with a few conditional assignments; the one in the spec is shown below.

```
if (idx >= 1 && idx <= N-2) last = idx;
else last = UNDEFINED;
```

After the replacement, the functions are already bounded in terms of computation, but the input is still an unbounded array.

**Bounding the input.** For programs that do not modify their inputs, the input array can be treated as an uninterpreted function. In other words, the input array is an entity whose only discernible property is that accesses with the same index produce the same value. Programs like the example in Section 10.2 that do modify their input array are modeled by the compiler as receiving an immutable array as input, copying it into a mutable array, and then returning the modified array as an output at the end of the computation.

The next problem is to represent the uninterpreted function finitely. We observe that the scalar functions `scalar_spec2` and `scalar_sketch2` read only a finite number of input grid elements, which lets us borrow a technique originally proposed by Ackerman [1] and used extensively in hardware verification [14]. The function `in_fn` below implements the semantics of an uninterpreted function under the restriction that the function is called with at most four different values of the argument. (For this example, we can restrict ourselves to four symbolic values because the scalar functions `reduced_spec` and `reduced_sketch`, shown below, will together make no more than four dynamic calls to `in_fn` for a given value of their input parameter `idx`.) The function `in_fn` implements the desired semantics by returning the same symbolic value whenever called with the same value of the argument `in_idx`.

```
float in_fn(int in_idx) {
  if (in_idx == i_0) return v_0;
  if (in_idx == i_1) return v_1;
  if (in_idx == i_2) return v_2;
  if (in_idx == i_3) return v_3;
}
```

This use of uninterpreted functions is considered a form of abstraction, because we are representing an unbounded grid using only four scalars by throwing away all the information about those cells that were not accessed on a particular invocation of the scalar functions.

The final step is to represent the symbolic values $i_k$ and $v_k$ with constructs from the SKETCH language. (Recall that SKETCH relies on a Boolean satisfiability solver which does not support symbolic manipulations.) The non-symbolic version of `in_fn` shown below, expressed entirely in SKETCH, implements the symbolic values $i_k$ by remembering the concrete values of arguments in a global array. The symbolic values $v_k$ are implemented as function arguments. This treatment will make the synthesizer carry out its reasoning under all possible values of $v_k$, which is equivalent to viewing $v_k$ as symbolic values.

```
float in_fn(int in_idx, float[4] v) {
  static int[4] g; static int gi=0;  // globals
  g[gi++] = in_idx;
  if (in_idx == g[0]) return v[0];
  if (in_idx == g[1]) return v[1];
  if (in_idx == g[2]) return v[2];
  if (in_idx == g[3]) return v[3];
}
```

As an optional optimization, our system allows the user to assert that the sketch computes `out[idx]` using only the input elements used by the spec, as opposed to arbitrary input elements. That is, the array `g` is set only in the spec; the sketch only reads its values and asserts that one of them matches. This is the case for almost all implementations, because if a sketch used any other entry, its value would have to get canceled out in order for the spec and the sketch to be equivalent. Using this assumption, we can reduce the number of comparisons on each call to the uninterpreted function, since we only need to compare the index to those indices used by the spec, but not with those used by the sketch. With this optimization, the compiler will ignore any implementation that violates the assumption, while never producing an incorrect implementation.

**Putting it all together.** Let us assume for the sake of simplicity that `f`, `g`, and `h` are identity functions. Then, the symbolic solver reduces the evaluation of the latest iteration expression to guarded assignments as shown below.

```
float reduced_spec(float[4] v, int N, int idx) {
  if (idx < 1 || idx > N-2) return 0;
  return in_fn(idx, v) + in_fn(idx, v);
}
float reduced_sketch(float v[4], int N, int idx) {
  int last = idx - ??1;
  if (last < 0 || last > N-??2) return 0;
  return in_fn(last+??3, v) + in_fn(last+??4, v);
}
```

These functions define a finite sketch problem which can be solved by the finite SKETCH solver. In this case, the SKETCH solver can easily prove that the abstracted sketch and spec are equivalent for the following value assignments to holes: $??_2 = 3$, $??_1 = ??_3 = ??_4 = 1$.

The control values can then be applied to the original sketch, and the construction will guarantee that the resulting implementation is equivalent to the original spec.

**Abstracting integers and floating point values.** Here, we focus on complications arising from modeling integer and floating-point values.

The reduced functions produced by the above algorithm lead to intractable SAT problems if translated directly to Boolean circuits, mainly due to the presence of floating point variables in the reduced programs. Additionally, modeling floating point values in their full IEEE glory would make the equivalence criterion overly strict, ruling out many optimizations employed by programmers who often choose to assume associativity of floating point numbers.

There are numerous approaches in the literature to handle floating point arithmetic in the context of model-checking and verification, most of them relying on uninterpreted functions [24, 52]. While it is relatively easy to replace floating point operations with uninterpreted functions, there is a simpler approach that works remarkably well in our domain. The key observation is that the stencils we are generating are often linear functions in their floating point arguments: both the reduced spec and sketch have the property that if you set their integer inputs to any fixed value, the remaining function will be a linear function

(it may be a different linear function for different values of the integer inputs). The stencil appears linear to the solver because it performs verification separately for each combination of integer input values. It is trivial to verify this property statically from a DAG representation of the reduced spec and reduced sketch. When this property holds—as is the case with all the benchmarks presented in this paper—the compiler can safely replace all floating point inputs with 1-bit integers without losing soundness, provided that it grows the integer representation whenever necessary to avoid arithmetic overflow. The soundness argument should be obvious from the fact that in order to test the equivalence of two $k$-dimensional linear functions over the reals, one only needs to test them with $k$ independent vectors.

Floating point constants are treated as free variables, and are also represented with a single bit. This treatment is sound but not complete because it loses algebraic relationships among constants. For example, after we replace 0.5 with a free variable $v_{0.5}$, we can no longer prove that `a*0.5 + a*0.5 == a`. This limitation did not prove to be an issue for any of the benchmarks we studied; optimizations of stencil codes do not rely on such equivalences very often.

After performing abstraction on the floating point values, the remaining problems involve only integers and Booleans. Our current approach is to translate these problems directly into circuits, which limits our scalability to representing integers with about 6 bits (3 bits for the hardest benchmarks). However, there are known scalable techniques that we can use to remove this limitation [15].

## 11.2 Algorithm Details

Here, we presents in detail the stencil reduction algorithm outlined in Section 11.1, focusing mainly on the first two steps (bounding the output and bounding the computations).

### 11.2.1 Preliminaries

We use standard compiler transformations to bring the input programs (*i.e.*, the specification and the sketch) into an intermediate normal form bearing the following properties:

**Loops.** All loops are normalized to the form `for (int i = e1; i < e2; ++i)`, where `i` is the uniquely named *main induction variable* of the loop. Remaining loop induction

variables are expressed as a function of `i`.

**Function calls.** All calls are inlined. Recursive calls to sketched functions are replaced with calls to their specifications (which must be non-recursive). Proving equivalence of the spec and a recursive sketch after this transformation constitutes a proof by induction, under the assumption that the recursion is well founded (*i.e.*, that all recursive calls eventually terminate).

Normalized programs obey the following syntax:

$$
\begin{array}{lll}
\text{Expressions} & e & ::= n \mid \texttt{true} \mid \texttt{false} \mid \texttt{x} \mid \texttt{x[}e\texttt{]} \mid e_1\ \texttt{op}\ e_2 \\
& & \mid f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_k\texttt{)} \\
& & \mid \texttt{switch}\ e\ \texttt{case}\ n_1\texttt{:}\ e_1\texttt{;}\ldots\texttt{case}\ n_k\texttt{:}\ e_k \\
\text{Statements} & c & ::= \texttt{x:=}e \mid \texttt{x[}e_1\texttt{]:=}e_2 \mid \texttt{skip} \\
& & \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid c_1\texttt{;}\ c_2 \\
& & \mid \texttt{for (i =}\ e_1\texttt{; i <}\ e_2\texttt{; ++i)}\ c \\
\text{Functions} & f & ::= \texttt{def f(x}_1\texttt{,}\ldots\texttt{,x}_k\texttt{)}\ c\ \texttt{return}\ e
\end{array}
$$

An additional semantic restriction is that loop bounds must be invariant with respect to the induction variable of their loop. The bounds can, of course, relate to outer induction variables. This restriction could be relaxed, but the complications involved are not justified in the domain of stencils. The domain is also restricted by the power of the algebraic solver used to eliminate the latest assignment expression (called `RD` in this section).

If the program satisfies the aforementioned constraints, the reduction about to be described will be sound, but only under the assumption that there is no integer overflow in either the spec or the sketch. This is because the symbolic solver used to eliminate the latest assignment expression may assume algebraic properties that do not hold in the presence of overflow (*e.g.*, `a-1<N` $\iff$ `a<N+1`).

### 11.2.2 Synthesizing Scalar Functions

We describe the algorithm that generates scalar functions and bounds their computation. These two steps are performed in an intertwined fashion and we describe them together.

One way to define a scalar function is to view it as a slice of the original stencil. More precisely, given a stencil computation `s(in)` returning a grid `out`, and an index `idx`,

the scalar function computes a slice of `s` with respect to `out[idx]`. While the original computation may read the entire (unbounded) input grid, the slice only reads a *bounded* number of input elements.

The slice is expressed recursively, using a functional language that resembles the one defined by the above syntax, but does not include statements. Each recursive call corresponds to a def-use edge in the slice of `out[idx]`. Since the slice incurs a bounded computation, the recursion is bounded. The result is thus finite and can be accepted by the finite SKETCH synthesizer.[1]

The slicing algorithm boils down to expressing `out[idx]` symbolically in terms of program inputs. To this end, we recursively substitute non-input variables in the expression `out[idx]` with the right-hand side values of their most recent assignments.

We refer to most recent assignments as *reaching definitions*. In contrast with traditional reaching definitions, which offer a static approximation of the dynamic behavior of the program, our reaching definitions are *concrete*: they are defined on the execution trace where each program point is reached by at most one definition for any program location. Since there is no ambiguity as to which definition most recently assigned the location, we obtain precise back-substitution in the sense that the fully substituted symbolic expression is executable and computes the value of `out[idx]`.

The procedure `RD`, which lies at the heart of the algorithm, computes the concrete reaching definition of a memory location.

$$RD : M \times P \times I \rightarrow P$$

The procedure maps a memory location $m \in M$, an execution point $p \in P$, and a program input $i \in I$ to the most recent execution point that defined the value of $m$ prior to $p$, under the input $i$. The set of memory locations $M$ consists of scalar variables and array elements. The set of execution points $P$ is the cross product of static program points with the loop iteration space. The input space $I$ includes the input grid together with any scalar arguments.

[1] If the input program violates the boundedness assumption, the algorithm described in this section will still produce a correct recursive representation of the slice, but the recursion will be unbounded and will depend on the inputs. When this program is fed to the SKETCH synthesizer, the synthesizer will attempt to inline the recursive calls an increasing number of times, to no avail; after a few tries, it will reach a predefined threshold for function inlining, and will inform the user that the sketch can not be resolved. Crucially, though, a violation of the assumption can not lead to a buggy implementation.

We are now ready to describe the abstraction algorithm. For each scalar variable `v` we create a function

$$\texttt{v\_fn} : P \times I \rightarrow \texttt{T}$$

that computes the value $n$ of `v` at execution point $p \in P$ under the input $i \in I$. The type `T` is a primitive type (**boolean**, **int**, or **double**). The function will be expressed in the functional language shown above. Similarly, for an array `a` (for simplicity, we assume that arrays are one-dimensional) we create a function

$$\texttt{a\_fn} : \texttt{Int} \times P \times I \rightarrow \texttt{T}$$

that computes the value of `a[idx]` for an index `idx` $\in$ `Int` at point $p \in P$ under the input $i \in I$. These two functions are called v-functions. The reduced function for a stencil `s` returning a grid `out` now becomes

```
double reduced_s(int idx, double[N] in) {
  return out_fn(idx, P_e, in);
}
```

where $P_\text{e}$ is the end point of the program execution trace.

v-functions are constructed via syntactic translation of the original program. A v-function first obtains the reaching definition and then (recursively) replaces all array and variable references on the right-hand side of the reaching definition with calls to appropriate v-functions. A v-function for variable `v` (or array access `a[idx]`) at execution point $p$ under input $i$, looks as follows.

1. Obtain the most recent execution point $p'$ where `v` (respectively, `a[idx]`) was defined prior to $p$ under input $i$.
2. Extract the static program statement $s$ executed at $p'$, and the right-hand side expression $e$ in $s$.
3. Return a valuation of a transformed expression $F(e)$ where
   (a) each variable sub-expression `v'` is replaced with `v'_fn(`$p'$`,`$i$`)`;
   (b) each array access sub-expression `a'[`$e'$`]` is replaced with `a'_fn(`$F(e')$`,`$p'$`,`$i$`)`.

To illustrate the process, consider the following example. The example is acyclic so that the reader need not be concerned with execution point representation for now.

```
    int[N] f(int[N] in, int a) {
s1:   int[N] out = 0;
s2:   int[N] A = in;
s3:   if (in[3] > in[4]) {
s4:     out[3] = A[3];
s5:     A[a] = in[5];
      }
s6:   if (A[5] > 0)
s7:     out[a] = A[out[a]];
      return out;
    }
```

The v-function of `out` for this example needs to handle three assignments to `out`:

```
out_fn(idx, p, i)) {
  p' = RD((out,idx), p, i);
  switch (P_s(p')) {  // extract the statement at p'
    case s1: return 0;
    case s4: return A_fn(3, p', i);
    case s7: return A_fn(out_fn(a, p', i), p', i);
  }
}
```

It remains to show how the function `RD` computes the concrete reaching definitions. First, we need to define the *execution point* $p \in P$. As mentioned in passing above, an execution point $p$ is a pair $(s,t)$, where $s$ is a (static instance of a) program statement and $t$ is a point from the iteration space $T$ of the program. The iteration point $t$ is defined as a valuation of loop induction variables that are in scope at the statement $s$ (these are exactly the induction variables of loops that enclose $s$). When $p = (s',t')$, we define `P_s`$(p) = s'$ and `P_t`$(p) = t'$. In the following, we use `T_map(`$t$`,'j',`$n$`)` to denote binding of an induction variable `j` to some value $n$ in iteration point $t$, and `T_get(`$t$`,'j')` to extract the currently bound value for `j`. Similarly, `I_get(`$i$`,'x')` extracts the value associated with (non-induction) variable`x` at input state $i$.

The trace of a program execution is a sequence of execution points. We define a total order $<_P$ on $P$ such that $p_1 <_P p_2$ iff $p_1$ executed before $p_2$. The *execution order* $p_1 <_P p_2$

is determined by the lexicographic order of the iteration points $\texttt{P\_t}(p_1)$ and $\texttt{P\_t}(p_2)$; if there is a tie, then $p_1$ and $p_2$ must be in the same loop iteration and their execution order is determined by their position in the program. Internally, we represent the execution point such that the $<_P$-test can be performed as a single lexicographic test. We define two execution point constants: $P_\mathrm{b}$ is the beginning of the execution and $P_\mathrm{e}$ is the end of the execution.

As statement guarded by conditionals may not execute in every iteration, the execution order $<_P$ alone is insufficient for determining the most recent definition. To reflect control conditions under which the statement executes, we define the predicate $q(p, i)$, which holds iff the statement $\texttt{P\_s}(p)$ executes at iteration point $\texttt{P\_t}(p)$ under the input $i$. Formally speaking, $q(p, i)$ is the disjunction of the path constraints for all paths that reach the execution point $q(p, i)$. Programs in our domain have structured control flow and loop bounds that are invariant with respect to their loop's induction variable. Therefore, the predicate $q(p, i)$ can be constructed syntactically as a conjunction of all the conditionals (including loop conditions) enclosing the statement $\texttt{P\_s}(p)$.

For example, consider statement $s_4$ in the code below. Let $p = (s_4, t)$ be an execution point associated with $s_4$ for some iteration point $t$. We form the predicate $q(p, i)$ for this execution point under some input state $i$: the constraint corresponding to the **if** statement in $s_3$ is $\texttt{A\_fn}(\texttt{T\_get}(t, \texttt{'j'}), p, i) > 0$; the constraint corresponding to the loop statement in $s_0$ and $s_1$ is $\texttt{T\_get}(t, \texttt{'j'}) \geq 0 \wedge \texttt{T\_get}(t, \texttt{'j'}) < \texttt{I\_get}(i, \texttt{'N'})$.

```
    double[N] f(double[N] in) {
      int [N] A, out;
s0:   for (int j=0;
s1:                j<N; ++j) {
s2:     A[j] = in[j];
s3:     if (A[j] > 0)
s4:       out[j] = in[j];
        else
s5:       out[j] = -in[j];

s6:     out[j] = sqrt(out[j]);
      }
    }
```

We are now ready to describe `RD`, the procedure for computing reaching definitions, for this case of an array access. Given a program location `v[idx]`, an execution point `p`, and input `i`, the procedure considers all execution points $p'$ that precede `p`, execute under the input `i`, and assign into location `v[e]` for some index expression `e` such that the value of `e` at execution point $p'$ equals `idx`. Among all such execution points, it selects the most recent one; if none meets all criteria, there is no reaching definition, and `RD` returns $P_\flat$.

```
RD((v,idx), p, i) {
   return max({P_♭} ∪ {p' | p' <_P p  ∧  q(p',i)
                          ∧ P_s(p')≡v[e]=e'
                          ∧ e_fn(p',i)=idx});
}
```

Our compiler uses algebraic reasoning to simplify procedure `RD`. The symbolic simplifier reasons with equalities, inequalities, and logical connectives. The simplification procedure relies on the fact that when we have an assignment of the form `x[g(j)] = e`, the constraint `g(j)` = `idx` often suffices to fully define the iteration space point in terms of `idx`, by inverting `g`. It then remains only to test whether the values thus derived satisfy the remaining constraints for qualifying execution points. Also, finding the most recent point of assignment is done in a staged manner, by first finding the most recent point corresponding to *each* assigning statement and then picking the most recent among them.

In the above example, we can find the last program point prior to point $p$ where `out[idx]` has been updated by the particular statement $s_4$ (if such a point exists), as follows:

```
p' = max({(s4,t) | (s4,t) <_P p
                 ∧ (T_get(t,'j')≥ 0
                    ∧ T_get(t,'j')<I_get(i,'N')
                    ∧ A_fn(T_get(t,'j'), (s3,t), i)> 0)
                 ∧ T_get(t,'j')=idx});
```

Note that the constraint `T_get(`$t$`,'j') = idx` fully defines the value of `j` to be equal to `idx`, so the statement above can be replaced with a couple of simple assignments.

```
t = T_map(new T,'j',idx);
p' = (s7,t);
if (! (p' < p && idx >= 0 && idx < I_get(i,'N')
       && A_fn(idx, (s3,t), i) > 0))
  p' = Pb;
```

To complete our example, we find statement-specific most recent points for each assigning statement in a similar manner, and pick the most recent among these points. The final v-function for `out` is shown in Figure 11.1.

```
int out_fn(int idx, P p, I i)  {
  T t = T_map(new T, 'j', idx);

  P p4 = new P(s4,t);
  if (! (p4 < p && idx >= 0 && idx < I_get(i,'N')
         && A_fn(idx, new P(s3,t), i) > 0))
    p4 = Pb;
  P p5 = new P(s5,t);
  if (! (p5 < p && idx >= 0 && idx < I_get(i,'N')
         && ! A_fn(idx, new P(s3,t), i) > 0))
    p5 = Pb;
  P p6 = new P(s6,t);
  if (! (p6 < p && idx >= 0 && idx < I_get(i,'N')))
    p6 = Pb;

  P p' = (p4 < p5 ?  (p5 < p6 ? p6 : p5) :
          (p4 < p6 ? p6 : p4));
  switch (P_s(p')) {
    case s4: return I_get(i, 'in', T_get(P_t(p'),'j'));
    case s5: return -I_get(i, 'in', T_get(P_t(p'),'j'));
    case s6:
      return sqrt(out_fn(T_get(P_t(p'),'j'), p', i));
  }
}
```

Figure 11.1: v-function for array out

# Chapter 12

# Empirical Evaluation

This section presents an empirical evaluation of our system using several kernels from the MultiGrid method as case studies. From the case studies, we were able to validate three basic claims.

**Scalability.** We prove that the system scales to complex real-world implementations of important kernels. For example, we were able to synthesize in a matter of minutes an implementation for a kernel that involved 14 different loops from a sketch that had 44 different holes.

**Usability.** The case studies also allow us to describe a typical use scenario for a sketching system. In particular, we describe how we were able to explore different implementation strategies, discarding those that don't work and refining those that do, without the risk of introducing bugs.

**Performance of generated code.** We show that creating these complex implementations is worth the effort. In particular, one of the implementations we sketched was over 8 times faster than the original reference implementation on a 1.3 GHz Itanium-2, even though they were compiled and optimized using the Intel Fortran compiler version 9.1, arguably one of the best compilers commercially available on the Itanium architecture. In other words, the sketch expressed optimization ideas which the compiler was unable to discover on its own from a naïve implementation.

This section is derived from the evaluation we presented [61], but the solver performance numbers have been updated to reflect the current state of the implementation. The

| | Loop | Holes | i-bits | Size | Iters | SAT | Total |
|---|---|---|---|---|---|---|---|
| `timeSkew` | 6 | 12 expr | 5 | 10469 | 23 | 238 | 263 |
| `cacheObv` | rec | 2 expr | 5 | 5313 | 2 | 141 | 165 |
| `interp1` | 3 | 111 | 4 | 2404 | 60 | 62 | 62 |
| `interp2` | 7 | 74 | 4 | 2446 | 45 | 45 | 46 |
| `rb3d1` | 6 | 36 | 5 | 7178 | 22 | 101 | 106 |
| `rb3d1Odd` | 14 | 43 | 4 | 32569 | 51 | 295 | 337 |
| `rb3d2` | 7 | 30 | 4 | 20783 | 35 | 66 | 84 |
| `rb2d1` | 4 | 16 | 6 | 1809 | 19 | 20 | 20 |
| `rb2d2` | 4 | 22 | 6 | 3868 | 22 | 66 | 68 |

Table 12.1: Solution time for the stencil benchmarks. Table columns specify: the number of loops present in the final implementation ("rec" stands for recursive); the number of holes the synthesizer filled in; the number of bits used to represent integers; the number of Boolean and arithmetic operations in the reduced problem after some simplification; the time (in seconds) spent in SAT solver queries; and the total time (in seconds) required to resolve the benchmark.

new performance numbers for the synthesizer are summarized in Table 12.1. The table lists all the stencil benchmarks described so far, together with the ones introduced in this section. The table also shows a few statistics to give a sense of the complexity of each benchmark; these include the number of loops in the final implementation, the number of holes filled by the synthesizer, and the size of the DAG representation of the constraint system. This last quantity is simply to give a measure of how complex the reduced problems are. The table also shows the solution time for each benchmark on the ThinkPad laptop with a single core Intel T1300 at 1.66GHz with 2MB of L2 cache and 1GB of memory that has been used so far for all the performance evaluations in the thesis. The table also shows the number of CEGIS iterations and what fraction of the time is spent solving SAT problems. For all these benchmarks the synthesizer used ABC [49] as the solution engine because we have found it to work better on stencil problems.

It is worth noting that most benchmarks appear to have grown in their size measure from their version in [61]. The reason for this has to do with array out of bounds checks. The synthesizer in [61] did not support array bounds checks, or any other kind of assertion in the body of loops. Therefore, it could produce a solution that wrote all the correct values inside the array, but also wrote a few values out of bounds. The `timeSkew` sketch was the only benchmark that every once in a while produced this kind of pathological solution. This

latest version of the implementation now supports assertion checking, and enforces memory safety. This is why many of the benchmarks are now a bit larger than before, and why some experienced small performance degradations even though the underlying solver got significantly faster.

Another interesting feature in the table is the number of CEGIS iterations, which for most benchmarks is fairly large compared with the iteration counts for the sequential benchmarks. Part of the explanation has to do with the huge candidate spaces for many of these benchmarks, but there is more to it than that. Recall that the domain specific transformation converted all the corner cases in the iteration space into corner cases in the input space. This eliminated a lot of redundancy in the representation, but it also means that now many inputs are needed to convey the information that a single input to the original program may have conveyed. Overall, though, the benefits of the transformation are significant; the original sketch synthesizer is unable to resolve most of the sketches in Table 12.1. Only `rb2d1` and `rb2d2` can be resolved by the standard SKETCH compiler, and even for those simple benchmarks, it can handle only fixed size grids up to size 4. Even for size 4, `rb2d1` takes 30 seconds, even though it only performs 6 CEGIS iterations compared with 19 using the domain specific synthesizer.

### 12.0.3 Sketching for MultiGrid

The MultiGrid algorithm is used for solving partial differential equations for a wide range of domains, including fluid dynamics and solid mechanics. It is composed of three main kernels: relax, interpolate, and restrict [13]. Each application of the relaxation routine produces a closer approximation to the solution, but with a very fine grid the low frequency components of the error in the approximation take too long to die out. To address this problem, MultiGrid computes corrections to the solution by creating a coarser problem (restrict), solving it recursively, and then mapping the correction back to a finer grid (interpolate). We have sketched implementations of relaxation and interpolation kernels in 3-D; the restrict kernel is quite similar to interpolate.

We sketched several implementation tricks from the literature and from hand optimized implementations of these kernels. In a couple of cases, it took less than half an hour to write and synthesize implementations that we estimate would have taken half a day if written by hand. Additionally, some of our sketched implementations were several times

faster than the clean reference implementations, even after the latter had been optimized by the Intel Fortran compiler.

**Relaxation.** The relaxation phase of MultiGrid starts with an approximation to the solution of the problem and uses it to compute a closer approximation to the solution. For our case study, we implemented a Red-Black Gauss-Seidel relaxation scheme for both a 2-D grid and a 3-D grid following more or less the same implementation strategies. The specification for the 3-D benchmark is shown below. The algorithm assigns a color to each cell on the grid in a checkered pattern, and then applies a six point stencil on the red cells, followed by another (same) stencil on the black cells. This widely used relaxation scheme has well known implementation strategies to optimize it for different architectures [25].

```
// red
for (int i = 1; i < N-1; ++i)
  for (int j = 1; j < N-1; ++j)
    for (int k = 1; k < N-1; ++k)
      if (i%2 == 0 ^ j%2 == 0 ^ k%2 == 0)
        out[i,j,k] = F(i,j,k, in);
// black
for (int i = 1; i < N-1; ++i)
  for (int j = 1; j < N-1; ++j)
    for (int k = 1; k < N-1; ++k)
      if (! (i%2 == 0 ^ j%2 == 0 ^ k%2 == 0))
        out[i,j] = F(i,j,k, out);
```

Here, `F(i,j,k, prev)` expands to

```
f[i,j,k] + v0*in[i,j,k] + v1*prev[i-1,j,k]
     + v2*prev[i+1,j,k] + v3*prev[i,j-1,k]
     + v4*prev[i,j+1,k] + v5*prev[i,j,k+1]
     + v6*prev[i,j,k-1];
```

The above specification is written in the simplest possible way, using the `xor` expression `i%2 == 0 ^ j%2 == 0 ^ k%2 == 0` to decide the color of each cell. These types of conditions tend to confuse traditional dependence analysis, so even a state-of-the-art compiler like the Intel compiler is unable to optimize the kernel fully.

In order to produce a better implementation, we used two implementation strategies from [25]. In this paper, Douglas *et al.* provide only high-level descriptions of their

optimization strategies (no pseudo-code), but those low-level details omitted in the paper are exactly what SKETCH can synthesize.

The first implementation we created is quite simple; it just eliminates the conditionals by computing the output in blocks of eight elements at a time: four red and four black. The sketch for the case where N is even is very simple; it has two loop-nests with unspecified bounds, each with four assignments of the form

```
out[2*i-??,2*j-??,2*k-??] = F(2*i-??,2*j-??,2*k-??, in);
```

The sketch describes the high-level idea that we compute first all the red cells four at a time, and then all the black cells, also four at a time. The sketches `rb2d1` and `rb3d1` from Table 12.1 correspond to the 2-D and 3-D instances of this sketch respectively. One can see that both sketches resolved quite fast despite having a large number of holes. Moreover, the resulting implementation is about 45% faster for the 3-D case and 70% faster for the 2-D case.

The implementation for the case where `N` is odd is considerably more complicated because one must cover a lot of corner cases, particularly in 3-D, where the final implementation is composed of 14 different loops. However, with the support of sketching, it is easy to construct the odd case from the even case. To do this, we took the 3-D implementation for the even case produced by the previous sketch, and simply sketched the corner cases on top of it. Using the implementation generated from the even case as a starting point allowed the sketch to scale; a sketch for the odd case that left everything unspecified proved to be intractable for the compiler. Fortunately, we did not have to start from scratch. We were able to exploit the fact that we already had a solution for the even case to make the odd case more tractable. The sketch for the red cells for the 2-D odd case is shown below. Statements 1 and 2 came from the implementation of the even case, and on top of it, we added a corner case for the last cell in each row (3), and the last row (4), Since we are not sure if the last cell in the last row (5) needs to be treated separately, we ask the solver to decide.

```
for (int i = ??; i < N/2-??; ++i){
  for (int j = ??; j < N/2-??; ++j){
    out[2*i-1,2*j-1] = F(2*i-1,2*j-1, in); //1
    out[2*i,2*j] = F(2*i,2*j, in);         //2
  }
```

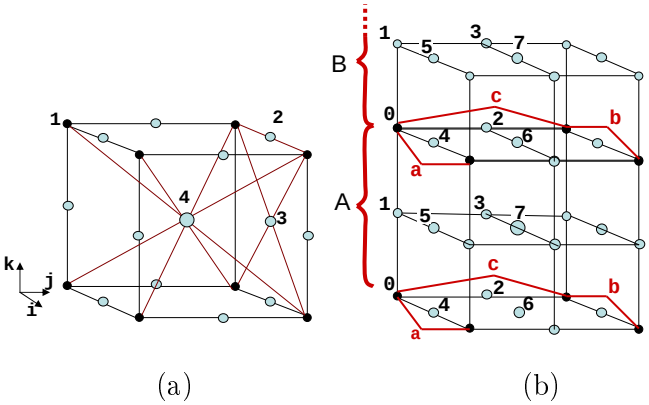
```
  out[2*i-??,N-??] = F(2*i-??,N-??, in);   //3
}
for (int j = ??; j < N/2-??; ++j){
  out[N-??,2*j-??] = F(N-??,2*j-??, in);   //4
}
if (??) out[N-??,N-??] = F(N-??,N-??, in); //5
```

Our second implementation for this benchmark uses another strategy mentioned in [25], namely computing the red and black cells together in a single pass through the array. In this case, careful attention is required in order to preserve the dependencies. We first implemented the trick in 2-D by creating a loop that updates both red and black cells as shown below, and then two more loops to handle the corner cases.

```
for (int i = ??; i < N/2-??; ++i) {
  for (int j = ??; j < N/2-??; ++j) {
    // red
    out[2*i-??,2*j-??] = F(2*i-??,2*j-??, in);
    out[2*i-??,2*j-??] = F(2*i-??,2*j-??, in);
    //black
    out[2*i-??,2*j-??] = F(2*i-??,2*j-??, out);
    out[2*i-??,2*j-??] = F(2*i-??,2*j-??, out);
  }
}
```

From the sketch, the synthesizer was able to discover that it had to compute the black cells with an offset with respect to the red cells in order to preserve dependencies. Note that neither the loop bounds nor the array access offsets are trivial; getting them right would have been quite challenging for the programmer.

```
for (int i = 2; i < N/2; ++i){
  for (int j = 1; j < N/2; ++j) {
    // red
    out[2*i-1,2*j-1] = F(2*i-1,2*j-1, in);
    out[2*i,2*j] = F(2*i,2*j, in);
    //black
    out[2*i-3,2*j-0] = F(2*i-3,2*j, out);
    out[2*i-2,2*j-1] = F(2*i-2,2*j-1, out);
  }
}
```




(a) (b)

Figure 12.1: (a) Stencil for interpolation distinguishes four different cases. Either the new point matches a point in the coarse grid (1), is in an edge in the old grid (2), in a face (3), or in the center of a cube formed by consecutive points in the old grid (4). (b) The optimized version will precompute the sums a, b and c.

As shown in Table 12.2, this optimization delivered a 60% performance improvement compared to the previous optimization, which was already 70% faster than the spec. The point of these numbers is not how good the optimized implementation is, but how bad the naïve one was. This shows that even if you use the best compiler available, you can not afford to write a naïve implementation; you need to do some hand optimization, and sketching is a very effective way to do that.

We tried to implement the same trick in 3-D. In this case, our sketch produced an implementation that computed the black cells for the plane $2i$ after computing the red cells in the plane $2i+2$. Unlike the 2-D case, however, this produced a performance degradation, probably due to the fact that the planes are too big to fit in the cache, so accessing too many of them at a time simply confuses the prefetcher with no benefit to performance.

**Interpolation.** The interpolation routine maps the values in a coarse grid to a finer grid of size $2N \times 2N \times 2N$, as illustrated in Figure 12.1. Points in the fine grid that correspond to points in the coarse one are copied, while the other points in the fine grid are computed by averaging the values of their neighbors in the coarse grid. As illustrated in Figure 12.1, this leads to four different cases, depending on whether we average 1, 2, 4 or 8 points. The following code shows a fragment of the specification, describing a few of the cases.

```
for (int i = 0; i < 2*N-2; ++i)
 for (int j = 0; j < 2*N-2; ++j)
  for (int k = 0; k < 2*N-2; ++k) {
```

```
  if (i%2 == 0 && j%2 == 0 && k%2 == 0)  // Case 1
    out[i,j,k] = in[i/2,j/2,k/2];

  if (i%2 == 1 && j%2 == 0 && k%2 == 0)  // Case 2
    out[i,j,k] =
      0.5 * (in[i/2,j/2,k/2] + in[i/2+1,j/2,k/2]);
  ...

  if (i%2 == 1 && j%2 == 1 && k%2 == 0)  // Case 3
    out[i,j,k] =
      0.25 * (in[i/2,j/2,k/2] + in[i/2+1,j/2+1,k/2]
      + in[i/2,j/2+1,k/2] + in[i/2+1,j/2,k/2]);
  ...
}
```

As in the previous sketch, we started by blocking the computation to eliminate all the conditionals in the specification. For each point in the coarse grid, there is a $2 \times 2 \times 2$ block in the fine grid which constitutes the smallest repeating pattern. Because the output grid is of size $(2\mathtt{N})^3$, odd grid sizes are not a problem.

The sketch was very easy to write because we left every single array offset unspecified, as well as the bounds of all loops. We only specified that on each iteration of the innermost loop, there were 8 assignments to entries of `out`, 1 for case `1`, 3 for case `2`, 3 for case `3`, and 1 for case `4`. The individual cases are shown below.

```
out[2*i+??,2*j+??,2*k+??] = in[i+??,j+??,k+??];
out[2*i+??,2*j+??,2*k+??] =
   0.5 * (in[i+??,j+??,k+??] + in[i+??,j+??,k+??]);
out[2*i+??,2*j+??,2*k+??] =
   0.25 * (in[i+??,j+??,k+??] + in[i+??,j+??,k+??] +
           in[i+??,j+??,k+??] + in[i+??,j+??,k+??]);
out[2*i+??,2*j+??,2*k+??] =
  0.125 * (in[i+??,j+??,k+??] + in[i+??,j+??,k+??] +
           in[i+??,j+??,k+??] + in[i+??,j+??,k+??] +
           in[i+??,j+??,k+??] + in[i+??,j+??,k+??] +
           in[i+??,j+??,k+??] + in[i+??,j+??,k+??]);
```

As shown in Table 12.2, this simple transformation allowed the implementation to run 8 times faster than the spec.

| Relax (Red-Black) 2-D | | | | Relax 3-D | | | | Interpolate | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| N | spec | rb2d1 | rb2d2 | N | spec | rb3d1 | rb3d2 | N | spec | interp1 | interp2 |
| 1000 | 17 | 10 | 6 | 100 | 15 | 10 | 9 | 75 | 141 | 18 | 18 |
| 2000 | 66 | 38 | 24 | 200 | 115 | 77 | 109 | 100 | 338 | 44 | 43 |
| 3000 | 153 | 84 | 54 | 300 | 385 | 236 | 634 | 150 | 1146 | 149 | 147 |
| 4000 | 264 | 148 | 97 | 400 | 923 | 585 | 1650 | 175 | 1935 | 238 | 232 |
| | | | | 500 | 1787 | 1174 | 2428 | 200 | 2822 | 339 | 335 |

Table 12.2: Running times of benchmarks implementations. The size of the grid for the Red-Black code is $\texttt{N}^2$ and $\texttt{N}^3$ for 2-D and 3-D respectively. The size of the fine grid for Interpolate is $(2\texttt{N})^3$. Time is in milliseconds.

The second sketch we did for this benchmark describes an optimization which is used by the HPF implementation of this kernel in the NAS benchmark suite [6]. The key insight behind this optimization is that a lot of the sums are computed more than once, so we can reuse some of them when computing different blocks. Figure 12.1(b) shows two consecutive blocks (A and B). The pairs `a`, `b` and `c` represent a partial sum of two points from the original grid. We can see that the partial sum `a` can be used to compute 6 different points: `A5`, `A7`, `B4`, `B5`, `B6`, `B7`. Similarly, the partial sums `b` and `c` can be reused in computing most of the other points. The implementation uses this insight by pre-computing the partial sums `a`, `a+b` and `c`; these are stored in temporary arrays for each value of (`i`, `j`), to make the rest of the computation easier to vectorize.

We wrote the sketch for this implementation trick in less than one hour. As before, the sketch leaves unspecified every single array offset and every single loop iteration bound. Below, one can see the loop that pre-computes the sub-expressions:

```
for (int i = ??; i < N-??; ++i) {
  float ta = in[i+??,j+??,k+??] + in[i+??,j+??,k+??];
  float tb = in[i+??,j+??,k+??] + in[i+??,j+??,k+??];
  float tc = in[i+??,j+??,k+??] + in[i+??,j+??,k+??];
  aplusb[i] = ta + tb;
  a[i] = ta;
  c[i] = tc;
}
```

The code for each of the expressions was sketched following Figure 12.1(b). In particular, the picture shows that the three points corresponding to case 2 are computed one from `a`, one from `c`, and one by adding the two vertices labeled `0`. Similarly, for points corresponding to case 3, one is computed from two entries from `c`, one from two entries from `a`, and one is just `a+b`. And finally, case 4 is the sum of two consecutive `a+b`. So the basic idea is clear from the picture, and can be sketched directly with the statements shown below. Nonetheless, the details of which offsets to use are not clear from the picture, so those are left unspecified for the solver to complete. The sketch resolves in less than three minutes.

```
output[k*2+??,j*2+??,i*2+??] = in[k+??,j+??,i+??];
output[k*2+??,j*2+??,i*2+??] =
  0.5 * (in[k+??,j+??,i+??] + in[k+??,j+??,i+??]);
output[k*2+??,j*2+??,i*2+??] = 0.5 * a[k+??];
output[k*2+??,j*2+??,i*2+??] = 0.5 * c[k+??];
output[k*2+??,j*2+??,i*2+??] =
  0.25 * (c[k+??] + c[k+??]);
output[k*2+??,j*2+??,i*2+??] =
  0.25 * (a[k+??] + a[k+??]);
output[k*2+??,j*2+??,i*2+??] = 0.25 * aplusb[k+??];
output[k*2+??,j*2+??,i*2+??] =
  0.125 * (aplusb[k+??] + aplusb[k+??]);
```

On the Itanium-2, this optimization had a very minimal impact on the performance compared with simple blocking. However, what is important is the fact that we were able to sketch the implementation trick, and get a complete implementation for it, all without the risk of introducing bugs. In fact, in the process of sketching these optimizations, we tried many other variations on the basic tricks. Some ideas were rejected by the compiler while others caused performance degradations. Nevertheless, we were able to try them easily and without introducing bugs.

Overall, the domain specific transformation allowed sketching to be applied successfully to the important domain of stencil computations. Without the domain specific transformations, the synthesizer would only be able to resolve the simplest stencils on very small grid sizes. More generally, we have shown that it is possible to dramatically improve the power of the synthesizer by preprocessing the sketch problem with domain specific program transformations. I believe that moving forward, this idea will allow for major breakthroughs in synthesis performance for the most challenging problem domains such as high-performance parallel programming.

# Chapter 13

# Conclusion

In the world of big software, where programs are constructed from complex frameworks by large development teams, the problems of scale and emergent complexity tend to capture most of the attention. But the problem of crafting efficient implementations of challenging algorithms has never gone away. Programming in the small is not a solved problem. In fact, the plateau in single-core performance is making this problem harder, as efficiency becomes more important, and parallelism adds a new dimension of implementation complexity. Modern languages and methodologies allow these implementations to be encapsulated and reused, but they do not address the fundamental difficulties behind these programming problems.

Software synthesis can help programmers coping with these implementation challenges by relieving them from the low-level reasoning, and allowing them to focus on the high-level algorithmic insights. We have argued, however, that making synthesis practical requires a new approach to the synthesis problem. The traditional approach to synthesis treats the human expert as an aid to the synthesizer. This approach forces the user to master the formalisms of the synthesizer in order to help it produce the desired code. In the hands of experts, these systems are very powerful, but new ideas are needed for their success to spread to the broader programming community.

Sketching offers a different approach to software synthesis. The key novelty in sketching is the use of partial programs to describe the insight behind an implementation while leaving the details unspecified. Sketching allows the programmer to communicate with the synthesizer using the familiar formalisms of imperative programming; the synthesizer is now an aid to the programmer.

The central question of this thesis was whether a synthesis approach based on combinatorial search over the possible completions for a partial program could be made to scale to real programming problems with astronomically large spaces of candidate solutions. The answer to this question is a clear yes. The thesis has shown how the SKETCH synthesizer is able to solve complex sketches from a variety of domains in a matter of minutes. Arguably, none of the sketches are very big, but a few such as AES or the MultiGrid stencil kernels constitute complete components from real systems.

The thesis has also introduced a number of important technical contributions. These include:

- The development of language extensions to allow programmers to write partial programs with clearly defined semantics.
- A formalization of the semantics of these partial programs and of the problem of sketch-based synthesis.
- The development of a novel approach to solving sketch problems based on the idea of counterexample guided inductive synthesis (CEGIS).
- The development of a SAT based inductive synthesis procedure from the formal definition of the synthesis semantics.
- The implementation and evaluation of a number of optimizations for systems of integer and boolean constraints based on dataflow analysis.
- The generalization of the CEGIS approach to the context of concurrent programs.
- The application of domain specific program transformations to simplify the sketching problem for the domain of stencil kernels.

That said, a number of research questions remain open. The most important to the success of sketching is whether this is indeed an intuitive programming model for most programmers. The personal experience of me and many of the users who have tried the system appears to suggest that it is, although some early users have complained about the difficulty of debugging sketches that turn out to have no valid solutions. It is hard to gauge to what extent this problem is an artifact of the poor quality of error messages reported by the current implementation, but the debugging problem is clearly an important area for

future research. When it comes to usability questions, however, anecdotal evidence is no substitute for systematic empirical evaluation in the form of user studies.

In short, this thesis has not solved all the problems of practical synthesis, but it has shown that given the current state of the art in constraint solving and bounded model checking, practical software synthesis is finally within reach.

# Appendix A

# List of sequential benchmarks

`compress_easy` This benchmark came from [69]. Given a bit-vector and a bit-mask, the task is select from the bit-vector all the bits selected by the bit-mask and pack them in the beginning of the word.
Bit-vector $C_{int}$ =18 $C_{bit}$ =0 $in_{bits}$ =16 53 lines

`compress_hard` A harder version of the benchmark above.
Bit-vector $C_{int}$ =41 $C_{bit}$ =0 $in_{bits}$ =16 56 lines

`doublyLinkedList` Sketches a remove method from a doubly linked list.
Datastructure $C_{int}$ =0 $C_{bit}$ =78 $in_{bits}$ =9 245 lines

`enqueue` Sketches an enqueue method for a Queue represented as a linked list.
Datastructure $C_{int}$ =481 $C_{bit}$ =48 $in_{bits}$ =15 108 lines

`karatsuba` This is the karatsuba multiplication described in Section 6.1.
Integer $C_{int}$ =1 $C_{bit}$ =56 $in_{bits}$ =16 214 lines

`listReverseEasy` An easy version of the list reversal example in the introduction. In this example, all the right-hand sides of the assignments are fully specified.
Datastructure $C_{int}$ =0 $C_{bit}$ =121 $in_{bits}$ =6 161 lines

`listReverseHarder` The list reversal benchmark from the introduction
Integer $C_{int}$ =0 $C_{bit}$ =25 $in_{bits}$ =5 40 lines

`log2` The integer *log* base 2 benchmark described in Section 2.4.
Bit-vector $C_{int}$ =5 $C_{bit}$ =320 $in_{bits}$ =32 46 lines

`log2VarLoop` Same as above, but leaves unspecified the number of iterations.
Bit-vector $C_{int}$ =9 $C_{bit}$ =512 $in_{bits}$ =32 46 lines

`logcount16` Counts the number of ones in a bit-vector in $log(N)$ steps using an idea similar to log-shifting from Section 6.1.
Bit-vector $C_{int}$ =9 $C_{bit}$ =256 $in_{bits}$ =16 26 lines

`logcount16_easy` Same as before but the user specifies that two masks should be symmetric.
Bit-vector $C_{int}$ =9 $C_{bit}$ =128 $in_{bits}$ =16 27 lines

`logcount8` Same as `logcount16` but for an 8-bit word.
Bit-vector $C_{int}$ =9 $C_{bit}$ =128 $in_{bits}$ =8 26 lines

`logcount8_easy` Same as `logcount16_easy` but for an 8-bit word.
Bit-vector $C_{int}$ =9 $C_{bit}$ =64 $in_{bits}$ =8 27 lines

`lss` This benchmark takes as input an array $[in_0, \ldots, in_W]$ of signed integers, and returns the value $max_{i,j}\{\sum_{i \leq k \leq j} in_k\}$. The problem is to compute this in linear time. This benchmark was contributed to us by a graduate student.
Integer $C_{int}$ =2 $C_{bit}$ =0 $in_{bits}$ =8 26 lines

`lss_easy` Simplified version of the above benchmark.
Integer $C_{int}$ =6 $C_{bit}$ =1 $in_{bits}$ =8 32 lines

`lss_harder` More difficult version of the above benchmark; in this instance the synthesizer has a lot more freedom than in the earlier cases.
Integer $C_{int}$ =9 $C_{bit}$ =0 $in_{bits}$ =8 32 lines

`lss_hardest` Most difficult version of lss benchmark.
Integer $C_{int}$ =44 $C_{bit}$ =0 $in_{bits}$ =8 32 lines

`merge_sort` Sketch of the well known merge sort procedure. An input array is partitioned and recursively sorted, and then an output array is populated with the values from the two halves. The synthesizer is asked to discover the function that decides what half should entry $i$ in the output array should come from, taking into account the corner case when one array has run out of values.
Integer $C_{int}$ =363 $C_{bit}$ =0 $in_{bits}$ =8 106 lines

`morton` Morton numbers sketch described in Section 6.1.
Bit-vector $C_{int}$ =18 $C_{bit}$ =512 $in_{bits}$ =32 23 lines

`morton_easier` Simplified version of morton number sketch described in Section 6.2.1
Bit-vector $C_{int}$ =9 $C_{bit}$ =256 $in_{bits}$ =16 21 lines

`morton_easiest` Simplified version of morton number sketch described in Section 6.2.1
Bit-vector $C_{int}$ =2 $C_{bit}$ =474 $in_{bits}$ =32 24 lines

`parity` Computes the xor of all the bits in a bit-vector using a strategy similar to log-shifting.
Bit-vector $C_{int}$ =10 $C_{bit}$ =0 $in_{bits}$ =32 19 lines

`Pollard` IBM Problem of the month Jan 2004, contributed by a graduate student. A read-only array with $N$ elements contains the values {1, . . . , N-1}, so it must contain a duplicate entry. Find the duplicate entry using only constant extra space.
Integer $C_{int}$ =10 $C_{bit}$ =0 $in_{bits}$ =34 52 lines

`polynomial` A toy benchmark to synthesize a polynomial of degree 4.
Datastructure $C_{int}$ =5 $C_{bit}$ =271 $in_{bits}$ =6 158 lines

`reverse` Reverse all the bits in a word in log-time using log-shifting.
Bit-vector $C_{int}$ =2 $C_{bit}$ =256 $in_{bits}$ =32 24 lines

`SetTest` Implement a treeSet from a HashSet specification, as described inSection 6.1.
Datastructure $C_{int}$ =0 $C_{bit}$ =19 $in_{bits}$ =20 37 lines

`tableBasedAddition` A toy benchmark, implements an addition of two 4-bit numbers through a table lookup. Models more complex table lookup benchmarks.
Bit-vector $C_{int}$ =0 $C_{bit}$ =1024 $in_{bits}$ =8 17 lines

`tutorial2` Toy benchmark explores the use of generators.
Integer $C_{int}$ =4 $C_{bit}$ =0 $in_{bits}$ =4 11 lines

`tutorial3` Toy benchmarks to show complex recursive generators.
Integer $C_{int}$ =259 $C_{bit}$ =0 $in_{bits}$ =6 34 lines

`xpose` Contributed by a graduate student from a problem he ran into during an internship. Models an SSE `shufps` function with a procedure and implements a transpose on a $4 \times 4$ matrix using this instruction.
Integer $C_{int}$ =38 $C_{bit}$ =96 $in_{bits}$ =32 26 lines

`xposeBit` Same as `xpose`, but takes bits instead of integers as inputs. Interestingly, it actually takes longer to solve because it needs more counterexamples to converge than the integer version.
Integer $C_{int}$ =38 $C_{bit}$ =96 $in_{bits}$ =16 36 lines

# Bibliography

[1] W. Ackermann. *Solvable cases of the decision problem.* Studies in Logic and the Foundations. of Mathematics. North-Holland,, 1954.

[2] D. Amit, N. Rinetzky, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *In 19th International Conference on Computer Aided Verification (CAV)*, 2007.

[3] S. E. Anderson. Bit twiddling hacks, 1997-2005. http://www-graphics.stanford.edu/ seander/bithacks.html.

[4] D. Andre and S. Russell. Programmable reinforcement learning agents. *Advances in Neural Information Processing Systems*, 13, 2001. MIT Press.

[5] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15(3):237–269, 1983.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[7] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[8] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 305–305, New York, NY, USA, 2003. ACM.

[9] L. Beckman, A. Haraldson, O. Oskarsson, and E. Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[10] M. Bickford, C. Kreitz, R. V. Renesse, and R. Constable. An experiment in formal design using meta-properties. In *In Proc. DISCEX-II Š01: The 2nd DARPA Information Survivability Conference and Exposition. IEEE*, 2001.

[11] A. Biere. Resolve and expand. In *Proceedings of the 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, 2005.

[12] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[13] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM, 2000.

[14] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.

[15] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proc. TACAS 2007*, March 2007.

[16] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21(5), December 1984.

[17] M. Burstein, D. Mcdermott, D. R. Smith, and S. J. Westfold. Derivation of glue code for agent interoperation. In *In Proc. 4th IntŠl. Conf. on Autonomous Agents*, pages 277–284. ACM Press, 2000.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[19] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371, May 2003.

[20] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[21] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. $10^{th}$ Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[22] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501. ACM Press, 1993.

[23] R. L. Constable. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.

[24] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Int. J. Parallel Program.*, 34(1):61–91, 2006.

[25] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal.*, 10:21–40, 2000.

[26] T. Emerson. Development of a constraint-based airlift scheduler by program synthesis from formal specifications. In *Proceedings of the 1999 Conference on Automated Software Engineering.* IEEE Computer Society Press, 1999.

[27] Advanced encryption standard (AES). U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, November 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[28] B. Fischer and J. Schumann. Autobayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3):483–508, May 2003.

[29] M. Frigo and S. Johnson. Fftw: An adaptive software architecture for the fft. In *ICASSP conference proceedings*, volume 3, pages 1381–1384, 1998.

[30] M. Frigo and V. Strumpen. The memory behavior of cache oblivious stencil computations. *The Journal of Supercomputing*, 39(2):93–112, 2007.

[31] V. Ganesh. *Decision procedures for bit-vectors, arrays and integers.* PhD thesis, Stanford, CA, USA, 2007. Adviser-David L. Dill.

[32] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* PhD thesis, 1994.

[33] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[34] C. K. Gomard. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. *ACM Trans. Program. Lang. Syst.*, 14(2):147–172, 1992.

[35] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS '05: 9th International Conference on Principles of Distributed Systems*, volume 3974, pages 3–16. Springer, 2005.

[36] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[37] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. pages 235–239. Springer, 2003.

[38] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[39] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[40] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[41] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[42] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, New York, NY, USA, 2006. ACM Press.

[43] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. A. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In B. Calder and B. G. Zorn, editors, *Memory System Performance*, pages 36–43. ACM, 2005.

[44] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[45] Z. Manna and R. Waldinger. Synthesis: Dreams => programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, 1979.

[46] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[47] A. W. Mazurkiewicz. Basic notions of trace theory. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 285–363, London, UK, 1989. Springer-Verlag.

[48] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[49] A. Mishchenko, S. Chatterjee, and R. Brayton. Dag-aware AIG rewriting: A fresh look at combinational logic synthesis. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 532–535, New York, NY, USA, 2006. ACM Press.

[50] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 652–671, London, UK, 1989. Springer-Verlag.

[51] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Foundations of Computer Science 1990*. IEEE, 1990.

[52] A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (cvt). *International Journal on Software Tools for Technology Transfer (STTT)*, 2, December 1998.

[53] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in high performance fortran. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–20, New York, NY, USA, 1997. ACM Press.

[54] H. Samulowitz and F. Bacchus. F.: Binary clause reasoning in qbf. In *In: Proc. of SAT. LNCS 4121*, pages 353–367. Springer, 2006.

[55] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[56] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):115–133, 2004.

[57] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

[58] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[59] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[60] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, 1999.

[61] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, volume 42, pages 167–178, New York, NY, USA, 2007. ACM.

[62] A. Solar-Lezama, C. Jones, G. Arnold, and R. Bodík. Sketching concurrent datastructures. In *PLDI 08*, 2008.

[63] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–294, New York, NY, USA, 2005. ACM Press.

[64] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.

[65] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 125–135, New York, NY, USA, 2008. ACM.

[66] M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 341–353, New York, NY, USA, 2006. ACM.

[67] M. T. Vechev, E. Yahav, D. F. Bacon, and N. Rinetzky. Cgcexplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 456–467, New York, NY, USA, 2007. ACM.

[68] W. Visser and K. Havelund. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12. Press, 2000.

[69] H. S. Warren. *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[70] D. Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):1–221, 2002.

[71] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 351–363, 2005.