Toward Revamping Discrete Structures: Concurrency through Functional / Constraint Programming Integration

Bruce Bolick, Ganesh Gopalakrishnan, Charles Jacobsen and Tony Tuttle
School of Computing
University of Utah, Salt Lake City
UT 84112, USA
ganesh@cs.utah.edu

***Abstract*—It is important to teach the *foundations* of both the practice of concurrency and the analysis of concurrency early in the undergraduate education. A natural place to bring in such foundations is in classes taught under the name *Discrete Structures* where many mathematical concepts are typically introduced. As extra lecture time is simply unavailable, the new material must mesh with concepts currently being taught, and will ideally help reinforce existing material. In this paper, we report on our ongoing experiments where we class-room test a collection of Functional Programming and Constraint Based Reasoning approaches in Discrete Structures. We present preliminary feedback that is encouraging, as well as the draft of a textbook and exercises. We also present our plans to ensure wider adoption, and the creation of community resources to further this thrust.**

***Keywords*-discrete structures; functional programming; constraint satisfaction; concurrency; interleavings.**

I. Introduction

Concurrency education has attained a position of central importance, yet the wide gap between demand and supply is quite palpable. In order to remedy the situation, every effort must be taken to introduce concurrency as an ongoing theme throughout one's education, and in a manner that meshes with and also reinforces the presentation of other topics one needs to learn in order to be a successful computer scientist.

Ideally, concurrency must be taught keeping two thrusts in mind. First, we must prepare students for the job market. This requires that they possess skills pertaining to productive parallel programming. Second, we are far from achieving a good understanding of how to exploit concurrency in practice. In other words, the ideas in this space will be in a constant state of flux. Therefore, students must have the wherewithal to learn on their own, after they leave school.

This paper is on introducing some of the core ideas that can help students adopt modern practices and continue to learn. Our emphasis is also on introducing these ideas into core undergraduate classes so that students can derive downstream benefits of having learned these topics early in their studies. More specifically, our emphasis is on introducing these ideas into a course called "*Discrete Structures*" that many computer-science departments have. This course typically introduces basic ideas such as mathematical logic, relations, functions, graphs, combinatorics, etc., based on books such as [1]. This severely limits the number of new things we can introduce. Also, whatever we end up choosing must really fit well with the topics already being taught in such courses. Based on these considerations, we choose *Functional Programming* and *Constraint Satisfaction* as the two topics to build upon.

In the rest of this introduction, we present how these two topics can benefit both the discrete structures education agenda as well as the concurrency education agenda. The remainder of the paper discusses the specifics of what we are experimenting with in our own offerings of Discrete Structures. We conclude with thoughts on further work, and how a community-wide effort might be undertaken.

Benefits for Discrete Structures Courses: Many Discrete Structures courses are offered the traditional way, emphasizing chalk-board based lectures as well as on-paper problem-solving skills. While these are central to one's mastery of the topics, and paper is perhaps the easiest of "programming environments" in which to lay out mathematical data structures and slowly compute on them through symbol pushing, leaving these courses entirely as "pencil and paper" has drawbacks from the point of view of effective Discrete Structures education, and is also a lost opportunity as far as the building of concurrency foundations goes.

- Capitalizing on the power of modern functional programming languages and constraint solving techniques, one can minimize the apprehension that many students initially face when confronted with mathematical topics. While one cannot come to experience the full range of power offered by mathematical functions or first-order logic quantifiers, they can at least experience how these ideas play out in a quantifier-free setting, or in the setting of recursively defined functions.
- By typing in programs and seeing them execute, students tend to remember and appreciate the true import of the ideas they are being exposed to. The benefits of this approach have already been made amply clear by pioneering pieces of work by Abelson and Sussman [2], and also recently Downey [3], [4].
- They also get to experience how ideas turn into practice.

For instance, seeing truth-tables alone leaves students with a warped impression of their utility, as they are guaranteed exponential, and cannot be illustrated at scale. Formulating puzzles and other constraint situations, on the other hand, allows a student to see that these structures can handle millions of propositional variables, and thus serve as *power tools* that can solve real-world problems.

Benefits for Modern Practices/Long-term Learning: Many present-day languages for parallel programming offer a rich collection of functional programming primitives. To use these languages effectively, students must possess basic grounding in functional programming as well as the ability to recognize situations in which these functional constructs can be effectively used. While entirely functional-style parallel programs are still a rarity, the use of functional constructs is fast-growing. Higher order functions such as `map`, `reduce`, and `filter` have permeated languages at multiple levels, and find increasing use in parallel programming [5], [6]. Considering how ideas in computer science are cyclic every few decades, and functional programming has been seen as fundamental to parallelism at least from 1978 [7], [8] it is quite natural to add functional programming as a *must learn* into one's basic skill-set to permit life-long learning. Languages such as Scala [9], X10 [10], Habanero Java [11], Copperhead [12] and Phalanx [13] (to name a few) employ interesting combinations of parallel programming constructs and functional programming constructs. Even mainstream programming languages such as C#, C++, and Java 8 now support the use of Lambda expressions.

Our Proposal: In this paper, we present excerpts from a discrete structures course being offered the second time during Fall 2013. A companion textbook is also being written [14], and all our software will be released open source at the end of this course. These ideas are introduced as natural extensions to existing discrete structures topics. They can therefore be exploited by those wanting to build further parallel and discrete programming topics in later courses.

Our choice of the programming language for illustration is Python, which is gaining considerable attention in many circles—all the way from its pedagogical uses to serious usage in HPC packages such as NumPy [15] and SciPY [16]. It also finds uses in CUDA and OpenCL programming (e.g., Pycuda [17] and PyOpenCL [18]). We also employ the world-renowned SMT solver from Microsoft Research (namely, `Z3` [19]) embedded into Python as a package `Z3Py` (freely available for academic use).

Roadmap: The rest of this paper is organized as follows. §II provides details of the topics that we propose for modern discrete structures classes. §III takes a deep-dive into the topic of computing program interleavings, and touches upon the variety of approaches one can consider teaching. §IV provides additional discussions and also details of student and instructor material that can be taken as a starting point to evolve the theme of this paper further, and generate course material as well as exercises. Concluding remarks are in §V.

II. Overview of Illustrations

We now illustrate a progression of topics that capture our proposal a bit more concretely. The exercises in this section were already class-room tested during Spring 2013, with positive feedback from the students.

Set comprehensions: Students can create and combine sets in Python with syntax similar to standard set notation, and carry out 'proofs' using specific sets. Here is an example of a set written in Python:

```
>>> A = { i for i in range(2,17) \
          if (i%2 == 0) }
set([2, 4, 6, 8, 10, 12, 14, 16])
```

Python has good support for functional programming– map, reduce, lambdas, and higher-order functions. The expression below calculates 10! using reduce:

```
# Factorial 10
>>> reduce(lambda x,y: x*y, range(1,11))
3628800
```

Familiar identities can be directly cast into recursive programs.

Typical Exercise: Show that $\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1}$.

Recursive Formulation and Solution:

```
def chooseRec(N, k):
    if N == k or k == 0:
        return 1
    else:
        return chooseRec(N - 1, k) + chooseRec(N - 1, k - 1)
```

Pascal's Triangle via recursion and comprehension:

```
>>> for N in range(8):
      print [chooseRec(N,k) for k in range(N+1)]
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

Map and Reduce: # Vowels in Gettysburg The following code reinforces more functional programming concepts:

```
Gettysburg = """Four score and seven years ago \
our fathers brought forth on this continent, a \
new nation, ... government of the people, by \
the people, for the people, shall not perish \
from the earth."""

vowels = {'a','e','i','o','u','A','E','I','O','U'}

reduce(lambda x,y:x+y, map(lambda x: 1 if x in \
           vowels else 0, Gettysburg))
```

Map Coloring via `Z3Py`

This illustrates the notion of an *adjacency list* and also how *graph coloring* can be formulated declaratively and solved using `Z3Py`.

```
from z3 import *

# Adjacency list of the graph of western US states
#
grWest =
 { 'AZ' : ['CA', 'NV', 'UT', 'NM'],
   'MT' : ['ID', 'WY', 'ND', 'SD'],
   'WY' : ['MT', 'ID', 'UT', 'CO', 'SD', 'NE'],
   'UT' : ['ID', 'NV', 'AZ', 'CO', 'WY'],
   'ID' : ['WA', 'OR', 'NV', 'MT', 'WY', 'UT'],
   'CO' : ['WY', 'UT', 'NM', 'NE', 'KS', 'OK'],
   'WA' : ['ID', 'OR'],...
 }

# Sanity-check that the graph is symmetric
ChkGraphIsSymmetric(grWest)

# Obtain the keys
stLstWest = grWest.keys()  # [ 'WA', 'OR', ..]

# Obtain the number of states
NStates = len(stLstWest)

# Form a list of constraint variables
LInts = Ints(stLstWest)

# Pairings of state names and constraint vars
stNamIntDict = { stLstWest[i] : LInts[i] for i in \
                  range(NStates) }

# Try for 3-coloring
CMax = 3

# Create a solver
s = Solver()

# Generate and add constraints
for state in stLstWest:

    #-- Default color range constraints
    s.add(stNamIntDict[state] >= 0)
    s.add(stNamIntDict[state] <= CMax)

    #-- These are the graph coloring constraints
    s.add(And([stNamIntDict[state] != \
                  stNamIntDict[adjState] \
                       for adjState in grWest[state]]))

# See if colorable
#
rslt = s.check()

# If so, find a model; else report failure
if (rslt == sat):
              print "The coloring of states is"
              print(s.model())
else:
   print "There is no " + str(CMax) + \
            " coloring for the given graph."
```

III. Computing Program Interleavings

In this section, we present two alternate solutions to the same problem, namely how to compute the number of thread interleavings in a shared memory situation. When two threads interleave, their actions riffle-shuffle, as if with decks of cards. The first solution to calculate the growing number of interleavings (using recursion) has been class-room tested, and helps reinforce combinatorics. The second solution (using constraint solving using `z3py`) has not been class-room tested yet. Depending on the makeup of the class and available time, it can be offered either during normal lectures or as extra material for independent projects. It helps reinforce mathematical logic, and exposes students to formal correctness checking tools such as [20], [21].

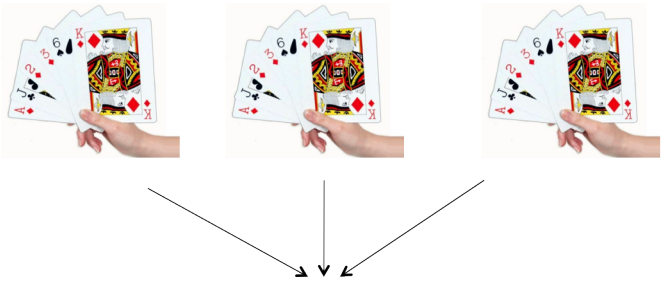

Figure 1. Shuffling Decks of Cards

Suppose we riffle-shuffle decks of cards (Figure 1). How many shuffled sequences can we create?

- Shuffling N decks of k cards each generates $(N \cdot k)!/(k!)^N$ shuffles.
- So for 5 decks of 5 cards, each, we can generate over 6×10^{14} riffle-shuffles. Taking advantage of the familiar context of cards and shuffles, one can emphasize how fast the number of concurrent interleavings of threads grow—in this example, even with 5 threads, each executing 5 atomic steps each.

Python to the shuffling rescue:
Let us obtain an indication of how many shuffles there are for 2 decks of N cards.

```
def shuffles(L1,L2):
    if (L1==[]):
      return [L2]
    else:
      if (L2==[]):
        return [L1]
      else:
        return
         list(map(lambda x:\
               ([L1[0]] + x), shuffles(L1[1:], L2))) +
         list(map(lambda x:\
                ([L2[0]] + x), shuffles(L1, L2[1:])))
```

Based on the structure of this recursive program, one can teach how to prove the equation for the number of shuffles by induction.

SMT formulation of Interleavings / Riffle Shuffles

We now build up ideas leading to SMT-based encoding of interleavings. We describe a *translation function* Γ that translates a program statement into a set of contraints that must be satisfied by a given scheduler and number of threads. This translation was introduced in [22] and recently

employed in [23] in a symbolic race checker for OpenMP programs.

$v = 1$ ***with 2 threads:*** Suppose the assignment statement `v = 1` is being carried out concurrently by two threads.

Our approach is to treat v as an array indexed by time, containing the values of v at different time points, where $v[-1]$ is the initial value of v. (If v were an array, we would simply add one more dimension to v.)

For this simple example, the scheduler will only perform a *write* operation for the two threads (2 operations total). Let

- $v_1^{t_1}$ = the time the scheduler schedules a write for thread 1,
- $v_1^{t_2}$ = the time the scheduler schedules a write for thread 2,

the *schedule identifiers* SId for the operations, taking distinct values in $\{0, 1\}$ as there are only two operations. (Schedule identifiers are like timestamps and combine two pieces of information: which thread is accessing it (superscript), and where in the code the access is occurring (subscript), forming single static assignment indices.)

With this notation,

$$\Gamma(\texttt{v = 1}) = \left\{v[v_1^{t_1}] = 1,\ v[v_1^{t_2}] = 1\right\}$$

and

$$v = [v_0, 1, 1].$$

To say that t_1 accesses (writes) into v first, we can throw in the constraint $v_1^{t_1} < v_1^{t_2}$. To say that either access order is possible, we do not throw in any constraint.

$v = v + 1$ ***with 2 threads:*** Now, things get more interesting when we translate `v = v + 1` assuming 2 threads. On each thread, a read and a write must be performed, giving four total operations and the identifiers $v_1^{t_1}$, $v_1^{t_2}$, $v_2^{t_1}$, and $v_2^{t_2}$, taking on distinct values in $\{0, 1, 2, 3\}$.

Now, it is clear that $v[v_2^{t_1}] = v[v_1^{t_1}] + 1$ and $v[v_2^{t_2}] = v[v_1^{t_2}] + 1$ model how "assignment works." It is also clear that $v_2^{t_1} > v_1^{t_1}$ and $v_2^{t_2} > v_1^{t_2}$ model that the L-value is updated only after the R-value is obtained. Now what about the R-value itself? This depends on "who wrote v last." This is precisely why we include $v[v_1^{t_1}] = v[v_1^{t_1} - 1]$ and $v[v_1^{t_2}] = v[v_1^{t_2} - 1]$. These constraints say that the R-value is defined by the access occurring at the "previous" time instant. Here, we do not pin down what the action performed at the previous moment was; we only specify that the action occurred one time step earlier with respect to fetching the R-value. During this earlier time, we might either have defined the L-value from the other `v=v+1` assignment (in which case the effect of that assignment statement would be seen) or merely have accessed the R-value of the other assignment statement (in which case the effect of the other assignment statement would not be seen). It is interesting that this system, in one fell swoop, models all the six schedules possible. We get

$$\Gamma(\texttt{v = v+1}) = \left\{\begin{array}{c} v[v_2^{t_1}] = v[v_1^{t_1}] + 1, \\ v[v_2^{t_2}] = v[v_1^{t_2}] + 1, \\ v_2^{t_1} > v_1^{t_1}, \\ v_2^{t_2} > v_1^{t_2}, \\ v[v_1^{t_1}] = v[v_1^{t_1} - 1], \\ v[v_1^{t_2}] = v[v_1^{t_2} - 1] \end{array}\right\}$$

where the values for v now depend on the values of the schedule identifiers.

Example: Suppose $v[-1] = 5$, $v_2^{t_2} = 2$, $v_2^{t_1} = 3$, $v_1^{t_2} = 1$, and $v_1^{t_1} = 0$. Then we have expressed these constraints: $v[3] = v[0] + 1 \ \wedge \ v[2] = v[1] + 1 \ \wedge \ v[1] = v[0]$. In this example, we are modeling the following schedule that, overall, increments v by 1, and not 2: (i) $v[1] = v[0]$ models that thread t_2 also "enjoys" the initial value of v in addition to t_1; (ii) $v[2] = v[1] + 1$ models that thread t_2 now does the update of this v; (iii) finally $v[3] = v[0] + 1$ models that t_1 now takes the value it had read "long ago," is incrementing that value, and depositing it into v.

Modeling barriers and other synchronizations: We can model synchronization primitives quite naturally in this setting by introducing suitable constraints on the schedule IDs. See [22] for details.

`Z3py` based schedule generation experiments

Here is a complete script, including execution results, of interleaving generation illustrated using Z3py.

```
# Path setup
# import sys
# sys.path.append("z3/build")
# import z3

# Initial value of v
v0 = 5

# Model v as a function from Int -> Int
v = z3.Function("v", z3.IntSort(), z3.IntSort())

# Four scheduling variables
v_1_t1 = z3.Int("v_1_t1")
v_2_t1 = z3.Int("v_2_t1")
v_1_t2 = z3.Int("v_1_t2")
v_2_t2 = z3.Int("v_2_t2")
idx_vars = [v_1_t1, v_2_t1, v_1_t2, v_2_t2]

###--- Create a solver and add main constraints ---###
s = z3.Solver()

# Define v(-1) to be the initial value of v
s.add(v(-1) == v0)

# All scheduling variables are distinct and in {0,1,2,3}
s.add([z3.And(idx >= 0, idx <= 3) for idx in idx_vars])
s.add(z3.Distinct(idx_vars))

# Semantics of assignment statement
s.add(v(v_2_t1) == v(v_1_t1) + 1)
s.add(v(v_2_t2) == v(v_1_t2) + 1)

# Scheduling order
s.add(v_2_t1 > v_1_t1)
s.add(v_2_t2 > v_1_t2)

# Interleaving effect
s.add(v(v_1_t1) == v(v_1_t1 - 1))
```

```
s.add(v(v_1_t2) == v(v_1_t2 - 1))

# Generate all possible interleavings
il = []
while s.check() == z3.sat:

    m = s.model()
    s.add( z3.Or( [idx != m.evaluate(idx) for \
                      idx in idx_vars] ) )

    il.append(m)

print "Found " + str(len(il)) + " interleavings."
print

for m in il:
    print
    print "###--- Interleaving ---###"
    print
    print "Timeline for v:",
    print ( str(m.evaluate(v(0))) + " -> " +
            str(m.evaluate(v(1))) + " -> " +
            str(m.evaluate(v(2))) + " -> " +
            str(m.evaluate(v(3))) )
    print
    print "Other variables:"
    print
    print "v_1_t1: " + str(m.evaluate(v_1_t1))
    print "v_2_t1: " + str(m.evaluate(v_2_t1))
    print "v_1_t2: " + str(m.evaluate(v_1_t2))
    print "v_2_t2: " + str(m.evaluate(v_2_t2))

#*******************************************************
#*******************************************************
# HERE ARE THE EXECUTION RESULTS
#*******************************************************
#*******************************************************

>>> execfile("interleavings.py")
Found 6 interleavings.


###--- Interleaving ---###

Timeline for v: 5 -> 6 -> 6 -> 7

Other variables:

v_1_t1: 0
v_2_t1: 1
v_1_t2: 2
v_2_t2: 3

###--- Interleaving ---###

Timeline for v: 5 -> 5 -> 6 -> 6

Other variables:

v_1_t1: 1
v_2_t1: 3
v_1_t2: 0
v_2_t2: 2

###--- Interleaving ---###

Timeline for v: 5 -> 6 -> 6 -> 7

Other variables:

v_1_t1: 2
v_2_t1: 3
v_1_t2: 0
v_2_t2: 1

###--- Interleaving ---###

Timeline for v: 5 -> 5 -> 6 -> 6

Other variables:

v_1_t1: 1
v_2_t1: 2
v_1_t2: 0
v_2_t2: 3

###--- Interleaving ---###

Timeline for v: 5 -> 5 -> 6 -> 6

Other variables:

v_1_t1: 0
v_2_t1: 2
v_1_t2: 1
v_2_t2: 3

###--- Interleaving ---###

Timeline for v: 5 -> 5 -> 6 -> 6

Other variables:

v_1_t1: 0
v_2_t1: 3
v_1_t2: 1
v_2_t2: 2
>>>
```

IV. Discussions, Including Student Benefits

Most of the material accompanying this paper is already present in our CS 2100 book draft available at http://www.cs.utah.edu/~ganesh. All exercises and a virtual machine containing the necessary software has been created, and can be provided to a common repository. Our use of Python also fits naturally with Python based concurrency education espoused by many (e.g., [24]), and we are hopeful that a common repository of this material can ultimately be created and kept updated, well-vetted, and documented.

Bloom Taxonomy: According to Bloom's taxonomy, the students will achieve the following benefits vis-a-vis the following concepts:

- Knowledge applying Functional / Declarative programming principles
- Constraint satisfaction and applied Mathematical Logic
- Applications of Map/Reduce
- Familiarity with concurrency and the growth of the number of interleavings (including synchronization primitives not shown here)

We wish to reiterate the reason for emphasizing fundamental concepts as much as concrete examples of today's languages and artifacts. As many new ideas are explored in the the upcoming world of parallelism, computations are bound to become more heterogeneous and asynchronous in nature. As one more glimpse at how rapidly things are changing, we would like to elaborate on a recent paper [25] that introduces the idea of employing constraints during dynamic scheduling. In this paper, the authors consider iterative and fixed-point algorithms for graphs and simulations. In these simulations, a data item only needs to be updated if its neighbors change. With data-driven parallelism, the task of updating the data item can be tied to the item's neighbors, so

that the task will be executed when one of them is modified, saving on unnecessary computations. The authors found that in order to be effective, data-driven parallelism needs to be constrained: Tasks need to be executed in a certain order (LIFO versus FIFO) and in phases. They achieved significant speedup over traditional fixed-point parallel algorithms for the Single-Source Shortest Path and other graph-related problems.

V. Conclusions

It is clear that many basic topics need to be covered before students are adquately prepared to face the "parallel future." The message of this paper has been to argue that one standard course—namely *Discrete Structures*—can be revamped without disrupting its traditional offerings too much, and yet yield significant downstream benefits to students, both for the remainder of their studies, as well as for life-long learning. As one detailed study, we showed how declarative and constraint-based thinking of thread interleavings can be introduced in this setting. The code for the proposed exercises was presented within three pages of this paper, and is being viewed as a "nugget" that, with suitable background preparation, most existing discrete structures courses can incorporate without too much disruption. Clearly, community-wide effort (e.g., through the NSF/IEEE-TCPP Curriculum Initiative, [26]) can help vet the material sufficiently, and make it suitable for wider adoption and also help keep the material up to date.

Parallelism and concurrency will increasingly need new programming approaches that emphasize specification of the "what" and not the "how," so that compilers can take maximum liberties with the code, instead of preserving all the semantics (intended and unintended) of imperative updates. Introducing this thinking early in CS curricula can better prepare students to absorb concurrency in following courses.

Acknowledgment

The authors would like to thank students in their Fall 2013 class of CS 2100 for their willingness to participate in this new approach to learning Discrete Structures.

References

[1] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 1999.

[2] Harold Abelson and Gerald J. Sussman. Structure and interpretation of computer programs, 1996. http://mitpress.mit.edu/sicp/full-text/book/book.html.

[3] Allen B. Downey. *Think Python - How to Think Like a Computer Scientist*. O'Reilly, 2012.

[4] Allen B. Downey. *The Little Book of Semaphores*. The Green Tea Press.

[5] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 181–192, 2012.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[7] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[8] Guy E. Blelloch. *Prefix Sums and Their Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1993.

[9] Mohsen Lesani, Martin Odersky, and Rachid Guerraoui. Concurrent Programming Paradigms, A Comparison in Scala. Technical report, 2009.

[10] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the Twentieth Annual ACM SIGPLAN Conference on Obj ect-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 519–538, October 2005.

[11] Habanero Multicore Software Research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 735–736, New York, NY, USA, 2009. ACM.

[12] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[13] Michael Garland, Manjunath Kudlur, and Yili Zheng. Designing a unified programming model for heterogeneous machines. In *SC '12: Proc. Conference on High Performance Computing Networking, Storage and Analysis*, November 2012.

[14] A Discrete Structures book being written for CS 2100 may be found here: http://www.cs.utah.edu/~ganesh. A second edition revision is being prepared.

[15] http://www.numpy.org/.

[16] http://www.scipy.org/.

[17] https://pypi.python.org/pypi/pycuda.

[18] https://pypi.python.org/pypi/pyopencl.

[19] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[20] Guodong Li, Peng Li, Geof Sawaga, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic verification and test generation for GPUs. In *PPoPP*, 2012.

[21] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, 2012.

[22] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In Foundations of Software Engineering, 2010. http://www.cs.utah.edu/fv/PUG tool.

[23] Hongyi Ma, Qichang Chen, Liqiang Wang, Chunhua Liao, and Daniel Quinlan. Openmp-checker: Detecting concurrency errors of openmp programs using hybrid program analysis, September 2012.

[24] Steven Bogaerts and Joshua Stough. Python for parallelism in introductory computer science education. In *Supercomputing*, 2012. http://sc12.supercomputing.org/schedule/event_detail.php?evid=eps105.

[25] Tim Harris, Yossi Lev, Victor Luchangco, Virendra Marathe, and Mark Moir. Constrained Data-Driven Parallelism. In *5th USENIX Workshop on Hot Topics in Parallelism (HotPar 13)*, San Jose, CA, USA, June 2013.

[26] NSF/IEEE-TCPP Curriculum Initiative. http://www.cs.gsu.edu/~tcpp/curriculum/.