Domain-Specific Symbolic Compilation

Rastislav Bodik[1], Kartik Chandra[2], Phitchaya Mangpo Phothilimthana[3], and Nathaniel Yazdani[4]

1 **University of Washington, Seattle, WA, USA**
`bodik@cs.washington.edu`
2 **Gunn High School, Palo Alto, CA, USA**
`kartikchandra@acm.org`
3 **University of California, Berkeley, CA, USA**
`mangpo@cs.berkeley.edu`
4 **University of Washington, Seattle, WA, USA**
`nyazdani@cs.washington.edu`

Abstract

A *symbolic compiler* translates a program to symbolic constraints, automatically reducing model checking and synthesis to constraint solving. We show that new applications of constraint solving require domain-specific encodings that yield orders of magnitude improvements in solver efficiency. Unfortunately, these encodings cannot be obtained with today's symbolic compilation.

We introduce *symbolic languages* that encapsulate domain-specific encodings under abstractions that behave as their non-symbolic counterparts: client code using the abstractions can be tested and debugged on concrete inputs. When client code is symbolically compiled, the resulting constraints use domain-specific encodings.

We demonstrate the idea on the first fully symbolic checker of type systems; a program partitioner; and a parallelizer of tree computations. In each of these case studies, symbolic languages improved on classical symbolic compilers by orders of magnitude.

1998 ACM Subject Classification D.2.2 [Software Engineering] Design Tools and Techniques, D.3.3 [Programming Languages] Language Constructs and Features

Keywords and phrases Symbolic evaluation, constraint solvers, program synthesis

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.2

1 Introduction

A symbolic compiler translates a program `p` to constraints that model its behavior [3, 18, 20]. The unknowns in the constraints typically represent the symbolic inputs to `p`, and the solution to the constraints is an input that induces a particular program behavior. For example, with a symbolic compiler and a solver, by just writing program `p`, we obtain a program checker – producing an input to `p` that leads to an assertion failure – for free.

The applications of symbolic compilation become even more interesting when the input to the program `p` is itself a program:

- If the program `p` is an interpreter, then constraint solving can find a program that forces the interpreter into a violation due to unsoundness in its type system. In Section 3, we explore finding such witnesses to unsoundness by symbolically compiling interpreters.
- If `p` is a type checker, then constraint solving performs type inference. In Section 4, we partition a program onto a many-core processor. We model this program transformation with a hardware-specific place type system, relying on symbolic compilation of type checkers to produce constraints for type inference.

- If `p` is an "execution runtime" for parallel programs, then constraint solving finds a parallel execution strategy, effectively parallelizing `p` for us. In Section 5, we synthesize parallel evaluators for attribute grammars by modeling the strategies as schedules and symbolically compiling interpreters of such schedules.

More formally, assume we have access to a solver `sol` that accepts a constraint ϕ and returns a solution, *i.e.,* a value x such that $\phi(x)$ holds. If no such x exists, then `sol` returns $\perp$. A symbolic compiler `sym` translates a program `p` into a logical formula ϕ that models the input-output semantics of `p`. It is convenient to think of a symbolic compiler as an execution inverter: `sym` accepts `p` and an output value y and produces a formula ϕ over the program input variable `x` such that the solution x to ϕ makes the program output y.

Model checking and program synthesis are two common applications of symbolic compilation. In bounded model checking, we want to compute a program input that leads to a failure. First, we modify the program so that failed assertions exit the program, returning a special value *fail*. The call $\texttt{sol}(\texttt{sym}(\texttt{p}, \textit{fail}))$ produces the failing input if one exists.

In inductive synthesis, we have a sketch program $\texttt{def sk}(\texttt{x}, \texttt{h}) = e$ where e uses an (unknown) function `h`. We want to find a function `f` such that substituting `f` for `h` gives `sk` the desired behavior. The behavior is often given with an input-output pair of values (x_0, y_0), *i.e.*, we want $\texttt{sk}(x_0, \texttt{f})$ to evaluate to y_0. In many settings, y_0 is simply *success* or *fail*. Symbolic compilation produces such a function `f` with the call $\texttt{sol}(\texttt{sym}(\texttt{sk}(x_0), y_0))$. The notation $\texttt{sk}(\texttt{x}_0)$ is a partially applied function `sk`, *i.e.*, a function of `h`. Note that to produce a function, the solver need not be second-order; the function `f` can be represented as a list of constants that define a derivation of the syntax of `f` from a suitable grammar.

Revisiting the three case studies clarifies the task of symbolic compilation:

- *Checking soundness of type systems.* We want to check the type system of an interpreter `int` that is composed of a type checker and an evaluator. Assume that the interpreter outputs *fail* when a program passes the type checker but fails in the evaluator. The call $\texttt{sol}(\texttt{sym}(\texttt{int}, \textit{fail}))$ then finds a program that is deemed type-safe but encounters a run-time violation. (We assume the interpreted programs have no parameters.) The benefit of using symbolic compilation is that one needs just an implementation of the interpreter. There is no need for fuzzers or tools that translate language semantics to constraints.
- *Program partitioning.* A spatial type system maps variables and operations to CPU cores, partitioning the program. If the typechecker has two parameters, a program `r` and the values of `r`'s type variables, then the call $\texttt{sol}(\texttt{sym}(\texttt{typechecker}(\texttt{r}), \textit{success})$ produces the type assignment to the program that partitions the program satisfying all program and hardware constraints checked by the type checker. Symbolic evaluation produces type constraints where unknowns are the type variables. Solving performs type inference. Formulating type inference as constraint solving is nothing new, of course. Our goal is to make this idea easier to apply by automatically obtaining high-performance type inference from a type checker.
- *Parallelizing tree computations.* We want to efficiently compute the attributes of a tree `t` defined by an attribute grammar `G`. The parallel tree evaluator may need to perform multiple tree traversals, some bottom-up, some top-down, some in-order, subject to value dependences in `G`. The evaluation strategy can be described with a schedule of traversals. The schedule language is defined with an interpreter that reads the grammar, the tree, and the schedule. Symbolic evaluation of the interpreter can give us a legal schedule for free with the call $\texttt{sol}(\texttt{sym}(\texttt{int}(\texttt{G}, \texttt{t}), \textit{success})$. Constraint solving sidesteps the error-prone process of analyzing the grammar and operationally constructing a valid schedule.

2 Architectures for Symbolic Compilation

We discuss three approaches for generating constraints. We first compare two existing architectures – constraint generators and general-purpose specializing symbolic compilers – and then introduce domain-specific symbolic compilers.

We describe the architectures by composing interpreters, compilers, and specializers. Borrowing the notation from [4], we summarize here their definitions:

- interpreter $\texttt{int} : [\![\texttt{int}]\!](\texttt{p}, \texttt{x}) = [\![\texttt{p}]\!]\ \texttt{x}$
- compiler $\texttt{comp} : [\![[\![\texttt{comp}]\!]\ \texttt{p}]\!]\ \texttt{x} = [\![\texttt{p}]\!]\ \texttt{x}$
- specializer $\texttt{s} : [\![[\![\texttt{s}]\!]\ (\texttt{p}, \texttt{x}_\texttt{s})]\!]\ \texttt{x}_\texttt{d} = [\![\texttt{p}]\!](\texttt{x}_\texttt{s}, \texttt{x}_\texttt{d})$
- symbolic compiler $\texttt{sym} : [\![\texttt{p}]\!]\ ([\![\texttt{sol}]\!]\ ([\![\texttt{sym}]\!]\ (\texttt{p}, \texttt{y}))) = \texttt{y}$

2.1 Constraint generators

A generator `gen` reads a problem instance and produces constraints whose solution solves the problem instance. As our running example, consider the synthesis of schedules for parallel tree evaluation that we introduced above. The problem instance is an attribute grammar `G` and the call `sol(gen(G))` produces the schedule for `G`.[1]

Notice that a constraint generator `gen` is not asked to invert a program, unlike the symbolic compiler `sym`. This frees the author of the generator to employ a clever problem-specific encoding. For example, Gulwani *et al.* phrased synthesis of loop-free programs as constraint-based synthesis of a network that connects available instructions [6]. Kuchcinski solved scheduling and resource assignment by modeling a system as a set of finite-domain variables [10]. Hamza *et al.* synthesized linear-time programs by converting an automaton recognizing the input/output relation [7].

On the other hand, since the generator receives only a problem instance but not the program to be inverted, the semantics of generated constraints must entirely come from the author of the generator. Consider again the synthesis of schedules: `sym` received the schedule interpreter, which it can use to automatically produce constraints that encode schedules. In contrast, the semantics of schedules must be hard-coded into `gen` by the programmer.

Our running example shows why implementing generators is challenging. The programmer needs to wrangle the semantics of three languages – the language of attribute grammars `AG`, the language of schedules `Sch`, and the constraints language Φ – reasoning across a four-step indirection:

1. The programmer reads the specifications of the three languages and writes a constraint generator `gen`.
2. The generator `gen` reads the grammar `G` and outputs a constraint ϕ.
3. The solver `sol` reads ϕ and produces a solution σ.
4. The solution σ indirectly encodes a schedule $\texttt{s} \in \texttt{Sch}$.
5. The schedule `s` evaluates a tree according to the input grammar $\texttt{G} \in \texttt{AG}$.

The programmer must ensure that the generator written in Step 1 produces a schedule that in Step 4 correctly encodes, say, in-order traversals and in Step 5 evaluates the tree in accordance with the attribute grammar semantics. This reasoning may explain why in

[1] The solution to constraints must be typically converted back to the problem domain. For example, if using SAT constraints, a Boolean vector that solves the SAT problem is translated to a program in the scheduling language. This code-generation problem is important but we ignore its automation in this paper.

our previous work on synthesizing schedules, we failed to fully debug our generator once the schedule language became moderately sophisticated.

2.2 General-Purpose Specializing Symbolic Compilers

This is the architecture of Sketch [17] and to a large extent Rosette [20]. The architecture has three components and relies heavily on specialization:

1. $\texttt{int} : (D \to D)_L \to D \to D$, an interpreter implemented in a metaprogramming language L_m. The interpreter implements the language L of the input program `p`. It accepts the program $\texttt{p} : (D \to D)$ and `p`'s input, producing `p`'s output.
2. $\texttt{s} : (D \to D \to D)_{L_m} \to D \to (D \to D)_{L_s}$, a specializer of programs in L_m producing programs in L_s. In particular, `s` will specialize `int` with respect to `p`, producing a residual program $\texttt{int}_\texttt{p}$.
3. $\texttt{xlate} : (D \to D)_{L_s} \to D \to \Phi$ translates a symbolic program to the language of constraints Φ. `xlate` also receives the output value $y \in D$ and produces a formula $\phi(\texttt{x})$ that is satisfied iff `p(x)` outputs y.

The symbolic compiler is thus $\texttt{sym}(\texttt{p},\texttt{y}) \triangleq \texttt{xlate}(\texttt{s}(\texttt{int},\texttt{p}),\texttt{y})$. In Rosette, L_m is a subset of Racket maintaining many metaprogramming facilities of Racket; L_s is the symbolic expression language; and C can be one of several subsets of SMT languages, such as the bitvector language. Note that `s` and `xlate` are part of the framework, while `int` is developed by the user.

The core of symbolic evaluation happens in the specializer which explores all paths of the program, producing a functional residual program that reflects the shape of the final constraints. The translator `xlate` performs algebraic optimizations followed by local code generation from the residual program to the constraint language.

The downside of this architecture is that symbolic compilation must typically follow the forward symbolic execution that merges constraints under their path conditions [9]. This algorithm may suffer from path explosion and does not lend itself to constraints other than SAT or SMT. Thus, integer linear programming (ILP) constraints – often domain-specific and highly efficient – are often the constraints of choice produced by constraint generators.

2.3 Domain-specific symbolic compilers

We modify the specializing architecture by introducing a new abstraction for implementing the interpreter of L. This interpreter, $\texttt{int}_{Ld}$, now has two parts:

1. $\texttt{int}_L$, an interpreter of L implemented in the *domain-specific symbolic* language L_d.
2. $\texttt{int}_d$, an interpreter of L_d implemented in a metaprogramming language L_m.

The symbolic compiler pipeline is now $\texttt{xlate}(\texttt{s}(\texttt{int}_{Ld},\texttt{p}),\texttt{y}))$. Ideally, the interpreter $\texttt{int}_d$ meets two informally stated properties: (1) symbolic evaluation of $\texttt{int}_d$ produces easier-to-solve constraints; and (2) symbolic evaluation of $\texttt{int}_d$ is faster than that of `int`, for example because L_d reduces path explosion.

This paper shows that domain-specific symbolic compilers can be implemented as a library on top of a classical symbolic compiler such as Rosette [19]. The library provides a *symbolic language* that implements the domain-specific encoding while hiding the encoding from both the programmer and the underlying symbolic compiler.

In Section 3, we check type systems for soundness errors. We introduce a *Bonsai tree* that serves as the symbolic input into a type checker and an interpreter, which are implemented on top of the Bonsai library. Symbolic evaluation of the two components produces constraints

that encode a space of abstract syntax trees (ASTs). The solution is a witness: a tree that succeeds in the type checker but fails in the interpreter. The Bonsai tree has the usual interface but internally produces a special encoding that allows symbolically evaluating the interpreter on trees that are not necessarily syntactically correct or type correct. This is key to finding witnesses, for the first time, without enumerating or sampling the program space, allowing us to compute the witness for a tricky soundness bug [1].

In Section 4, we partition a program onto a many-core processor. Mapping of program operations to cores is modeled with a place type system ensuring that each code fragment fits into its core. Partitioning is thus type inference. To infer types, we symbolically evaluate the type checker with respect to a program whose type variables are symbolic. We design a symbolic language for querying properties of the symbolic location of a computation. Under this abstraction, we switch from the standard SMT encoding to an ILP encoding. The resulting ILP encoding solves previously inaccessible partitioning problems.

Finally, in Section 5, we synthesize parallel tree programs as used in page layout and data visualizations. The programs are formalized as schedules for evaluation of attribute grammars. We design a *symbolic trace language*, an abstraction for writing interpreters of such schedules. Under this abstraction, we can (1) sidestep the expensive symbolic state that is maintained by the standard symbolic evaluator and (2) switch from ensuring that all dependences are met to ensuring that all anti-dependences are avoided.

3 Checking Type Systems with Bonsai Trees

Model checking of type systems. Bonsai uses model checking to search for soundness errors in type systems. The user provides a typechecker and an interpreter for their language, and Bonsai searches for a *counterexample* program that passes the typechecker while causing the interpreter to crash. If such a counterexample can be found, then it is evidence of a soundness bug in the type system. Furthermore, such a counterexample provides helpful feedback for the user to understand and fix the bug. On the other hand, if *no* counterexample can be found, the user has some assurance that the typechecker is sound.

The most common existing typechecker-checking technique is *fuzzing*. A fuzzer generates random terms and uses them to test a typechecker and interpreter. Fuzzers may sample from the space of syntactically-correct terms or the space of well-typed terms; however, in both cases, the probability of generating a counterexample by chance is extremely low. Thus, fuzzers often need hours or days of guessing to find even simple type checker bugs (an example of a "simple" bug is assigning `cons` the return type `a` instead of `Listof a`).

Bonsai can be regarded as a final successor to typechecker fuzzing: rather than randomly sampling from the space of syntactically-correct or well-typed terms, Bonsai symbolically compiles an executable language specification to constraints, and then utilizes the backward reasoning of a constraint solver to sample *directly* from the space of counterexamples. This makes Bonsai much more efficient than traditional fuzzers: Bonsai finds the above bug in just 1.3 seconds compared to hours or days needed by fuzzers.

Bonsai consists of the algorithm shown in Figure 1. First, Bonsai initializes a symbolic representation of A, the set of all trees up to some maximum size m. Next, it computes symbolic representations of the subsets of A that (a) are syntactically valid; (b) pass the typechecker; (c) fail in the interpreter. Finally, it asks the solver to find a tree in the intersection of these three sets. If such a tree exists, it represents a counterexample.

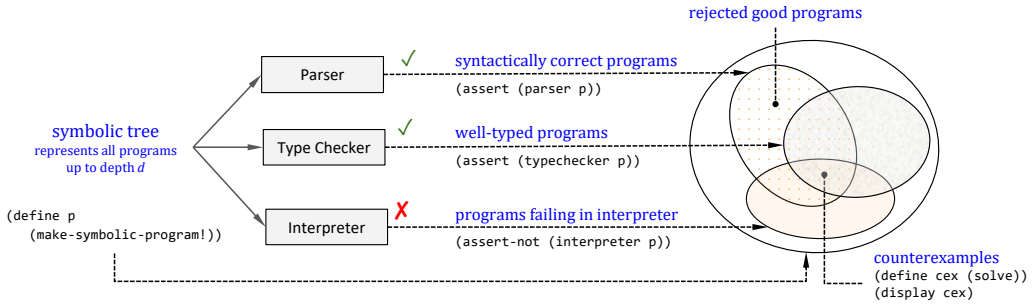


Figure 1 Bonsai performs three independent symbolic evaluations, interestingly executing the interpreter also on trees that are both syntactically and type-incorrect.

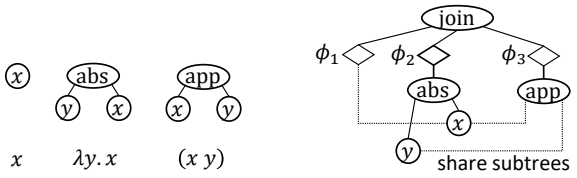
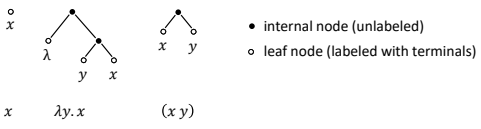
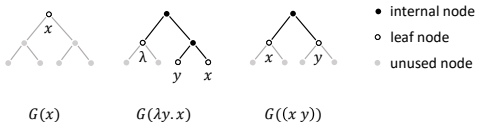


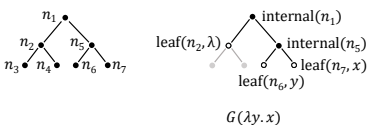
Figure 2 The classical symbolic representation of a set of trees grows very quickly even when small trees are merged, even if their subtrees are shared.



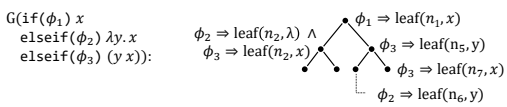
(a) Bonsai trees for x, $\lambda y.x$, and $x(y)$.



(b) Embedding (a) in perfect binary trees.



(c) Symbolic encoding of the tree for $\lambda y.x$.



(d) Merging three trees under path conditions ϕ_i.

Figure 3 A stepwise overview of the BONSAI tree encoding.

General-purpose symbolic evaluation. To perform symbolic evaluation of a type checker, we need to create a symbolic abstract syntax tree that represents A, the set of concrete abstract syntax trees. The standard approach would produce trees such as those shown in Figure 2, where sets of trees are merged by creating symbolic choices to select among potential children at each node. This merged symbolic tree could then be supplied to an existing type checker and interpreter by symbolically evaluating them on the tree.

Though this classical approach would work in principle, it fails to scale to trees that are deep enough to explore large programs. Furthermore, each operation on such a symbolic tree causes its representation to grow even more complex, and the large data structures prevent scalable symbolic execution. Thus, the challenge here is to optimize the speed of symbolic compilation, rather than the speed of solving.

Domain-specific symbolic evaluation. Bonsai solves this problem by creating a new encoding for sets of trees that limits growth by efficiently merging trees within the set. This "Bonsai tree" is compatible with a standard symbolic evaluator, and language engineers can use this symbolic tree nearly as if it were a concrete tree. Figure 3 gives a stepwise explanation of the Bonsai symbolic tree, starting from concrete Bonsai trees for three program terms (a). These concrete trees are embedded in a perfect binary tree (b). The embedding is represented with two predicates for each node: the first determines whether the node is internal or a leaf; the second determines the terminal for leaves (c). By allowing the predicates to be symbolic expressions, a single tree can represent multiple Bonsai trees. In (d), we show how symbolic Bonsai trees arise; here we merge three trees at an if-statement.

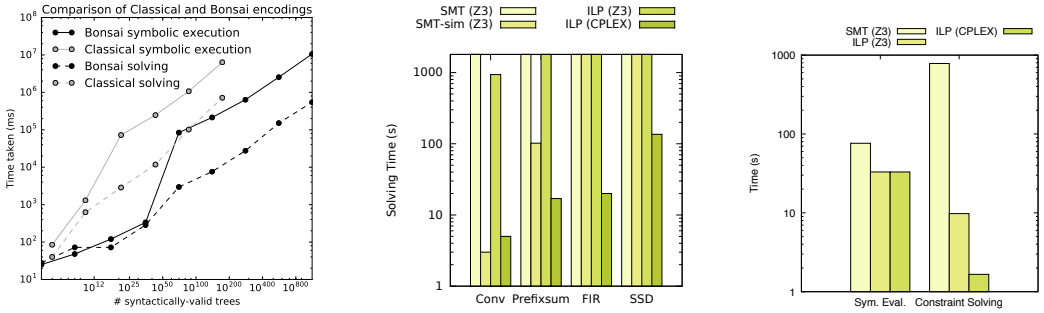
Despite having a different underlying representation, Bonsai trees can be easily manipulated by programmers, just as if they were concrete trees. Bonsai provides utilities for creating, modifying, and pattern-matching with symbolic trees, allowing programmers to implement typecheckers and interpreters without having to focus on the details of symbolic execution.

Evaluation. Figure 4a shows an empirical comparison between the classical and Bonsai encodings. Symbolic terms of various sizes were executed with identical typecheckers and interpreters, varying only the underlying encoding. Bonsai's encoding was consistently several orders of magnitude faster. In under an hour, Bonsai explores programs much larger than counterexamples created by human experts who report soundness bugs, thus providing users with a margin of assurance.

Bonsai has reproduced many soundness bugs in a variety of languages, notably including (1) unsound argument covariance in a model of Java, and (2) a subtle issue with Scala's existential types, discovered in 2016 by Nada Amin and Ross Tate [1]. Slight modifications to the algorithm also allow users to ask intriguing new questions that fuzzers cannot easily answer, such as "On what programs do typecheckers t_1 and t_2 disagree?" or "Does my typechecker reject programs that don't fail?" Finally, by making the *typechecker* symbolic, Bonsai can synthesize suggestions for how to fix an unsound type system.

4 Program Partitioning

The Code Partitioning Problem. Compilers for fine-grain many-core architectures must partition the program into tiny code fragments mapped onto physical cores. Chlorophyll [15, 16] is a language for GA144, an ultra-low-power processor with 144 tiny cores [5]. The Chlorophyll type system ensures that no fragment overflows the 64-word capacity of its core.



(a) Bonsai tree **(b)** Program partitioning **(c)** Tree scheduling

Figure 4 Experimental evaluations.

Listing 1 Original type checker, ensuring that code fragments fit into cores.

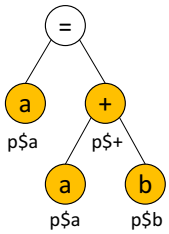
```
1  (define cores-space (make-vector n-cores 0)) ; space used up on each core
2  (define (inc-space p size)
3            (vector-set! cores-space p (+ (vector-ref cores-space p) size)))
4
5  ; Increase code size whenever core p sends a value to core r.
6  (define (comm p r) (when (not (= p r)) (begin (inc-space p 2) (inc-space r 2))))
7
8  ; Increase code size for broadcast communication from p to ps. ps may contain duplicates.
9  (define (broadcast p ps)
10   (define remote-ps (length (remove p (unique ps))));# of unique cores in ps excluding p
11   (inc-space p (* 2 remote-ps))   ; space used in the sender core
12   (for ([r ps]) (inc-space r 2))) ; space used in the receiver cores
13
14 (define (count-space node) ; Count space needed by an AST node.
15   (cond
16    [(var? node) (inc-space (place-type node) 1)]
17    [(binexpr? node) ; The inputs to this operation come from binexpr-e1 and binexpr-e2.
18     (define p (place-type node))
19     (inc-space p (size node)) ; space taken by the operation
20     (comm (place-type (binexpr-e1 node)) p)  ; Add space for communication code when
21     (comm (place-type (binexpr-e2 node)) p)] ; operands come from other cores.
22    [(if? node)
23     ; If is replicated on all cores that run any part of if's body.
24     ; We omit inc-space here.
25     ; The condition result is broadcast to all cores used in if's body.
26     (broadcast (place-type (if-test node))
27                (append (all-cores (if-then node)) (all-cores (if-else node))))]
28   ...))
29
30 (tree-map count-space ast)
31 (for ([space cores-space]) (assert (< space core-capacity)))
32 (minimize (apply + cores-space)) ; used during inference only
```

Each variable and operation have a *place type* whose value is a core ID. The type checker in Listing 1 computes the code size of each fragment. The `tree-map` function traverses a program AST in the post-order fashion and applies the function `count-space` on each node in the AST (line 28). The checker accumulates the space taken by each node (e.g. a `binexpr` node on line 18), and space occupied by communication code, for both one-to-one communication (e.g. sending operand values to an operator on lines 19–20) and broadcast communication (e.g. sending a condition result to all nodes in the body of `if` on lines 24–25).

Automatic Program Partitioning as Type Inference. When a program omits some place types, the compiler infers them, effectively partitioning the program. Chlorophyll implements the type inference using Rosette [19, 20], which symbolically evaluates the type checker in Listing 1 with respect to the program. The type checker needs no changes; we only need to

Listing 2 Type checker in resource language, producing ILP constraints.

```
1  (define n (make-parameter #f)) ; a parameter procedure for dynamic binding
2  (define (comm p r) (inc-space (n) (* 2 (+ (different? p r (n)) (different? r p (n))))))
3  (define (broadcast p ps)
4    (inc-space (n) (* 2 (+ (count-different p ps (n)) ;; space used in the sender core
5                           (different? ps p (n)) )))) ;; space used in the receiver cores
6
7  ;; the function count-space is changed in one place (see text); inc-space is unchanged
8
9  (for ([i n-cores]) (parameterize ([n i]) (tree-map count-space ast)))
10 (for ([space cores-space]) (assert (< space core-capacity)))
11 (minimize (apply + cores-space))
```



(a) Example program AST. Each node is annotated with its place type below. The yellow nodes are the ones that have been interpreted.

```
(inc-space p$a 1) ;; Line 15, node = a [def]
(inc-space p$a 1) ;; Line 15, node = a [use]
(inc-space p$b 1) ;; Line 15, node = b
(inc-space p$+ 1)  ;; Line 18, node = +
(comm p$a p$+)     ;; Line 19, comm a -> +
(comm p$b p$+)     ;; Line 20, comm b -> +
```

(b) Residual type checking program after traversing the yellow nodes in the AST on the left (post-order). The line numbers in the comments indicate where the expressions come from from Listing 1.

Figure 5 Running example of program partitioning

initialize the (unknown) place types in the program to symbolic values (i.e., `p$0, p$1, ...`).

Figure 5a shows an example of a program AST with unknown places. Each node in the AST is annotated with its symbolic place type. Figure 5b shows the conceptual partially-evaluated type checker after checking the yellow nodes in the example AST; concrete expressions are fully evaluated, and the expressions with symbolic variables remain. After we symbolically evaluate the residual type checker in Figure 5b, we obtain `cores-space` shown in Figure 6a. Rosette then uses Z3 to solve the generated SMT constraints on `cores-space` (line 29 of Listing 1) and minimize the total code size (line 30 of Listing 1).

Hence, we obtain our type inference just by implementing a type checker. The development process requires little effort, but the type inference is slow at inferring place types.

Symbolic Evaluation to ILP Constraints. It is known that partitioning and scheduling problems can be solved efficiently using ILP [21, 14, 13, 8]. However, if we follow the standard way of generating ILP constraints, we will not be able to simply turn type checking into type inference. Here, we turn to our key idea and introduce a symbolic language that will generate ILP constraints. The programmer implements the type checker as before but in our *resource language*. The programmer is prohibited from writing programs with symbolic path conditions because these path conditions create non-linear constraints. If a program contains a symbolic path condition, the compiler will raise an exception. Resource language is embedded in Rosette. It provides additional operations: `mapped-to?`, `different?`, and `count-different`, as described in Table 1.

We make four minimal changes to our original type checker, shown in Listing 2. First, we traverse the AST once for every core (line 9). Each iteration i is responsible for accumulating space used in core i. Second, in the function `count-space`, we change the expression to increase the size of core p by the size of the operation of *node* from `(inc-space p (size node))` to `(inc-space (n) (* (size node) (mapped-to? p (n))))`. The previous call produces a non-

```
> cores-space
#(;; core 0
  (+ (ite (= p$a 0) 1 0) ;; a [def]
     (ite (= p$a 0) 1 0) ;; a [use]
     (ite (= p$b 0) 1 0) ;; b
     (ite (= p$+ 0) 1 0) ;; +
     (ite
       (and (or (= p$+ 0) (= p$a 0))
            (! (= p$+ p$a)))
       2 0)              ;; a -> +
     (ite
       (and (or (= p$+ 0) (= p$b 0))
            (! (= p$+ p$b)))
       2 0))             ;; b -> +

  ;; core 1
  (+ ...)
)
```

(a) Original symbolic expression generated from the original type checker (Listing 1)

```
> cores-space
#(;; core 0
  (+ (* 1 Mpn(p$a,0)) ;; a [def]
     (* 1 Mpn(p$a,0)) ;; a [use]
     (* 1 Mpn(p$b,0)) ;; b
     (* 1 Mpn(p$+,0)) ;; +
     (* 2 Remote_prn(p$+,p$a,0)) ;; a -> +
     (* 2 Remote_prn(p$a,p$+,0))
     (* 2 Remote_prn(p$+,p$b,0)) ;; b -> +
     (* 2 Remote_prn(p$b,p$+,0)))
  ;; core 1
  (+ ...))

> (asserts) ;; global assertions
((and (<= 0 Mpn(p$a,0)) (>= 1 Mpn(p$a,0)))
 (and (<= 0 Mpn(p$a,1)) (>= 1 Mpn(p$a,1)))
 (= 1 (+ Mpn(p$a,0) Mpn(p$a,1))) ;; a
 ...
 (<= 0 Remote_prn(p$+,p$a,0))    ;; a -> +
 (>= 1 Remote_prn(p$+,p$a,0))
 (>= Remote_prn(p$+,p$a,0)
     (- Mpn(p$+,0) Mpn(p$a,0)))
 ...
)
```

(b) ILP symbolic expression generated from the modified type checker (Listing 2)

Figure 6 Symbolic expression of space occupied in each core after running a type checker on the yellow nodes in the example AST (Figure 5a).

Table 1 Description of resource language operations. Sym/conc stands for symbolic or concrete.

Function	Type	Description

`(mapped-to? p n)`	`p: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if place p is core n (i.e. $p = n$), otherwise returns 0

`(different? p r n)`	`p: sym/conc integer`
	`r: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if places p and r are different, and place p is core n (i.e. $(p \neq r) \wedge (p = n)$), otherwise returns 0

`(different? ps r n)`	`ps: list of sym/conc integers`
	`r: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if there is at least one place p in ps such that $(p \neq r) \wedge (p = n)$, otherwise returns 0

`(count-different p rs n)`	`p: sym/conc integer`
	`rs: list of sym/conc integers`
	`n: concrete integer`
	`return: sym/conc integer`
		returns a number of unique places in rs that differ from p if place p is core n, otherwise returns 0

linear equation because the first argument p, which is symbolic, to `inc-space` is used as a path condition. Third, we avoid symbolic path conditions inside the function `comm` by using `(different? p r (n))` to compute the size of code for sending data at core (n), and similarly for receiving data. Last, in the function `broadcast`, we utilize `count-different` to compute space taken by code for broadcasting a value to a set of cores.

Implementation. Table 2 details the implementation of the additional operations provided by our symbolic language. Under the abstraction, `(mapped-to? p n)` creates symbolic variables $M_{pn}(p, n')$ for all $n' \in N$ – where N is a set of values that p can take – and returns $M_{pn}(p, n)$;

Table 2 Implementation of resource language operations. $sum^{\dagger}$ and $offset^{\dagger}$ are temporary variables.

Function → return	Created Variables	Additional Assertions
`(mapped-to? p n)` → $M_{pn}(p,n)$	$\forall n' \in N, M_{pn}(p,n')$	$\forall n' \in N, 0 \leq M_{pn}(p,n') \leq 1$ $\sum_{n' \in N} M_{pn}(p,n') = 1$
`(different? p r n)` → $Remote_{prn}(p,r,n)$	$Remote_{prn}(p,r,n)$	$0 \leq Remote_{prn(p,r,n)} \leq 1$ $Remote_{prn(p,r,n)} \geq M_{pn}(p,n) - M_{pn}(r,n)$
`(different? p rs n)` → $Remote_{prsn}(p,rs,n)$	$Remote_{prsn}(p,rs,n)$	$0 \leq Remote_{prsn(p,rs,n)} \leq 1$ $\forall r \in rs, Remote_{prsn}(p,rs,n) \geq Remote_{prn}(p,r,n)$
`(count-different p rs n)` → $Count_{prsn}(p,rs,n)$	$Count_{prsn}(p,rs,n)$ $\forall n' \in N, M^{*}_{rsn}(n')$	$\forall n \in N, 0 \leq M^{*}_{rsn}(n) \leq 1$ $\forall n \in N, r \in rs, M^{*}_{rsn}(n) \geq M_{pn}(r,n)$ $sum^{\dagger} = \sum_{n' \in \{N - \{n\}\}} M^{*}_{rsn}(n')$ $offset^{\dagger} = (M_{pn}(p,n) - 1) \times MAX_{INT}$ $Count_{prsn}(p,rs,n) \geq 0$ $Count_{prsn}(p,rs,n) \geq sum^{\dagger} + offset^{\dagger}$

$M_{pn}(p,n) = 1$ if $p = n$, and $M_{pn}(p,n) = 0$ otherwise. Since p can be mapped to only one value, the function adds the constraint $\sum_{n' \in N} M_{pn}(p,n') = 1$ into the global list of assertions. `(different? p r n)` creates and returns a variable $Remote_{prn}(p,r,n)$, as well as adds $Remote_{prn(p,r,n)} \geq M_{pn}(p,n) - M_{pn}(r,n)$ and $0 \leq Remote_{prn(p,r,n)} \leq 1$ to the global list of assertions. Note that $Remote_{prn(p,r,n)}$ can be either 0 or 1 when $p = r$, which is not what we want. However, this equation is valid if $Remote_{prn(p,r,n)}$ is (indirectly) minimized, so it is 0 when $p = r$ as we expect. The validity check happens when **`minimize`** is called.

Our approach requires no change in Rosette's internals. The additional operations simply generate Rosette assertions. We implement our custom **`minimize`** function, which performs the validity check before calling Rosette's **`minimize`**.

Figure 6b show the symbolic expression of `cores-space` along with additional assertions after symbolically executing the modified type checker on the yellow nodes of the AST in Figure 5a. Notice that the new expression is linear, while the original one is not.

Evaluation. The ILP encoding produced by our abstraction solves problems inaccessible to the SMT-based partitioner, and it is faster than the SMT encoding optimized for the domain of partitioning problems (namely, flattening deeply nested `ite` expressions). Figure 4b shows the median time to partition four benchmarks across three runs. We set the timeout to 30 minutes. In summary, SMT always timed out; domain-optimized SMT constraints solved half of the benchmarks; the ILP encoding solved all benchmarks.

5 Synthesis of Parallel Tree Programs

Attribute Grammars and Static Scheduling. Tree computations such as document layout for data visualization or CSS are naturally specified as attribute grammars [12]. For the sake of efficiency, high-performance layout engines in web browsers schedule these tree computations statically, by assigning the statement that computes an attribute to a predetermined position in a sequence of tree traversals. Static scheduling avoids the overhead of determining dynamically when an attribute is ready to be computed.

We express a static schedule for an attribute grammar as a program in a domain-specific language of *tree traversal schedules*, L_S. A schedule consists of tree traversal passes, each

Listing 3 The original interpreter of tree traversal schedules, $\texttt{int}_S$.

```
1  (define (int_S G t s)
2    (match s
3      [(seq s_1 s_2)
4       (int_S G t s_1)
5       (int_S G t s_2)]
6      [(par s_1 s_2)
7       ; check data independence of forward
8       ; and backward orders
9       (int_S G (copy t) (seq s_2 s_1))
10      (int_S G t        (seq s_1 s_2))]
11     [(pre visits)
12      (preorder (visitor G visits) t)]
13     [(post visits)
14      (postorder (visitor G visits) t)]))
15
16 (define ((visitor G visits) node)
17   (let ([class (get-class G node)])
18     (for ([slot (get-slots visits class)])
19       (eval class node slot))))
20
21 (define (eval class node slot)
22   (let* ([rule (get-rule class slot)]
23          [attr (target node rule)])
24     (for ([dep (get-deps node rule)])
25       (assert (ready? dep)))
26     (assert (not (ready? attr)))
27     (set-ready! attr)))
```

Listing 4 The interpreter $\texttt{int}_{ST}$, written with the symbolic trace language.

```
1  (define (int_ST G t s)
2    (match s
3      [(seq s_1 s_2)
4       (int_ST G t s_1)
5       (int_ST G t s_2)]
6      [(par s_1 s_2)
7       (parallel
8         (int_ST G t s_1)
9         (int_ST G t s_2))]
10     [(pre visits)
11      (preorder (visitor G visits) t)]
12     [(post visits)
13      (postorder (visitor G visits) t)]))
14
15 (define ((visitor G visits) node)
16   (let ([class (get-class G node)])
17    (for ([slot* (get-slots visits class)])
18       (fork ([slot slot*])
19         (eval class node slot)))))
20
21 (define (eval class node slot)
22   (let* ([rule (get-rule class slot)]
23          [attr (target node rule)])
24     (for ([dep (get-deps node rule)])
25       (read dep))
26     (write attr)
27     (step)))
```

of which executes statements from the attribute grammar. For instance, the schedule `post{ Inner{w, h}, Leaf{w, h} } ; pre{ Inner{x, y}, Leaf{x, y} }` performs a post-order traversal computing `w` then `h` at both `Inner` and `Leaf` nodes and then performs a pre-order traversal computing `x` then `y` at both `Inner` and `Leaf` nodes. A statement to execute is indicated by the name of the target attribute, since attributes and statements to compute them correspond one-to-one.

Listing 3 presents a definitional interpreter for the scheduling language L_S. The interpreter checks the correctness of a schedule on a given input tree. Among the checks are the absence of reads from uninitialized attributes (line 25) and single assignment (line 26), which together ensure that data dependencies are satisfied.

Schedule Synthesis. To synthesize a legal schedule, we first define the space of candidate schedules by creating a partially symbolic schedule, such as `post{ Inner{`$??_1$`, `$??_2$`}, Leaf{`$??_3$`, `$??_4$`} } ; pre{ Inner{`$??_5$`, `$??_6$`}, Leaf{`$??_7$`, `$??_8$`} }`, where $??_i$ indicates a symbolic choice ranging over the statements from the relevant node class (*e.g.*, $??_1$ ranges over the statements for `Inner` nodes). This schedule is desugared to a schedule-generating function $\texttt{sch} : D^8 \to L_{Sch}$ whose parameters control the symbolic choices $??_i$.

Note that the schedule is partly concrete; it specifies two concrete traversals, leaving symbolic only the *slots* in the node visitors. This limited symbolic nature simplifies symbolic compilation. To consider other traversal patterns, we can apply a standard technique for prioritized enumeration of sketches [2].

General-Purpose Symbolic Evaluation. With the schedule-generating function `sch` in hand, the call $\texttt{sol}(\texttt{sym}(\texttt{int}_S(\texttt{G}, \texttt{t}) \circ \texttt{sch}, \mathit{success}))$ synthesizes the slots for the partially concrete schedule correct on a tree `t`. When evaluating a symbolic choice $??_i$, symbolic evaluation considers each alternative concrete statement (line 18 in Listing 3), generates the constraints stating that the dependencies are ready and the target has not been computed (lines 25 and

Listing 5 Example L_T program.

```
(define ??1 (choose "x := y + z" "y := 2 * x" "x := 3"))
(define ??2 (choose "x := y + z" "y := 2 * x" "x := 3"))
(define ??3 (choose "x := y + z" "y := 2 * x" "x := 3"))
(define x (alloc))
(define y (alloc))
(define z (alloc))

(for ([??i (list ??1 ??2 ??3)])  ; execute a program with three slots
  (fork ([stmt ??i])  ; for each possible statement in this slot
    (let ([var (lookup (lhs stmt))]
          [expr (rhs stmt)])
      (for ([ref (refs expr)])  ; for all locations in the right-hand side
        (read ref))
      (write var)
      (step))))
```

Table 3 Operations of the symbolic trace language L_T (each returning `(void)` unless noted otherwise), where `host` refers to the pure subset of the host language.

Operation	Type	Description
(choose$v_1 \ldots v_n$)	v_i: concrete value return: symbolic choice	returns a symbolic choice from the given values, to construct a program hole ??
(alloc)	return: concrete location	returns a fresh concrete location
(read l)	l: concrete location	logs a read from the given location
(write l)	l: concrete location	logs a write to the given location
(step)		advances the program to the next statement
(fork ([x c]) e)	x: variable in host c: symbolic choice e: expression in host	evaluates e for each concrete alternative of c, bound to x, under an appropriate guard
(parallel $e_1 e_2$)	e_1: expression in host e_2: expression in host	evaluates e_1 then e_2 while checking for conflicting usage of locations

26), sets the target attribute as ready, updates the program state (line 27), and then merges alternative states.

The constraints resulting from this evaluation present a challenge for the SMT solver. The symbolic state encodes whether a concrete attribute is ready at a given execution step as a function of symbolic choices (*i.e.*, which statement goes into which slot). We hypothesize that this state formulation prevents the solver to learn from failed guesses: if placing statement s_1 before s_2 leads to a dependence violation, the solver will happily try to place s_1 before s_2 into some other pair of slots.

Domain-Specific Symbolic Evaluation. To make constraint solving more efficient, we express the interpreter of schedules in the *symbolic trace language* L_T. The syntax of this language is summarized in Table 3, and a small example program in L_T is shown in Listing 5. The new version of the interpreter, $\texttt{int}_{ST}$, is in Listing 4.

The symbolic trace language understands only dependency relationships carried through *locations*, write-once memory objects with fully abstract contents. A location `l` is generated with `(alloc)` and used with `(read l)` and `(write l)`. In the context of attribute grammars, a location corresponds to a particular attribute somewhere in the tree.

As a trace program executes, we generate efficient ILP constraints for these correctness conditions:

Listing 6 Excerpt of the residual program in L_T from partial evaluation of $\texttt{int}_{ST}$ with respect to the symbolic schedule `sch`, showing the call to `(eval Inner root` $??_1$`)`.

```
; construct the symbolic schedule
(define ??_1
  (choose "self.x := 0" ; guard = b_{1,x}
          "self.y := 0" ; guard = b_{1,y}
          "self.w := left.w + right.w" ; guard = b_{1,w}
          "self.h := left.h + right.h")) ; guard = b_{1,h}
...
; assign each attribute a freshly allocated location
(set! root.x (alloc))
(set! root.y (alloc))
...
; expansion of (fork ([slot slot*]) (eval ??_1))
; case for ??_1 = "self.x := 0"
(write root.x #:guard {b_{1,x}})
(step #:guard {b_{1,x}})
; case for ??_1 = "self.y := 0"
(write root.y #:guard {b_{1,y}})
(step #:guard {b_{1,y}})
; case for ??_1 = "self.w := left.w + right.w"
(read root.left.w #:guard {b_{1,w}})
(read root.right.w #:guard {b_{1,w}})
(write root.w #:guard {b_{1,w}})
(step #:guard {b_{1,w}})
; case for ??_1 = "self.h := left.h + right.h"
(read root.left.h #:guard {b_{1,h}})
(read root.right.h #:guard {b_{1,h}})
(write root.h #:guard {b_{1,h}})
(step #:guard {b_{1,h}})
...
```

1. Every location is written at most once.
2. Every read to a location is preceded by the write to that location.
3. Concurrent threads are data-independent (*i.e.*, locations read by a thread are disjoint from locations written by any concurrent thread).

Note that the symbolic trace language requires annotating with `fork` those code fragments that must be explored under alternative symbolic choices. Each such choice evaluates under a *guard* (analogous to a path condition in traditional symbolic evaluation) that records the current set of assumptions about symbolic choices. For instance, when `(fork ([x (choose` x_1 x_2`)]) ...)` explores the path for x_1, the current guard will be extended (since uses of `fork` may be nested) with the assumption that `x =` x_1, and a similar process then happens for x_2. The `fork` is analogous to Rosette's `for/all` and serves the same role of controlling where alternative paths are merged.

Adopting the symbolic trace language requires only a handful of straightforward changes to the interpreter in Listing 3, to turn its checks into trace events. The modified interpreter is shown in Listing 4. Listing 6 shows an excerpt of the residual program generated by the call $\texttt{s}(\texttt{int}_{ST}(\texttt{G},\texttt{t}) \circ \texttt{sch})$, where `s` is the program specializer.

Implementation. The symbolic trace language is implemented as an embedded domain-specific language in Rosette. We leverage the restricted nature of L_T to generate efficient ILP constraints. We mention in this paper only the encoding of dependences, which was responsible for the bulk of the improvement over the previous SMT encoding.

Collectively, the generated constraints must ensure that all *dependences* are satisfied, which means that all reads from a location follow the write into that location. A straightforward encoding is to require that the step counter of the read is higher than the step counter of the write.

However, a less obvious encoding improves the solver's performance by several orders of magnitude. Rather than ensuring that all dependences are met, we pose the equivalent constraints ensuring that no *antidependences* exist, which means that the write must not happen after any of the reads from the location. The advantage of ruling out antidependences over requiring dependences is that in ILP, it seems easier to solve the constraint "if a write happens here, then *none* of the reads must have happened before," than it is to solve "if a read happens here, then a write *must* have happened before." We hypothesize that this encoding wins over alternative ILP encodings because ruling out antidependences provides the solver the analogue of conflict clauses that it would need to learn itself.

We want to point that the concept of antidependences does not exist in the original schedule interpreter. A general symbolic compiler thus cannot switch from dependences to ruling out antidependences. Doing so would require relative deep and global reasoning.

Evaluation. Figure 4c compares our domain-specific symbolic evaluator against the general-purpose evaluator. We evaluate the performance of symbolic evaluation and constraint solving. The benchmarks synthesize tree traversal schedules for an attribute grammar that encodes the treemap data visualization [11]. For each attribute grammar, symbolic evaluation is done with a set of example trees chosen by an automated tool to sufficiently cover the grammar. An enumeration of candidate partially symbolic schedules is used. Each measurement is the median value from three runs. The domain-specific encoding on the CPLEX solver improves the solving time by three orders of magnitude.

6 Summary and Future Directions

Contribution to Solver-Aided DSLs. Our domain-specific symbolic compilation idea originated from solver-aided DSLs (SDSLs). SDSL is attractive because it is the most promising technique we have today for automatically constructing program synthesizers: simply write an interpreter, and obtain a symbolic compiler for free. SDSLs take advantage of the domain in two ways. First, DSL programs are compact, reducing the search space explored by synthesizers. Second, an interpreter for a DSL can be written to increase the opportunities for specialization.

Rosette, its meta-language, and SMT solvers together served as an excellent tool to build SDSLs. However, as we moved to larger and more complex problems, both the symbolic compilation and solving emerged as a bottleneck. SMT solvers also ran out of steam. Our proposed solution delivers the promise of building domain-specific synthesizers that can solve larger, more complex problems. We show that a domain-specific symbolic compiler, built on top of Rosette, can produce a custom encoding of constraints and utilize a more efficient solver for that particular domain, such as an ILP solver.

Evaluation. Our ultimate goal is to obtain efficient constraints generated by an intuitive, expressive abstraction. In this paper, we showed that our domain-specific symbolic compiler can generate efficient constraints, reducing the time of solving and symbolic evaluation, sometimes by orders of magnitude. We are encouraged that good constraints can be generated by symbolically evaluating a program, as that opens new ways to write specifications.

Our findings are preliminary when it comes to expressiveness. While we are happy with the experience of expressing our interpreters and checkers, we have not stressed the abstractions enough to identify their limits. For example, we did not try to use the Bonsai tree (Section 3) on type checkers that also perform inference; the resource language (Section 4)

may need extensions to support runtime data migration; and the trace language (Section 5) may fail on incremental tree evaluation because it needs data-dependent traversals that visit only a subtree. These new applications may require new abstractions. However, we hope that the three designs will serve at as an inspiration.

Checking the usage. Symbolic languages are not intended to be used without care, and a type system should thus ensure proper usage of symbolic language operations. However, the code using the abstraction can sometimes break the abstraction even through seemingly unrelated code (because the client and the abstraction are symbolically evaluated together). For example, using a conditional expression in the partitioning type checker (Listing 2) may cause the symbolic evaluation to produce a symbolic path condition, which makes the program inexpressible in ILP constraints. It is an open question what restrictions we should impose on the client to avoid such surprises.

Combining symbolic languages. New challenges arise when we compose two symbolic languages. For example, to distribute nodes of an unbounded tree onto CPU cores, we may want to combine some variants of the trace language from Section 5 and the resource language from Section 4. The former would schedule traversals while the latter would map data onto cores. What do we expect when both languages are used in the same interpreter? If scheduling and data placement happen to be separate problems, it would be desirable if the symbolic compilation of the interpreter produce two independent sets of constraints. If the problems are intertwined, the interpreter writer should be able to control the approximation: for example, fix the tree distribution first, then find the best traversal strategy.

Converting solutions to programs. We have blissfully assumed that the solution returned by the solver *is* the desired program. Naturally, the solution must be converted to program syntax, and the conversion becomes trickier as we add more levels of abstraction. Specializing symbolic compilers (Rosette [20] and Sketch [17]) automate the conversion, but that functionality is broken by our insertion of the symbolic language layer. It seems possible to define the conversion as an inversion of symbolic compilation, and perhaps this view could lead us towards automatic construction of solution-to-program converters.

References

1. Nada Amin and Ross Tate. Java and scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA*, 2016.
2. James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *POPL*, 2016.
3. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 – April 2, 2004. Proceedings*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
4. Robert Glück. Is there a fourth futamura projection? In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 51–60, 2009.
5. GreenArrays. *Product Brief: GreenArrays Architecture*, 2010. URL: `http://www.greenarraychips.com/home/documents/greg/PB002-100822-GA-Arch.pdf`.

6 Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

7 Jad Hamza, Barbara Jobstmann, and Viktor Kuncak. Synthesis for regular specifications over unbounded domains. In *FMCAD*, 2010.

8 Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, 2015.

9 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

10 Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):355–383, July 2003.

11 Leo Meyerovich, Matthew Torok, Eric Atkinson, and Rastislav Bodik. Parallel schedule synthesis for attribute grammars. In *PPoPP*, 2013.

12 Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *WWW*, 2010.

13 Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *PLDI*, 2013.

14 Jens Palsberg and Mayur Naik. ILP-based Resource-aware Compilation, 2004.

15 Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

16 Phitchaya Mangpo Phothilimthana, Michael Schuldt, and Rastislav Bodik. Compiling a gesture recognition application for a low-power spatial architecture. In *Proceedings of Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, 2016.

17 Sketch: a synthesizer language and compiler. URL: `http://bitbucket.org/gatoatigrado/sketch-frontend`.

18 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, New York, NY, USA, 2006.

19 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

20 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, 2014.

21 Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *PLDI*, 2000.