Deriving Divide-and-Conquer Dynamic Programming Algorithms Using Solver-Aided Transformations

Double-blind submission

Abstract

We introduce a framework allowing domain experts to manipulate computational terms in the interest of deriving better, more efficient implementations. It employs deductive reasoning to generate provably correct efficient implementations from a very high-level specification of an algorithm, and inductive constraint-based synthesis to improve automation. Semantic information is encoded into program terms through the use of refinement types.

In this paper, we develop the technique in the context of a system called Bellmania that uses solver-aided tactics to derive parallel divide-and-conquer implementations of dynamic programming algorithms that have better locality and are significantly more efficient than traditional loop-based implementations. Bellmania includes a high-level language for specifying dynamic programming algorithms and a calculus that facilitates gradual transformation of these specifications into efficient implementations. These transformations formalize the divide-and-conquer technique; a visualization interface helps users to interactively guide the process, while an SMT-based back-end certifies each step and takes care of low-level reasoning required for parallelism.

We have used the system to generate provably correct implementations of several algorithms, including some important algorithms from computational biology, and show that the performance is comparable to that of the best manually optimized code.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming

1. Introduction

Software synthesis aims to close the gap between descriptions of software components, such as algorithms and systems, and their implementations as computer programs. Dynamic Programming (DP) algorithms offer a prominent example of how large this gap can be. For instance, consider Algorithm 1, which correspond to the well known DP algorithm to compute the optimal parenthesization for a chain of matrix multiplications.

Algorithm 1 A naïve loop implementation

```
for i = 1..n do  G_{i(i+1)} := x_i                    ▷ Initialize
for i = (n − 2)..0 do                                ▷ Compute
    for j = (i + 2)..n do
        G_ij := min_{i<k<j} G_ik + G_kj + w_ikj
```

The algorithm computes an $n \times n$ region of a DP table G via a simple row-major order. This algorithm is simple, but a direct C implementation of it will be $50\times$ slower than the best manually optimized implementation. One reason for this poor performance is the high rate of cache misses incurred by reading the ranges G_{ik} and G_{kj}, $i < k < j$, repeatedly for every iteration of the loops on i and j. Memory reads dominate the running time of this algorithm, so high speedups can be gained by localizing memory access.

The state-of-the-art implementation uses a *divide and conquer* approach both to improve memory performance and to increase the asymptotic degree of parallelism [30]. An excerpt of the pseudo-code for such an implementation is shown in Algorithm 2. In the optimized version, the programmer has to determine a number of low-level details, including the correct order of calls—some of which can be run in parallel—as well as some rather involved index arithmetic. When implemented in C++, the full version is, in fact, more than ten times longer than the naïve one and considerably more complicated. It is also much more difficult to verify, since the programmer would have to provide invariants and contracts for a much larger set of loops and possibly recursive functions. Parallelism only adds to the complexity of this task.

In this paper, we present a new system called Bellmania[1], which allows an expert to interactively generate parallel divide-and-conquer implementations of dynamic programming algorithms, which are provably correct relative

[1] Named so as a tribute to Richard Bellman.

to a high-level specification of the code in Algorithm 1. We show that the resulting implementations are, on average, $16\times$ faster than the original versions and within 22% from a high-performance hand crafted implementation, with speedups as high as $46\times$. Their structure will also make them *cache oblivious* [16] and *cache adaptive* [3], just like the hand crafted implementations are.

Bellmania is a deductive synthesis system in the tradition of systems like Kids [26], and more recently Fiat [13]. These systems derive an implementation from a specification in a correct-by-construction manner by applying a sequence of deductive reasoning steps. Thanks to the correct-by-construction approach, the potentially complex final artifact does not have to be independently verified, making the approach ideal for potentially complex implementations like the ones we target.

Traditionally, the major shortcoming of deductive synthesis has been the effort required of the user in order to guide the derivation process towards a correct solution. In this work, we reduce this effort using three core techniques. First, we show that a small number of domain specific tactics, combined with a new notation to jointly describe the specification and the implementation, can enable the derivation to be described succinctly at a high-level of abstraction. Secondly, we show the value of incorporating an SMT solver into the derivation process; in particular, we can afford to use tactics that are only correct when some side conditions hold, where these conditions are potentially complex logical assertions, without having to burden the user with proving those premises. Finally, we show that by incorporating *solver-based inductive synthesis*, which generalizes from concrete values and execution traces, into the derivation process, we can automate away many low-level decisions, allowing for shorter, simpler derivations. We use the term *solver-aided tactics* to refer to this combination of solver-based inductive synthesis and solver-checked premises within a tactic.

Algorithm 2 An optimized divide-and-conquer version

A$[1..n]$, where: (*snippet*)
procedure A$[s..e]$
  if $e - s < b$ **then**
    for $i = e..s$ **do**
      for $j = \max\{s, i\}..e$ **do**
        $G_{ij} := \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$
  else
    A$\left[s..\lfloor\frac{s+e}{2}\rfloor\right]$
    A$\left[\lfloor\frac{s+e}{2}\rfloor+1..e\right]$
    B$\left[s..\lfloor\frac{s+e}{2}\rfloor\ ,\ \lfloor\frac{s+e}{2}\rfloor+1..e\right]$
procedure B$[s_0..e_0\ ,\ s_1..e_1]$
  if $e - s < b$ **then** ...
  else
    B$\left[\lfloor\frac{s_0+e_0}{2}\rfloor+1..e_0\ ,\ s_1..\lfloor\frac{s_1+e_1}{2}\rfloor\right]$
    C$\left[s_0..\lfloor\frac{s_0+e_0}{2}\rfloor\ ,\ s_1..\lfloor\frac{s_1+e_1}{2}\rfloor\ ,\ \lfloor\frac{s_1+e_1}{2}\rfloor+1..e_1\right]$
    ⋮
procedure C$[s_0..e_0\ ,\ s_1..e_1\ ,\ s_2..e_2]$
  ⋮

Overall, the paper makes the following contributions.

- A new formalism used to describe a wide class of dynamic programming algorithms, capable of bridging the gap between the high-level specification and the divide-and-conquer implementation of them.
- An interesting application of refinement types for tracking dependencies between sub-computations, making it possible to automatically generate parallel implementations from it.
- The idea of using *solver-aided tactics*, demonstrating their applicability and utility in the derivation of divide-and-conquer dynamic programming implementations.
- A suite of solver-aided tactics for dynamic programming and an overview of the proofs of their soundness, assuming only the soundness of the underlying SMT solver.
- A description of Bellmania, the first system capable of generating provably correct implementations of divide-and-conquer dynamic programming. Our evaluation shows that the code it generates is comparable to manually tuned implementations written by experts in terms of performance.

Dynamic Programming is central to many important domains ranging from logistics to computational biology — e.g., a recent textbook [14] lists 11 applications of DP in bioinformatics just in its introductory chapter, with many more in chapters that follow. Increasing performance and reliability of DP implementations can therefore have significant impact. More generally, we believe that this work serves as an important test case for a new approach to combining inductive and deductive synthesis which could have an impact beyond this domain.

2. Overview

Most readers are likely familiar with the Dynamic Programming (DP) technique of Richard Bellman [2] to construct an optimal solution to a problem by combining together optimal solutions to many overlapping sub-problems. The key to DP is to exploit the overlap and reuse computed values to explore exponential-sized solution spaces in polynomial time. Dynamic programs are usually described through recurrence relations that specify how to decompose sub-problems, and is typically implemented using a DP table where each cell holds the computed solution for one of these sub-problems. The table can be filled by visiting each cell once in some predetermined order, but recent research has shown that it is possible to achieve order-of-magnitude performance improvements over this standard implementation approach by developing *divide-and-conquer* implemen-

tation strategies that recursively partition the space of subproblems into smaller subspaces [3, 7–10, 30].

Before delving into how Bellmania supports the process of generating such an implementation, it is useful to understand how a traditional iterative implementation works. For this, we will use the optimal parenthesization algorithm from the introduction (Algorithm 1). The problem description is as follows: given a sequence of factors $a_0 \cdots a_{n-1}$, the goal is to discover a minimal-cost placement of parentheses in the product expression assuming that multiplication is associative but not commutative. The cost of reading a_i is given by x_i, and that the cost for multiplying $\Pi(a_{i..(k-1)})$ by $\Pi(a_{k..(j-1)})$ is given by w_{ikj}. The specification of the algorithm is shown in Figure 1; the values x_i and w_{ikj} are inputs to the algorithm, and the output is a table G, where each element G_{ij} is the lowest cost for parenthesizing $a_{i..(j-1)}$, with G_{0n} being the overall optimum.

Iterative Algorithm. Using the standard dynamic programming method, anyone who has read [12] would compute this recurrence with an iterative program by understanding the dependency pattern: to compute the $\min_{i<k<j}(\cdots)$ expression in Figure 1 the algorithm needs to enumerate k and gather information from all cells below and to the left of G_{ij}. In particular, each value G_{ij} is computed from other values $G_{i'j'}$ with higher row indexes $i' > i$ and lower column indexes $j' < j$. Therefore, considering G as a two-dimensional array, it can be filled in a single sweep from left to right and from bottom to top, as done in Algorithm 1.

Divide-and-Conquer Algorithm. To illustrate the main concepts underlying Bellmania and the key ideas in deriving divide-and-conquer implementations, we will walk through the first few steps that an algorithms expert — whom we will call Richard — would follow using Bellmania to generate a provably correct divide-and-conquer implementation of the optimal parenthesization algorithm.

In the Bellmania development model, Richard will start with the specification from Figure 1, and progressively manipulate it to get the specification in a form that reflects the recursive structure of the divide-and-conquer implementation. At any step in the transformation, Bellmania can generate code from the partially transformed specification. Code generated from the initial specification will yield an implementation like Algorithm 1, whereas generating code from

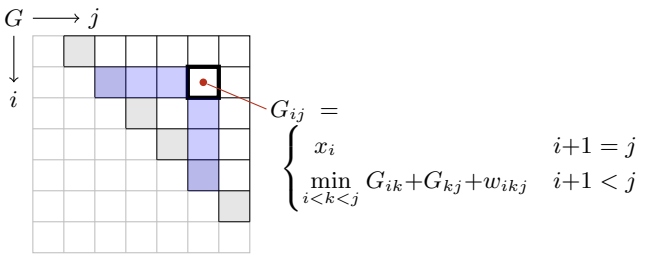

Figure 1. Recurrence equation and cell-level dependencies.

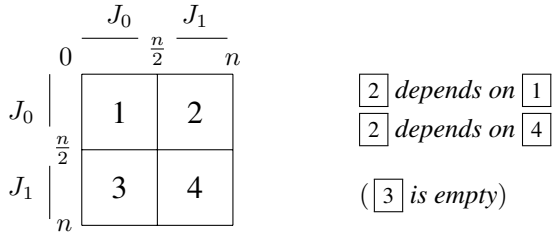

Figure 2. Dividing a two-dimensional array into quadrants; the dependencies for the case of the running example are shown on the right.

the fully transformed specification will yield the divide-and-conquer implementation that we want. In the rest of the text, we will use the term *program* to refer to any of the partially transformed specifications.

Figure 4 provides a visual description of the initial stages of the transformation process. The figure includes block diagrams illustrating how the program at a given stage in the transformation will compute its output table from its input. For example, the first row corresponds to the program before any transformations take place, i.e. the initial specification. At this stage, the program involves a single loop nest that reads from the entire array and writes to the entire array; solid arrows in the diagram denote data dependency. The transformation from one stage to the next, the dashed arrows, is performed by the application of *tactics* that represent a high-level refinement concept.

As a first step in the transformation, Richard would like to partition the two-dimensional array G into quadrants, as illustrated in Figure 2. In Bellmania, the partition is accomplished by applying the Slice tactic, illustrated graphically at the top of Figure 4. In order to escape the need to reason about concrete array indices, Bellmania provides an abstract view where the partitions are labeled J_0, J_1. The effect of Slice is shown in text in Figure 3(a) — the figure trades accuracy for succinctness by omitting the base case of the recurrence for now. Due to the structure of the problem, namely $i < j$, the bottom-left quadrant ($\boxed{3}$ in Figure 2) is empty. Slicing therefore produces only three partitions.

The computation of $\boxed{1}$ (the top-left quadrant) does not depend on any of the other computations, so Richard applies the Stratify tactic, which separates an independent computation step as a separate loop. This is equivalent to rewriting the specification as in Figure 3(b): the first computation is given a special name $G^{\boxed{1}}$, then the following computations read data either from $G^{\boxed{1}}$ (when the indices are in $\boxed{1}$) or from G (otherwise), which is denoted by $G^{\boxed{1}}/G$. The "/" operator is part of the Bellmania language and will be defined formally in Section 3. Bellmania checks the data dependencies and verifies that the transformation is sound.

Repeating Stratify results in a three-step computation, as seen in Figure 4(c), from which Richard can obtain the pro-

Slice $\quad i,j : \langle J_0 \times J_0 \mid J_0 \times J_1 \mid J_1 \times J_1 \rangle \qquad (a)$

$$\forall i,j \in \boxed{1}.\ G_{ij} = \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$$
$$\forall i,j \in \boxed{2}.\ G_{ij} = \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$$
$$\forall i,j \in \boxed{4}.\ G_{ij} = \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$$

Stratify $\boxed{1} \qquad (b)$

$$\forall i,j \in \boxed{1}.\ G^{\boxed{1}}_{ij} = \min_{i<k<j} G^{\boxed{1}}_{ik} + G^{\boxed{1}}_{kj} + w_{ikj}$$
$$\forall i,j \in \boxed{2}.\ G_{ij} = \min_{i<k<j} (G^{\boxed{1}}/G)_{ik} + (G^{\boxed{1}}/G)_{kj} + w_{ikj}$$
$$\forall i,j \in \boxed{4}.\ G_{ij} = \min_{i<k<j} (G^{\boxed{1}}/G)_{ik} + (G^{\boxed{1}}/G)_{kj} + w_{ikj}$$

Figure 3. The first two steps in the development, represented as logical specifications.

gram in Algorithm 3. This already gives some performance gain, since computations of $\boxed{1}$ and $\boxed{4}$ can now run in parallel. Bellmania is capable of sound reasoning about parallelism, using knowledge encoded via types of sub-terms, showing that two threads are race-free when they work on different regions of the table. This is handled automatically by the code generator.

At this point, Richard notices that $G^{\boxed{1}}$ is just a smaller version of the entire array G; he invokes another tactic called Synth, which automatically synthesizes a recursive call $\mathrm{A}\left[G^{\boxed{1}}\right]$ (presented using abstract index ranges as A^{J_0}).

Similarly, $G^{\boxed{4}}$ is also a smaller version of G, this time with indices from J_1. Synth figures it out automatically as well, synthesizing a call $\mathrm{A}\left[G^{\boxed{4}}\right]$. The remaining part, $G^{\boxed{2}}$, is essentially different, so Richard gives it a new name, "B", which becomes a new synthesis task.[2]

Applying the same strategy will eventually lead Richard to further break down and transform the computation of B into multiple recursive sub-computations, further improving the locality of the resulting algorithm until a true divide-and-conquer solution is obtained. Bellmania generates code for all procedures encountered throughout the development. In this case, three recursive procedures are generated. The base case of the recursion is when the region becomes small enough to just run the loop version.[3]

[2] Richard's choice of names is consistent with the literature.

Algorithm 3 Simplified Arbiter — Sliced and Stratified

procedure $\mathrm{A}[G]$
- **for** $i = (n-2)..0 \cap J_0$ **do** ▷ *Compute* $\boxed{1}$
  - **for** $j = (i+2)..n \cap J_0$ **do**
    - $G_{ij} := \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$
- **for** $i = (n-2)..0 \cap J_1$ **do** ▷ *Compute* $\boxed{4}$
  - **for** $j = (i+2)..n \cap J_1$ **do**
    - $G_{ij} := \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$
- **for** $i = (n-2)..0 \cap J_0$ **do** ▷ *Compute* $\boxed{2}$
  - **for** $j = (i+2)..n \cap J_1$ **do**
    - $G_{ij} := \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$

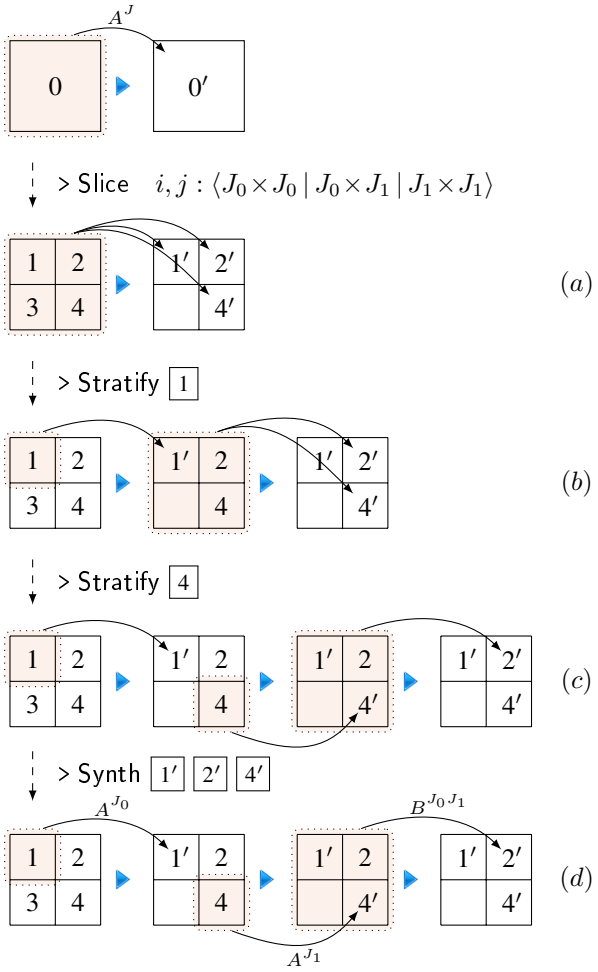

Figure 4. Overview of tactic semantics in Bellmania.

[3] The optimal base case size of the region can be found by auto-tuning (taken as an input in the current version of the compiler)

Algorithm 4 Parenthesis — Recursive Version

procedure $\mathrm{A}[G]$
- **if** G *is very small* **then** *run iterative version*
- **else**
  - $\mathrm{A}\left[G^{\boxed{1}}\right]$ ▷ *Compute* $\boxed{1}$
  - $\mathrm{A}\left[G^{\boxed{4}}\right]$ ▷ *Compute* $\boxed{4}$
  - $\mathrm{B}\left[G^{\boxed{1}}, G^{\boxed{4}}, G^{\boxed{2}}\right]$ ▷ *Compute* $\boxed{2}$

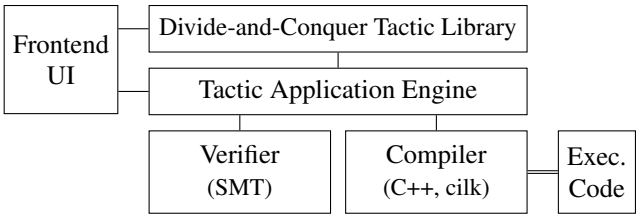

Figure 5. Overall design of Bellmania.

As is well illustrated by the example, this line of reasoning can get quite complicated for most dynamic programming algorithms, and producing a correct divide-and-conquer algorithm for a given dynamic programming problem is considered quite difficult even by the researchers who originally pioneered the technique. Fortunately, Bellmania is able to mechanize most of the technical details, allowing Richard and other algorithm designers to focus on their area of expertise, try different strategies, and eventually produce a *certified* implementation of the algorithm.

Overall, it took Richard only 4 steps to construct Algorithm 4, and a total of 30 steps to construct all three phases of the Parenthesis algorithm, comprising an implementation that is 46× faster than a parallel implementation of Algorithm 1 using a state-of-the-art parallelizing compiler. The user is greatly assisted by tactics like Synth, which carries out the monotonic and error-prone task of choosing the right parameters for each recursive call; also, mistakes are identified early in the development thanks to automatic verification, saving hours of debugging later on. The resulting code is much easier to maintain, because the artifact is not just the optimized C++ code, but also the Bellmania specification and the script that encodes the optimization strategy.

Once a divide-and-conquer algorithm is found, generating an optimal implementation still requires some additional work, such as finding the right point at which to switch to an iterative algorithm to leverage SIMD parallelism as well as low-level tuning and compiler optimization; these steps are performed by more traditional compiler optimization techniques as discussed in Section 6.

System Design

The design of the Bellmania system (Figure 5) contains a generic core — called the *Tactic Application Engine* (TAE) — on top of which a library of *tactics* specific to dynamic programming is built. While it is possible to extend the library, it already contains enough tactics to successfully develop algorithms for a family of problems, so that the user of the system only needs to apply existing tactics by issuing commands through a GUI and watching the program evolve. The TAE has a back-end that verifies conjectures, and is in charge of making sure that tactic applications represent valid rewritings of the program. Finally, the programs created this way are transferred to a compilation back-end, where some automatic static analysis is applied and then executable (C++) code is emitted.

```
Slice (find (θ ↦ ?)) (? ⟨J0 × J0, J0 × J1, J1 × J1⟩)
Stratify "/" (fixee A) A ψ
Stratify "/" (fixee A) A ψ
A B C ↦ SynthAuto . ... ψ
```

Figure 6. Bellmania script used to generate Algorithm 4.

The trusted core is small, comprising of: (a) a type checker (see Section 3.4), (b) a term substitution procedure (see Section 4), (c) a formula simplifier (see Section 5.1), (d) the SMT solver, and (e) the compiler.

An example for the concrete syntax is shown in Figure 6. A full listing of the scripts for our running examples, as well as screenshots of the UI, can be found in Appendix C.1.

3. A Unified Language

For the purpose of manipulating programs and reasoning about them, we first set up a small language that can easily express our programs (that is, partially transformed specifications), and can also be translated to executable code in a straightforward fashion. We choose a functional setting, with scalar types for array indices ($i, j, k : J$ in the example) and values stored in the arrays, typically real numbers ($G_{ij} : \mathbb{R}$ in the example). Arrays are encoded as functions, or arrow types, mapping indices to values (e.g. $G : J^2 \to \mathbb{R}$). We want to have a notion of array elements that are uninitialized, so we assume every scalar type contains the special value $\perp$, corresponding to an undefined value.

Our language should be able to express operations such as Slice from the previous section. To achieve this, we introduce *predicate abstraction* over the index types, by extending the type system with subtyping and *refinement types*. An index type J can be partitioned into subtypes $\langle J_0 \mid J_1 \rangle$ (two or more), meaning that we define predicates $\widehat{J_0}, \widehat{J_1} : J \to \mathbb{B}$ (where $\mathbb{B}$ is the Boolean type) and the refinement types $J_0 = \{v : J \mid \widehat{J_0}(v)\}$ and $J_1 = \{v : J \mid \widehat{J_1}(v)\}$, which become subtypes of J. Usually, we can omit the "hat" and use J_0 as a synonym for $\widehat{J_0}$ when it is clear from the context that it designates a type. We would then provide types for sub-arrays, e.g. $G^{\boxed{2}} : J_0 \times J_1 \to \mathbb{R}$.

To refer to a region of a given array, we define a *guard* operator, which is parameterized by subtypes of the indices, for example $G^{\boxed{2}} = [G]_{J_0 \times J_1}$. To combine computations on different parts of the array, we use a lifting of the ternary condition operator (known as `?:`) to functions, where the condition is implied by the operands; e.g. (Figure 2)

$$G = G^{\boxed{1}} / G^{\boxed{2}} / G^{\boxed{4}} = \lambda ij. \begin{cases} G^{\boxed{1}}_{ij} & i, j \in J_0 \\ G^{\boxed{2}}_{ij} & i \in J_0 \wedge j \in J_1 \\ G^{\boxed{4}}_{ij} & i, j \in J_1 \end{cases}$$

Formal set-up The Bellmania language is based on the polymorphic λ-calculus, that is, simply typed λ-calculus with universally quantified type variables (also known as *System F*). The following subsections contain formal definitions for language constructs.

We write abstraction terms as $(v : \mathcal{T}) \mapsto e$, where $\mathcal{T}$ is the type of the argument v and e is the body, instead of the traditional notation $\lambda(v : \mathcal{T}).\, e$, mainly due to aesthetic reasons but also because we hope this will look more familiar to intended users. Curried functions $(v_1 : \mathcal{T}_1) \mapsto (v_2 : \mathcal{T}_2) \mapsto \cdots \mapsto (v_n : \mathcal{T}_n) \mapsto e$ are abbreviated as $(v_1 : \mathcal{T}_1) \cdots (v_n : \mathcal{T}_n) \mapsto e$. Argument types may be omitted when they can be inferred from the body.

The semantics differ slightly from that of traditional functional languages: arrow types $\mathcal{T}_1 \to \mathcal{T}_2$ are interpreted as **mappings** from values of type $\mathcal{T}_1$ to values of type $\mathcal{T}_2$. Algebraically, interpretations of types, $[\![\mathcal{T}_1]\!]$, $[\![\mathcal{T}_2]\!]$, are sets, and interpretations of arrow-typed terms, $f : \mathcal{T}_1 \to \mathcal{T}_2$, are **partial functions** — $[\![f]\!] : [\![\mathcal{T}_1]\!] \rightharpoonup [\![\mathcal{T}_2]\!]$. This implies that a term $t : \mathcal{T}$ may have an *undefined* value, $[\![t]\!] = \bot_{\mathcal{T}}$ (We would shorten it to $[\![t]\!] = \bot$ when the type is either insignificant or understood from the context). For simplicity, we shall identify $\bot_{\mathcal{T}_1 \to \mathcal{T}_2}$ with the empty mapping $(v : \mathcal{T}_1) \mapsto \bot_{\mathcal{T}_2}$.

All functions are naturally extended, so that $f \bot = \bot$.

3.1 Operators

The core language is augmented with the following intrinsic operators:

- A fixed point operator $\mathrm{fix}\, f$, with denotational semantics

$$[\![\mathrm{fix}\, f]\!] \;=\; \theta \quad \text{s.t.} \quad [\![f]\!]\, \theta = \theta$$

  we assume that recurrences given in specifications are well-defined, such that $[\![f]\!]$ has a single fixed point. In other words, we ignore nonterminating computations — we assume that the user provides only terminating recurrences in specifications.

- A guard operator $[\,]_{\square}$, which comes in two flavors:

$$[x]_{cond} = \begin{cases} x & cond \\ \bot & \neg cond \end{cases}$$
$$[f]_{P_1 \times P_2 \times \cdots P_n} = \overline{x} \mapsto [f\, \overline{x}]_{\bigwedge P_i(x_i)}$$

  where $\overline{x} = x_1 \cdots x_n$. This second form can be used to refer to quadrants of an array; in this form, it always produces a term of type $\square \to \mathcal{R}$, where $\mathcal{R}$ is the domain of f.

- A slash operator $/$:

$$\text{For scalars } x, y : \mathcal{S} \quad x/y = \begin{cases} x & \text{if } x \neq \bot \\ y & \text{if } x = \bot \end{cases}$$
$$\text{For } f, g : \mathcal{T}_1 \to \mathcal{T}_2 \quad f/g = (v : \mathcal{T}_1) \mapsto (f\, v)/(g\, v)$$

  This operator is typically used to combine computations done on parts of the array. For example,

$$\psi \mapsto \left[f\, \psi\right]_{I_0} \Big/ \left[g\, \psi\right]_{I_1}$$

  combines a result of f in the lower indices of a (one-dimensional) array with a result of g in the higher indices (I_0 and I_1, respectively, are the index subsets). Notice that this does not limit the areas from which f and g read; they are free to access the entire domain of ψ.

In our concrete syntax, function application and fix take precedence over $/$, and the body of $\mapsto$ spans as far as possible (like λv in λ-calculus).

3.2 Primitives

The standard library contains some common primitives:

- $\mathbb{R}$, a type for real numbers; $\mathbb{N}$ for natural numbers; $\mathbb{B}$ for Boolean true/false.
- $= : \forall \mathcal{T}.\, \mathcal{T} \to \mathcal{T} \to \mathbb{B}$, always interpreted as equality.
- $+, - : \forall \mathcal{T}.\, \mathcal{T} \to \mathcal{T} \to \mathcal{T}$, polymorphic binary operators.
- $< : \forall \mathcal{T}.\, \mathcal{T} \to \mathcal{T} \to \mathbb{B}$, a polymorphic order relation.
- $cons : \forall \mathcal{T}.\, \mathcal{T} \to (\mathbb{N} \to \mathcal{T}) \to (\mathbb{N} \to \mathcal{T}), nil : \forall \mathcal{T}.\, \mathbb{N} \to \mathcal{T}$, list constructors.
- $\min, \max, \Sigma : \forall \mathcal{T}\mathcal{S}.\, (\mathcal{T} \to \mathcal{S}) \to \mathcal{S}$, reduction (aggregation) operators on ordered/unordered collections. The collection is represented by a mapping $f : \mathcal{T} \to \mathcal{S}$, so that e.g.

$$[\![\min f]\!] = \min \left\{ [\![f]\!]\, v \;\middle|\; v \in [\![\mathcal{T}]\!], [\![f]\!]\, v \neq \bot \right\}$$

  The collections are assumed to be finite.

3.3 Additional Notation

We also adopt some syntactic sugar to make complex terms more manageable:

- $x \gg f \;=\; f\, x$ for application from the left.
- $\langle t_1, \cdots, t_n \rangle \;=\; cons\ t_1\ (cons \cdots (cons\ t_n\ nil) \cdots)$ for fixed-length lists.

3.4 Types and Type Qualifiers

We extend the type system with predicate abstraction in the form of logically qualified data types (Liquid Types [24]). These are refinement types restricted via a set of abstraction predicates, called *qualifiers*, which are defined over the base types. Contrary to the general use of refinement types, the purpose of these qualifiers is not to check a program for safety and reject ill-typed programs, but rather to serve as annotations for tactics, to convey information to the solver for use in the proof, and later to help the compiler properly emit memory access and parallelization primitives.

More specifically, typing $f : \{v : \mathcal{T}_1 \mid P(v)\} \to \mathcal{T}_2$ would mean that $f\, x$ can only be defined where $P(x)$ is true;

otherwise, $f\,x = \bot$. It **does not** mean that the compiler has to prove $P(x)$ at the point where the term $f\,x$ occurs.

As such, we define a Bellmania program to be well-typed iff it is well-typed without the annotations (in its *raw form*). Qualifiers are processed as a separate pass to properly annotate sub-terms.

Some qualifiers are built-in, and more can defined by the user. To keep the syntax simple, we somewhat limit the use of qualifiers, allowing only the following forms:

- $\{v : \mathcal{T} \mid P(v)\}$, abbreviated as $\mathcal{T} \cap P$. When the signature of P is known (which is usually the case), it is enough to write P.
- $\{v : \mathcal{T} \mid P(v) \wedge Q(v)\}$, abbreviated as $\mathcal{T} \cap P \cap Q$, or just $P \cap Q$. This extends to any number of conjuncts of the same form.
- $(x : \mathcal{T}_1) \to \{v : \mathcal{T}_2 \mid R(x, v)\} \to \mathcal{T}_3$, abbreviated as $\big((\mathcal{T}_1 \times \mathcal{T}_2) \cap R\big) \to \mathcal{T}_3$. The qualifier argument x **must** be the preceding argument; this extends to predicates of any arity (that is, a k-ary predicate in a qualifier is applied to the k arguments to the left of it, including the one where it appears).

The type refinement constructors $\cap$ and $\times$ may be composed to create *droplets*, using the abstract syntax in Figure 7. Note that the language does not include tuple types; hence all function types are implicitly curried, even when using $\times$. Droplets can express conjunctions of qualifiers, as long as their argument sets are either disjoint or contained, but not overlapping; for example,

$$x : \{v : \mathcal{T}_1 \mid P(v)\} \to \{v : \mathcal{T}_2 \mid Q(v) \wedge R(x, v)\} \to \mathcal{T}_3$$

can be written as $\big((P \times Q) \cap R\big) \to \mathcal{T}_3$, but

$$x : \mathcal{T}_1 \to y : \{v : \mathcal{T}_2 \mid R(x, v)\} \to \{v : \mathcal{T}_3 \mid R(y, v)\} \to \mathcal{T}_4$$

cannot be represented as a droplet, except by extending the vocabulary of qualifiers.

As with any refinement type system, we define the *shape* of a droplet to be the raw type obtained from it by removing all qualifiers.

Example. The array G computed in the Parenthesis example (Figure 1) can be typed using:

$$G : ((J \times J) \cap <) \to \mathbb{R}$$

This states that $G\,i\,j$ is only defined for $i < j$. It doesn't *force* it to be defined, as it is still a partial function.

Typing Rules

As mentioned earlier, annotations are ignored when type-checking a term. This gives a simple characterization of type safety without the need to explictly write any new typing rules. It also means that for $f : \mathcal{T}_1 \to \mathcal{T}_2$, $x : \mathcal{T}_3$, we obtain $f\,x : \mathcal{T}_2$ whenever $\mathcal{T}_1$ and $\mathcal{T}_3$ have the same shape. This requires some explanation.

$$\begin{array}{lcll} d & ::= & e^1 \;\mid\; e^k \to d & \\ e^1 & ::= & \mathcal{T} & \textit{for scalar type } \mathcal{T} \\ e^{k+l} & ::= & e^k \times e^l & \\ e^k & ::= & e^k \cap P & \textit{for } k\textit{-ary predicate symbol } P \end{array}$$

Figure 7. Syntax of type qualifiers (droplets). k, l are positive integers that stand for dimension indexes.

Considering a (partial) function $\mathcal{T} \to \mathcal{S}$ to be a set of pairs of elements $\langle x, y\rangle$ from its domain $\mathcal{T}$ and range $\mathcal{S}$, respectively, it is clear to see that any function of type $\mathcal{T}_1 \to \mathcal{S}_1$, such that $[\![\mathcal{T}_1]\!] \subseteq [\![\mathcal{T}]\!]$, $[\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{S}]\!]$, is *also*, by definition, a function of type $\mathcal{T} \to \mathcal{S}$, since $[\![\mathcal{T}_1]\!] \times [\![\mathcal{S}_1]\!] \subseteq [\![\mathcal{T}]\!] \times [\![\mathcal{S}]\!]$. If we define subtyping as inclusion of the domains, i.e. $\mathcal{T}_1 <: \mathcal{T}$ whenever $[\![\mathcal{T}_1]\!] \subseteq [\![\mathcal{T}]\!]$, this translates into:

$$\mathcal{T}_1 <: \mathcal{T} \;\wedge\; \mathcal{S}_1 <: \mathcal{S} \;\;\Rightarrow\;\; (\mathcal{T}_1 \to \mathcal{S}_1) <: (\mathcal{T} \to \mathcal{S})$$

In this case, the type constructor $\to$ is **covariant** in both arguments.[4] With this in mind, a function $g : (\mathcal{T} \to \mathcal{S}) \to \mathcal{S}_2$ can be called with an argument $a : \mathcal{T}_1 \to \mathcal{S}_1$, by regular subtyping rules, and $g\,a : \mathcal{S}_2$.

When the argument's type is not a subtype of the expected type, but has the same shape, it is *coerced* to the required type by restricting values to the desired proper subset.

$$\text{For } h : \mathcal{T} \to \mathcal{S} \qquad [\![h\,a]\!] \;=\; [\![h]\!]\big([\![a]\!] :: \mathcal{T}\big)$$

Where :: is defined as follows:

- For scalar (non-arrow) type $\mathcal{T}$,

$$x :: \mathcal{T} \;=\; \begin{cases} x & \text{if } x \in [\![\mathcal{T}]\!] \\ \bot & \text{if } x \notin [\![\mathcal{T}]\!] \end{cases}$$

- $f :: \mathcal{T} \to \mathcal{S} \;=\; x \mapsto \big(f\,(x :: \mathcal{T})\big) :: \mathcal{S}$

We extend our abstract syntax with an explicit *cast operator* $t :: \mathcal{T}$ following this semantics. Notice that this is not the same as $t : \mathcal{T}$, which is a *type judgement*.

Type Inference

Base types are inferred normally as in a classical Hindley-Milner type system [21]. The operators (Section 3.1) behave like polymorphic constants with the following types:

$$\text{fix} \;:\; \forall\mathcal{T}.\; (\mathcal{T} \to \mathcal{T}) \to \mathcal{T} \qquad / \;:\; \forall\mathcal{T}.\; \mathcal{T} \to \mathcal{T} \to \mathcal{T}$$
$$(::\mathcal{T}) \;:\; \text{shape}[\mathcal{T}] \to \text{shape}[\mathcal{T}]$$

As for $[\,]_\square$, for all typing purposes the first variant is a no-op, and the second variant is just syntactic sugar for $:: \square \to \_$, where $\_$ is a fresh type variable.

[4] This is different from classical view, and holds in this case because we chose to interpret functions as *mappings* (see beginning of this section).

Any type variables occurring in type expressions are resolved at that stage through unification. In particular, it means that type variables are always assigned raw types.

Qualifiers are also inferred by propagating them up and down the syntax tree. Since the program already typechecks once the base types are in place, the problem is no longer one of finding *valid* annotations, but rather of *tightening* them as much as possible without introducing semantics-changing coercions. For example, the term $(f :: I \to (I \cap P))\, i$ may be assigned the type I, but it can also be assigned $I \cap P$ without changing its semantics.

Qualifiers are propagated by defining a *type intersection* operator $\sqcap$ that takes two droplets of the same shape $\mathcal{T}_1, \mathcal{T}_2$ and returns a droplet with a conjunction of all the qualifiers occuring in either $\mathcal{T}_1$ or $\mathcal{T}_2$. The operator is defined in terms of the corresponding liquid types:

- If $\mathcal{T}_1 = \{v : \mathcal{T} \mid \varphi_1\}$ and $\mathcal{T}_2 = \{v : \mathcal{T} \mid \varphi_2\}$,
$$\mathcal{T}_1 \sqcap \mathcal{T}_2 \;=\; \{v : \mathcal{T} \mid \varphi_1 \wedge \varphi_2\}$$
- If $\mathcal{T}_1 = x : \mathcal{S}_1 \to \mathcal{S}_2$, $\mathcal{T}_2 = x : \mathcal{S}_3 \to \mathcal{S}_4$ (named arguments are normalized so that $\mathcal{T}_1$ and $\mathcal{T}_2$ use the same names),
$$\mathcal{T}_1 \sqcap \mathcal{T}_2 \;=\; x : (\mathcal{S}_1 \sqcap \mathcal{S}_3) \to (\mathcal{S}_2 \sqcap \mathcal{S}_4)$$

We then define the *type refinement* steps for terms. They are listed in Figure 8. These rules are applied continuously until a fixed point is reached. The resulting types are eventually converted back to droplet form (expressed via $\cap$ and $\times$); qualifiers that cannot be expressed in droplets are discarded.

Note that two syntactically identical terms in different sub-trees may be assigned different types by this method. This is a desirable property, as (some) context information gets encoded in the type that way.

The interested reader can find an example of how type inference is applied in Appendix C.2.

Example

The specification of the Parenthesis problem (Figure 1) will be written as

$$\begin{aligned} &x : J \to \mathbb{R} \\ &w : J^3 \to \mathbb{R} \\ &G \;=\; \text{fix}\ (\theta : J^2_< \to \mathbb{R})\, i\, j \mapsto \big[x_i\big]_{i+1=j} \Big/ \\ &\qquad\qquad\qquad\qquad \min k \mapsto \theta_{ik} + \theta_{kj} + w_{ikj} \end{aligned}$$

$J^2_<$ is used here as an abbreviation for $(J \times J) \cap <$. We also use f_{xy} as a more readable alternative typography for $f\, x\, y$, where f is a function and x, y are its arguments.

Note that the range for k in $\min k \mapsto \cdots$ is implicit, given the type of θ:

$$\theta_{ik} \neq \bot \Rightarrow i < k \quad \text{and} \quad \theta_{kj} \neq \bot \Rightarrow k < j$$

4. Tactics

We now define the method with which our framework transforms program terms, by means of *tactics*. A tactic is a scheme of equalities that can be used for rewriting. When applied to a program term, any occurrence of the **left-hand side** is replaced by the **right-hand side**.[5] A valid application of a tactic is an instance of the scheme that is well-typed and logically valid (that is, the two sides have the same interpretation in any structure that interprets the free variables occurring in the equality).

The application of tactics yields a sequence of program terms, each of which is checked to be equivalent to the previous one. We refer to this sequence by the name *development*.

We associate with each tactic some *proof obligations*, listed after the word ***Obligations*** in the following paragraphs. When applying a tactic instance, these obligations are also instantiated and given to an automated prover. If verified successfully, they entail the validity of the instance. Note that the equality itself can be used as the tactic's proof obligation, if it is easy enough to prove automatically; in such cases we write "***Obligations*:** tactic."

The following are the major tactics provided by our framework. More tactic definitions are given in Appendix B.

Slice $\qquad f \;=\; \big[f\big]_{X_1} \Big/ \big[f\big]_{X_2} \Big/ \cdots \Big/ \big[f\big]_{X_r}$

This tactic partitions a mapping into sub-regions. Each X_i may be a cross product ($\times$) according to the arity of f.

***Obligations*:** tactic.

Informally, the recombination expression is equal to f when $X_{1..r}$ "cover" all the defined points of f (also known as the *support* of f).

Stratify $\qquad \text{fix}(f \mathbin{»} g) \;=\; (\text{fix}\, f) \mathbin{»} \big(\psi \mapsto \text{fix}(\widehat{\psi} \mathbin{»} g)\big)$

where $\widehat{\psi}$ abbreviates $\theta \mapsto \psi$, with fresh variable θ.

This tactic is used to break a long (recursive) computation into simpler sub-computations. ψ may be fresh, or it may reuse a variable already occurring in g, rebinding those occurrences. An example is required to understand why this is useful.

Let $h = \theta \mapsto \big([t_0]_{I_0} / [t_1]_{I_1}\big)$, where $t_{0,1}$ are terms with free variable θ. Stratification is done by setting $f = \theta \mapsto t_0$ and $g = f\,\theta \mapsto [f\,\theta]_{I_0} / [t_1]_{I_1}$. We get $\text{fix}\, h = \text{fix}(f \mathbin{»} g)$; Stratify breaks it into $\text{fix}\, f$ and $\psi \mapsto \text{fix}(\widehat{\psi} \mathbin{»} g)$, which get simplified with β-reduction, giving the expression the form:

$$\big(\text{fix}(\theta \mapsto t_0)\big) \mathbin{»} \Big(\psi \mapsto \text{fix}\,\big(\theta \mapsto [\psi]_{I_0} / [t_1]_{I_1}\big)\Big)$$

This precisely encodes our intuition of computing the fixed point of t_0 first, then t_1 based on the result of t_0.

***Obligations*:** Let $h = f \mathbin{»} g$ and $g' = \psi \mapsto \widehat{\psi} \mathbin{»} g$. Let θ, ζ be fresh variables.

$$f\,(g'\,\zeta\,\theta) \;=\; f\,\zeta \qquad g'\,(f\,\theta)\,\theta \;=\; h\,\theta \tag{4.1}$$

[5] This is also a standard convention in Coq [1], for example.

Core language

$$\frac{e = v \qquad \Gamma, v : \mathcal{T}_1 \vdash e : \mathcal{T}_0}{\Gamma, v : \mathcal{T}_1 \;\vdash\; e : \mathcal{T}_0 \sqcap \mathcal{T}_1} \qquad \frac{e = e_1\, e_2 \qquad \Gamma \vdash e : \mathcal{T},\ e_1 : \mathcal{T}_1 \to \mathcal{S}_1,\ e_2 : \mathcal{T}_2}{\Gamma \;\vdash\; e : \mathcal{T} \sqcap \mathcal{S}_1, \quad e_2 : \mathcal{T}_1 \sqcap \mathcal{T}_2, \quad e_1 : (\mathcal{T}_1 \to \mathcal{S}_1) \sqcap (\mathcal{T}_2 \to \mathcal{T})}$$

$$\frac{e = (v : \mathcal{T}) \mapsto e_1 \qquad \Gamma \vdash e : \mathcal{T}_0 \to \mathcal{S}_0 \qquad \Gamma, v : \mathcal{T} \sqcap \mathcal{T}_0 \vdash e_1 : \mathcal{T}_1}{\Gamma \;\vdash\; e : (\mathcal{T}_0 \to \mathcal{S}_0) \sqcap (\mathcal{T} \to \mathcal{T}_1) \qquad \Gamma, v : \mathcal{T} \sqcap \mathcal{T}_0 \;\vdash\; e_1 : \mathcal{T}_1 \sqcap \mathcal{S}_0}$$

Extensions

$$\frac{e = \mathrm{fix}\, e_1 \qquad \Gamma \vdash e : \mathcal{T},\ e_1 : \mathcal{T}_1 \to \mathcal{T}_2}{\Gamma \vdash e : \mathcal{T} \sqcap \mathcal{T}_2} \qquad \frac{e = e_1 / e_2 \qquad \Gamma \vdash e : \mathcal{T},\ e_1 : \mathcal{T}_1,\ e_2 : \mathcal{T}_2}{\Gamma \vdash e_1 : \mathcal{T}_1 \sqcap \mathcal{T},\ e_2 : \mathcal{T}_2 \sqcap \mathcal{T}}$$

$$\frac{e = [e_1]_{cond} \qquad \Gamma \vdash e : \mathcal{T}, e_1 : \mathcal{T}_1}{\Gamma \vdash e : \mathcal{T} \sqcap \mathcal{T}_1,\ e_1 : \mathcal{T} \sqcap \mathcal{T}_1} \qquad \frac{e = e_1 :: \mathcal{T} \qquad \Gamma \vdash e : \mathcal{T}_0, e_1 : \mathcal{T}_1}{\Gamma \vdash e : \mathcal{T} \sqcap \mathcal{T}_0 \sqcap \mathcal{T}_1,\ e_1 : \mathcal{T} \sqcap \mathcal{T}_0 \sqcap \mathcal{T}_1}$$

Figure 8. Type refinement rules, for inferring qualifiers in sub-expressions.

Synth $\quad \mathrm{fix}\,(h_1 \,/\, \cdots \,/\, h_r) \;=\; f_1 :: \mathcal{T}_1 \,/\, \cdots \,/\, f_r :: \mathcal{T}_r$

This tactic is used to generate recursive calls to sub-programs. For $i = 1..r$, f_i is one of the following: $\mathrm{fix}\, h_i$, $h_i\, \psi$, or $t\, \psi$, where ψ is some variable and t is a term corresponding to a previously defined subroutine (A, B, C in the example). Bellmania chooses these values automatically (see Section 4.1), but the user may override it.

***Obligations*:** Let $h = h_1 / \cdots / h_r$, and let $\mathcal{T} \to \mathcal{T}$ be the shape of h. For each f_i, depending on the form of f_i:

- If $f_i \cong \mathrm{fix}\, f$ (for some f) —
  $h :: (\mathcal{T} \to \mathcal{Y}) = h :: (\mathcal{Y} \to \mathcal{Y}) = f :: (\mathcal{Y} \to \mathcal{T})$ for some $\mathcal{Y}$ which is a subtype of $\mathcal{T}$ and a supertype of $\mathcal{T}_i$.
- If f_i does not contain any "fix" terms —
  $h\,(h\,\theta) :: \mathcal{T}_i = f_i :: \mathcal{T}_i$ for a fresh variable θ.

$\cong$ denotes syntactic congruence up to β-reduction.

Theorem 4.1. *Let $s = s'$ be an instance of one of the tactics introduced in this section. let $a_i = b_i$, $i = 1..k$, be the proof obligations. If $[\![a_i]\!] = [\![b_i]\!]$ for all interpretations of the free variables of a_i and b_i, then $[\![s]\!] = [\![s']\!]$ for all interpretations of the free variables of s and s'.*

Proof is given in Appendix A.

Example

The naïve implementation of Algorithm 1 can be written as

$$\begin{aligned} \Psi \;&=\; i\, j \mapsto [x_i]_{i+1=j} \\ A^{\,J} \;&=\; \psi \mapsto \mathrm{fix}(\theta : J^2_{<} \to \mathbb{R})\ (i : J)\ (j : J) \;\mapsto \\ &\qquad \min \langle\, \psi_{ij}, \\ &\qquad\quad \min\,(k : J) \mapsto \theta_{ik} + \theta_{kj} + w_{ikj} \,\rangle \end{aligned} \tag{4.2}$$

As mentioned in Section 2, the first step Richard does is to apply Slice to the program, thus introducing a partitioning into quadrants.

Slice

$f \;=\; \theta\, i\, j \mapsto \cdots$

$X_1 \;=\; \_ \times J_0 \times J_0 \qquad X_2 \;=\; \_ \times J_0 \times J_1$

$X_3 \;=\; \_ \times J_1 \times J_1$

(*each "_" is a fresh type variable*)

$$A^{\,J} = \psi \mapsto \mathrm{fix} \begin{array}{|c|c|} \hline \boxed{1} & \boxed{2} \\ \hline & \boxed{4} \\ \hline \end{array} \tag{4.3}$$

$$\begin{aligned} \boxed{1} \;&=\; \theta\ (i : J_0)\ (j : J_0) \mapsto \\ &\qquad \min \langle\, \psi_{ij}, \min\,(k : J) \mapsto \theta_{ik} + \theta_{kj} + w_{ikj} \,\rangle \\ \boxed{2} \;&=\; \theta\ (i : J_0)\ (j : J_1) \mapsto \\ &\qquad \min \langle\, \psi_{ij}, \min\,(k : J) \mapsto \theta_{ik} + \theta_{kj} + w_{ikj} \,\rangle \\ \boxed{4} \;&=\; \theta\ (i : J_1)\ (j : J_1) \mapsto \\ &\qquad \min \langle\, \psi_{ij}, \min\,(k : J) \mapsto \theta_{ik} + \theta_{kj} + w_{ikj} \,\rangle \end{aligned}$$

With repeated applications of Slice, a program may grow to become quite large; to make large program terms easy to read and refer to, we provide boxed numerals as labels for sub-terms, using them as abbreviations where they occur in the containing expression.

In addition, to allude to the reader's intuition, expressions of the form $a/b/c/d$ will be written as $\begin{array}{c|c} a & b \\ \hline c & d \end{array}$ when the slices represent quadrants.

The type of θ in (4.3) is still $J^2 \cap < \to \mathbb{R}$. In order to avoid too much clutter caused by type terms, the Bellmania UI uses a heuristic and only displays some of them. By hovering, the user can inspect types that are not shown.

The result is a functional representation of Figure 3(a); with the added term ψ_{ij} it allows A to accept an input array as a parameter and minimize values that are already stored in it; not quite crucial for this particular instance where the input is just Ψ, but in some cases it will become necessary to split a minimization operation into several sub-ranges, to achieve the desired memory locality.

Stratify $\boxed{1}$

$f = \begin{array}{c|c} \boxed{1} & \widehat{\psi} \\ \hline \widehat{\psi} & \widehat{\psi} \end{array}$ (*recall that* $\widehat{\psi} = \theta \mapsto \psi$)

$g = z \mapsto \begin{array}{c|c} z & \boxed{2} \\ \hline & \boxed{4} \end{array} \qquad \psi = \psi$

$$A^J = \psi \mapsto \left(\text{fix} \begin{array}{c|c} \boxed{1} & \widehat{\psi} \\ \hline \widehat{\psi} & \widehat{\psi} \end{array} \right) \gg \psi \mapsto \text{fix} \begin{array}{c|c} \widehat{\psi} & \boxed{2} \\ \hline & \boxed{4} \end{array} \tag{4.4}$$

$\boxed{1}\boxed{2}\boxed{4}$ *as in* (4.3)

The reason for this particular choice of f and g is as explained in Section 4. Richard does not have to worry too much about them, because they are encoded in the tactic application engine, so that Bellmania knows how to build them automatically based on the term being stratified ($\boxed{1}$ in this case).

Notice that an existing variable ψ is reused, rebinding any occurrences within $\boxed{2}$, $\boxed{4}$. This effect is desirable, as it limits the context of the expression: the inner ψ shadows the outer ψ, meaning $\boxed{2}$, $\boxed{4}$ do not need to access the data that was input to $\boxed{1}$, only its output; therefore $\boxed{1}$ can be computed in-place. The proof obligations for Stratify make sure this transition is valid.

At this point Richard can either do another Stratify or a Synth. The order is insignificant, but to be consistent with Figure 4, let us assume he chooses the former.

Stratify $\boxed{1}$

$f = \begin{array}{c|c} \boxed{1} & \widehat{\psi} \\ \hline \widehat{\psi} & \widehat{\psi} \end{array}$ (*recall that* $\widehat{\psi} = \theta \mapsto \psi$)

$g = z \mapsto \begin{array}{c|c} z & \boxed{2} \\ \hline & \boxed{4} \end{array} \qquad \psi = \psi$

$$A^J = \psi \mapsto \left(\text{fix} \begin{array}{c|c} \boxed{1} & \widehat{\psi} \\ \hline \widehat{\psi} & \widehat{\psi} \end{array} \right) \gg \psi \mapsto \left(\text{fix} \begin{array}{c|c} \widehat{\psi} & \boxed{2} \\ \hline \widehat{\psi} & \widehat{\psi} \end{array} \right) \gg \psi \mapsto \text{fix} \begin{array}{c|c} \widehat{\psi} & \widehat{\psi} \\ \hline & \boxed{4} \end{array} \tag{4.5}$$

$\boxed{1}\boxed{2}\boxed{4}$ *as in* (4.3)

Synth $\boxed{1}$

$h_1 = \boxed{1} \qquad h_{2,3,4} = \widehat{\psi}$

$f_1 = A^{J_0}\ \psi \qquad f_{2,3,4} = \psi$

$\mathcal{Y} = J_0^2 \to \mathbb{R}$

Synth $\boxed{4}$

$h_{1,2} = \widehat{\psi} \qquad h_3 = \boxed{4}$

$f_{1,2} = \psi \qquad f_3 = A^{J_1}\ \psi$

$\mathcal{Y} = J_1^2 \to \mathbb{R}$

$$A^J = \psi \mapsto \begin{array}{c|c} A^{J_0}\ \psi & \psi \\ \hline \psi & \psi \end{array} \gg \psi \mapsto \left(\text{fix} \begin{array}{c|c} \widehat{\psi} & \boxed{2} \\ \hline \widehat{\psi} & \widehat{\psi} \end{array} \right) \gg \psi \mapsto \begin{array}{c|c} \psi & \psi \\ \hline & A^{J_1}\ \psi \end{array} \tag{4.6}$$

$\boxed{2}$ *as in* (4.3)

For $\boxed{2}$, the situation is slightly more complicated because no instance of the routine A matches the specification and there are no other routines to choose from. Richard first defines a new routine $B^{J_0 J_1}$ by carving the respective subexpression from the developed program A^J. Notice that B has two parameters, because it depends not only on the index range, but also on the particular partitioning.

4.1 Synthesis-powered Synth Tactic

As mentioned in Sections 1 and 2, the user is assisted by automatic inference while applying tactics. In particular, the Synth tactic requires the user to specify a subroutine to call and parameters to call it with. In addition, the subtype $\mathcal{Y}$ is required to complete the correctness proof. To automate this task, Bellmania employs Counterexample-guided Inductive Synthesis (CEGIS), a software synthesis technique implemented in the tool SKETCH [28]. The proof obligations, along with the possible space of parameter assignments taken from the set of sub-types defined during Slice, are translated to SKETCH. Since SKETCH uses bounded domains, the result is then verified using full SMT.

In addition to considering explicitly defined sub-types, the synthesizer also tries small variations of them to cover corner cases. When index arithmetic is used, the range for a sub-call may have to be extended by a row or column on one or more sides. For each index sub-type $\mathcal{T} \subseteq J$, Bellmania also tries $\mathcal{T} \cup (\mathcal{T} \pm 1)$ for filling in values of parameters:

$$\mathcal{T} + 1 = \{i + 1 \mid i \in \mathcal{T}\} \qquad \mathcal{T} - 1 = \{i - 1 \mid i \in \mathcal{T}\}$$

While the number of combinations is not huge, it is usually hard for the user to figure out which exact call should be made. Since Synth is used extensively throughout the development, This kind of automation greatly improves overall usability.

SKETCH times for the running example range 15–45 seconds. For comparison, covering the same search space for a typical invocation via exhaustive search in C++ took about $1\frac{1}{2}$ hours.

5. Automating Proofs

This section describes the encoding of proof obligations in (many-sorted) first-order logic, and the ways in which type information is used in discharging them.

Each base type is associated with a sort. The qualifiers are naturally encoded as predicate symbols with appropriate sorts. In the following paragraphs, we use a type and its associated sort interchangeably, and the meaning should be clear from the context.

Each free variable and each node in the formula syntax tree are assigned two symbols: a function symbol representing the values, and a predicate symbol representing the support, that is, the set of tuples for which there is a mapping. For example, a variable $f : J \to \mathbb{R}$ will be assigned a function $f^1 : J \to \mathbb{R}$ and a predicate $|f| : J \to \mathbb{B}$. The superscript indictes the function's arity, and the vertical bars indicate the support.

For refinement-typed symbols, the first-order symbols are still defined in terms of the shape, and an assumption concerning the support is emitted. For example, for $g : (J \cap P) \to \mathbb{R}$, the symbols $g^1 : J \to \mathbb{R}$, $|g| : J \to \mathbb{B}$ are defined, with the assumption $\forall \alpha : J.\ |g|(\alpha) \Rightarrow P(\alpha)$.

Assumptions are similarly created for nodes of the syntax tree of the formula to be verified. We define the *enclosure* of a node to be the ordered set of all the variables bound by ancestor abstraction nodes ($v \mapsto \ldots$). Since the interpretation of the node depends on the values of these variables, it is "skolemized", i.e., its type is prefixed by the types of enclosing variables. For example, if $e : \mathcal{T}$, then inside a term $(v : \mathcal{S}) \mapsto \cdots e \cdots$ it would be treated as type $\mathcal{S} \to \mathcal{T}$.

Typically, the goal is an equality between functions $f = g$. This naturally translates to first-order logic as

$$\forall \overline{\alpha}.\ \big(|f|(\overline{\alpha}) \Leftrightarrow |g|(\overline{\alpha})\big) \wedge \big(|f|(\overline{\alpha}) \Rightarrow f^k(\overline{\alpha}) = g^k(\overline{\alpha})\big)$$

First-class functions. When a function is being used as an argument in an application term, we take its arrow type $\mathcal{T} \to \mathcal{S}$ and create a *faux sort* $F_{\mathcal{T}\to\mathcal{S}}$, an "apply" operator $@ : F_{\mathcal{T}\to\mathcal{S}} \to \mathcal{T} \to \mathcal{S}$, and the *extensionality axiom* —

$$\forall \alpha\alpha'.\ \big(\forall \beta.\ @(\alpha, \beta) = @(\alpha', \beta)\big) \Rightarrow \alpha = \alpha' \qquad (5.1)$$

Then for each such function symbol $f^k : \mathcal{T} \to \mathcal{S}$ used as argument, we create its *reflection* $f^0 : F_{\mathcal{T}\to\mathcal{S}}$ defined by

$$\forall \overline{\alpha}.\ @(f^0, \overline{\alpha}) = f^k(\overline{\alpha}) \qquad (5.2)$$

5.1 Simplification

When f,g of the goal $f = g$, are abstraction terms, the above can be simplified by introducing k fresh variables, $\overline{x} = x_1 \cdots x_k$, and writing the goal as $f\,\overline{x} = g\,\overline{x}$. The types of $\overline{x}$ are inferred from the types of f and g (which should have the same shape). We can then apply β-reduction as a simplification step. This dramatically reduces the number of quantifiers in the first-order theory representing the goal, making SMT solving feasible.

Moreover, if the goal has the form $f\,t_1 = f\,t_2$ (e.g. Stratify, Section 4) it may be worth trying to prove that $t_1 :: \mathcal{T} = t_2 :: \mathcal{T}$, where $f : \mathcal{T} \to \mathcal{S}$. This trivially entails the goal and is (usually) much easier to prove.

Another useful technique is common subexpression elimination, which is quite standard in compiler optimizations. Tactic applications tend to create copies of terms, so merging identical subexpressions into a single symbol can drastically reduce the size of the SMT encoding.

6. Code Generation

We built a compiler for programs in Bellmania language that generates efficient C++ code parallelized with Intel Cilk constructs. The compiler uses type information from the development to improve the quality of generated code. From the types of sub-terms corresponding to array indices, the compiler extracts information about what region of the array is read by each computation, and from the types of λ-bound index variable it constructs loops that write to the appropriate regions. The compiler utilizes the SMT solver to infer dependency constraints as in [18], and figures out the direction of each loop (ascending or descending). In addition, inter-quadrant dependencies can be used to determine which computations can be run in parallel (at the granularity of function calls) based on a fork-join model; two calls are considered non-conflicting if each write region is disjoint from the others' read and write regions. Disjointness can be decided using propositional logic through the predicate abstraction induced by type qualifiers.

The compiler also employs more traditional optimization techniques to further simplify the code and improve its running time: (1) lifts conditionals to loop bounds via simple interval analysis, (2) eliminates redundant iterations and comparisons that can be resolved at compile time, and (3) identifies loops that read non-contiguous memory blocks and applies copy optimization [20] automatically to better utilize caches. Examples of generated code are included in the supplemental material with this submission.

7. Empirical Evaluation

We implemented our technique and used it to generate cache-oblivious divide-and-conquer implementations of three algorithms that were used as benchmarks in [30], and a few others.

$$
\begin{aligned}
&w^x : ((I \times I) \cap <) \to J \to \mathbb{R} \\
&w^y : ((J \times J) \cap <) \to I \to \mathbb{R} \\
&G \;=\; \mathrm{fix}\, \theta\, i\, j \mapsto \big[0\big]_{i=j=0} \Big/ \big[w^y_{0j0}\big]_{i=0} \Big/ \big[w^x_{0i0}\big]_{j=0} \Big/ \\
&\qquad\qquad \min\, \langle\, \theta_{(i-1)(j-1)} + c_{ij}, \\
&\qquad\qquad\quad \min p \mapsto \theta_{pj} + w^x_{pij}, \\
&\qquad\qquad\quad \min q \mapsto \theta_{iq} + w^y_{qji}\, \rangle
\end{aligned}
$$

Figure 9. Specifications for the Gap problem.

Parenthesis problem. Our running example; Compute an optimal placement of parentheses in a long chain of multiplication, e.g. of matrices, where the inputs are cost functions x_i for accessing the i-th element and w_{ikj} for multiplying elements $[i, k)$ by elements $[k, j)$.

Gap problem. A generalized minimal edit distance problem. Given two input strings $\overline{x} = x_1 \cdots x_m$ and $\overline{y} = y_1 \cdots y_n$, compute the cost of transforming x into y by any combination of the following steps: (i) Replacing x_i with y_j, at cost c_{ij}, (ii) Deleting $x_{p+1} \cdots x_q$, at cost w^x_{pq}, (iii) Inserting $y_{p+1} \cdots y_q$ in $\overline{x}$, at cost w^y_{pq}. The corresponding recurrence is shown in Figure 9.

Protein Accordion Folding problem. A protein can be viewed as a string $\mathcal{P}_{1..n}$ over an alphabet of amino acids. The protein folds itself in a way that minimizes potential energy. Some of the acids are *hydrophobic*; minimization of the total hydrophobic area exposed to water is a major driving force of the folding process. One possible model is packing $\mathcal{P}$ in a two-dimensional square lattice in a way that maximizes the number of pairs of hydrophobic elements, where the shape of the fold is an *accordion*, alternating between going down and going up.

We also exercised our system on a number of textbook problems: the Longest Common Subsequence (LCS) problem, the Knapsack problem, and the Bitonic Traveling Salesman problem.

7.1 Estimation of User Effort

Because Bellmania is an interactive system, we try to give a measure as to how much effort a typical user has to invest to complete a development for the DP algorithms that comprise our test suite. To get an idea of how domain experts think about the problem, we consult [11], where descriptions are conveniently provided in the form of data-flow diagrams for each step of computation. We compare the sizes of these diagrams, which we call "conceptual size", with the number of tactics required to derive the respective implementation in Bellmania. The results of the comparison are given in Table 1, where "# phases" indicates how many recursive subroutines are included in the algorithm description (and in the respective development) and the two other column give the development size and conceptual size of each benchmark. The development size is within $2\times$ of the conceptual size in most cases.

	# phases	Bellmania # tactics	Conceptual size
Paren	3	30	22
Gap	3	53	27
Protein	4	47	28
LCS	1	5	5
Knapsack	2	49	16
Bitonic	3	32	16

Table 1. Sizes of synthesis scripts compared to conceptual problem size (see Section 7.1).

A sample diagram and Bellmania transcript for the running example are included in the supplemental material.

7.2 Implementation Details

The tactic application engine is implemented in Scala. We implemeted a prototype IDE using HTML5 and AngularJS, which communicates with the engine by sending and receiving program terms serialized as JSON. Our system supports using either Z3 or CVC4 as the back-end SMT solver for discharging proof obligations required for soundness proofs. Synthesis of recursive calls is done by translating the program to SKETCH, which solves a the correct assignment to type parameters. To argue for the feasibility of our system, we include SMT solver running time for the verification of the three most used tactics (figures are for CVC4), as well as time required for SKETCH synthesis, in Table 2. We consider an average delay of ~10 seconds to be reasonable, even for an interactive environment such as Bellmania.

Tactics are implemented as small Scala classes. It is possible for the more advanced user to extend the library by writing such classes. To give an idea, on top of the generic TAE the Stratify tactic was coded in 12 lines of Scala, including the functionality that breaks a function h into two functions f and g.

The compiler for programs in Bellmania is implemented in Python and generates C++ code containing Intel Cilk constructs for parallelization. Table 3 shows performance improvement for our auto-generated implementation (AUTO) on the state-of-the-art optimized parallel loop implementation (LOOPDP) from [30]. It also compares AUTO with manually optimized recursive implementations CO_Opt and COZ for the three problems from [30]. Our compiler automatically does *copy optimization* as done in CO_Opt and COZ. COZ also incorporates a low-level optimization of using Z-order layout of the array, which is out of scope for this paper. N is the problem size and B is the base case size for using loops instead of recursion. It can be seen from the table that our implementation performs close to the manually optimized code and can be optimized further by hand. Figure 10 depicts the performance of these implementations on one sample instance as a function of problem size, and shows the scalability of the generated code.

	Verification			Synthesis
	Slice	Stratify	Synth	Sketch
Paren	0.9	8.7	0.9	24.5
Gap	0.6	6.8	1.4	11.6
Protein	0.9	3.8	0.7	9.5
LCS	0.9	1.9	0.5	3.2
Knapsack	0.3	1.9	0.4	5.3
Bitonic	0.9	7.2	0.6	10.1

Table 2. Average proof search time for proof obligations and avegrage synthesis time for Synth parameters (seconds).

	Speedup w.r.t parallel LOOPDP on 6 cores CPU (12 workers), B=64			
	N	CO_Opt	COZ	AUTO
Parenthesis	16384	32x	50x	46x
Gap	16384	21x	34x	30x
Protein	16384	2.2x	2.6x	1.4x
LCS	45000	–	–	1.5x
Bitonic	45000	–	–	4.2x

Table 3. Performance of different C++ implementations

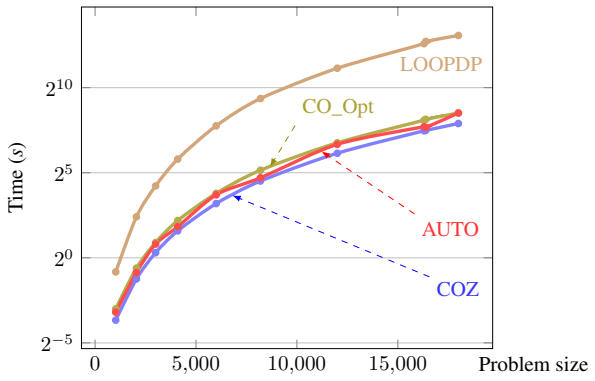

Figure 10. Performance comparison for parallelized implementations for Gap problem on 6-core Intel i7 CPU

8. Related Work

Classical work by Smith *et al.* [25] presents rule-based transformation, stringing it tightly with program verification. This lay the foundation for semi-automatic programming [4, 6, 26]. More recently, a similar approach was introduced into Leon [19], leveraging deductive tools as a way to boost CEGIS, thereby covering more programs. Bellmania takes a dual approach, where automated techniques based on SMT are leveraged to support and improve deductive synthesis.

Inductive synthesis has been the focus of renewed interest thanks to the discovery of techniques that leverage SAT/SMT solvers to symbolically represent and search very large spaces of possible programs [17, 27, 31], and the use of counterexample-guided inductive synthesis (CEGIS), which allows one to leverage inductive techniques to find programs that satisfy more general specifications. Our work is also inspired by the StreamBit project [29], which introduced the idea of transformation rules with missing details that can be inferred by a symbolic search procedure.

Fiat [13] is another recent system that admits stepwise transformation of specifications into programs via a refinement calculus. While Bellmania offloads proofs to SMT and SKETCH, Fiat uses decision procedures in Coq, reling heavily on deductive reasoning and uses Ltac scripts for automation. The intended users of Fiat is regular software developers who invoke pre-packaged scripts, whereas Bellmania targets domain experts who exercise more control over the generated code.

Broadly speaking, the Bellmania system could have been implemented as a library on top of a framework such as Coq or Why3 [15] using binding to SMT solvers provided by these frameworks. The decision not to do so was merely a design choice, to facilitate easier integration with our UI and with SKETCH.

Polyhedral compilers offer some optimizations for the same domain of problem via tiling [5, 22]. While showing significant speedups, these compilers cannot produce divide-and-conquer optimizations, which were proved to be more effective by [30].

Autogen [11] is a most recent advance that employs dynamic analysis to discover a program's access pattern and learn a decomposition that can be used to generate a divide-and-conquer implementation. The two methods are complementary, since Autogen does not provide correctness guarantees, so the user might be able to use insights from Autogen while developing certified code in Bellmania.

Pu *et al.* [23] have shown that recurrences for DP can be generated automatically from a non-recursive specification of the optimization problem. This is orthogonal; in Bellmania, the recurrence is the input, and the output is an efficient divide-and-conquer implementation.

9. Conclusion

The examples in this paper show that a few well-placed tactics can cover a wide range of program transformations. The introduction of solver-aided tactics allowed us to make the library of tactics smaller, by enabling the design of higher-level, more generic tactics. Their small number gives the hope that end-users with some mathematical background will be able to use the system without the steep learning curve that is usually associated with proof assistants. This can be a valuable tool for algorithms research.

But solver-aided tactics should not be seen as specific to divide-and-conquer algorithms, or even to algorithms. The same approach can be applied to other domains. Domain knowledge can be used to craft specialized tactics, providing users with the power to use a high-level DSL to specify their requirements, without sacrificing performance.

References

[1] The Coq proof assistant, reference manual. `https://coq.inria.fr/refman`.

[2] R. E. Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[3] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 958–971, 2014.

[4] L. Blaine and A. Goldberg. DTRE - a semi-automatic transformation system. In *Constructing Programs from Specifications*, pages 165–204. Elsevier, 1991.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, 2008.

[6] M. Butler and T. Långbacka. Program derivation using the refinement calculator. In *Theorem Proving in Higher Order Logics: 9th International Conference, volume 1125 of Lecture Notes in Computer Science*, pages 93–108. Springer Verlag, 1996.

[7] R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 591–600, 2006.

[8] R. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 207–216, 2008.

[9] R. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47(4):878–919, 2010.

[10] R. Chowdhury, H.-S. Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(3):495–510, 2010.

[11] R. Chowdhury, P. Ganapathi, J. J. Tithi, C. Bachmeier, B. C. Kuszmaul, C. E. Leiserson, A. Solar-Lezama, and Y. Tang. Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, page 10, 2016.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[13] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 689–700, 2015.

[14] R. Durbin, S. R. Eddy, A. Krogh, and G. J. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[15] J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *ESOP*, Lecture Notes in Computer Science, pages 125–128. Springer, 2013.

[16] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, 1999.

[17] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011.

[18] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[19] E. Kneuss, V. Kuncak, I. Kuraj, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[20] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 63–74, 1991.

[21] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[22] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

[23] Y. Pu, R. Bodík, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 83–98, 2011.

[24] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, 2008.

[25] D. R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[26] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Software Eng.*, 16(9):1024–1043, 1990.

[27] A. Solar-Lezama. The sketching approach to program synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, pages 4–13, 2009.

[28] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[29] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 281–294, 2005.

[30] J. J. Tithi, P. Ganapathi, A. Talati, S. Agarwal, and R. Chowdhury. High-performance energy-efficient recursive dynamic programming using matrix-multiplication-like flexible kernels. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, 2015.

[31] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, 2013.