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

Shachar Itzhaky Rohit Singh
Armando Solar-Lezama
Kuat Yessenov Yongquan Lu Charles Leiserson
MIT, USA

Rezaul Chowdhury
Stony Brook University, NY, USA

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 verifies 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

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
        $G_{ij} := \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$
```

Keywords Divide-and-conquer, Dynamic Programming, Program Synthesis, Theorem Proving, Refinement Types, SMT

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.[1] The first loop fills the diagonal of an array with some initial values, and the second loop computes off-diagonal elements by reading existing elements.

The algorithm computes an $n \times n$ region of a DP table G via a standard row-major order. This algorithm is simple, but a direct C implementation of it turns out to be about $10\times$ slower than the best manually optimized implementation (as we will see in Section 7.2). One reason for this poor performance is the high rate of cache misses incurred by reading the ranges G_{ik} and G_{kj}, for $i < k < j$, repeatedly on every iteration of the loops over 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 [32]. An excerpt of the pseudo-code for such an implementation is shown in Algorithm 2. In the optimized version, the

[1] `https://en.wikipedia.org/wiki/Matrix_chain_multiplication`

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[2], which allows an expert to interactively generate parallel divide-and-conquer implementations of dynamic programming algorithms, which are provably correct relative to a high-level specification of the code in Algorithm 1. We show that the resulting implementations are 1.4–46× faster than the original versions and within 2–60% of a high-performance hand crafted implementation. Their structure will also make them *cache oblivious* [18] and *cache adaptive* [4], just like the hand crafted implementations are.

Bellmania is a deductive synthesis system in the tradition of systems like Kids [27], and more recently Fiat [14]. 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.

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

Algorithm 2 An optimized divide-and-conquer version

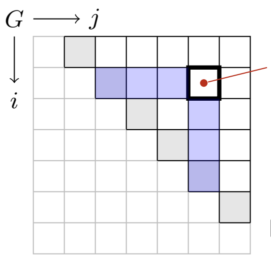
```
A[1..n], where:                                   (snippet)
procedure A[$s..e$]
    if $e - s < b$ then
        for $i = e..s$ do
            for $j = \max\{s, i\}..e$ do
                $G_{ij} := \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$
    else
        A[$s..\lfloor\frac{s+e}{2}\rfloor$]
     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

2. Overview

Most readers are likely familiar with the Dynamic Programming (DP) technique of Richard Bellman [3] 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* implementation strategies that recursively partition the space of subproblems into smaller subspaces [4, 8–11, 32].

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.

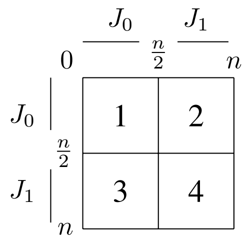
Iterative Algorithm. Using the standard dynamic programming method, anyone who has read [13] would compute this recurrence with an iterative program by understanding the dependency pattern: to compute the $\min_{i<k<j}(\cdots)$ expression in Figure 1 the algorithm needs to enumerate k and gather information 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,



$$G_{ij} = \begin{cases} x_i & i{+}1 = j \\ \min\limits_{i<k<j} G_{ik}{+}G_{kj}{+}w_{ikj} & i{+}1 < j \end{cases}$$

▢ base case 🟪 dependencies ▢ empty

Figure 1. Recurrence equation and cell-level dependencies.



$\boxed{2}$ *depends on* $\boxed{1}$

$\boxed{2}$ *depends on* $\boxed{4}$

($\boxed{3}$ *is empty*)

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

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 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

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.

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.

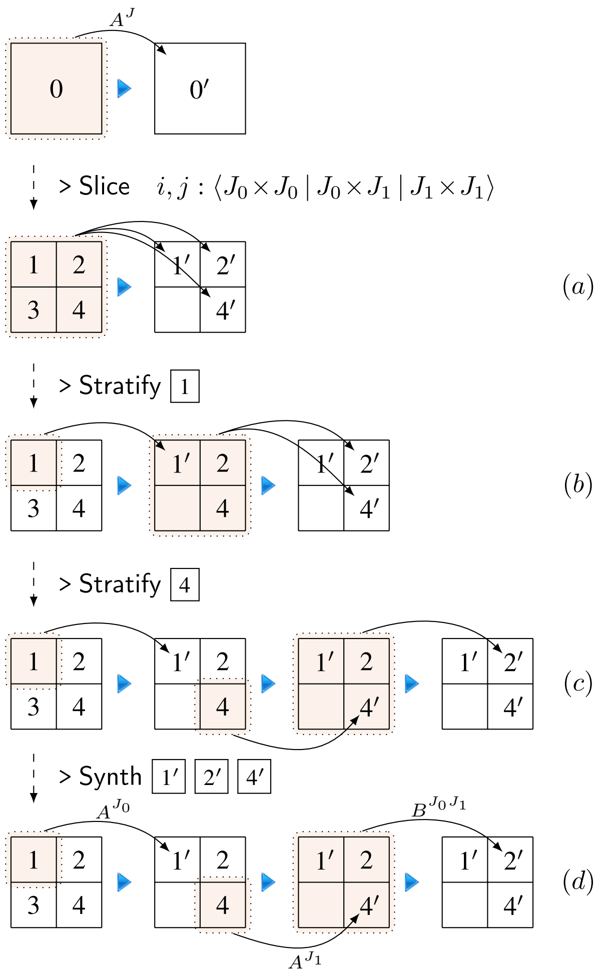
The computation of $\boxed{1}$ (the top-left quadrant) does not depend on any of the other computations, so Richard applies the Stratify tactic, which separates an independent computation step as a separate loop. This is equivalent to rewriting the specification as in Figure 3(b): the first computation 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 program 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 $\mathrm{A}\big[G^{\boxed{1}}\big]$ (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 $\mathrm{A}\big[G^{\boxed{4}}\big]$. The remaining part, $G^{\boxed{2}}$, is essentially different, so Richard gives it a new name, "B", which becomes a new synthesis task.[3]

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



A^J
0
0′
> Slice $i,j : \langle J_0 \times J_0 \mid J_0 \times J_1 \mid J_1 \times J_1 \rangle$
1 2 3 4
1′ 2′ 4′
(a)
> Stratify $\boxed{1}$
(b)
> Stratify $\boxed{4}$
(c)
> Synth $\boxed{1'}$ $\boxed{2'}$ $\boxed{4'}$
A^{J_0}
$B^{J_0 J_1}$
A^{J_1}
(d)

Figure 4. Overview of tactic semantics in Bellmania.

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

Algorithm 3 Parenthesis — Sliced and Stratified

```
procedure 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
      $G_{ij} := \min_{i<k<j} G_{ik} + G_{kj} + w_{ikj}$
```

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.[4]

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.

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 (see Section 6). In its current version, Bellmania does not emit machine-checkable proofs; to the extent that SMT solvers can emit such proofs, these could be adapted to a proof of refinement (in the sense of [14]). The compiler has to be formally verified separately. These two tasks require considerable engineering effort and are left for the future.

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.

Interaction Model

The intended use pattern for Bellmania is by repeatedly issuing commands to apply tactics inside a REPL that keeps

[4] 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 A[$G$]
  if $G$ is very small then run iterative version
  else
    A[$G^{\boxed{1}}$]                                  ▷ Compute $\boxed{1}$
    A[$G^{\boxed{4}}$]                                  ▷ Compute $\boxed{4}$
    B[$G^{\boxed{1}}, G^{\boxed{4}}, G^{\boxed{2}}$]    ▷ Compute $\boxed{2}$
```

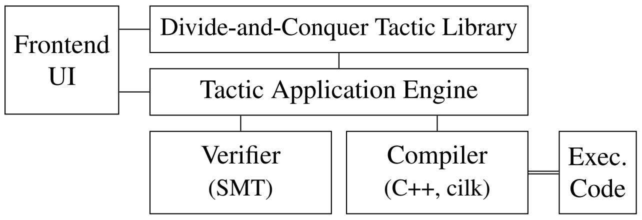


Figure 5. Overall design of Bellmania.

`Slice (find (`$\theta \mapsto ?$`)) (? `$\langle J_0{\times}J_0,\ J_0{\times}J_1,\ J_1{\times}J_1\rangle$`)`
`Stratify "/" (fixee `$\boxed{A}$`) `$\boxed{A}\ \psi$
`Stratify "/" (fixee `$\boxed{A}$`) `$\boxed{A}\ \psi$
$\boxed{A}\ \boxed{B}\ \boxed{C} \mapsto$ `SynthAuto . ...` ψ

Figure 6. Bellmania script used to generate Algorithm 4.

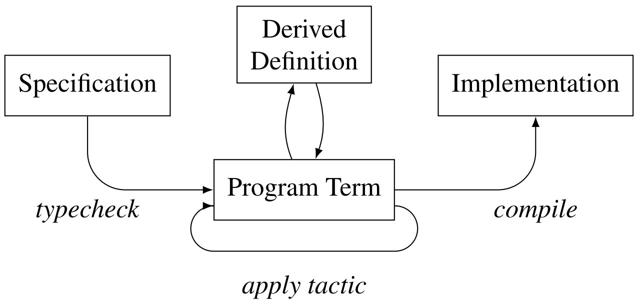


Figure 7. Interaction workflow in Bellmania.

showing to the user the resulting *program term* (Figure 7). To begin the interaction, the user types in a *specification* written in the Bellmania language (as defined in Section 3), which is typechecked by the system, producing a term. This term then becomes the focus of the development, and further transformations apply to it; the user issues *apply tactic* commands, one at a time, using a syntax similar to Figure 6. Tactic applications are also typechecked, as well as the resulting program, since type information can be refined by including the context (see Section 3.4). During the development, sub-computations may be discovered that require further drilling down (such as "B" in the last example). These result in *derived* terms that take the focus until they, too, have been fully transformed. When the program and all its sub-computations are fully developed and expressed as a divide-and-conquer algorithm, the user invokes the compiler back-end that emits C++ code.

In the following sections, we describe the programming language and formally define the tactics that were used in the example above. We then show how to formalize the same intuition as we had there, using this new instrument.

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 $\bot$, 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)

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

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 $\operatorname{fix} f$, with denotational semantics
$$[\![\operatorname{fix} f]\!] = \theta \quad \text{s.t.} \quad [\![f]\!]\,\theta = \theta$$
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 $[\,]_{\Box}$, 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 $\Box \to \mathcal{R}$, 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.
- $\mathit{cons} : \forall\mathcal{T}.\ \mathcal{T} \to (\mathbb{N}{\to}\mathcal{T}) \to (\mathbb{N}{\to}\mathcal{T}), \mathit{nil} : \forall\mathcal{T}.\ \mathbb{N} \to \mathcal{T}$, 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.
- $\langle t_1, \cdots, t_n\rangle = \mathit{cons}\ t_1\ (\mathit{cons} \cdots (\mathit{cons}\ t_n\ \mathit{nil}) \cdots)$ 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 [25]). These are refinement types that provide a natural encoding of predicate abstraction by restricting refinements to set of abstraction predicates, called *qualifiers*, which are defined over the base types. Contrary to their general use, the purpose of these qualiiers in Bellmania 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 proofs, 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 the type as 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* (tiny bits of liquid), using the abstract syntax in Figure 8. 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.

Through the use of droplets, we manage to harness refinement types without being to cumbersome and requiring the user to write verbose annotations. To this end, we trade off some expressive power for succinctness.

Example

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

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

Figure 8. Syntax of type qualifiers (droplets). k, l are positive integers that stand for dimension indexes.

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

$J^2_<$ is used here as an abbreviation for $(J \times J) \cap <$. We also use f_{xy} as a more readable typographic alternative 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:

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

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* (the raw type obtained by removing all qualifiers). This requires some explanation.

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.[5] 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:

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

- 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 [23]. 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.

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 $\mathcal{T}_1 = x : \mathcal{S}_1 \to \mathcal{S}_2$, $\mathcal{T}_2 = x : \mathcal{S}_3 \to \mathcal{S}_4$ (normalized so that $\mathcal{T}_1$ and $\mathcal{T}_2$ use the same names for arguments),

$$\mathcal{T}_1 \sqcap \mathcal{T}_2 \;=\; x : (\mathcal{S}_1 \sqcap \mathcal{S}_3) \to (\mathcal{S}_2 \sqcap \mathcal{S}_4)$$

We then define the *type refinement* steps for terms. They are listed in Figure 9. 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.

The top three rules apply to ordinary constructs of typed λ-calculus: the first rule propagates qualifiers from the environment to leaf expressions that use a declared variable;

Core language

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

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

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

the next two rules propagate information across application and refinement terms. For an application $f\,i$, both the type of i and the domain of f can be refined via the other, e.g. if $f : I_0 \to \mathbb{R}$ and $i : I$, then i can be refined to I_0; symmetrically, if $f : I \to \mathbb{R}$ and $i : I_0$, then f can be refined to $I_0 \to \mathbb{R}$. Similarly, the range of f and the type of $f\,i$ can refine each other. The rule for abstraction terms is analogous, except that information about the type of the argument v is passed through the environment.

The bottom rules pertain to language extensions defined previously in this section. For $/$, qualifiers can safely seep down to sub-expressions (but not back up), e.g. if $x/y : I_0$, it is safe to assert $x : I_0$, $y : I_0$ without changing the semantics of the expression. For casts, x and $x :: \mathcal{T}$ are tied to have the same refinements, and those of $\mathcal{T}$ are appended. A condition $[x]_{\square}$ ties the sub-term type in the same way, but without adding any qualifiers. The rule for fix is the only one that requires some thought: the type of $\mathrm{fix}\, f$ cannot be forced down on f; but since, by definition, $\mathrm{fix}\, f = f\,(\mathrm{fix}\, f)$ must hold, then any restrictions imposed by the range of f can be asserted for $\mathrm{fix}\, f$.

Note that two syntactically identical terms in different subtrees 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.

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**.[6] 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. In some cases, the obligations are strictly stronger than the validity of the rewrite, which means the TAE is not complete and may fail to apply a valid transformation. It is important to note that the user does not have to specify, or in fact be aware of, any proof obligations; they are defined as part of the tactic library.

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

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

This tactic partitions a mapping into sub-regions. Each X_i may be a cross product ($\times$) according to the arity of f. For example, $X_1 = J_0 \times J_0$, $X_2 = J_0 \times J_1$, $X_3 = J_1 \times J_1$.

***Obligations*:** just the equality above.

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).

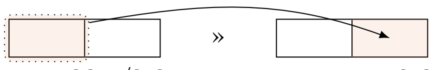
[6] This is also a standard convention in Coq [1], for example.

Stratify $\quad \mathrm{fix}(f \gg g) \;=\; (\mathrm{fix}\, f) \gg \big(\psi \mapsto \mathrm{fix}(\widehat{\psi} \gg g)\big)$

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

This tactic is used to break a long (recursive) computation into steps. ψ is bound the result of the first computation (f), and then used as the input for the second step (g). Typically, we apply Stratify to a computation that has previously been Sliced (although that does not have to be the case); to understand the efficacy of Stratify, we present a minimal example.

Suppose the program $\mathrm{fix}\,\theta \mapsto \big([t]_{I_0}/[t]_{I_1}\big)$, where $t = i \mapsto \theta_{i-1} + 1$. We would like to obtain a program that computes the lower half I_0 first, then the upper half I_1; that is, $\big(\mathrm{fix}\,\theta \mapsto [t]_{I_0}/[\psi]_{I_1}\big) \gg \big(\psi \mapsto \mathrm{fix}\,\theta \mapsto [\psi]_{I_0}/[t]_{I_1}\big)$.



$\mathrm{fix}\,\theta \mapsto [t]_{I_0}/[\psi]_{I_1} \qquad \psi \mapsto \mathrm{fix}\,\theta \mapsto [\psi]_{I_0}/[t]_{I_1}$

This can be obtained by choosing $f = \theta \mapsto [t]_{I_0}/[\psi]_{I_1}$ and $g = f\,\theta \mapsto [f\,\theta]_{I_0}/[t]_{I_1}$.

Notice how $f\,\theta$ is used as a placeholder for the sub-computation f in g. On the one hand, $f \gg g = \theta \mapsto \big[[t]_{I_0}/[\psi]_{I_1}\big]_{I_0}/[t]_{I_1}$, which, modulo some simplification, is equivalent to the original term $\theta \mapsto [t]_{I_0}/[t]_{I_1}$. On the other hand, when instantiating Stratify, $f\,\theta$ becomes $\widehat{\psi}\,\theta$, which is equivalent to ψ, giving the term for the second phase.

ψ may be fresh, or it may reuse a variable already occurring in g, rebinding those occurrences. This precisely encodes our intuition of computing the first fixed point *in situ*, then the second one based on the result of the first.

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

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

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

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

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

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

Slice
$f \;=\; \theta\,i\,j \mapsto \cdots$
$X_1 \;=\; \_ \times J_0 \times J_0 \quad X_2 \;=\; \_ \times J_0 \times J_1$
$X_3 \;=\; \_ \times J_1 \times J_1$

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

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

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

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 $\boxed{1}$, $\boxed{2}$, etc. as labels for sub-terms, using them as abbreviations for these terms 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 types of the quadrants should be clear from the context; here, their types are

$$\boxed{1} : \_\!\times J_0 \!\times J_0 \!\to\! \mathbb{R}, \;\; \boxed{2} : \_\!\times J_0 \!\times J_1 \!\to\! \mathbb{R}, \;\; \boxed{4} : \_\!\times J_1 \!\times J_1 \!\to\! \mathbb{R}$$

The type of θ in (4.3) is still $J^2_{<} \to \mathbb{R}$. 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 an argument and minimize values that are already stored in it; not quite crucial for this particular instance where the input is just Ψ, but very useful in cases where it is necessary to split a minimization operation into several sub-ranges, to achieve the desired memory locality.

Stratify $\boxed{1}$

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

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

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

$\boxed{1}\boxed{2}\boxed{4}$ *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 $\boxed{1}$

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

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

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

$\boxed{1}\boxed{2}\boxed{4}$ *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 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. Next, Richard will carry on developing B in a similar manner.

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 [29]. 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 [20], 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. Type information is heavily utilized here: occurrences of array variables signify reads, so their types denote (an over-approximation of) the region being "read from", whereas return types denote the region being "written to". 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, (3) identifies loops that read non-contiguous memory blocks and applies copy optimization [22] automatically to better utilize caches. Examples of generated code are included in the Bellmania repository [2], and (4) follows a simple heuristic that inserts loop vectorization directives.

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 [32], and a few others.

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

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

Figure 10. Specifications for the Gap problem.

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 10.

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 Implementation Details

The tactic application engine is implemented in Scala. We implemented 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 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 1. 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 back-end is implemented as another component in Scala, processing ASTs and generating C++ code containing Intel Cilk constructs for parallelization. It employs a thin intermediate representation layer at which the optimizations of Section 6 are applied.

	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 1. Average proof search time for proof obligations and average synthesis time for Synth parameters (seconds).

	Speedup w.r.t parallel LOOPDP on 16 cores CPU (16 workers), B=64			
	N	CO_Opt	COZ	AUTO
Parenthesis	16384	9.8x	11.4x	11.1x
Gap	16384	6.6x	8.4x	8.5x
Protein	16384	5.1x	5.5x	3.1x
LCS	45000	–	–	3.9x
Bitonic	45000	–	–	3.8x

Table 2. Performance of different C++ implementations

7.2 Experimental Results

Table 2 shows performance improvement for our auto-generated implementation (AUTO) on the state-of-the-art optimized parallel loop implementation (LOOPDP) that was used by [32]. It also compares AUTO with manually optimized recursive implementations CO_Opt and COZ for the three problems from [32][7]. 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. Figure 11 depicts the performance of these implementations on one sample instance as a function of problem size, and shows the scalability of the generated code.

7.2.1 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

[7] Speedups in the table are lower than in [32]; this should be attributed to improvement done to the baseline LOOPDP.

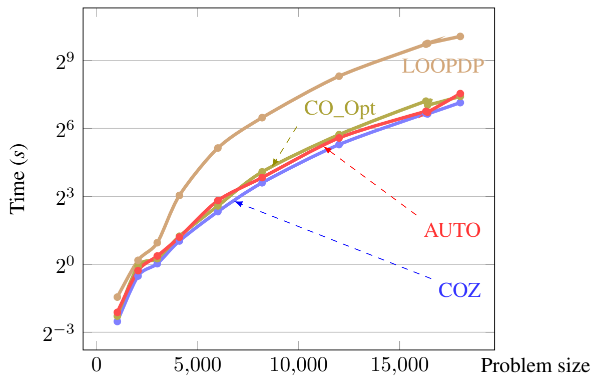


Figure 11. Performance comparison for parallelized implementations for Gap problem on 16-core Intel Xeon 2.4GHz CPU

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

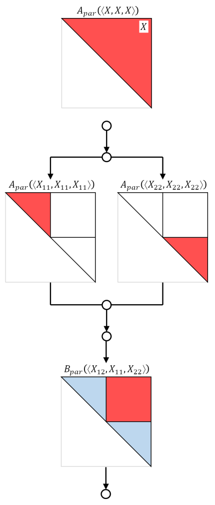
Table 3. Sizes of synthesis scripts compared to conceptual problem size (see Section 7.2.1).

our test suite. To get an idea of how domain experts think about the problem, we consult [12], where descriptions are conveniently provided in the form of data-flow diagrams for each step of computation. An example for such a diagram, corresponding to our Algorithm 4, is shown in Figure 12; in this case, we count it as 4 steps.

We compare the sizes of these diagrams, which we label "# steps", with the number of tactic applications required to derive the respective implementation in Bellmania. The results of the comparison are given in Table 3, where "# phases" indicates how many recursive subroutines are included in the algorithm description (and in the respective development) and the two other columns give the sizes of the description vs. that of the development. The development size is within $2\times$ of the diagrams' size in most cases, peaking at $3\times$.

Diagrams and complete Bellmania transcript for the running example are included in Appendix C.1.

Admittedly, the measured volume ratio provides a crude proxy for usability. A controlled user study should be run to provide true evidence, but that goes beyond the scope of this paper. Still, the results are reassuring since through the course of developing the tool we were able to reduce the number of reasoning steps from hundreds to a few dozens, and also make them high-level. We were also able to detect mistakes in existing (hand written) formulations of divide-and-conquer algorithms, that eluded discovery by experts. The additional examples in the appendix should furnish sufficient intuition to conceive a qualitative opinion on the efficacy of the approach and the prototype.



$A_{par}(\langle X,X,X\rangle)$
X
$A_{par}(\langle X_{11},X_{11},X_{11}\rangle)$
$A_{par}(\langle X_{22},X_{22},X_{22}\rangle)$
$B_{par}(\langle X_{12},X_{11},X_{22}\rangle)$

Figure 12. An example diagram (from [12])

8. Further Improvements

Overall usability can be greatly enhanced with some syntactic sugar for program terms and tactic application commands. The current syntax is part of a proof-of-concept UI. User experience can also be enriched with direct manipulation gestures, such as pointing to a sub-term in the program rather than using boxed literals to refer to them, and "carving" sub-computations from the current term by selecting and dragging.

The tactic library can be extended with a number of important optimization for this domain. *Dimensionality reduction* is a technique that allows to lower an n-dimensional DP problem to $n-1$ dimensions by computing layer by layer. This has an acute difference from merely buffering the most recent layer; without going too much into details, we point out that intermediate layers computed this way may differ from the corresponding projection of the n-dimensional table, while the final layer is identical. This tactic can enable a whole sub-class of high dimension DP problems.

Some low-level tactics can boost performance as well: the Protein benchmark runs faster using the hand-coded version due to a subtle case splitting around the innermost loop, that reduces an expression of the form $min(k, 2j-i+1)$ to just $2j-i+1$, allowing in turn to factor an addend out of the loop (k is the innermost loop variable). Doing the same optimization by hand on the code produced by Bellmania leads to the same speedup, but this boost cannot be claimed until it is automated.

Z-ordering of the array is also a general technique and can be automated as part of the compiler back-end. However, this and other optimizations (such as vectorization and copy optimization mentioned in Section 6) are delicate and sometimes even degrade performance, so they better be auto-tuned rather than applied heuristically.

9. Related Work

Classical work by Smith *et al.* [26] presents rule-based transformation, stringing it tightly with program verification. This lay the foundation for semi-automatic programming [5, 6, 27]. Later, these concepts have been applied to divide-and-conquer in a disciplined way in [7, 31]; these address divide-and-conquer in the classical sense of [13] (Chapter 4), focusing on parallelism. In Bellmania, more focus is put on re-ordering of array reads and writes, following and mechanizing techniques related to DP from [8, 9]. In fact, traditional parallelism is taking "for granted" for our aggregation operators, since they are associative and methods such as [17] apply rather trivially, and translated into the tactics Slice, Assoc, and Distrib. On top of these algebraic transformations, Bellmania allows clever re-orderings, especially through Stratify and Let. (Some of the these tactics appear in the appendix.) More recently, a similar approach was introduced into Leon [21], 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 [19, 28, 33], 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 [30], which introduced the idea of transformation rules with missing details that can be inferred by a symbolic search procedure.

Fiat [14] 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 [16] 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.

Autogen [12] 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: it works for a class of problems that obey a "small world assumption", meaning that all possible behaviors are demonstrated by traces of bounded, known size. Crucially, in Autogen it is the user's responsibility to determine whether the input problem falls within this category; if it does not, Autogen will not fail but instead generate an incorrect implementation. This is a fundamental difference stemming from Autogen's use of dynamic traces *vs.* purely deductive reasoning in Bellmania. Still, the user might be able to use insights from Autogen to develop verified code in Bellmania, where size bounds are not required.

Pu *et al.* [24] 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. Obviously, the recurrence produced by [24] can be used as input to Bellmania, providing an even higher-level end-to-end reasoning.

10. 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.

Moreover, limiting the number of tactics shrinks the space in which to search for programs, so that an additional level automation may be achieved via AI or ML methods. As more developments are done by humans and collected in a database, those algorithms would become more adept in predicting the next step of the construction.

In a broader outlook, the technique for formalizing transformation tactics is not particularly attached to divide-and-conquer algorithms and their implementations. In this work, we constructed a generic tactic application engine, on top of which the tactics required for our domain were easy to implement. This gives rise to the hope that, in the future, the same approach can be applied to other domains, in the interest of encoding domain knowledge, providing better DSLs that offer users the power to write high-level programs without sacrificing performance.

Acknowledgments

This work is supported by NSF Grants CCF-1139056, CCF-1439084 and CNS-1553510. We thank Shoaib Kamil and Adobe research for their feedback and support.

References

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

[2] Bellmania repository on github. `https://github.com/corwin-of-amber/bellmaniac/`.

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

[4] 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.

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

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

[7] W.-N. Chin, J. Darlington, and Y. Guo. Parallelizing conditional recurrences. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*, Euro-Par '96, pages 579–586, 1996.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 135–146, 1994. ISBN 0-89791-662-X.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

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

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

[28] 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.

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

[30] 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.

[31] Y. M. Teo, W.-N. Chin, and S. H. Tan. Deriving efficient parallel programs for complex recurrences. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation*, PASCO '97, pages 101–110, 1997.

[32] 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.

[33] 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.

A. Proof of Soundness

Theorem A.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. For the tactics with ***Obligations*:** tactic, the theorem is trivial.

> For Stratify, let f, g be partial functions such that

$$\forall \theta, \zeta. \quad f\,(g\,\zeta\,\theta) \;=\; f\,\zeta \quad\wedge\quad g\,(f\,\theta)\,\theta \;=\; h\,\theta$$

Assume that $\zeta = \text{fix}\, f$ and $\theta = \text{fix}(g\,\zeta)$. That is, $f\,\zeta = \zeta$ and $g\,\zeta\,\theta = \theta$. Then —

$$h\,\theta = g\,(f\,\theta)\,\theta = g\,(f\,(g\,\zeta\,\theta))\,\theta = g\,(f\,\zeta)\,\theta = \theta$$

So $\theta = \text{fix}\, h$. We get $\text{fix}\, h = \text{fix}\,\big(g\,(\text{fix}\, f)\big)$; equivalently,

$$\text{fix}\, h = (\text{fix}\, f) \,\text{»}\, \big(\psi \mapsto \text{fix}(g\,\psi)\big)$$

Now instantiate h, f, and g, with $f \,\text{»}\, g$, f, and g' from (4.1), and we obtain the equality in the tactic.

> For Synth, (*i*) assume $f_i = \text{fix}\, g$ and

$$h :: \mathcal{T} \to \mathcal{Y} = h :: \mathcal{Y} \to \mathcal{Y} = g :: \mathcal{Y} \to \mathcal{T}$$

Intuitively, $\mathcal{Y}$ "cuts out" a region of an array $\theta :: \mathcal{T}$ given as input to h and g. This area is self-contained, in the sense that only elements in $\mathcal{Y}$ are needed to compute elements in $\mathcal{Y}$, as indicated by the refined type $\mathcal{Y} \to \mathcal{Y}$.

Notice that from the premise follows $g :: \mathcal{Y} \to \mathcal{T} = g :: \mathcal{Y} \to \mathcal{Y}$. We use the following corollary:

Corollary. Let $f : \mathcal{T} \to \mathcal{T}$; if either $f :: \mathcal{T} \to \mathcal{Y} = f :: \mathcal{Y} \to \mathcal{Y}$ or $f :: \mathcal{Y} \to \mathcal{T} = f :: \mathcal{Y} \to \mathcal{Y}$, then $(\text{fix}\, f) :: \mathcal{Y} = \text{fix}(f :: \mathcal{Y} \to \mathcal{Y})$.

Proof follows later in this appendix.

From the corollary, and for the given h and g, we learn that $(\text{fix}\, h) :: \mathcal{Y} = \text{fix}(h :: \mathcal{Y} \to \mathcal{Y})$, and also $(\text{fix}\, g) :: \mathcal{Y} = \text{fix}(g :: \mathcal{Y} \to \mathcal{Y})$. Since $h :: \mathcal{Y} \to \mathcal{Y} = g :: \mathcal{Y} \to \mathcal{Y}$, we get $(\text{fix}\, h) :: \mathcal{Y} = (\text{fix}\, g) :: \mathcal{Y}$; now, $\mathcal{Y}$ is a supertype of $\mathcal{T}_i$, so $(\theta :: \mathcal{Y}) :: \mathcal{T}_i = \theta :: \mathcal{T}_i$:

$$\begin{aligned}(\text{fix}\, h) :: \mathcal{T}_i &= ((\text{fix}\, h) :: \mathcal{Y}) :: \mathcal{T}_i = ((\text{fix}\, g) :: \mathcal{Y}) :: \mathcal{T}_i = \\ &= (\text{fix}\, g) :: \mathcal{T}_i = f_i :: \mathcal{T}_i\end{aligned}$$

(ii) Assume $h\,(h\,\theta) :: \mathcal{T}_i = f_i :: \mathcal{T}_i$ holds for any $\theta : \mathcal{T}$, then in particlar, for $\theta = \text{fix}\ h$, we get $h\,(h\ \text{fix}\, h) :: \mathcal{T}_i = f_i :: \mathcal{T}_i$. Since $h\,(h\ \text{fix}\, h) = \text{fix}\, h$, we obtain the conjecture $(\text{fix}\, h) :: \mathcal{T}_i = f_i :: \mathcal{T}_i$. □

Our reliance on the termination of fix expressions may seem conspicuous, since some of these expressions are generated automatically by the system. However, a closer look reveals that whenever such a computation is introduced, the set of the recursive calls it makes is a subset of those made by the existing one. Therefore, if the original recurrence terminates, so does the new one. In any case, all the recurrences in our development have a trivial termination argument (the indexes i,j change monotonically between calls), so practically, this should never become a problem.

We now prove the corollary from the proof of Synth.

Corollary. Let $f : \mathcal{T} \to \mathcal{T}$; if either $f :: \mathcal{T} \to \mathcal{Y} = f :: \mathcal{Y} \to \mathcal{Y}$ or $f :: \mathcal{Y} \to \mathcal{T} = f :: \mathcal{Y} \to \mathcal{Y}$, then $(\text{fix}\, f) :: \mathcal{Y} = \text{fix}(f :: \mathcal{Y} \to \mathcal{Y})$.

Proof.

For the first case, assume $\theta = \text{fix}\, f$,

$$\begin{aligned}\theta :: \mathcal{Y} &= (\theta \,\text{»}\, f) :: \mathcal{Y} = \theta \,\text{»}\, (f :: \mathcal{T} \to \mathcal{Y}) = \\ &= \theta \,\text{»}\, (f :: \mathcal{Y} \to \mathcal{Y}) = (\theta :: \mathcal{Y}) \,\text{»}\, (f :: \mathcal{Y} \to \mathcal{Y})\end{aligned}$$

This means that $\theta :: \mathcal{Y} = \text{fix}(f :: \mathcal{Y} \to \mathcal{Y})$, as desired. For the second case, from domain theory we know that $\text{fix}\, f = f^k\bot$ for some $k \geq 1$. We prove by induction that $f^k\bot = (f :: \mathcal{Y} \to \mathcal{Y})^k\bot$.

For $k = 1$,

$$f\,\bot = f\,(\bot :: \mathcal{Y}) = (f :: \mathcal{Y} \to \mathcal{T})\,\bot = (f :: \mathcal{Y} \to \mathcal{Y})\,\bot$$

Assume $f^k\bot = (f :: \mathcal{Y} \to \mathcal{Y})^k\bot$, then definitely $f^k\bot = f^k\bot :: \mathcal{Y}$. Therefore,

$$\begin{aligned}f^{k+1}\bot &= (f^k\bot) \,\text{»}\, f = (f^k\bot :: \mathcal{Y}) \,\text{»}\, f = \\ &= (f^k\bot) \,\text{»}\, (f :: \mathcal{Y} \to \mathcal{T}) = \\ &= ((f :: \mathcal{Y} \to \mathcal{Y})^k\bot) \,\text{»}\, (f :: \mathcal{Y} \to \mathcal{Y}) = \\ &= (f :: \mathcal{Y} \to \mathcal{Y})^{k+1}\bot\end{aligned}$$

From this we learn that $\text{fix}\, f = \text{fix}(f :: \mathcal{Y} \to \mathcal{Y}) = (\text{fix}\, f) :: \mathcal{Y}$.

B. More Tactics

For most of the tactics below, the proof obligation is exactly the equality that that expresses the rewrite. We signify this with the notation ***Obligations*:** $\star$.

Shrink

$$f \;=\; f :: \mathcal{T}$$

Used to specify tighter qualifiers for the type of a sub-term.

***Obligations*:** $\star$.

For arrow-typed terms, this essentially requires to prove that f is only defined for arguments in the domain of $\mathcal{T}$, and that the values are in the range of $\mathcal{T}$. This can be seen as a special case of Slice with $r = 1$, with the additional feature of specifying the range as well.

Associativity

$$\mathrm{reduce}\,\big\langle\, \mathrm{reduce}\langle\overline{x}_1\rangle, \cdots, \mathrm{reduce}\langle\overline{x}_r\rangle \big\rangle \;=\; \mathrm{reduce}\langle\overline{x}_1, \cdots, \overline{x}_r\rangle$$

where reduce is a built-in aggregation ($\min$, $\max$, Σ), and $\overline{x}_i$ are lists of terms (of the same type). If any of $\overline{x}_i$ is of length one, $\mathrm{reduce}\langle\overline{x}_i\rangle$ can be replaced by $\overline{x}_i$.

***Obligations*:** none.

Distributivity

Let e be an expression with a hole, $e[\Box] = (\cdots \Box \cdots)$.

$$\begin{aligned} e[t_1/\cdots/t_r] &= e[t_1]/\cdots/e[t_r] \\ e[t_1/\cdots/t_r] &= \mathrm{reduce}\langle e[t_1], \cdots, e[t_r]\rangle \\ \mathrm{reduce}\, e[t_1/\cdots/t_r] &= \mathrm{reduce}\langle \mathrm{reduce}\, e[t_1], \cdots, \mathrm{reduce}\, e[t_r]\rangle \end{aligned}$$

This tactic provides several alternatives for different uses of aggregations. Clearly, $/$ does not distribute over any expression; we give just a few examples where this tactic is applicable.

- $(x/y) + 1 \;=\; (x+1)\,/\,(y+1)$
- $x/0 \;=\; \max\langle x, 0\rangle$ (for $x : \mathbb{N}$)
- $\min\big([f]_{J_0}\,/\,[f]_{J_1}\big) \;=\; \min\langle \min[f]_{J_0}\,,\;\min[f]_{J_1}\rangle$

***Obligations*:** $\star$.

Elimination

$$e[t] \;=\; e[\bot]$$

Used to eliminate a sub-term that is either always undefined or has no effect in the context in which it occurs.

***Obligations*:** $\star$.

Let Insertion

Let e be an expression with a hole, $e[\Box] = (\cdots x_1 \mapsto \cdots x_k \mapsto \cdots \Box \cdots)$, where $x_{1..k} \mapsto$ are abstraction terms enclosing $\Box$. The bodies may contain arbitrary terms in addition to these abstractions.

$$\begin{aligned} e[t] &= (\overline{x} \mapsto t) \text{ » } z \mapsto e[z\,\overline{x}] \\ e[\mathrm{reduce}\langle\overline{a}, \overline{b}\rangle] &= (\overline{x} \mapsto \mathrm{reduce}\langle\overline{a}\rangle) \\ &\qquad \text{ » } z \mapsto e[\mathrm{reduce}\langle z\,\overline{x}, \overline{b}\rangle] \end{aligned}$$

where $\overline{x} = x_{1..k}$, and z is a fresh variable. This tactic also has a special version that involves reduce. The items in $\langle\overline{a}, \overline{b}\rangle$ may be interleaved, since $\min, \max, \Sigma$ all happen to be commutative.[8]

***Obligations*:** tactic, if z occurs free in e; otherwise none.

Let Insertion $[\mathrm{reduce}]$

$$\begin{aligned} e[\mathrm{reduce}\langle\overline{a}, \overline{b}\rangle] &= (\overline{x} \mapsto \mathrm{reduce}\langle\overline{a}\rangle) \\ &\qquad \text{ » } z \mapsto e[\mathrm{reduce}\langle z\,\overline{x}, \overline{b}\rangle] \end{aligned}$$

where $\overline{x} = x_{1..k}$, and z a fresh variable.

***Obligations*:** $\star$, if z occurs free in e; otherwise none.

Padding

$$t \;=\; \big(t\,/\,f_1/\cdots/f_r\big) :: \mathcal{T}$$

where $\mathcal{T}$ is the type of t. This tactic is commonly used with Let insertion, to make the type of a sub-computation match the type of the entire term.

***Obligations*:** $\star$.

Pull Out

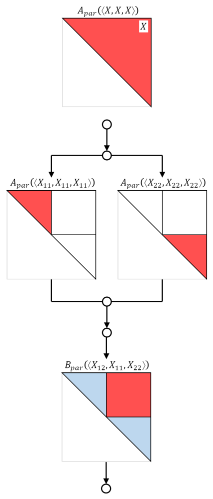
For $e[\Box]$ as defined previously:

$$z \;=\; \overline{x} \mapsto t$$

where z is a fresh variable.

Similar to Let Insertion, but does not change the original term; instead, it is used to single out and name a particular expression t, preserving the context in which it occurs in $e[t]$. It is not a tactic *per se*, as it does not actually effect any transformation on $e[t]$; instead, it is designed to increase readability of the development and simplify successive human-computer interaction.

[8] If non-commutative functions get added in the future, then this will change into $\langle\overline{a}, \overline{b}, \overline{c}\rangle$ non-interleaving, with the right hand side being $(\overline{x} \mapsto \mathrm{reduce}\langle\overline{b}\rangle)$ » $z \mapsto e[\mathrm{reduce}\langle\overline{a}, z\,\overline{x}, \overline{c}\rangle]$.



$A_{par}(\langle X,X,X\rangle)$
X
$A_{par}(\langle X_{11},X_{11},X_{11}\rangle)$
$A_{par}(\langle X_{22},X_{22},X_{22}\rangle)$
$B_{par}(\langle X_{12},X_{11},X_{22}\rangle)$

Slice (find ($\theta \mapsto$?)) (? $\langle J_0 \times J_0$, $J_0 \times J_1$, $J_1 \times J_1 \rangle$)
Stratify "/" (fixee $\boxed{A}$) $\boxed{A}$ ψ
Stratify "/" (fixee $\boxed{A}$) $\boxed{A}$ ψ
$\boxed{A}$ $\boxed{B}$ $\boxed{C}$ $\mapsto$ SynthAuto . ... ψ

Figure 13. Development of subroutine A of the Parenthesis problem as conceptually described in [12] (top) and using Bellmania (bottom).

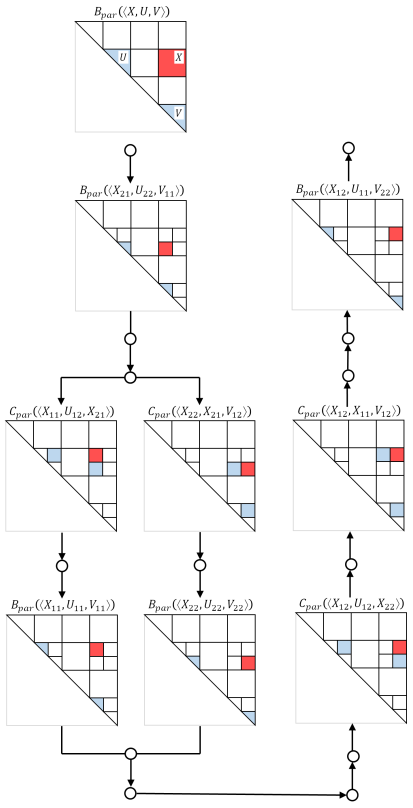
C. Examples

Many aspects of Bellmania are best illustrated via examples. While the main sections include many such examples, some more elaborated ones may prove useful and interesting.

C.1 Development of Parenthesis (with diagrams)

The running example from Sections 1 and 2 has a total of three subroutines, which we label A, B, and C. As promised, we include original design diagrams by the technique's authors taken from [12]. The blocks (triangles) in the diagrams represent intermediate steps of the computation. Below each diagram, we show a transcript of what the user has to type in when using Bellmania to carry out the same development.

Boxed letters in the scripts are used to refer to sub-terms of the current program (in Section 4 we used boxed digits, but in the actual UI we use letters because there are more of them). Reading the scripts in a non-interactive setting might be hard since the reader cannot observe the program; they are listed here just to give an idea of the size and structure. Bellmania repository [2] contains some screen-shots of an interactive session.



$B_{par}(\langle X,U,V\rangle)$
U
X
V
$B_{par}(\langle X_{21},U_{22},V_{11}\rangle)$
$B_{par}(\langle X_{12},U_{11},V_{22}\rangle)$
$C_{par}(\langle X_{11},U_{12},X_{21}\rangle)$
$C_{par}(\langle X_{22},X_{21},V_{12}\rangle)$
$C_{par}(\langle X_{12},X_{11},V_{12}\rangle)$
$B_{par}(\langle X_{11},U_{11},V_{11}\rangle)$
$B_{par}(\langle X_{22},U_{22},V_{22}\rangle)$
$C_{par}(\langle X_{12},U_{12},X_{22}\rangle)$

Slice f (? $\langle K_0 \times K_2$, $K_0 \times K_3$, $K_1 \times K_2$, $K_1 \times K_3 \rangle$)
$\boxed{D}$ $\mapsto$ Stratify "/" (fixee .) . ψ
$\boxed{C}$ $\mapsto$ Stratify "/" (fixee .) . ψ
$\boxed{E}$ $\mapsto$ Stratify "/" (fixee .) . ψ

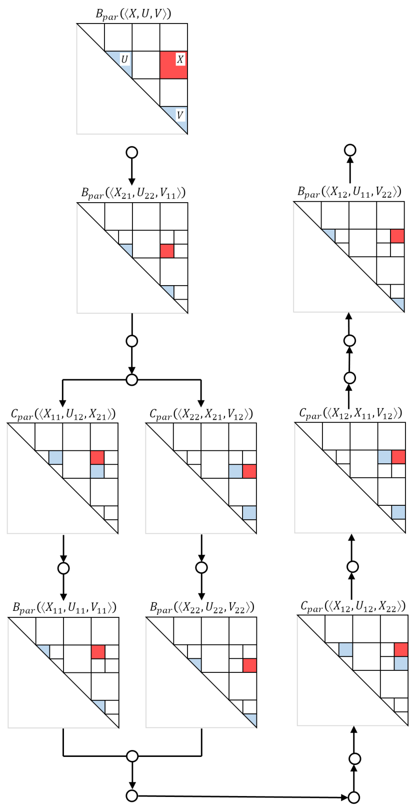
$\langle$Slice (find (k $\mapsto$?)) $\langle K_0$, K_1, K_2, $K_3 \rangle$,
 Slice (find (k $\mapsto$?)) $\langle K_1$, K_2, $K_3 \rangle$,
 Slice (find (k $\mapsto$?)) $\langle K_0$, K_1, $K_2 \rangle$ $\rangle$

Distrib min
Assoc min

$\langle$Stratify min (fixee $\boxed{A}$) $\langle \boxed{G}, \boxed{J} \rangle$ ψ,
 Stratify min (fixee $\boxed{B}$) $\langle \boxed{M}, \boxed{O} \rangle$ ψ,
 Stratify min (fixee $\boxed{C}$) $\langle \boxed{R}, \boxed{T} \rangle$ ψ $\rangle$
Stratify min (fixee $\boxed{A}$) $\langle \boxed{I}, \boxed{K} \rangle$ ψ

$\boxed{I}$ $\boxed{S}$ $\boxed{Z}$ $\underline{\boxed{G}}$ $\boxed{M}$ $\boxed{P}$ $\boxed{W}$ $\underline{\boxed{D}}$ $\mapsto$ SynthAuto . ... ψ

Figure 14. Same, for subroutine B of Parenthesis.



$B_{par}(\langle X,U,V\rangle)$
U
X
V
$B_{par}(\langle X_{21},U_{22},V_{11}\rangle)$
$B_{par}(\langle X_{12},U_{11},V_{22}\rangle)$
$C_{par}(\langle X_{11},U_{12},X_{21}\rangle)$
$C_{par}(\langle X_{22},X_{21},V_{12}\rangle)$
$C_{par}(\langle X_{12},X_{11},V_{12}\rangle)$
$B_{par}(\langle X_{11},U_{11},V_{11}\rangle)$
$B_{par}(\langle X_{22},U_{22},V_{22}\rangle)$
$C_{par}(\langle X_{12},U_{12},X_{22}\rangle)$

Slice (find (i $\mapsto$?)) $\langle \mathrm{L}_0\times\mathrm{L}_4, \mathrm{L}_0\times\mathrm{L}_5, \mathrm{L}_1\times\mathrm{L}_4, \mathrm{L}_1\times\mathrm{L}_5\rangle$
Let "/" (slasher $\boxed{\mathrm{A}}$) $\boxed{\mathrm{A}}$ ψ
Let "/" (slasher $\boxed{\mathrm{A}}$) $\boxed{\mathrm{A}}$ ψ
Let "/" (slasher $\boxed{\mathrm{A}}$) $\boxed{\mathrm{A}}$ ψ
Slice (findAll ($k \mapsto$?)) $\langle L_2, L_3\rangle$

Distrib min
Assoc min

$\langle$Let min (slasher $\boxed{\mathrm{A}}$) $\langle\boxed{\mathrm{E}},\boxed{\mathrm{G}}\rangle$ ψ,
 Let min (slasher $\boxed{\mathrm{B}}$) $\langle\boxed{\mathrm{H}},\boxed{\mathrm{J}}\rangle$ ψ,
 Let min (slasher $\boxed{\mathrm{C}}$) $\langle\boxed{\mathrm{K}},\boxed{\mathrm{M}}\rangle$ ψ,
 Let min (slasher $\boxed{\mathrm{D}}$) $\langle\boxed{\mathrm{N}},\boxed{\mathrm{P}}\rangle$ ψ $\rangle$

$\boxed{\mathrm{A}}$ $\boxed{\mathrm{B}}$ $\boxed{\mathrm{C}}$ $\boxed{\mathrm{D}}$ $\boxed{\mathrm{E}}$ $\boxed{\mathrm{F}}$ $\boxed{\mathrm{G}}$ $\boxed{\mathrm{H}}$ $\mapsto$ SynthAuto . ... ψ

Figure 15. Same, for subroutine C of Parenthesis

C.2 Qualified Type Inference

We provide an example of how qualifiers are inferred in program terms.

Example

Assume that:

- I, T are types
- $\widehat{I_0} : I \to \mathbb{B}$ is a unary qualifier
- $0 : T$ is a constant
- $\mathcal{S}$ a type variable,

Consider the term $(f : I_0 \to \mathcal{S})\, i \mapsto f\, i\, i \,/\, 0$. The first step of Hindley-Milner inference will induce the following type shapes through unification:

$$\underbrace{\underbrace{(f : I \to \mathcal{S})^{0}}_{I \to I \to T}\quad \underbrace{i^{0}}_{I}\quad \mapsto\quad \underbrace{\underbrace{\underbrace{\underbrace{f^{1}}_{I \to I \to T}\ \underbrace{i^{1}}_{I}}_{I \to T}\ \underbrace{i^{2}}_{I}}_{T}\quad /\quad \underbrace{0}_{T}}_{T}}_{(I \to I \to T) \to I \to T}$$

Superscript numerals denote different occurrences of the same variable. In this case, the type variable $\mathcal{S}$ has been assigned $I \to T$.

The process would have stopped here if it weren't for the qualifier I_0 used in the type for f. At this point we can use type refinements to get more accurate types for f and i in the body of the function term.

$$\frac{f : I_0 \to I \to T,\ i : I \ \vdash\ f^{1} : I \to I \to T}{f : I_0 \to I \to T,\ i : I \ \vdash\ f^{1} : I_0 \to I \to T}$$

Notice that $(I \to I \to T) \sqcap (I_0 \to I \to T) = I_0 \to I \to T$. Truthfully, in this case this is quite a trivial result.

Let $\Gamma = \{f : I_0 \to I \to T,\ i : I\}$.

$$\frac{\Gamma\ \vdash\ (f^{1}\, i^{1}) : I \to T,\ \ f^{1} : I_0 \to I \to T,\ \ i^{1} : I}{\Gamma\ \vdash\ (f^{1}\, i^{1}) : I \to T,\ \ f^{1} : I_0 \to I \to T,\ \ i^{1} : I_0}$$

The types of f^{1} and $f^{1}\, i^{1}$ have not changed, but the type of i^{1} was lowered to I_0.

After applying the typing rules similarly to all the subterms, we get the inferred types as shown:

$$\underbrace{\underbrace{(f : I_0 \to \mathcal{S})}_{I_0 \to I \to T}\quad \underbrace{i}_{I}\quad \mapsto\quad \underbrace{\underbrace{\underbrace{\underbrace{f}_{I_0 \to I \to T}\ \underbrace{i}_{I_0}}_{I_0 \to T}\ \underbrace{i}_{I}}_{T}\quad /\quad \underbrace{0}_{T}}_{T}}_{(I_0 \to I \to T) \to I \to T}$$