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

Abstract

We introduce *solver-aided tactics* as a mechanism for transforming computational terms in the interest of deriving better, more efficient implementations. Solver-aided tactics allow us to combine deductive and constraint-based synthesis to generate provably correct efficient implementations from a very high-level specification of an algorithm by chaining together a small number of high-level transformation steps. Our solver-aided tactics also leverage a type system, which incorporates predicate abstraction to associate semantic information with program terms, to guide transformations and provide enough context for automated proofs.

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. In particular, it provides solver-aided tactics that formalize the divide-and-conquer technique and a visualization interface to help users to interactively guide the transformation process. 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

Keywords Synthesis; Dynamic Programming

1. Introduction

There are two broad categories of software synthesis techniques: *inductive* approaches, which generalize from concrete values or execution traces, and *deductive* approaches, which derive an implementation from a specification through deductive reasoning steps. 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 [13, 21, 25], and the use of counterexample-guided inductive synthesis (CEGIS), which allows one to leverage inductive techniques to find programs that satisfy more general specifications. Deductive techniques, however, still hold some important advantages over inductive approaches; in particular, their scalability is not limited by the power of a checking oracle, because the correctness of the implementation is guaranteed by construction.

In this paper, we present a new approach to interactive deductive synthesis based on *solver-aided tactics* that preserves the benefits of deductive synthesis techniques but reduces the burden on the user by relying on two important innovations: (a) the use of inductive synthesis to discover important details of the low-level steps needed to achieve a transformation, (b) the use of a type system based on predicate abstraction (liquid types) that associates semantic information with program terms, enabling automated verification of the validity of a transformation. Both of these innovations rely on aggressive use of SMT solvers to discharge complex proof obligations that would otherwise have to be discharged interactively with significant manual effort.

We believe the approach has the potential to be generally applicable to a variety of synthesis problems, but in this paper, we focus on a particular domain of *divide-and-conquer dynamic programming* algorithms. Specifically, we have developed a system called *Bellmania* that uses solver-aided tactics specialized for this domain to help an algorithm designer derive divide-and-conquer dynamic programming al-

gorithms from a high-level specification. As we illustrate in the next section, this domain is challenging not just as a synthesis target, but also for human experts. Therefore, in addition to serving as a test bed for a new synthesis approach, the development of Bellmania is a significant achievement in itself.

Our work on solver-aided tactics builds on prior work on the StreamBit project [23], which introduced the idea of transformation rules with missing details that can be inferred by a symbolic search procedure, as well as the pioneering work on the Leon synthesizer, which has explored the use of deductive techniques to improve the scalability of inductive synthesis. However, our approach is unique in the way it leverages inductive synthesis and liquid types in the context of deductive synthesis: (a) the solver can use inductive synthesis to search for the detailed parameters required for a transformation, (b) The solver can prove validity of side conditions that ensure the soundness of each individual transformation, (c) the tactics can leverage information from logically qualified types in the program in order to guide the transformation. The flexibility of being able to rely on the solver to check the validity of transformations means that we do not have to be conservative when checking if a transformation can be applied. In addition, the reliance on the synthesizer to fill in parameters and proof details means that the user has to type less and has less chance for error.

Overall, we make the following contributions.

- We introduce *solver-aided tactics* as a way to raise the level of abstraction of deductive synthesis.
- We develop a small library of these formal tactics that can be used to systematically transform a class of problem specifications, expressed as recurrences in a simple functional language, into equivalent divide-and-conquer programs that admit cache-oblivious parallel implementations.
- We prove that these tactics are semantics-preserving, assuming some side conditions are met at the point when the tactic is applied.
- We show that the side conditions can be effectively translated into first-order closed formulas, and verified automatically by SMT solvers.
- We demonstrates the first system capable of generating provably correct implementations of divide-and-conquer implementations from a high-level description of the algorithm.
- We measure the performance of automatically generated code and show that it is comparable to manually tuned reference implementations written by experts.

2. Overview

Most readers are likely familiar with the Dynamic Programming (DP) technique of Richard Bellman [1] 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 in order to explore otherwise exponential-sized problem spaces in polynomial time. Dynamic programs are usually described through recurrence relations that specify how the cells in a DP table must be filled using solutions already computed for other cells, 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 (see, e.g., [24]). For example, Tithi *et al.* have shown that for classical DP problems such as Floyd-Warshall, the parallel divide-and-conquer implementation is 8x faster across a range of problem sizes compared with a parallel tiled implementation thanks to the better temporal locality and the additional optimization opportunities exposed by partitioning [24]. These performance differences matter because DP is central to many important domains ranging from logistics to computational biology; as an illustrative example, a recent textbook [11] on biological sequence analysis lists 11 applications of DP in bioinformatics just in its introductory chapter, with many more in chapters that follow.

To illustrate the key concepts undelying Bellmania, we will walk through the first few steps that an algorithms expert — whom we will call Richard — would follow to generate a provably correct divide-and-conquer implementation of a DP algorithm. As a motivating example, we consider the Simplified Arbiter problem. Two processes x and y must be scheduled to run n and m seconds, respectively, on a single processor, using one-second slots. Execution starts at $t = 0$. The cost for scheduling the slots $[a..b)$ of x after having scheduled slots $[0..c)$ of y is given by w^x_{abc}, and the cost for schedulting the slots $[a..b)$ of y after scheduling $[0..c)$ of x is given by w^y_{abc}.

The optimal cost for scheduling the first i slots of x and the first j slots of y is given by the recurrence G_{ij} in Figure 1. When i is zero, it means that only y has been scheduled, so the cost is w^y_{0j0}, and similarly when j is zero, the cost is w^x_{0i0}. When i and j are both positive, there are two options: either the schedule ends with an allocation to x, where slots $[p..i)$ of x were scheduled at $t = p + j$, and

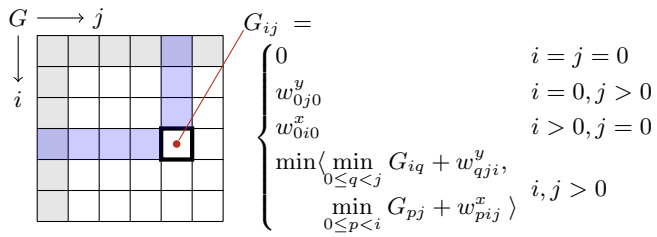

Figure 1. Recurrence equation and cell-level dependencies.

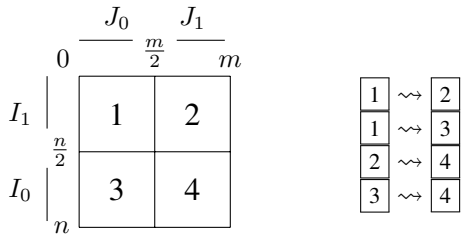

Figure 2. Dividing a two-dimensional array into quadrants; the dependencies are shown on the right.

the cost is $G_{pj} + w^x_{pij}$; or it ends with an allocation to y, where slots $[q..j)$ of y were scheduled at $t = i + q$, and the cost is $G_{iq} + w^y_{qji}$. The minimum over all respective $p < i$ and $q < j$ is taken. Eventually, the optimal cost of the entire schedule is given by G_{nm}.

Iterative Algorithm. Using a standard dynamic programming method, our algorithm expert Richard would compute this recurrence with an iterative program by understanding the dependency pattern: to compute the $\min\langle\cdots\rangle$ expression in Figure 1 and find the optimal values for p and q, the algorithm needs information from all cells above and to the left of G_{ij}. In particular, each value G_{ij} is computed from other values $G_{i'j'}$ with lower indexes, $i' < i,\ j' < j$. Therefore, considering G as a two-dimensional array, it can be filled in a single pass from left to right and from top to bottom, as shown in Algorithm 1.

Algorithm 1 Iterative Simplified Arbiter

$G_{00} := 0$ ▷ *Initialize*
for $i = 1..n$ **do** $G_{i0} := w^x_{0i0}$
for $j = 1..m$ **do** $G_{0j} := w^y_{0j0}$
for $i = 1..n$ **do** ▷ *Compute*
  for $j = 1..m$ **do**
    $G_{ij} := \min\langle \min_{0\le q<j} G_{iq} + w^y_{qji},\ \min_{0\le p<i} G_{pj} + w^x_{pij} \rangle$

Divide-and-Conquer Algorithm. Divide-and-conquer is a common algorithm development pattern ([9], chapter 4) that has recently been applied to DP ([5–8]). This approach has the benefit of yielding cache-oblivious implementations by increasing memory locality while preserving parallelism. With divide-and-conquer, the DP table is partitioned into regions, and each region is expressed as a sub-problem to be solved.

We will now describe how Richard approaches the running example using Bellmania. He would like to partition the two-dimensional array G into quadrants, as illustrated in Figure 2. In Bellmania, this is accomplished by applying the Slice tactic, illustrated graphically at the top of Figure 5. *Tactics* are transformation steps that manipulate the program, and represent a high-level refinement concept. The partitions are labeled I_0, I_1 and J_0, J_1 for row index ranges and column index ranges, respectively. Slicing gives the analog of the specification in Figure 3(*i*). The expression inside $\min\langle\cdots\rangle$ is shortened for space, but it is the same as in Algorithm 1.

Slice $i : \langle I_0|I_1\rangle$ $j : \langle J_0|J_1\rangle$ (i)
$\forall i,j \in \boxed{1}.\ G_{ij} = \min\langle\cdots G_{iq}\cdots G_{pj}\cdots\rangle$
$\forall i,j \in \boxed{2}.\ G_{ij} = \min\langle\cdots G_{iq}\cdots G_{pj}\cdots\rangle$
$\forall i,j \in \boxed{3}.\ G_{ij} = \min\langle\cdots G_{iq}\cdots G_{pj}\cdots\rangle$
$\forall i,j \in \boxed{4}.\ G_{ij} = \min\langle\cdots G_{iq}\cdots G_{pj}\cdots\rangle$
Stratify $\boxed{1}$ (ii)
$\forall i,j \in \boxed{1}.\ G^{\boxed{1}}_{ij} = \min\langle\cdots G^{\boxed{1}}_{iq}\cdots G^{\boxed{1}}_{pj}\cdots\rangle$
$\forall i,j \in \boxed{2}.\ G_{ij} = \min\langle\cdots (G^{\boxed{1}}/G)_{iq}\cdots (G/G^{\boxed{1}})_{pj}\cdots\rangle$
$\forall i,j \in \boxed{3}.\ G_{ij} = \min\langle\cdots (G/G^{\boxed{1}})_{iq}\cdots (G/G^{\boxed{1}})_{pj}\cdots\rangle$
$\forall i,j \in \boxed{4}.\ G_{ij} = \min\langle\cdots (G/G^{\boxed{1}})_{iq}\cdots (G/G^{\boxed{1}})_{pj}\cdots\rangle$

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

Following the same reasoning as in the iterative case, computing $\boxed{1}$ does not depend on any of the other computations. Richard applies the Stratify tactic, which encodes exactly this intuition: it separates an independent computation step as a separate loop. This is equivalent to rewriting the specification as in Figure 3(*ii*): 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 would result in a four-step computation as seen in Figure 4, from which Richard can obtain the program in Algorithm 2 (only the first two steps are shown; remaining steps are analogous). This already gives some performance gain, since the compututations $\boxed{2}$ and $\boxed{3}$ can now run in parallel. However, this is not what Richard wants; so he changes the development of this procedure, which he calls "A", to produce a recursive divide-and-conquer algorithm.

At this point, Richard notices that *Compute* $\boxed{1}$ is just a smaller version of the original *Compute*; so following Stratify $\boxed{1}$, he invokes Synth, which automatically synthesizes a recursive call $\mathrm{A}\big[G_{(0..\frac{n}{2})(0..\frac{m}{2})}\big]$ (presented using abstract index ranges as $A^{I_0 J_0}$).

The other steps require some further algebraic manipulation. Observe that the computation of $\boxed{2}$ is **not** equivalent to $\mathrm{A}\big[G_{(0..\frac{n}{2})(\frac{m}{2}..m)}\big]$, because when $\frac{m}{2} \le j < m$, the range $0 \le q < j$ leads to some accesses G_{iq} lying outside of $\boxed{2}$.

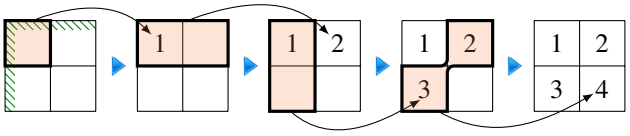

Figure 4. Stratified computation for Simplified Arbiter.

The array is initially empty except for the hatched area that is filled by *Initialize*. Shaded areas indicate the region that is read at each step. The arrows point at the quadrant that is written to.

Algorithm 2 Simplified Arbiter — Sliced and Stratified

procedure A$[G]$

  for $i = 1..\frac{n}{2}$ **do** ▷ *Compute* 1

    for $j = 1..\frac{m}{2}$ **do**

    $G_{ij} := \min\langle \min_{0\le q<j} G_{iq} + w^y_{qji}, \min_{0\le p<i} G_{pj} + w^x_{pij} \rangle$

  for $i = 1..\frac{n}{2}$ **do** ▷ *Compute* 2

    for $j = \frac{m}{2}{+}1..m$ **do**

    $G_{ij} := \min\langle \min_{0\le q<j} G_{iq} + w^y_{qji}, \min_{0\le p<i} G_{pj} + w^x_{pij} \rangle$

  ⋮ ▷ *Compute* 3

Algorithm 3 Simplified Arbiter — Sliced Even More

procedure A$[G]$

  A$\left[G_{(0..\frac{n}{2})(0..\frac{m}{2})}\right]$ ▷ *Compute* 1

  for $i = 1..\frac{n}{2}$ **do** ▷ *Compute* 2

    for $j = \frac{m}{2}{+}1..m$ **do**

    $G_{ij} := \min\langle \min_{0\le q<\frac{m}{2}} G_{iq} + w^y_{qji} \rangle$ ▷ (left)

  for $i = 1..\frac{n}{2}$ **do**

    for $j = \frac{m}{2}{+}1..m$ **do**

    $G_{ij} := \min\langle G_{ij}, \min_{\frac{m}{2}\le q<j} G_{iq} + w^y_{qji}, \min_{0\le p<i} G_{pj} + w^x_{pij} \rangle$ ▷ (right)

  ⋮

Richard addressed this problem by splitting the range of q using the Slice tactic again, effectively breaking the original $\min\langle\cdots\rangle$ into two: one where $0 \le q < \frac{m}{2}$, and one where $\frac{m}{2} \le q < j$. He uses Stratify to organize the loops in the program such that both loops write to the same area, namely 2, where the second loop reads data written by the first loop, as shown in Algorithm 3. It is important to notice that the first loop only reads from 1, while the second loop only reads from 2.

The computation of 2:(right) is now a true copy of A, with sub-matrix $G_{(0..\frac{n}{2})(\frac{m}{2}..m)}$ (with the small caveat, that A has to be changed to include the term G_{ij} in the $\min\langle\ \rangle$

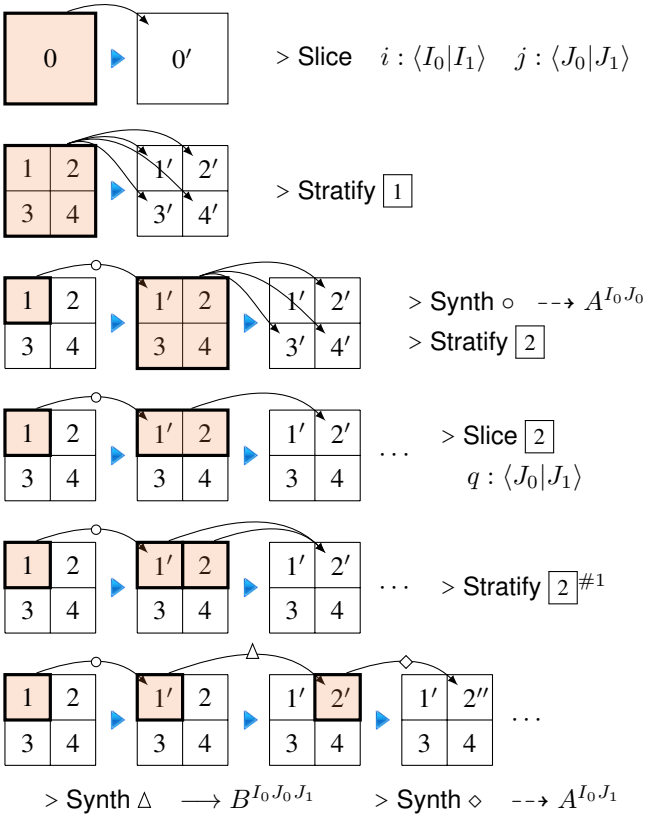

Figure 5. Overview of tactic semantics in Bellmania.

expression). Again, Bellmania synthesizes the sub-call and proves the equivalence. Running Synth on 2:(left) will reveal that it is a new computation, to which Richard gives the name "B".

After repeating the same reasoning steps to 3 and 4, Richard finally has the version in Algorithm 4. The base case (when G is small) is added automatically by the Bellmania compiler. The specific size bound needs to be tuned for performance.[1]

Algorithm 4 Simplified Arbiter — Recursive Version

procedure A$[G]$

  if G *is very small* **then** *run iterative version*

  else

    A$\left[G_{(0..\frac{n}{2})(0..\frac{m}{2})}\right]$ ▷ *Compute* 1

    B$\left[G_{(0..\frac{n}{2})(0..\frac{m}{2})}, G_{(0..\frac{n}{2})(\frac{m}{2}..m)}\right]$ ▷ *Compute* 2

    A$\left[G_{(0..\frac{n}{2})(\frac{m}{2}..m)}\right]$

    C$\left[G_{(0..\frac{n}{2})(0..\frac{m}{2})}, G_{(\frac{n}{2}..n)(0..\frac{m}{2})}\right]$ ▷ *Compute* 3

    A$\left[G_{(\frac{n}{2}..n)(0..\frac{m}{2})}\right]$

    B$\left[G_{(\frac{n}{2}..n)(0..\frac{m}{2})}, G_{(\frac{n}{2}..n)(\frac{m}{2}..m)}\right]$ ▷ *Compute* 4

    C$\left[G_{(0..\frac{n}{2})(\frac{m}{2}..m)}, G_{(\frac{n}{2}..n)(\frac{m}{2}..m)}\right]$

    A$\left[G_{(0..\frac{n}{2})(\frac{m}{2}..m)}\right]$

[1] The auto-tuning step is not implemented in the current version.

Richard must then use the same strategy to further break down and transform the computations of B and C, each into four recursive sub-computations, further improving the locality of the resulting algorithm. Eventually, through these transformations, he can succeed in breaking the computation of G into recursive sub-computations leading to a true divide-and-conquer algorithm.

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, the reasoning can be mechanized in Bellmania, which allows Richard and other algorithm designers to produce an implementation of this algorithm as well as a **machine-checked proof** of correctness through a series of high-level tactic application.

Overall, it took Richard only about 10 steps to construct Algorithm 4, and a total of 26 steps to construct all three steps of the Simplified Arbiter, comprising an implementation that is 10× faster than a parallel Algorithm 1 generated by a state-of-the-art parallelizing compiler. The user is greatly assisted by tactics like Synth, that carry 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.

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.

In the following sections, we describe the different components of Bellmania. The system utilizes *solver-aided tactics* to manipulate a given specification and generate provably correct pseudo-code; this approach is demonstrated by engineering specialized tactics for the domain of divide-and-conquer DP.

3. A Unified Language

We first set up a formal language that we will use to describe computations and reason about them. Bellmania uses the same language for specifications and for programs. Its core is the polymorphic λ-calculus, that is, simply typed λ-calculus with universally quantified type variables (also known as *System F*).

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 $(v_1 : \mathcal{T}_1) \cdots (v_n : \mathcal{T}_n) \mapsto e$.

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

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

3.1 Operators

The core language is augmented with the following intrinsic operators:

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

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

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

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

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

  where $\overline{x} = x_1 \cdots x_n$. This second form can be used to refer to quadrants of an array; for example, in **??**, $\boxed{2} \equiv [G]_{J_0 \times J_1}$.

- 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 takes precedence over $/$, and the bodies of $\mapsto$ spans as far as possible.

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.

- $= : \mathcal{T} \to \mathcal{T} \to \mathbb{B}$, always interpreted as equality.
- $+, - : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$, polymorphic binary operators.
- $< : \mathcal{T} \to \mathcal{T} \to \mathbb{B}$, a polymorphic order relation.
- $cons : \mathcal{T} \to (\mathbb{N} \to \mathcal{T}) \to (\mathbb{N} \to \mathcal{T}), nil : \mathbb{N} \to \mathcal{T}$, list constructors.
- $\min, \max, \Sigma : (\mathcal{T} \to \mathcal{S}) \to \mathcal{S}$, 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 expected to be finite.

3.3 Additional Notation

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

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

3.4 Types and Type Qualifiers

We extend the type system with predicate abstraction in the form of logically qualified data types (Liquid Types [18]). These are refinement types restricted via a set of abstraction predicates, called *qualifiers*, which are defined over the base types. Contradictory to the general use of refinement types, the purpose of these qualifiers is not to check a program for safety and reject ill-typed program, but rather to serve as annotations for tactics, to convey information to the solver for use in the proof, and later to help the compiler to properly schedule parallel computations.

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 almost always), it is enough to write P.
- $\{v : \mathcal{T} \mid P(v) \wedge Q(v)\}$, abbreviated as $\mathcal{T} \cap P \cap Q$, or just $P \cap Q$. This extends to any number of conjuncts of the same form.
- $(x : \mathcal{T}_2) \to \{v : \mathcal{T}_2 \mid R(x, v)\} \to \mathcal{T}_3$, 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).

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

Figure 6. Syntax of type qualifiers (droplets). k, l are positive integers that stand for dimension indexes.

The type refinement operators $\cap$ and $\times$ may be composed to create *droplets*, using the abstract syntax in Figure 6. Note that the language does not define tuple types; hence there is no distinction between curried and uncurried function types. 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.

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

Example. The inputs w^x, w^y to the Simplified Arbiter (Figure 1) can be typed using these droplets:

$$w^x : ((I \times I) \cap <) \to J \to \mathbb{R}$$
$$w^y : ((J \times J) \cap <) \to I \to \mathbb{R}$$

This states that $w\, p\, i\, j$ is only defined for $p < i$. It doesn't *force* it to be defined, as it is still a partial function. This property is in fact useful: we now have a mechanism for specifying that some schedules are impossible, by setting the respective $w\, p\, i\, j = \bot$!

Typing Rules

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

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.[2] 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, but has the same shape as that of the expected type, it is *coerced* to the required type by restricting the values to the desired proper subset.

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

Where :: is defined as follows:

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

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

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

We extend our abstract syntax with an explicit *cast operator* $t \ :: \ \mathcal{T}$ following this semantics. Notice that the second case of $[\,]_\square$ can be defined as syntactic sugar for ::, $[t]_\square = t :: \square \to \_$, where _ is a fresh type variable to be inferred.

Type Inference

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

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

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

Qualifiers are also inferred by essentially 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, $(f \ :: \ I \ \to \ (I \cap P))\,i$ 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:

[2] This is different from classical view, and holds in this case because we interpret functions as *mappings*.

- If $\mathcal{T}_1 = \{v : \mathcal{T} \mid \varphi_1\}$ and $\mathcal{T}_2 = \{v : \mathcal{T} \mid \varphi_2\}$,

$$\mathcal{T}_1 \sqcap \mathcal{T}_2 \ = \ \{v : \mathcal{T} \mid \varphi_1 \wedge \varphi_2\}$$

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

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

We then define the *type refinement* steps for terms. They are listed in Figure 7. These rules are applied continuously until a fixed point is reached. The resulting types are eventually converted back to droplet form (expressed via $\cap$ and $\times$); qualifiers that cannot be expressed in droplets are discarded.

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

Example. Let $I_0 \subseteq I$ be a unary predicate, and $0 : T$ a constant. The expression $f\,(i : I_0) \mapsto f\,i\,i \ / \ 0$ will induce the following type inferences:

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

$$\underbrace{\underbrace{\underbrace{\underbrace{(I_0 \to I \to T) \to I \to T}_{T}}_{T}}_{I_0 \to T}}_{I \to T}$$

Example

The specification of the Simplified Arbiter (Figure 1) will be written as

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

We use f_{xy} as a more readable alternative typography for $f\,x\,y$, where f is a function and x, y are its arguments.

Note that the ranges for $\min p$ and $\min q$ are implicit, from the types of w^x, w^y:

$$w^x_{pij} \neq \bot \Rightarrow p < i \quad \text{and} \quad w^y_{qji} \neq \bot \Rightarrow q < j$$

4. Tactics

We now define the method by which that 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

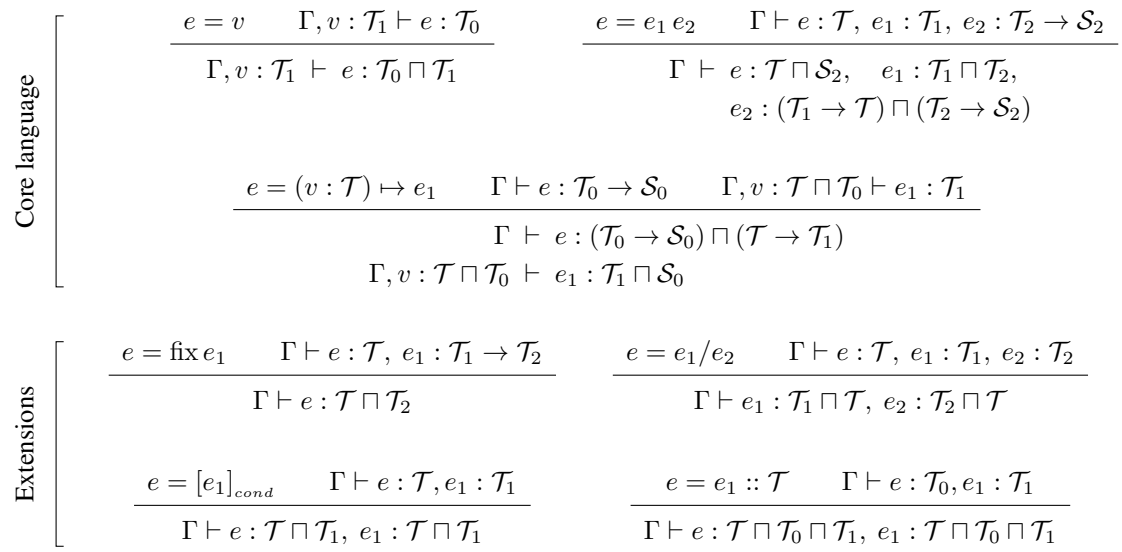

Figure 7. Type refinement rules, for inferring qualifiers in sub-expressions.

left-hand side is replaced by the **right-hand side**.[3] 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 paragraph. 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. Clearly the tactic itself can be used as its proof obligation, if it is easy enough to prove automatically; in such cases we write "***Obligations*:** tactic."

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

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

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

***Obligations*:** tactic.

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

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

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

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

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

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

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

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

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

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

Proof is given in Section 4.2.

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 subprograms. 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 $f_i \cong \text{fix}\, f$ —
  $h :: (\mathcal{T} \to \mathcal{Y}) = h :: (\mathcal{Y} \to \mathcal{Y}) = g :: (\mathcal{Y} \to \mathcal{T})$ 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.

We use $\cong$ to denote syntactic congruence up to β-reduction.

4.1 Synthesis-supported 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 CEGIS, a software synthesis technique implemented in the tool SKETCH [22]. 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.

While the size of the search space is not huge, typically in the order of ~50 alternatives, it is usually hard for the user to figure out which exact call should be made. Since Synth is used extensively throughout the develoment, This kind of automation greatly improves overall usability.

4.2 Soundness

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. 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,

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

Now instantiate h, f, and g, with $f \mathbin{\triangleright} g$, f, and g' from Equation (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 is included in the 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.

5. Automated Proofs

This section describes the encoding of proof obligations in (many-sorted) first-order logic, and the ways by 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 interchangebly, 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 operator $@ : F_{\mathcal{T}\to\mathcal{S}} \to \mathcal{T} \to \mathcal{S}$, and the *extensionality axiom* —

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

And for each such function symbol $f^k : \mathcal{T} \to \mathcal{S}$ used as argument, 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 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 (1) figures out directions of loops for fixed point computations by inferring the dependency constraints [14] from the available type information using an SMT solver, (2) uses type information to parallelize non-conflicting function calls and loops inside a function, (3) lifts conditionals via simple interval analysis to optimize the loop bounds and performs some traditional compiler transformations to reduce the number of iterations and comparisons, and (4) identifies loops that read non-contiguous memory blocks and applies copy optimization [16] automatically to better utilize caches. Examples of generated code are included in the supplemental material with this submission.

$$x :: J \to \mathbb{R}$$
$$w :: (J \times J \times J) \to \mathbb{R}$$
$$E \;=\; \mathrm{fix}\,\theta\,i\,j \mapsto \big[x_{ij}\big]_{i+1=j} \Big/ \min k \mapsto \theta_{ik} + \theta_{kj} + w_{ikj}$$

Figure 8. Specifications for the Parenthesis problem.

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 [24], and a few others.

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}.

Parenthesis problem. Compute an optimal placements of parenthesis in a long chain of multiplication, e.g. of matrices, where the input is are cost functions x_i for accessing the i-th element and w_{ikj} for multiplying elements $[i, k)$ by elements $[k, j)$. The corresponding recurrence is shown in Figure 8.

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.

The tactic application engine is implemented in Scala. We implemeted a prototype IDE using HTML5 and AngularJS, which communicates with the engine by sending and receiving program terms serialized as JSON. Our system supports using either Z3 or CVC4 as the back-end SMT solver for discharging proof obligations required for soundness proofs. Synthesis of recursive calls is done by translating the program to Sketch, which employs CEGIS to find the correct assignment to parameters. To argue for the feasibility of our

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

Table 1. Performance of different C++ implementations

	#	Verification			Synthesis
		Slice	Stratify	Synth	Sketch
Gap	3	1.5	6.4	0.2	8
Paren	3	0.8	16.5	0.8	11.2
Accordion	4	0.6	3.8	0.4	6.1
LCS	1	0.2	1.5	0.4	3.1
Knapsack	2	0.3	1.6	0.4	4.6
Bitonic	3	0.7	1.9	0.7	6.4

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

system, we include solver running time for the verification of the three most used tactics, as well as time required for Sketch synthesis, in Table 2. We consider an average delay of ~10 seconds to be reasonable, even for an interactive environment such as Bellmania.

The compiler for programs in Bellmania is implemented in Python and generates C++ code containing Intel Cilk constructs for parallelization. Table 1 shows performance improvement for our auto-generated implementation (AUTO) on the state-of-the-art optimized parallel loop implementation (LOOPDP) from [24]. It also compares AUTO with manually optimized recursive implementations CO_Opt and COZ for the three problems from [24]. Our compiler automatically does *copy optimization* as done in CO_Opt and COZ. COZ also incorporates a low-level optimization of using Z-order layout of the array, which is out of scope for this paper. N is the problem size and B is the base case size for using loops instead of recursion. It can be seen from the table that our implementation performs close to the manually optimized code and can be optimized further by hand. Figure 9 depicts the performance of these implementations on one sample instance as a function of problem size, and shows the scalability of our technique.

8. Related Work

Classical work by Smith *et al.* [19] presents rule-based transformation, stringing it tightly with program verification. This lay the foundation for semi-automatic programming [2, 4, 20]. More recently, a similar approach was introduced into Leon [15], leveraging deductive tools as a way to boost CEGIS, thereby covering more programs. Bellmania takes a

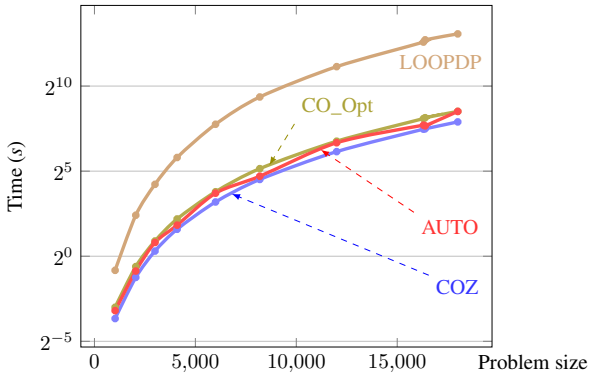

Figure 9. Performance comparison for parallelized implementations for Gap problem on 6-core Intel i7 CPU

dual approach, where automated techniques based on SMT are leveraged to support and improve deductive synthesis.

Fiat [10] is another recent system that admits stepwise transformation of specifications into programs via a refinement calculus. While Bellmania offloads proofs to automated solvers, Fiat formalizes refinements using the Coq proof assistant. The user is then responsible of proving the correctness of the derivation using a library of symbolic proof tactics.

Broadly speaking, the Bellmania system could have been implemented as a library on top of a framework such as Coq or Why3 [12] using binding to SMT solvers provided by these frameworks. The decision not to do so was merely a design choice.

Polyhedral compilers offer some optimizations for the same domain of problem via tiling [3, 17]. While showing significant speedups, these compilers cannot produce divide-and-conquer optimizations, which were proved to be more effective by [24].

9. Conclusion

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

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

References

[1] R. E. Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003. ISBN 0486428095.

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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. ISBN 0-521-62971-3.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010. ISBN 978-1-4244-7559-9.

[18] 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.

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

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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. More Tactics

Shrink

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

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

***Obligations*:** tactic.

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

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

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 $x : \mathbb{N}$)
- $\min\left([f]_{J_0} \,/\, [f]_{J_1}\right) \;=\; \min\langle \min[f]_{J_0}\,,\; \min[f]_{J_1}\rangle$

***Obligations*:** tactic.

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*:** tactic.

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.

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

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.[4]

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

Let Insertion [reduce]

$$e[\mathrm{reduce}\langle \overline{a}, \overline{b}\rangle] \;=\; (\overline{x} \mapsto \mathrm{reduce}\langle \overline{a}\rangle) \mathbin{»} z \mapsto e[\mathrm{reduce}\langle z\,\overline{x}, \overline{b}\rangle]$$

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

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

Padding

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

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*:** tactic.

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.

B. More Proofs

We prove the corollary from Section 4.2.

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 $(\mathrm{fix}\, f) :: \mathcal{Y} = \mathrm{fix}(f :: \mathcal{Y} \to \mathcal{Y})$.

Proof.

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

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

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

[4] 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 $(\overline{x} \mapsto \mathrm{reduce}\langle \overline{b}\rangle) \mathbin{»} z \mapsto e[\mathrm{reduce}\langle \overline{a}, z\,\overline{x}, \overline{c}\rangle]$.

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,

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

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

C. Example Development

The following example shows in more detail the first phase of the development for the Simplified Arbiter test case, to give the reader a flavor of the technique and the way the system does reasoning.

In Bellmania language, the Simplified Arbiter example is specified by a base case Ψ corresponding to *Initialize*, and a computation part corresponding to *Compute*.

$$\Psi \;=\; \operatorname{fix}\Big(\theta\, i\, j \mapsto [0]_{i=j=0} \;/\; [w^y_{0j0}]_{i=0} \;/\; [w^x_{0i0}]_{j=0}\Big) \tag{C.1}$$

$$\begin{aligned} A^{IJ} \;=\; \psi \mapsto \operatorname{fix} \theta \underset{(I)}{i}\ \underset{(J)}{j} \mapsto \min \langle\ & \psi_{ij} \\ & \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ & \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji}\ \rangle \end{aligned} \tag{C.2}$$

Vertical typeset was used to save some horizontal space, but $\underset{(\mathcal{T})}{v}$ should be read as just $v : \mathcal{T}$.

After Richard applies Slice, he gets the four quadrants $I_0 \times J_0$, $I_0 \times J_1$, $I_1 \times J_0$, $I_1 \times J_1$ (Figure 2). The system defined unary qualifiers with the axioms:

$$\begin{array}{ll} \forall i{:}I.\ I_0(i) \vee I_1(i) & \forall i_0{:}I_0,\ i_1{:}I_1.\ i_0 < i_1 \\ \forall j{:}J.\ J_0(j) \vee J_1(j) & \forall j_0{:}J_0,\ j_1{:}J_1.\ j_0 < j_1 \end{array}$$

The program is just about to grow quite large; to make such terms easy to read and refer to, we provide boxed letters as labels for sub-terms, using them as abbreviations where they occur in the larger 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.

Slice

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

$X_1 \;=\; \_ \times I_0 \times J_0 \quad X_2 \;=\; \_ \times I_0 \times J_1$

$X_3 \;=\; \_ \times I_1 \times J_0 \quad X_4 \;=\; \_ \times I_1 \times J_1$

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

$$\begin{aligned} A^{IJ} \;=\; \psi \mapsto \operatorname{fix} & \begin{array}{c|c} \boxed{\mathrm{A}} & \boxed{\mathrm{B}} \\ \hline \boxed{\mathrm{C}} & \boxed{\mathrm{D}} \end{array} \\ \boxed{\mathrm{A}} \;=\; \theta \underset{(I_0)}{i}\ \underset{(J_0)}{j} \mapsto \min \langle\ & \psi_{ij} \\ & \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ & \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji}\ \rangle \\ \boxed{\mathrm{B}} \;=\; \theta \underset{(I_0)}{i}\ \underset{(J_1)}{j} \mapsto \min \langle\ & \psi_{ij} \\ & \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ & \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji}\ \rangle \\ \boxed{\mathrm{C}} \;=\; \theta \underset{(I_1)}{i}\ \underset{(J_0)}{j} \mapsto \min \langle\ & \psi_{ij} \\ & \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ & \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji}\ \rangle \\ \boxed{\mathrm{D}} \;=\; \theta \underset{(I_1)}{i}\ \underset{(J_1)}{j} \mapsto \min \langle\ & \psi_{ij} \\ & \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ & \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji}\ \rangle \end{aligned} \tag{C.3}$$

Let

$e[\Box] \;=\; \begin{array}{c|c} \Box & \boxed{\mathrm{B}} \\ \hline \boxed{\mathrm{C}} & \boxed{\mathrm{D}} \end{array} \qquad t \;=\; \boxed{\mathrm{A}}$

$$A^{IJ} \;=\; \psi \mapsto \operatorname{fix}\left(\boxed{\mathrm{A}} \gg z \mapsto \begin{array}{c|c} z & \boxed{\mathrm{B}} \\ \hline \boxed{\mathrm{C}} & \boxed{\mathrm{D}} \end{array}\right) \tag{C.4}$$

Stratify[with Padding]

$f \;=\; \begin{array}{c|c} \boxed{\mathrm{A}} & \dot\psi \\ \hline \dot\psi & \dot\psi \end{array} \qquad$ (*recall that* $\dot\psi = \theta \mapsto \psi$)

$g \;=\; z \mapsto \begin{array}{c|c} z & \boxed{\mathrm{B}} \\ \hline \boxed{\mathrm{C}} & \boxed{\mathrm{D}} \end{array} \qquad \psi = \psi$

$$A^{IJ} \;=\; \psi \mapsto \operatorname{fix} \begin{array}{c|c} \boxed{\mathrm{A}} & \dot\psi \\ \hline \dot\psi & \dot\psi \end{array} \gg \psi \mapsto \operatorname{fix} \begin{array}{c|c} \dot\psi & \boxed{\mathrm{B}} \\ \hline \boxed{\mathrm{C}} & \boxed{\mathrm{D}} \end{array} \tag{C.5}$$

Notice that an existing variable ψ is reused, rebinding any occurrences within $\boxed{\mathrm{B}}$, $\boxed{\mathrm{C}}$, $\boxed{\mathrm{D}}$. This effect is useful, as it limits the context of the expression: the inner ψ shadows the outer ψ, meaning $\boxed{\mathrm{B}}$, $\boxed{\mathrm{C}}$, $\boxed{\mathrm{D}}$ do not need to access the data that was input to $\boxed{\mathrm{A}}$, only its output.

The sequence Let, Stratify[with Padding] is now applied in the same manner to $\boxed{\mathrm{B}}$ and $\boxed{\mathrm{C}}$ (see Figure 5). We do not list the applications as they are analogous to the previous ones.

$$A^{IJ} = \psi \mapsto \mathrm{fix}\,\frac{\boxed{\mathrm{A}} \mid \dot\psi}{\dot\psi \mid \dot\psi} \gg \psi \mapsto \mathrm{fix}\,\frac{\dot\psi \mid \boxed{\mathrm{B}}}{\dot\psi \mid \dot\psi} \gg \psi \mapsto \mathrm{fix}\,\frac{\dot\psi \mid \dot\psi}{\boxed{\mathrm{C}} \mid \dot\psi} \gg \psi \mapsto \mathrm{fix}\,\frac{\dot\psi \mid \dot\psi}{\dot\psi \mid \boxed{\mathrm{D}}} \tag{C.6}$$

Synth

$h_1 = \boxed{\mathrm{A}}$
$h_{2,3,4} = \dot\psi$
$f_1 = \theta \underset{(I_0)}{i} \underset{(J_0)}{j} \mapsto \min \langle\, \psi_{ij}, \min \underset{(I_0)}{p} \mapsto \theta_{pj} + w_{pij}, \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji} \,\rangle$
$f_{2,3,4} = \dot\psi$

$$A^{IJ} = \psi \mapsto \frac{A_\psi^{I_0 J_0} \mid \psi}{\psi \mid \psi} \gg \psi \mapsto \mathrm{fix}\,\frac{\dot\psi \mid \boxed{\mathrm{B}}}{\dot\psi \mid \dot\psi} \gg \psi \mapsto \mathrm{fix}\,\frac{\dot\psi \mid \dot\psi}{\boxed{\mathrm{C}} \mid \dot\psi} \gg \psi \mapsto \mathrm{fix}\,\frac{\dot\psi \mid \dot\psi}{\dot\psi \mid \boxed{\mathrm{D}}} \tag{C.7}$$

We note that $\mathrm{fix}\, f_1 = A_\psi^{I_0 J_0}$ are identical (up to β-reduction), which is the whole reason f_1 was chosen. Also, we took the liberty to simplify $\mathrm{fix}\,\dot\psi$ into ψ — although this is not necessary — just to display a shorter term.

The next few tactics will focus on the subterm $\boxed{\mathrm{B}}$ from (C.3).

$$\boxed{\mathrm{B}} = \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min \langle\, \psi_{ij}, \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji} \,\rangle \tag{C.8}$$

Slice

$f = \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji}$
$X_1 = J_0 \to \_ \qquad X_2 = J_1 \to \_$

$$\boxed{\mathrm{B}} = \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min \langle\, \psi_{ij}, \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \min \big((\underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}) \,/\, (\underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}) \big) \,\rangle \tag{C.9}$$

For the intuition behind this, see the top-right part of Figure 10. The colors represent cell ranges that will be read by different sub-routines (presumably running on different cores). The range of q is split into the part that lies within $\boxed{1}$ ($q \in J_0$) and the one that lies within $\boxed{2}$ ($q \in J_1$). The same reasoning is applied to the other quadrants.

Distributivity

$e[\Box] = \min \Box$
$t_1 = \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}$
$t_2 = \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}$

Associativity

$\mathrm{reduce} = \min$
$\overline{x}_1 = \psi_{ij}$
$\overline{x}_2 = \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}$
$\overline{x}_3 = \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}\,,\ \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}$

$$\boxed{\mathrm{B}} = \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min \langle\, \psi_{ij}, \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}, \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji} \,\rangle \tag{C.10}$$

Let[reduce]

$e[\Box] = \dfrac{\dot\psi \mid \theta\, i\, j \mapsto \Box}{\dot\psi \mid \dot\psi}$
$\overline{a} = \psi_{ij},\ \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}$
$\overline{b} = \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}$

$$\frac{\dot\psi \mid \boxed{\mathrm{B}}}{\dot\psi \mid \dot\psi} = \mathrm{fix}\left(\boxed{\mathrm{E}} \gg z \mapsto \frac{\dot\psi \mid \boxed{\mathrm{F}}}{\dot\psi \mid \dot\psi} \right)$$
$$\boxed{\mathrm{E}} = \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min\langle \psi_{ij}, \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji} \rangle$$
$$\boxed{\mathrm{F}} = \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min\langle z_{\theta ij}, \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji} \rangle \tag{C.11}$$

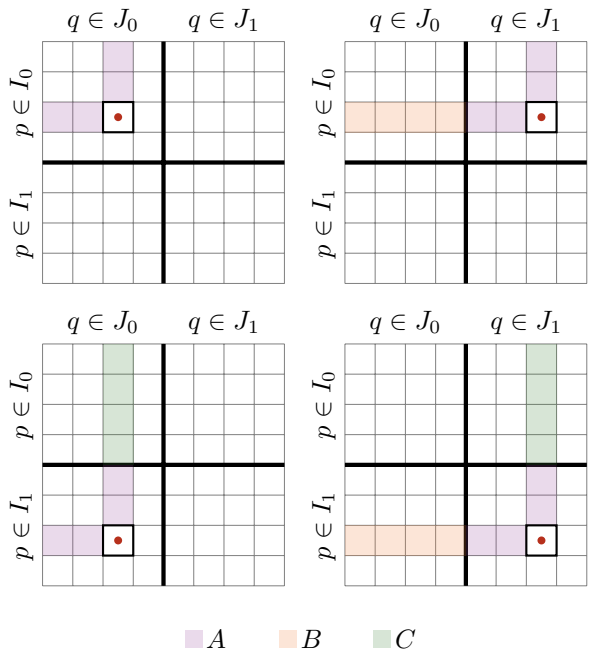

Figure 10. The strategy for applications of Slice in the case study.

Stratify[with Padding]

$$f = \begin{array}{c|c} \dot\psi & \boxed{\mathrm{E}} \\ \hline \dot\psi & \dot\psi \end{array}$$

$$g = z \mapsto \begin{array}{c|c} \dot\psi & \boxed{\mathrm{F}} \\ \hline \dot\psi & \dot\psi \end{array} \qquad \psi = \psi$$

$$\mathrm{fix}\begin{array}{c|c} \dot\psi & \boxed{\mathrm{B}} \\ \hline \dot\psi & \dot\psi \end{array} = \mathrm{fix}\begin{array}{c|c} \dot\psi & \boxed{\mathrm{E}} \\ \hline \dot\psi & \dot\psi \end{array} \gg \psi \mapsto \mathrm{fix}\begin{array}{c|c} \dot\psi & \boxed{\mathrm{F}} \\ \hline \dot\psi & \dot\psi \end{array}$$

$$\begin{aligned} \boxed{\mathrm{E}} &= \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min\langle \psi_{ij}, \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji} \rangle \\ \boxed{\mathrm{F}} &= \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min\langle \psi_{ij}, \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji} \rangle \end{aligned} \tag{C.12}$$

Define

$$\begin{aligned} B^{I J_0 J_1} = \big(\psi \mapsto \mathrm{fix}\, \theta \underset{(I)}{i} \underset{(J)}{j} \mapsto \min\langle \psi_{ij}, \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji} \rangle\big) \\ :: \big((I \times J_0) \to \mathbb{R}\big) \to \big((I \times J_1) \to \mathbb{R}\big) \end{aligned} \tag{C.13}$$

Synth

$$\begin{aligned} h_2 &= \boxed{\mathrm{E}} \\ h_{1,3,4} &= \dot\psi \\ f_2 &= \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min\langle \psi_{ij}, \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji} \rangle \\ f_{1,3,4} &= \dot\psi \end{aligned}$$

Synth

$$\begin{aligned} h_2 &= \boxed{\mathrm{F}} \\ h_{1,3,4} &= \dot\psi \\ f_2 &= \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min\langle \psi_{ij}, \min \underset{(I_0)}{p} \mapsto \theta_{pj} + w_{pij}, \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji} \rangle \\ f_{1,3,4} &= \dot\psi \end{aligned}$$

$$\mathrm{fix}\begin{array}{c|c} \dot\psi & \boxed{\mathrm{B}} \\ \hline \dot\psi & \dot\psi \end{array} = \begin{array}{c|c} \psi & B_\psi^{I_0 J_0 J_1} \\ \hline \psi & \psi \end{array} \gg \psi \mapsto \begin{array}{c|c} \psi & A_\psi^{I_0 J_1} \\ \hline \psi & \psi \end{array} \tag{C.14}$$

In a similar manner, we will obtain the following:

$$\mathrm{fix}\begin{array}{c|c} \dot\psi & \dot\psi \\ \hline \boxed{\mathrm{C}} & \dot\psi \end{array} = \begin{array}{c|c} \psi & \psi \\ \hline C_\psi^{I_0 I_1 J_0} & \psi \end{array} \gg \psi \mapsto \begin{array}{c|c} \psi & \psi \\ \hline A_\psi^{I_1 J_0} & \psi \end{array} \tag{C.15}$$

$$\begin{aligned} C^{I_0 I_1 J} = \big(\psi \mapsto \mathrm{fix}\, \theta \underset{(I)}{i} \underset{(J)}{j} \mapsto \min\langle \psi_{ij}, \min \underset{(I_0)}{p} \mapsto \theta_{pj} + w_{pij} \rangle\big) \\ :: \big((I_0 \times J) \to \mathbb{R}\big) \to \big((I_1 \times J) \to \mathbb{R}\big) \end{aligned} \tag{C.16}$$

And —

$$\mathrm{fix}\begin{array}{c|c} \dot\psi & \dot\psi \\ \hline \dot\psi & \boxed{\mathrm{D}} \end{array} = \begin{array}{c|c} \psi & \psi \\ \hline \psi & B_\psi^{I_1 J_0 J_1} \end{array} \gg \psi \mapsto \begin{array}{c|c} \psi & \psi \\ \hline \psi & C_\psi^{I_0 I_1 J_1} \end{array} \gg \psi \mapsto \begin{array}{c|c} \psi & \psi \\ \hline \psi & A_\psi^{I_1 J_1} \end{array} \tag{C.17}$$

This gives the stratified version as shown in Figure 11. The read and write regions are already encoded in the types of A, B, C in (C.7), (C.14), (C.15), and (C.17).

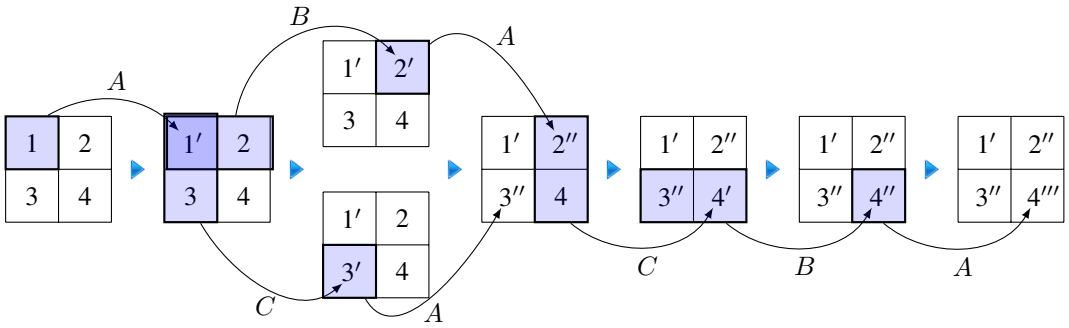

Figure 11. Fully divide-and-conquered version of A^{IJ} in the example development.