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

Double-blind submission

Abstract

We introduce *solver-aided tactics* as a mechanism for transforming computational terms, expressed in a specialized form of λ-calculus, intro equivalent terms in the interest of deriving better, more efficient implementations. By stringing chains of transformations, a programmer can start with a compact specification of an algorithm and develop it towards concrete executable code, while addressing performance issues involved.

Solver-aided tactics are unique in two ways: (a) They combine symbolic pattern-matching with SMT technology to ensure a correct transformation; this means that a lot of the burden of proof is moved from the user of the system to automated solvers. (b) They make use of type annotations within program terms to maintain and utilize properties of those computations, in order to guide transformations and provide enough context for the automated proofs. For this purpose, we employ predicate abstraction as part of the type system.

We present Bellmania, a framework that brings together novel techniques from both inductive and deductive synthesis disciplines. Bellmania comprises of a language for specifying dynamic programming algorithms as recurrences, and a calculus that facilitates gradual transformation of these specifications into efficient implementations. In particular, it provides tactics that formalize the Divide-and-Conquer technique to derive parallel implementations that optimally utilize memory caches.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program synthesis

Keywords Synthesis; Dynamic Programming; SMT; Automated Reasoning; Transformational

1. Introduction

Software synthesis techniques can be broadly classified into two categories: *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 techniques have been the focus of significant renewed interest thanks to two important developments: (a) the discovery of techniques that leverage SAT/SMT solvers to symbolically represent and search very large spaces of possible programs [12, 19, 22], and (b) the use of counterexample-guided inductive synthesis (CEGIS), which allows one to leverage inductive techniques to find programs that satisfy more general specifications as long as one has access to an oracle that can check whether a given candidate solution satisfies the specification [19]. 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 deductive synthesis based on *solver-aided tactics* that preserves the benefits of deductive synthesis techniques but reduces the burden on the user by relying heavily on the ability of SMT solvers to reason about the validity of transformations and lift the level of abstraction of deductive transformation rules. 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 algorithms 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 [20], 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 the SMT solver in the context of deductive synthesis: (a) the solver can directly check that a program term is equivalent to another term, thus merging branches and reducing repetitive manual labor, (b) The solver can prove validity of side conditions that ensure the soundness of each individual transformation, (c) the tactics can leverage infomation 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 don not have to compute a priori the set of all conditions under which a transformaiton can be applied. Even a transformation that is only sometimes correct can be useful as long as we check the validity of every application.

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 transform a class of recurrence specifications, written in a simple functional language, into equivalent divide-and-conquer

programs, that admit parallel cache-local implementations, in a principled, systematic manner.

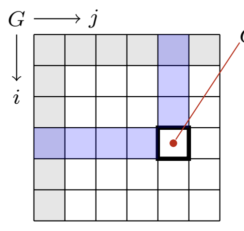
- We prove that these tactics are semantics-preserving, assuming some side conditions are met at the point when the tactic is applied.
- We show that the side conditions can be effectively translated into first-order closed formulas, and verified automatically by SMT solvers.
- We demonstrates the first system capable of generating provably correct implementations of divide-and-conquer implementations from a high-level description of the algorithm.

2. Divide-and-Conquer DP

Most readers are likely familiar with the Dynamic Programming (DP) technique of Richard Bellman [2] to construct an optimal solution to a problem by combining together optimal solutions to many overlapping sub-problems. The key to DP is to exploit the overlap 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., [21]). 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 [21]. 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.

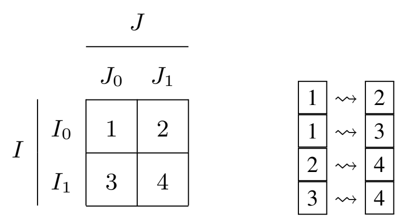
Before diving into the details of how solver-aided tactics can be used to derive divide-and-conquer dynamic programming, it is important to explain how an algorithms expert—we will call him Richard—would go about designing such an implementation by hand. As a motivating example, we consider the Simplified Arbiter problem. Two processes x and y are scheduled to run $|x|$ and $|y|$ time slots, respectively. Execution starts at $t = 0$, and the length of each time slot is one time unit. The cost for scheduling the slots $[a..b)$ of x at $t = a+c$ is given by w_{abc}, and the cost for schedulting the slots $[a..b)$ of y at same $t = a + c$ is given by w'_{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'_{0j0}, and similarly when j is zero, the cost is w_{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 the cost is $G_{pj} + w_{pij}$; or it ends with an allocation to y, where slots $[q..j)$ of y were scheduled at $t = i + q$, and the cost is $G_{iq} + w'_{qji}$. The minimum over all respective $p < i$ and $q < j$ is taken. Eventually, the optimal cost of the entire schedule is given by $G_{|x||y|}$.



$$G_{ij} = \begin{cases} 0 & i = j = 0 \\ w'_{0j0} & i = 0, j > 0 \\ w_{0i0} & i > 0, j = 0 \\ \min\langle \min\limits_{0\le q<j} G_{iq} + w'_{qji}, \\ \qquad \min\limits_{0\le p<i} G_{pj} + w_{pij} \rangle & i, j > 0 \end{cases}$$

Figure 1. Recurrence equation and cell-level dependencies.



$\boxed{1} \rightsquigarrow \boxed{2}$
$\boxed{1} \rightsquigarrow \boxed{3}$
$\boxed{2} \rightsquigarrow \boxed{4}$
$\boxed{3} \rightsquigarrow \boxed{4}$

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

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.

Algorithm 1 Iterative Simplified Arbiter

```
$G_{00} := 0$
**for** $j = 1..|y|$ **do** $G_{0j} := w'_{0j0}$
**for** $i = 1..|x|$ **do**
    $G_{i0} := w_{0i0}$
    **for** $j = 1..|y|$ **do**
        $G_{ij} := \min\langle \min_{0\le q<j} G_{iq} + w'_{qji}, \min_{0\le p<i} G_{pj} + w_{pij} \rangle$
    **end for**
**end for**
```

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]). The DP table is partitioned into regions, and each region is expressed as a sub-problem to be solved.

For the running example Richard takes the two-dimensional array G and partitions it into quadrants, as illustrated in Figure 2. He then applies the same reasoning as in the iterative case, concluding that the computations of $\boxed{2}$ and $\boxed{3}$ depend on $\boxed{1}$, and that the computation of $\boxed{4}$ depends on $\boxed{2}$ and $\boxed{3}$.

He *stratifies* the computations on these quadrants into the following four steps:

1: Compute $\boxed{1}$ (using only input data w, w').
2: Compute $\boxed{2}$ using data from $\boxed{1}$.
3: Compute $\boxed{3}$ using data from $\boxed{1}$.
4: Compute $\boxed{4}$ using data from $\boxed{2}$ and $\boxed{3}$.

Each step depends only on a subset of the steps that came before it, as illustrated by Figure 3. However, this is not yet a divide-and-conquer algorithm: of the four steps, only step $\boxed{1}$ is an instance of the original problem; all the other steps look somewhat different from the original problem and lack significant locality. With some algebraic manipulation, however, it is possible to define each of the four steps above recursively, leading to a true divide-and-conquer algorithm with very high locality and significantly improved performance relative to the iterative algorithm.

To illustrate how Richard can perform this transformation, we first introduce a small amount of notation. We define I and J to be the index sets for the rows and columns, respectively. We then define partitions $I = I_0 \cup I_1$ and $J = J_0 \cup J_1$ as in Figure 2. G is now parameterized on those index sets;

$$G^{\widehat{I}\widehat{J}}_{(i:\widehat{I})\,(j:\widehat{J})} = \begin{cases} 0 & i=j=0 \\ w_{0j0} & i=0, j>0 \\ w'_{0i0} & i>0, j=0 \\ \min\langle \min\limits_{q\in\widehat{J}\cap[0,j)} G^{\widehat{I}\widehat{J}}_{iq} + w'_{qji}, & \\ \qquad \min\limits_{p\in\widehat{I}\cap[0,i)} G^{\widehat{I}\widehat{J}}_{pj} + w_{pij} \rangle & i,j>0 \end{cases} \tag{2.1}$$

The computation of $\boxed{1}$ corresponds to the value of G^{IJ} within the sub-domain $I_0 \times J_0$, and it also corresponds exactly to $G^{I_0J_0}$. This is due to the fact that for $j \in J_0$, $J \cap [0,j) = J_0 \cap [0,j)$, and for $i \in I_0$, $I \cap [0,i) = I_0 \cap [0,i)$, so for those i and j, $G^{I_0J_0} = G^{IJ}$.

On the other hand, the computation of $\boxed{2}$ is **not** equivalent to $G^{I_0J_1}$, because the range when $j \in J_1$, $J \cap [0,j) \neq J_1 \cap [0,j)$. The problem stems from the fact that some of the recursive references to G_{iq} are outside of $\boxed{2}$.

Richard addressed this problem by splitting the range J, adding yet range another parameter to G.

$$G^{\widehat{I}\widehat{J}\widehat{J}'}_{(i:\widehat{I})\,(j:\widehat{J})} = \begin{cases} 0 & i=j=0 \\ w_{0j0} & i=0, j>0 \\ w'_{0i0} & i>0, j=0 \\ \min\langle \min\limits_{q\in\widehat{J}'} G^{\widehat{I}\widehat{J}'\widehat{J}'}_{iq} + w'_{qji}, & \\ \qquad \min\limits_{q\in\widehat{J}\cap[0,j)} G^{\widehat{I}\widehat{J}\widehat{J}'}_{iq} + w'_{qji}, & i,j>0 \\ \qquad \min\limits_{p\in\widehat{I}\cap[0,i)} G^{\widehat{I}\widehat{J}\widehat{J}'}_{pj} + w_{pij} \rangle & \end{cases} \tag{2.2}$$

The computation of $\boxed{2}$ is now a copy of $G^{I_0J_1J_0}$. G can be further generalized in the following way:

$$H^{\widehat{I}\widehat{J}\widehat{J}'}_{(i:\widehat{I})\,(j:\widehat{J})\,\psi} = \begin{cases} 0 & i=j=0 \\ w_{0j0} & i=0, j>0 \\ w'_{0i0} & i>0, j=0 \\ \min\langle \psi_{ij}, & \\ \qquad \min\limits_{q\in\widehat{J}\cap[0,j)} G^{\widehat{I}\widehat{J}\widehat{J}'}_{iq} + w'_{qji}, & i,j>0 \\ \qquad \min\limits_{p\in\widehat{I}\cap[0,i)} G^{\widehat{I}\widehat{J}\widehat{J}'}_{pj} + w_{pij} \rangle & \end{cases} \tag{2.3}$$

Richard immediately sees that both G and $\boxed{1}$ can be expressed in terms of H^{IJJ}, $H^{I_0J_0J_0}$ by making $\psi_{ij} = \infty$, but more importantly, that $\boxed{2}$ can now also be expressed in terms of $H^{I_0J_1J_0}$ by making $\psi_{ij} = \min\limits_{q\in J_0} G^{I_0J_0J_0}_{iq} + w'_{qji}$. Moreover, he replaces the calls to G inside H with recursive calls to H itself, so that the extra parameter

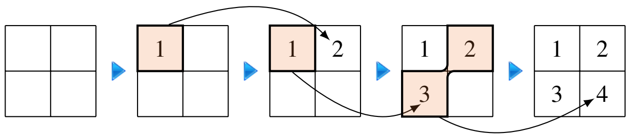


Figure 3. Stratified computation for Simplified Arbiter. The array is initially empty; shaded areas indicate the region that is read at each step.

$\widehat{J}'$ is no longer needed:

$$H^{\widehat{I}\widehat{J}}_{(i:\widehat{I})\,(j:\widehat{J})\,\psi} = \begin{cases} 0 & i=j=0 \\ w_{0j0} & i=0, j>0 \\ w'_{0i0} & i>0, j=0 \\ \min\langle \psi_{ij}, & \\ \qquad \min\limits_{q\in\widehat{J}\cap[0,j)} H^{\widehat{I}\widehat{J}}_{iq\psi} + w'_{qji}, & i,j>0 \\ \qquad \min\limits_{p\in\widehat{I}\cap[0,i)} H^{\widehat{I}\widehat{J}}_{pj\psi} + w_{pij} \rangle & \end{cases} \tag{2.4}$$

Richard must then use the same strategy to further generalize $H^{\widehat{I}\widehat{J}}$ so that it can also be used for $\boxed{3}$ and $\boxed{4}$. He will also transform the sub-computation of ψ into four recursive sub-computations, further improving the locality of the resulting algorithm. Eventually, through this kind of transformations, he can succeed in breaking the computation of G into recursive sub-computations leading to a true divide-and-conquer algorithm. Unfortunately, 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, these are the steps that are mechanized in Bellmania, allowing an algorithm designer like Richard to produce a provably correct implementation of this algorithm through a series of high-level commands. 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, but those steps can be performed by more traditional compiler optimization techniques that are outside the scope of this paper.

In the following sections, we describe the different components of Bellmania. The system utilizes *solver-aided tactics* to 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. 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 evaluate to 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 is augmented with the following intrinsic operators:

- A fixed point operator fix f, with the denotational semantics

$$[\![\text{fix}\, f]\!] \;=\; \sigma x.\, ([\![f]\!]\, x = x)$$

  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 slash operator $/$,

$$\text{For base type } \mathcal{S}, x, y : \mathcal{S} \quad x/y = \begin{cases} x & \text{if } x \neq \bot \\ y & \text{if } x = \bot \end{cases}$$

$$\text{For } f, g : \mathcal{T}_1 \to \mathcal{T}_2 \qquad f/g = (v : \mathcal{T}_1) \mapsto (f\,v)/(g\,v)$$

3.2 Types and Type Qualifiers

We extend the type system with predicate abstraction in the form of logically qualified data types (Liquid Types [16]). 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.

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 R 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 current one).

The type refinement operators $\cap$ and $\times$ may be composed to create *droplets*, using the abstract syntax in Figure 4. 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.

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

Figure 4. Syntax of type qualifiers (droplets). k, l are positive integer dimension indexes.

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, w' to the Simplified Arbiter (Figure 1) can be typed using these droplets:

$$\begin{array}{l} w : ((I \times I) \cap <) \to J \to \mathbb{R} \\ w' : ((J \times J) \cap <) \to I \to \mathbb{R} \end{array}$$

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!

Typing Rules

As mentioned earlier, annotations are ignored when typechecking a term. This gives a simple characteristic 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.[1] 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.

Type Inference

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

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

constants with the following types:

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

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:

- 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 e a sub-term, listed in Figure 5. 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:

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

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.
- $t\big|_{\Box}$ abbreviates $t :: \Box \to \_$, where "_" is a fresh type variable to be inferred. As a special case, $\big|$ can also be applied to scalar (non-arrow) expressions by supplying a Boolean condition, interpreted as a (nullary) qualifier:
$$x\big|_C \;=\; x :: (\_ \cap C) \;=\; \begin{cases} x & \text{if } C \\ \bot & \text{if } \neg C \end{cases}$$

3.4 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.
- $\min, \max, \Sigma \;:\; (\mathcal{T} \to \mathcal{S}) \to \mathcal{S}$, reduction (aggregation) operators on ordered/unordered collections. The collection is represented by a mapping $f : \mathcal{T} \to \mathcal{S}$, so that e.g.
$$[\![\min f]\!] = \min \big\{ [\![f]\!]\, v \;\big|\; v \in [\![\mathcal{T}]\!], [\![f]\!]\, v \neq \bot \big\}$$
The collections are expected to be finite.

Example

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

$$\begin{aligned} &w : ((I \times I) \cap <) \to J \to \mathbb{R} \\ &w' : ((J \times J) \cap <) \to I \to \mathbb{R} \\ &G \;=\; \text{fix}\,\theta\, i\, j \mapsto 0\big|_{i=0 \wedge j=0} \Big/ \, w'_{0j0}\big|_{i=0} \Big/ \, w_{0i0}\big|_{j=0} \Big/ \\ &\qquad\qquad \min \;\langle\; \min p \mapsto \theta_{pj} + w_{pij}, \\ &\qquad\qquad\qquad \min q \mapsto \theta_{iq} + w'_{qji} \;\rangle \end{aligned} \tag{3.1}$$

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

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

$$w_{pij} \neq \bot \to p < i \quad \text{and} \quad w_{qji} \neq \bot \to q < j$$

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 left-hand side is replaced by the right-hand side. 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

$$f \;=\; f\big|_{X_1} \Big/ \, f\big|_{X_2} \Big/ \, \cdots \, \Big/ \, f\big|_{X_r}$$

This tactic partitions a mapping into sub-regions. Each X_i may be a $\times$-expression 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).

Core language

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

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

Extensions

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

$$\frac{e = e_1 :: \mathcal{T} \qquad \Gamma \vdash e_1 : \mathcal{T}_1}{\Gamma \vdash e : \mathcal{T} \sqcap \mathcal{T}_1,\ e_1 : \mathcal{T} \sqcap \mathcal{T}_1}$$

Figure 5. Type refinement rules, for inferring qualifiers in sub-expressions.

Shrink

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

Used to extra-specify the type of a sub-term.

***Obligations*:** tactic.

For arrow-typed terms, this essentially requires to prove that f is undefined anyway for argument values outside the domain of $\mathcal{T}$, and that the defined values are in the range of $\mathcal{T}$.

Stratify

$$\mathrm{fix}(f \gg g) \ = \ \mathrm{fix}\, f \ \gg \ \psi \mapsto \mathrm{fix}(\dot{\psi} \gg g)$$

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

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

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

$$\begin{aligned} f\,(g'\,\zeta\,\theta) \ &= \ f\,\zeta \\ g'\,(f\,\theta)\,\theta \ &= \ h\,\theta \end{aligned} \tag{4.1}$$

Although the proof is not hard, we defer it to a later theorem.

Synth

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

This tactic is used to generate recursive calls to sub-programs. For $i = 1..r$, f_i is typically either h_i or a sub-term occurring earlier in the development.

***Obligations*:** Let $h = h_1 / \cdots / h_r$, let $\overline{\theta} = \theta_{1..r}$ be r fresh variables, and let $f = \theta_{1..r} \mapsto (f_1\,\theta_1) :: \mathcal{T}_1 / \cdots / (f_r\,\theta_r) :: \mathcal{T}_r$.

- $\mathcal{T}_{1..r}$ are disjoint mappings.
- **Either** $h\,(f\,\overline{\theta}) = f\,\overline{\theta}$
  or $h\,\theta = \theta \ \to \ (f_i\,\theta :: \mathcal{T}_i) = \theta :: \mathcal{T}_i$, $\theta :: \mathcal{T}_1 \,/\, \cdots \,/\, \theta :: \mathcal{T}_r = \theta$

(We give two alternatives, as the first is usually easier to prove, but may hold in less cases)

Example

For simplicity of the example, we assume that the input to the Simplified Arbiter problem satisfies triangle inequalities:

$$w_{pij} \le w_{pkj} + w_{kij} \qquad w'_{qji} \le w'_{qki} + w'_{kji} \tag{4.2}$$

for all (appropriately typed) $p < k < i,\ q < k < j$.

Starting from the specification in (3.1), Richard applies Synth to turn $\mathrm{fix}(\theta\,i\,j \ \mapsto \ \square / \min\langle\cdots\rangle)$ into $\mathrm{fix}\,\theta\,i\,j \ \mapsto \ \min\langle\square, \cdots\rangle$. While not absolutely necessary, we will see that it makes some expressions easier to handle later on.

Distributivity and Associativity (refer to Appendix A) are used to obtain the tactic parameter f_1; the proof is deferred until Synth is applied so that the prover can use the extra context.

$$\boxed{\begin{array}{l} \underline{\text{Synth}} \\ h_1 = \theta\,i\,j \mapsto 0|_{i=0 \wedge j=0} \,/\, w'_{0j0}|_{i=0} \,/\, w_{0i0}|_{j=0} \,/ \\ \qquad\qquad \min \langle\, \min p \mapsto \theta_{pj} + w_{pij}, \\ \qquad\qquad\qquad \min q \mapsto \theta_{iq} + w'_{qji} \,\rangle \\ f_1 = \theta\,i\,j \mapsto \min \langle\, 0|_{i=0 \wedge j=0} \,/\, w'_{0j0}|_{i=0} \,/\, w_{0i0}|_{j=0}, \\ \qquad\qquad\qquad \min p \mapsto \theta_{pj} + w_{pij}, \\ \qquad\qquad\qquad \min q \mapsto \theta_{iq} + w'_{qji} \,\rangle \end{array}}$$

$$\begin{aligned} G \ = \ \mathrm{fix}\,\theta\,i\,j \mapsto \min \langle\, & 0|_{i=0 \wedge j=0} \,/\, w'_{0j0}|_{i=0} \,/\, w_{0i0}|_{j=0}, \\ & \min p \mapsto \theta_{pj} + w_{pij}, \\ & \min q \mapsto \theta_{iq} + w'_{qji} \,\rangle \end{aligned} \tag{4.3}$$

He then applies Let Insertion, followed by Stratify, to separate the base case and obtain a more general form, similar to H of (2.3).

$$\boxed{\begin{array}{l}\underline{\text{Let}}\\ e[\square] \;=\; \theta\, i\, j \mapsto \min\langle\, \square, \\ \qquad\qquad\qquad\qquad \min p \mapsto \theta_{pj} + w_{pij}, \\ \qquad\qquad\qquad\qquad \min q \mapsto \theta_{iq} + w'_{qji} \,\rangle \\ t \;=\; 0|_{i=0\wedge j=0} \;/\; w'_{0j0}|_{i=0} \;/\; w_{0i0}|_{j=0}\end{array}}$$

$$\begin{aligned} G \;=\; \mathrm{fix}\,\big(\,&(\theta\, i\, j \mapsto 0|_{i=0\wedge j=0} \;/\; w'_{0j0}|_{i=0} \;/\; w_{0i0}|_{j=0}) \,»\\ &z\,\theta\, i\, j \mapsto \min\langle\, z_{\theta ij}, \\ &\qquad\qquad \min p \mapsto \theta_{pj} + w_{pij}, \\ &\qquad\qquad \min q \mapsto \theta_{iq} + w'_{qji} \,\rangle\,\big) \end{aligned} \tag{4.4}$$

$$\boxed{\begin{array}{l}\underline{\text{Stratify}}\\ f \;=\; \theta\, i\, j \mapsto 0|_{i=0\wedge j=0} \;/\; w'_{0j0}|_{i=0} \;/\; w_{0i0}|_{j=0} \\ g \;=\; z\,\theta\, i\, j \mapsto \min\langle z_{\theta ij}, \cdots\rangle\end{array}}$$

$$\begin{aligned} G \;=\; \mathrm{fix}\,\big(&\theta\, i\, j \mapsto 0|_{i=0\wedge j=0} \;/\; w'_{0j0}|_{i=0} \;/\; w_{0i0}|_{j=0}\big) \,»\\ &\psi \mapsto \mathrm{fix}\,\theta\, i\, j \mapsto \min\langle\, \psi_{ij} \\ &\qquad\qquad \min p \mapsto \theta_{pj} + w_{pij}, \\ &\qquad\qquad \min q \mapsto \theta_{iq} + w'_{qji} \,\rangle \end{aligned} \tag{4.5}$$

Setting the base case aside, let A^{IJ} be the second term, where the superscript parameterizes the types of i, j and p, q.

$$\begin{aligned} A^{IJ} \;=\; \psi \mapsto \mathrm{fix}\,\theta\, &\underset{(I)}{i}\; \underset{(J)}{j} \mapsto \min\langle\, \psi_{ij} \\ &\qquad \min \underset{(I)}{p} \;\mapsto \theta_{pj} + w_{pij}, \\ &\qquad \min \underset{(J)}{q} \;\mapsto \theta_{iq} + w'_{qji} \,\rangle \end{aligned} \tag{4.6}$$

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

This is exactly what Richard was after. A^{IJ} constitutes the first phase of the divide-and-conquer formalization.

4.1 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

$$\forall\theta,\zeta.\quad \begin{aligned} f\,(g\,\zeta\,\theta) \;&=\; f\,\zeta \\ g\,(f\,\theta)\,\theta \;&=\; h\,\theta \end{aligned}$$

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

$$\begin{aligned} f\,\zeta &= \zeta \\ g\,\zeta\,\theta &= \theta \end{aligned}$$

Then —

$$\begin{aligned} h\,\theta = g\,(f\,\theta)\,\theta &= g\,(f\,(g\,\zeta\,\theta))\,\theta = \\ &= g\,(f\,\zeta)\,\theta = \theta \end{aligned}$$

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

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

Now instantiate h, f, and g, with $f \,»\, g$, f, and g' from Section 4, and we obtain the equality in the tactic.

For Synth, (*i*) assume

$$\forall\overline{\theta}.\quad h\,(f\,\overline{\theta}) = f\,\overline{\theta}$$

and let $\theta = \theta_{1..r}$ such that $\theta_i = \mathrm{fix}\, f_i$. So $f_i\,\theta_i = \theta_i$. Let $\theta = \theta_1 :: \mathcal{T}_1/\cdots/\theta_r :: \mathcal{T}_r$.

$$\begin{aligned} f\,\overline{\theta} &= (f_1\,\theta_1) :: \mathcal{T}_1/\cdots/(f_r\,\theta_r) :: \mathcal{T}_r = \\ &= \theta_1 :: \mathcal{T}_1/\cdots/\theta_r :: \mathcal{T}_r = \theta \end{aligned}$$

$$h\,\theta = h\,(f\,\overline{\theta}) = f\,\overline{\theta} = \theta$$

Then $\theta = \mathrm{fix}\, h$;
We get $\mathrm{fix}\, h = (\mathrm{fix}\, f_1) :: \mathcal{T}_1/\cdots/(\mathrm{fix}\, f_r) :: \mathcal{T}_r$, as required.
(*ii*) assume

$$\forall\theta.\quad \begin{array}{l} h\,\theta = \theta \;\rightarrow\; \displaystyle\bigwedge_{i=1..r} (f_i\,\theta) :: \mathcal{T}_i = \theta :: \mathcal{T}_i \\ \theta = \theta :: \mathcal{T}_1 \;/\; \cdots \;/\; \theta :: \mathcal{T}_r \end{array}$$

and let $\theta = \mathrm{fix}\, h$. So $h\,\theta = \theta$, therefore $(f_i\,\theta) :: \mathcal{T}_i = \theta :: \mathcal{T}_i$. Since we assume that $\mathrm{fix}\, f_i$, we get by induction that $(\mathrm{fix}\, f_i) :: \mathcal{T}_i = \theta :: \mathcal{T}_i$. Hence,

$$(\mathrm{fix}\, f_1) :: \mathcal{T}_1/\cdots/(\mathrm{fix}\, f_r) :: \mathcal{T}_r = \theta :: \mathcal{T}_1/\cdots/\theta :: \mathcal{T}_r = \theta$$

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 is 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 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 $\forall\alpha : J.\ |g|(\alpha) \to P(\alpha)$.

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 $\mathcal{S} \to \mathcal{T}$.

When a function is being used as an argument in an application term (first-class functions), we take its arrow type $\mathcal{T} \to \mathcal{S}$ and create a corresponding *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* —

$$\forall \alpha\alpha'.\ \big(\forall \beta.\ @(\alpha,\beta) = @(\alpha',\beta)\big) \to \alpha = \alpha' \tag{5.1}$$

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

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

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

$$\forall \overline{a}.\ \big(|f|(\overline{\alpha}) \leftrightarrow |g|(\overline{\alpha})\big) \wedge \big(|f|(\overline{\alpha}) \to f^k(\overline{\alpha}) = g^k(\overline{\alpha})\big)$$

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 much easier to prove. To understand why this is a common case, consider (6.3) below: notice how g does not modify the quadrant from which f reads.

6. Case Studies

In addition to the somewhat artificial Simplified Arbiter, we tested our technique on two representative problems:

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

- Replacing x_i with y_j, at cost c_{ij}.
- Deleting $x_{p+1} \cdots x_q$, at cost w_{pq}.
- Inserting $y_{p+1} \cdots y_q$ in $\overline{x}$, at cost w'_{pq}.

The computation is given by the recurrence Figure 6.

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 7.

The next example shows more steps for the Simplified Arbiter test case, to give the reader a flavor of the technique.

Example

Further developing A^{IJ} (4.6) we apply Slice to get the four quadrants; I_0, I_1, J_0, J_1 (Figure 2) are defined as unary qualifiers with the axioms:

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

As a result, the term is going to grow quite large; to make such terms easy to read and refer to, we provide boxed letters as labels

$$\begin{aligned} w &:: ((J \times J) \cap <) \to \mathbb{R} \\ w' &:: ((K \times K) \cap <) \to \mathbb{R} \\ G &= \operatorname{fix} \theta\, i\, j \mapsto 0\big|_{i=0 \wedge j=0} \Big/ w'_{0j}\big|_{i=0} \Big/ w_{i0}\big|_{j=0} \Big/ \\ &\qquad \min \langle\, \theta_{(i-1)\,(j-1)} + c_{ij}, \\ &\qquad\quad \min p \mapsto \theta_{pj} + w_{pi}, \\ &\qquad\quad \min q \mapsto \theta_{iq} + w'_{qj} \,\rangle \end{aligned}$$

Figure 6. Specifications for the full version of the Gap DP problem.

$$\begin{aligned} x &:: J \to \mathbb{R} \\ w &:: (J \times J \times J) \to \mathbb{R} \\ E &= \operatorname{fix} \theta\, i\, j \mapsto x_{ij}\big|_{i+1=j} \Big/ \\ &\qquad \min k \mapsto \theta_{ik} + \theta_{kj} + w_{ikj} \end{aligned}$$

Figure 7. Specifications for the Parenthesis Assignment DP problem.

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.

<u>Slice</u>

$$\begin{array}{ll} f\ =\ \theta\, i\, j \mapsto \cdots & \\ X_1\ =\ \_ \times I_0 \times J_0 & X_2\ =\ \_ \times I_0 \times J_1 \\ X_3\ =\ \_ \times I_1 \times J_0 & X_4\ =\ \_ \times I_1 \times J_1 \end{array}$$

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

$$\begin{aligned} A^{IJ} &= \psi \mapsto \operatorname{fix} \begin{array}{c|c} \boxed{A} & \boxed{B} \\ \hline \boxed{C} & \boxed{D} \end{array} \\ \boxed{A} &= \theta \underset{(I_0)}{i} \underset{(J_0)}{j} \mapsto \min \langle\, \psi_{ij} \\ &\qquad \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ &\qquad \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji} \,\rangle \\ \boxed{B} &= \theta \underset{(I_0)}{i} \underset{(J_1)}{j} \mapsto \min \langle\, \psi_{ij} \\ &\qquad \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ &\qquad \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji} \,\rangle \\ \boxed{C} &= \theta \underset{(I_1)}{i} \underset{(J_0)}{j} \mapsto \min \langle\, \psi_{ij} \\ &\qquad \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ &\qquad \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji} \,\rangle \\ \boxed{D} &= \theta \underset{(I_1)}{i} \underset{(J_1)}{j} \mapsto \min \langle\, \psi_{ij} \\ &\qquad \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ &\qquad \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji} \,\rangle \end{aligned} \tag{6.1}$$

<u>Let</u>

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

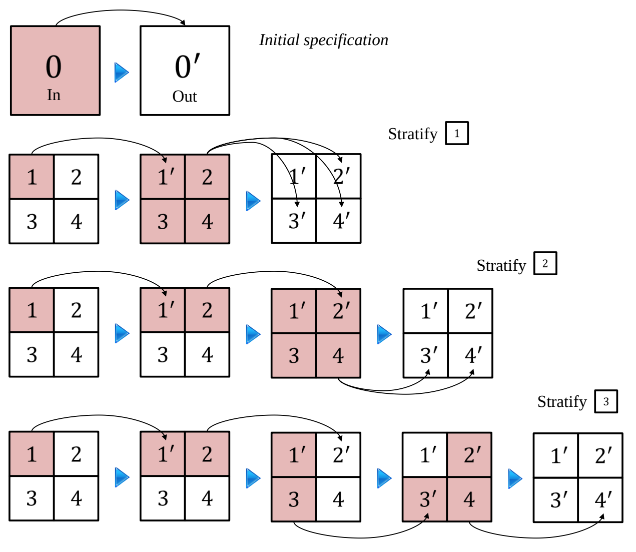


Figure 8. Stratification steps for phase "A" of Simplified Arbiter.

$$A^{IJ} = \psi \mapsto \text{fix}\left(\boxed{A} \gg z \mapsto \begin{array}{c|c} z & \boxed{B} \\ \hline \boxed{C} & \boxed{D} \end{array}\right) \tag{6.2}$$

Stratify[with Padding]

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

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

$$A^{IJ} = \psi \mapsto \text{fix} \begin{array}{c|c} \boxed{A} & \dot\psi \\ \hline \dot\psi & \dot\psi \end{array} \gg \psi \mapsto \text{fix} \begin{array}{c|c} \dot\psi & \boxed{B} \\ \hline \boxed{C} & \boxed{D} \end{array} \tag{6.3}$$

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

The sequence Let, Stratify[with Padding] is now applied in the same manner to $\boxed{B}$ and $\boxed{C}$ (see Figure 8). We do not list the applications as they are analogous to the previous ones.

$$\begin{aligned} A^{IJ} = \psi \mapsto \text{fix} \begin{array}{c|c} \boxed{A} & \dot\psi \\ \hline \dot\psi & \dot\psi \end{array} \gg \psi \mapsto \text{fix} \begin{array}{c|c} \dot\psi & \boxed{B} \\ \hline \dot\psi & \dot\psi \end{array} \gg \\ \psi \mapsto \text{fix} \begin{array}{c|c} \dot\psi & \dot\psi \\ \hline \boxed{C} & \dot\psi \end{array} \gg \psi \mapsto \text{fix} \begin{array}{c|c} \dot\psi & \dot\psi \\ \hline \dot\psi & \boxed{D} \end{array} \end{aligned} \tag{6.4}$$

Synth

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

$$\begin{aligned} A^{IJ} = \psi \mapsto \begin{array}{c|c} A_{\psi}^{I_0 J_0} & \psi \\ \hline \psi & \psi \end{array} \gg \psi \mapsto \text{fix} \begin{array}{c|c} \dot\psi & \boxed{B} \\ \hline \dot\psi & \dot\psi \end{array} \gg \\ \psi \mapsto \text{fix} \begin{array}{c|c} \dot\psi & \dot\psi \\ \hline \boxed{C} & \dot\psi \end{array} \gg \psi \mapsto \text{fix} \begin{array}{c|c} \dot\psi & \dot\psi \\ \hline \dot\psi & \boxed{D} \end{array} \end{aligned} \tag{6.5}$$

We note that 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 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 $\boxed{B}$ from (6.1).

$$\begin{aligned} \boxed{B} = \theta\ \underset{(I_0)}{i}\ \underset{(J_1)}{j} \mapsto \min \langle\ & \psi_{ij} \\ & \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ & \min \underset{(J)}{q} \mapsto \theta_{iq} + w'_{qji}\ \rangle \end{aligned} \tag{6.6}$$

Slice

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

$$\begin{aligned} \boxed{B} = \theta\ \underset{(I_0)}{i}\ \underset{(J_1)}{j} \mapsto \min \Big\langle\ & \psi_{ij} \\ & \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ & \min \Big(\big(\underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}\big)\ / \\ & \qquad \big(\underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}\big)\Big)\ \Big\rangle \end{aligned} \tag{6.7}$$

For the intuition behind this, see the top-right part of Figure 9. The colors represent cell ranges that will be read by different subroutines (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

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

Associativity

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

$$\begin{aligned} \boxed{B} = \theta\ \underset{(I_0)}{i}\ \underset{(J_1)}{j} \mapsto \min \Big\langle\ & \psi_{ij} \\ & \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ & \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}, \\ & \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}\ \Big\rangle \end{aligned} \tag{6.8}$$

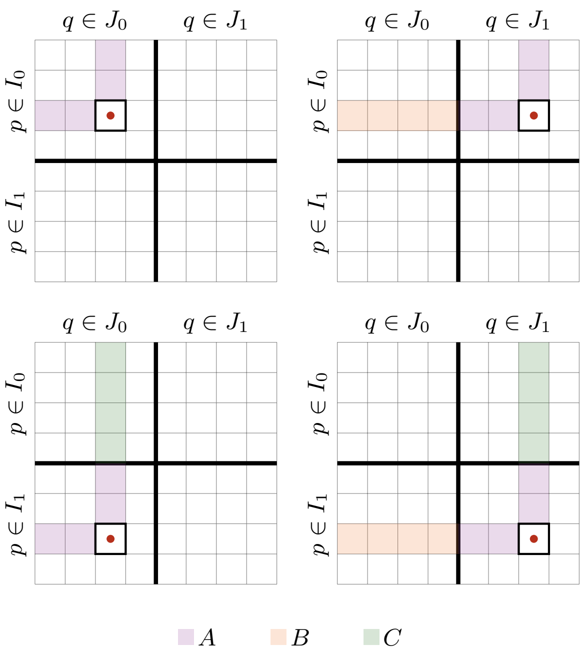


Figure 9. The strategy for applications of Slice in the case study.

Let[reduce]

$$e[\square] = \begin{array}{c|c} \dot\psi & \theta\, i\, j \mapsto \square \\ \hline \dot\psi & \dot\psi \end{array}$$
$$\bar{a} = \psi_{ij},\ \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}$$
$$\bar{b} = \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij},\quad \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}$$

$$\begin{array}{c|c} \dot\psi & \boxed{B} \\ \hline \dot\psi & \dot\psi \end{array} = \text{fix}\left(\boxed{E} \text{»} z \mapsto \begin{array}{c|c} \dot\psi & \boxed{F} \\ \hline \dot\psi & \dot\psi \end{array}\right)$$
$$\begin{aligned} \boxed{E} &= \theta\ \underset{(I_0)}{i}\ \underset{(J_1)}{j} \mapsto \min\langle \psi_{ij}, \\ &\qquad \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}\rangle \\ \boxed{F} &= \theta\ \underset{(I_0)}{i}\ \underset{(J_1)}{j} \mapsto \min\langle z_{\theta ij}, \\ &\qquad \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ &\qquad \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}\rangle \end{aligned} \tag{6.9}$$

Stratify[with Padding]

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

$$\text{fix} \begin{array}{c|c} \dot\psi & \boxed{B} \\ \hline \dot\psi & \dot\psi \end{array} = \text{fix} \begin{array}{c|c} \dot\psi & \boxed{E} \\ \hline \dot\psi & \dot\psi \end{array} \text{»} \psi \mapsto \text{fix} \begin{array}{c|c} \dot\psi & \boxed{F} \\ \hline \dot\psi & \dot\psi \end{array}$$
$$\begin{aligned} \boxed{E} &= \theta\ \underset{(I_0)}{i}\ \underset{(J_1)}{j} \mapsto \min\langle \psi_{ij}, \\ &\qquad \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}\rangle \\ \boxed{F} &= \theta\ \underset{(I_0)}{i}\ \underset{(J_1)}{j} \mapsto \min\langle \psi_{ij}, \\ &\qquad \min \underset{(I)}{p} \mapsto \theta_{pj} + w_{pij}, \\ &\qquad \min \underset{(J_1)}{q} \mapsto \theta_{iq} + w'_{qji}\rangle \end{aligned} \tag{6.10}$$

Define

$$\begin{aligned} B^{I J_0 J_1} \ = \ &\big(\psi \mapsto \\ &\text{fix}\, \theta\ \underset{(I)}{i}\ \underset{(J)}{j} \mapsto \min\langle \psi_{ij}, \\ &\qquad \min \underset{(J_0)}{q} \mapsto \theta_{iq} + w'_{qji}\rangle\big) \\ &:: \big((I \times J_0) \to \mathbb{R}\big) \to \big((I \times J_1) \to \mathbb{R}\big) \end{aligned} \tag{6.11}$$

Synth

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

Synth

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

$$\text{fix} \begin{array}{c|c} \dot\psi & \boxed{B} \\ \hline \dot\psi & \dot\psi \end{array} = \begin{array}{c|c} \psi & B_\psi^{I_0 J_0 J_1} \\ \hline \psi & \psi \end{array} \text{»} \psi \mapsto \begin{array}{c|c} \psi & A_\psi^{I_0 J_1} \\ \hline \psi & \psi \end{array} \tag{6.12}$$

In a similar manner, we will obtain the following:

$$\text{fix} \begin{array}{c|c} \dot\psi & \dot\psi \\ \hline \boxed{C} & \dot\psi \end{array} = \begin{array}{c|c} \psi & \psi \\ \hline C_\psi^{I_0 I_1 J_0} & \psi \end{array} \text{»} \psi \mapsto \begin{array}{c|c} \psi & \psi \\ \hline A_\psi^{I_1 J_0} & \psi \end{array} \tag{6.13}$$

$$\begin{aligned} C^{I_0 I_1 J} \ = \ &\big(\psi \mapsto \\ &\text{fix}\, \theta\ \underset{(I)}{i}\ \underset{(J)}{j} \mapsto \min\langle \psi_{ij}, \\ &\qquad \min \underset{(I_0)}{p} \mapsto \theta_{pj} + w_{pij}\rangle\big) \\ &:: \big((I_0 \times J) \to \mathbb{R}\big) \to \big((I_1 \times J) \to \mathbb{R}\big) \end{aligned} \tag{6.14}$$

And —

$$\begin{aligned} \text{fix} \begin{array}{c|c} \dot\psi & \dot\psi \\ \hline \dot\psi & \boxed{D} \end{array} &= \begin{array}{c|c} \psi & \psi \\ \hline \psi & B_\psi^{I_1 J_0 J_1} \end{array} \text{»} \psi \mapsto \begin{array}{c|c} \psi & \psi \\ \hline \psi & C_\psi^{I_0 I_1 J_1} \end{array} \\ &\text{»} \psi \mapsto \begin{array}{c|c} \psi & \psi \\ \hline \psi & A_\psi^{I_1 J_1} \end{array} \end{aligned} \tag{6.15}$$

	Solving time (*s*)	
	Z3	CVC4
Gap		
Synth	T/O	2.8
Stratify $\boxed{1}$	6.2	5.8
Stratify $\boxed{2}$	3.5	0.5
Let[reduce] + Stratify in $\boxed{2}$	3.1	1.9
Let[reduce] + Stratify in $\boxed{4}$ (*i*)	10	3.9
(*ii*)	5.7	4.7
(*iii*)	4.7	4.3
Parenthesis		
Stratify $\boxed{\diagdown}$ (*diagonal*)	0.4	0.1
Stratify $\boxed{1}$	1.8	2.5
Stratify $\boxed{2}$	1.7	0.6
Synth $\boxed{1}$	T/O	0.5
Synth $\boxed{2}$	T/O	0.5

Table 1. Proof search time for proof obligations in a few sample tactic applications.

This gives the stratified version as shown in Figure 10. The read and write regions are already encoded in the types of A, B, C in (6.5), (6.12), (6.13), and (6.15).

We finish this section by presenting some numbers: Table 1 shows measurements done using two SMT solvers, Z3 and CVC4, and a sample set of tactic applications used in the derivation of Gap and Parenthesis divide-and-conquer DP implementations. The measurements were taken using MacBook Air, 1.7GHz Intel Core i7, with 4GB of RAM. The numbers show wait times that are acceptable for interactive work. Synth goals are rather tricky as they involve quantified assumptions — Z3 timed out while trying to find the proof, but we managed to complete it successfully via CVC4 in all cases.

7. Related Work

Classical work by Smith *et al.* [17] presents rule-based transformation, stringing it tightly with program verification. This lay the foundation for semi-automatic programming [3, 4, 18]. More recently, a similar approach was introduced into Leon [14], 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.

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.

Our "$/$" operator can be compared to the separating disjunction "$*$" of Separation Logic [15], used to frame parts of the dynamic heap (which can be thought of as one large array), in particular while checking that a program only accesses the parts allocated to it in its precondition. While $*$ has the semantics of an existentially quantified predicate, Bellmania uses type qualifiers to explicitly specify a formula defining each part. In this sense, it is more closely related to Region Logic [1]. These formulas make encoding in first-order logic straightforward, and the use of Liquid Types allows for any number of dimensions and for decidable checking of domain inclusion and disjointness.

8. 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.

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.

A. More Tactics

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, $e[\square] = (\cdots \square \cdots)$.

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

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

***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, $e[\square] = (\cdots x_1 \mapsto \cdots x_k \mapsto \cdots \square \cdots)$, where $x_{1..k} \mapsto$ are abstraction terms enclosing $\square$. The bodies may contain arbitrary terms in addition to these abstractions.

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

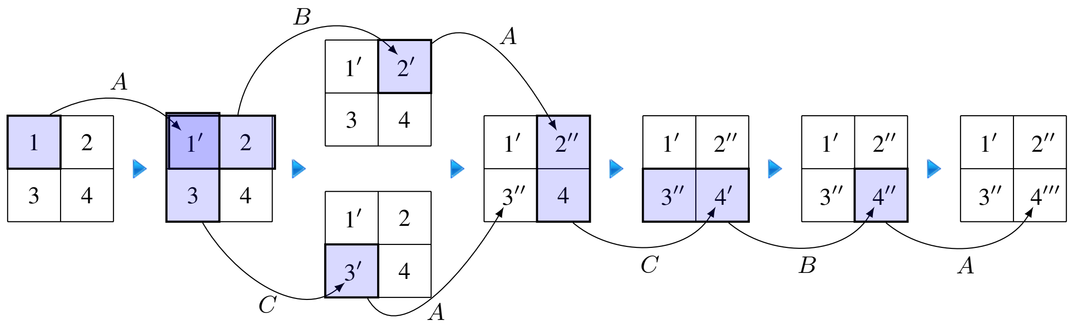


Figure 10. Fully divide-and-conquered version of A^{IJ} in the example development.

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.[2]

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

Let Insertion [reduce]

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

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

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

Padding

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

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*:** 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.

References

[1] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pages 387–411, 2008.

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

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

[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] 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.

[13] M. Jonnalagedda, T. Coppey, S. Stucki, T. Rompf, and M. Odersky. Staged parser combinators for efficient data processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 637–653, 2014.

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

[15] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, 2002.

[16] 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.

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.