Explanations for Dynamic Programming*

Martin Erwig, Prashant Kumar, and Alan Fern

Oregon State University
`[erwig|kumarpra|alan.fern]@oregonstate.edu`

Abstract. We present an approach for explaining dynamic programming that is based on computing with a granular representation of values that are typically aggregated during program execution. We demonstrate how to derive more detailed and meaningful explanations of program behavior from such a representation than would otherwise be possible. To illustrate the practicality of this approach we also present a Haskell library for dynamic programming that allows programmers to specify programs by recurrence relationships from which implementations are derived that can run with granular representation and produce explanations. The explanations essentially answer questions of why one result was obtained instead of another. While usually the alternatives have to be provided by a user, we will show that with our approach such alternatives can be in some cases anticipated and that corresponding explanations can be generated automatically.

1 Introduction

The need for program explanations arises whenever a program execution produces a result that differs from the user's expectation. The difference could be due to a bug in the program or to an incorrect expectation on part of the user. To find out, a programmer may employ a debugger to gain an understanding of the program's behavior [1, 2]. However, debugging is very costly and time consuming [3]. Moreover, the focus on fault localization makes debuggers not the most effective tools for program understanding, since they force the user to think in terms of low-level implementation details. In fact, debuggers typically already assume an understanding of the program by the programmer [4]. The work on customizable debugging operations is additional testimony to the limitations of generic debugging approaches [5, 6]. Finally, debugging is not an option for most users of software, simply because they are not programmers. Therefore, to generate program explanations we need to consider alternative methods.

One approach to producing explanations is to track data that is aggregated during a computation and keep the unaggregated representation that can later be queried to illustrate the effects of the performed computation. Specifically, as we illustrate in Section 2 we can maintain *value decompositions* of those data that are the basis for decisions in computations that might require explanations.

* This work is partially supported by DARPA under the grant N66001-17-2-4030 and by the National Science Foundation under the grant CCF-1717300.

Since our goal is to facilitate systematic explanations of decisions made by dynamic programming algorithms, we show in Section 3 how dynamic programming algorithms can be expressed as recurrence equations over semirings, and we present a Haskell implementation to demonstrate that the idea is feasible in practice. In Section 4 we demonstrate how to use this implementation to operate with value decompositions and produce explanations.

Value decompositions produce explanations for decisions. Specifically, they are used to answer questions such as “Why was A chosen over alternative B?” Alternatives against which decisions are to be explained are typically provided by users, but as we demonstrate in Section 5, sometimes they can be anticipated, which means that comparative explanations can be generated automatically. Finally, we compare our approach with related work in Section 6 and present some conclusions in Section 7. The main contributions of this paper are as follows.

– A framework based on semirings for expressing *dynamic programming algorithms* that supports the *computation with value decompositions*.
– An extension of the framework for the *automatic generation of explanations*.
– A method for the *automatic generation of examples* in explanations.
– An implementation of the approach as a *Haskell library*.

2 Explaining Decisions With Value Decompositions

Many decision and optimization algorithms select one or more alternatives from a set based on data gathered about different aspects for each alternative. For example, to decide between two vacation destinations one may rank weather (W), food (F), and price (P) on a point scale from 1 (poor) to 10 (great) and compute a total point score for each possible destination and then pick the one with the highest score.

This view can be formalized using the concepts of *value decomposition* and *valuation*. Given a set of categories C, a mapping $v : C \to \mathbb{R}$ is called a *value decomposition* (with respect to C). The (total) *value* of a value decomposition is defined as the sum of its components, that is, $\hat{v} = \sum_{(c,x)\in v} x$. A *valuation* for a set S (with respect to C) is a function φ that maps elements of S to corresponding value decompositions, that is, $\varphi : S \to \mathbb{R}^C$. We write $\hat{\varphi}(A)$ to denote the total value of A's value decomposition. In our example scenario lets consider two destinations $S = \{X, Y\}$ with the respective value decompositions $v_X = \{W \mapsto 7, F \mapsto 8, P \mapsto 1,\}$ and $v_Y = \{W \mapsto 4, F \mapsto 4, P \mapsto 9,\}$, which yields the valuation $\varphi = \{X \mapsto v_X, Y \mapsto v_Y\}$.

The elements of S can be ordered based on the valuation totals in an obvious way:

$$\forall A, B \in S.\ A > B \Leftrightarrow \hat{\varphi}(A) > \hat{\varphi}(B)$$

When a user asks about a program execution why A was selected over B, the obvious explanation is $\hat{\varphi}(A) > \hat{\varphi}(B)$, reporting the valuation totals. However, such an answer might not be useful, since it ignores the categories that link the raw numbers to the application domain and thus lacks a context for the user to interpret the numbers. In our example, destination Y would be selected since $\hat{\varphi}(Y) = 17 > \hat{\varphi}(X) = 16$, which might be surprising because X seems clearly so much better than Y in terms of weather and food.

If the value decomposition is maintained during the computation, we can generate a more detailed explanation. First, we can rewrite $\hat{\varphi}(A) > \hat{\varphi}(B)$ as $\hat{\varphi}(A) - \hat{\varphi}(B) > 0$, which suggests the definition of the *valuation difference* between two elements A and B as follows.

$$\delta(A, B) = \{(c, x - y) \mid (c, x) \in \varphi(A) \wedge (c, y) \in \varphi(B)\}$$

The total of the value difference $\hat{\delta}(A, B)$ is given by the sum of all components, just like the total of a value decomposition. In our example we have $\delta(Y, X) = \{W \mapsto -3, F \mapsto -4, P \mapsto 8\}$. It is clear that the value difference generally contains positive and negative entries and that for $\delta(A, B) > 0$ to be true the sum of the positive entries must exceed the absolute value of the sum of the negative entries. We call the negative components of a value difference its *barrier*. It is defined as follows.

$$\beta(A, B) = \{(c, x) \mid (c, x) \in \delta(A, B) \wedge x < 0\}$$

The total value $\hat{\beta}(A, B)$ is again the sum of all the components. In our example we have $\beta(Y, X) = \{W \mapsto -3, F \mapsto -4\}$ and $\hat{\beta}(Y, X) = -7$.

The decision to select A over B does not necessarily need as support all of the positive components of $\delta(A, B)$; any subset whose total is larger than $|\hat{\beta}(A, B)|$ will suffice. We call such a subset a *dominator*:[1]

$$\Delta(A, B) = \{D \mid D \subseteq \delta(A, B) \wedge \hat{D} > |\hat{\beta}(A, B)|\}$$

The only dominator in our toy example is $\Delta(Y, X) = \{P \mapsto 8\}$.

The smaller a dominator, the better it is suited as an explanation, since it requires fewer details to explain how the barrier is overcome. We therefore define the *minimal dominating set* (MDS) as follows.

$$\underline{\Delta}(A, B) = \{D \mid D \subseteq \Delta(A, B) \wedge D' \subset D \Rightarrow D' \notin \Delta(A, B)\}$$

Note that $\underline{\Delta}$ may contain multiple elements, which means that minimal dominators are not unique. In other words, a decision may have different minimally sized explanations. Again, due to the small size of our example, the only dominator is also the MDS in this case. Nevertheless, it captures the explantion that Y is preferred over X due to the extreme price difference.

[1] This definition allows dominators to contain negative components, which are counterproductive to the goal of dominators. However, the definition of minimal-size dominators will never produce a dominator with a negative component, so that the general definition does not hurt.

3 Dynamic Programming with Semirings

We show how to represent dynamic programming (DP) algorithms by semirings in Section 3.1 and how such a representation can automatically generate efficient implementations from recursive specifications in Haskell in Section 3.2. We illustrate the use of the library with an example in Sections 3.3 and 3.4.

3.1 Semirings and Dynamic Programming

A semiring is an algebraic structure $(S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, which consists of a nonempty set S with binary operations for addition ($\oplus$) and multiplication ($\otimes$) plus neutral elements zero ($\mathbf{0}$) and one ($\mathbf{1}$) [7]. Figure 1 lists the axioms that a semiring structure has to satisfy and several semiring examples.

$$\begin{aligned}
a \oplus (b \oplus c) &= (a \oplus b) \oplus c \\
a \oplus b &= b \oplus a \\
a \oplus \mathbf{0} &= \mathbf{0} \oplus a = a \\
a \otimes (b \otimes c) &= (a \otimes b) \otimes c \\
a \otimes \mathbf{1} &= \mathbf{1} \otimes a = a \\
a \otimes (b \oplus c) &= a \otimes b \oplus a \otimes c \\
(a \oplus b) \otimes c &= a \otimes c \oplus b \otimes c \\
a \otimes \mathbf{0} &= \mathbf{0} \otimes a = \mathbf{0}
\end{aligned}$$

Semiring	Set	$\oplus$	$\otimes$	$\mathbf{0}$	$\mathbf{1}$
Boolean	$\{true, false\}$	$\vee$	$\wedge$	*false*	*true*
Counting	$\mathbb{N}$	$+$	$\times$	0	1
Tropical (Min-Plus)	$\mathbb{R}^+ \cup \{\infty\}$	min	$+$	∞	0
Arctic (Max-Plus)	$\mathbb{R}^+ \cup \{-\infty\}$	max	$+$	$-\infty$	0
Viterbi	$[0, 1]$	max	$\times$	0	1

Fig. 1. Semiring axioms and examples.

A semiring $(S, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ with a partial order $\leq$ over S is *monotonic* if $\forall s, t, u \in S$, $(s \leq t) \Rightarrow (s \otimes u \leq t \otimes u)$ and $(s \leq t) \Rightarrow (u \otimes s \leq u \otimes t)$. A monotonic semiring ensures the so-called *optimal subproblem property*, which says that the optimal solution of a dynamic programming problem contains the optimal solutions of the subproblems into which the original problem was divided. This can be seen as follows [8]. Suppose the values s and t correspond to two solutions of a subproblem such that s is a better solution than t (that is, $s \leq t$). Further, suppose that u is the optimal solution of a set of subproblems that does not include the subproblems producing the values s and t. The monotonicity property ensures that s combined with u (and not t combined with u) always results in the optimal solution when the aforementioned subproblem is combined with the set of subproblems.

Dynamic programming algorithms can be described by recursive equations that use operations of a particular kind of semiring, and since monotonic semirings satisfy the optimal substructure property, the computations produce correct solutions. Note that we can slightly weaken the requirements for the optimal subproblem property. Since monotonicity doesn't depend on the absorption rule (which requires $a \otimes \mathbf{0} = \mathbf{0} \otimes a = \mathbf{0}$), the optimal subproblem property holds for DP algorithms that are based on what we call *quasi-semirings*, which are semirings for which the absorption rule doesn't hold. We will make use of this property later in Section 3.4 where we define a quasi-semiring for computing values "alongside" the values of a semiring.

3.2 A Haskell Library for Dynamic Programming

We have implemented a library for dynamic programming and semirings that is based on the DP library by Sasha Rush.[2] The first component is a representation of semirings. The semiring structure can be nicely captured in a Haskell type class. Of course, the required laws cannot be expressed in Haskell; it's the programmer's obligation to ensure that the laws hold for their instance definitions.

```
class Semiring a where
  zero, one :: a
  (<+>), (<.>) :: a -> a -> a

sconcat :: Semiring a => [a] -> a
sconcat = foldr (<+>) zero
```

Several Haskell packages exist that already define a `Semiring` type class (some of which are defunct). In general, previous approaches have the advantage that they integrate the `Semiring` class more tightly into the existing Haskell class hierarchy. For example, `zero` and `<+>` are essentially `mempty` and `mappend` of the class `Monoid`. Mainly for presentation reasons we decided to define the `Semiring` type class independently, since it allows the definition of instances through a single definition instead of being forced to split it into several ones.

To see how this library is used, consider the following implementation for computing Fibonacci numbers, which uses the Counting semiring, obtained by defining a number type as an instance of the `Semiring` class in the obvious way.[3]

```
instance Semiring Integer where
  {zero = 0; one = 1; (<+>) = (+); (<.>) = (*)}
```

The semiring recurrence representation is very similar to the well-known recursive definition, except for two notable differences are: First, recursive calls are made by a function `memo` to indicate when intermediate results of recursive calls should be stored in a table. Second, the implementation consists of two parts, (a) the definition of the recurrence relation that denotes a table-based, efficient implementation (`fibT`), and (b) an interface that simply executes the table-based implementation (`fib`).

[2] See `http://hackage.haskell.org/package/DP`. The code has not been maintained in some time and doesn't seem to work currently. Our implementation is available at `https://github.com/prashant007/XDP`.

[3] Note that the Counting semiring is *not* monotonic. The implementation of Fibonacci numbers is still correct, since the $\oplus$ function isn't used to select among different alternatives.

```
fibT :: DP Integer Integer
fibT 0 = zero
fibT 1 = one
fibT n = memo (n-1) <+> memo (n-2)

fib :: Integer -> Integer
fib n = runDP fibT n
```

This examples illustrates some of the major building blocks that are provided by the dynamic programming library.

- The functions `<+>` and `<.>` correspond to semiring addition ($\oplus$) and multiplication ($\otimes$), respectively.
- The type `DP t r` represents a dynamic programming computation. Parameter `t` represents the argument, corresponding to the table index on which recursion occurs, and `r` represents the result type of the computation.
- The function `memo` takes an index as input. The index can be thought of as the input to the smaller subproblems that need to be solved while solving a dynamic programming problem; it is the quantity on which the algorithm is recursively invoked. With `memo` a subproblem for a given input value is solved only once, and the result is stored in a table for potential reuse.
- The function `inj` (used later) turns any semiring value (different from **0** and **1**) into a `DP` value.
- The function `runDP` executes a dynamic programming specification `DP t r` that works on tables indexed by a type `t` by producing a function that computes results of type `r` from an initial value of type `t`.

3.3 Computing the Lengths of Shortest Paths

In its simplest form, a shortest path algorithm takes a graph together with a source and destination node as inputs and computes the length of the shortest path between the two nodes.

In the following, we show how a program for computing shortest paths can be systematically extended to support the generation of explanations in addition to the computed answers. We use the *Bellman-Ford* algorithm [9], which can be concisely described by the following recurrence relation in which SP denotes the length of the shortest path in a graph between the start node s and any other v with at most i number of edges. This algorithm works only for graphs with non-negative edge weights. We directly show the definition using the operations from the Min-Plus semiring (see Figure 1): $\oplus$ represents min, $\otimes$ represents numeric addition, and the constants **0** and **1** represent the additive and the multiplicative identity and stand for ∞ and 0, respectively.

$$SP(v,i) = \begin{cases} \mathbf{1} & i = 0 \land v = s \\ \mathbf{0} & i = 0 \land v \neq s \\ SP(v,i-1) \oplus \bigoplus_{(u,v)\in E}(SP(u,i-1) \otimes w(u,v)) & \text{otherwise} \end{cases}$$

Here E is the set of edges in the graph, and $w(u, v)$ denotes the weight of edge (u, v). This algorithm incrementally updates connection information between nodes. When all edge labels in a graph with n nodes are positive, the shortest path contains at most $n-1$ edges. Therefore, the shortest path to a node t can be obtained by the expression $SP(t, n)$. In each step the algorithm considers nodes that are one more edge away from the target node and updates the distance of the currently known shortest path.

Note that this formulation of the algorithm is actually more general than the original, since the operations can be taken from different semirings to express different computations. We will later take advantage of this generality by generating, in addition to the shortest path value, decomposed values, the path itself, and explanations.

Next we show how the shortest path algorithm can be expressed as a dynamic programming algorithm in our library. The Min-Plus semiring is implemented in Haskell through a class instance definition for the type constructor `Large` that adds ∞ to a number type. We need ∞ to represent the case when there isn't a path between two nodes.

```
data Large a = Finite a | Infinity deriving (Eq,Ord)

instance (Num a,Ord a) => Semiring (Large a) where
  {zero = Infinity; one = Finite 0; (<+>) = min; (<.>) = (+)}
```

The instance definitions for `Functor`, `Applicative`, and `Num` are all straightforward (they are basically the same as for `Maybe`), and we omit them here for brevity. One subtle, but important, difference between `Large` and `Maybe` is that `Infinity` is defined as the second constructor in the data definition, which makes it the largest element of the `Large` data type when an `Ord` instance is derived.

For the Haskell implementation of the algorithm, we represent edges as pairs of nodes and a graph as a list of edges paired with their lengths, see Figure 2. We use a multi-parameter type class `SP` to facilitate a generic implementation of the shortest path function that works for different edge label types (type parameter `l`) and types of results (type parameter `r`). As in the Fibonacci example, the implementation consists of two parts: (a) a recurrence specification of the DP algorithm (the function `sp`) and (b) the function `shortestPath` for actually running the described computation. Both functions have a default implementation that doesn't change for different class instances. The class consists of an additional member `result` that turns labeled edges into values of the DP result type `r`. The definition of the `sp` function is directly derived from the semiring representation of the Bellman-Ford recurrence relation. Note that the `memo` function in the definition of `sp` takes pairs as input and effectively denotes a recursion of the `sp` function, memoizing the output of each recursive call for later reuse. The second argument of the `<+>` function in the recursive case of the `sp` function implements the part $\bigoplus_{(u,v)\in E}(SP(u, i-1) \otimes w(u, v))$ of the recurrence relation. The function `sconcat` takes a list of values, namely all incoming edges at node `v`, and combines these using the semiring addition function `<+>`. Finally, the actual computation of a shortest path between two nodes is initiated by the function

```
type Node = Int
type Edge = (Node,Node)
type Graph l = [(Edge,l)]

noNodes :: Graph l -> Int
noNodes = length . nub . concatMap (\((p,q),_) -> [p,q])

class Semiring r => SP l r where
  result :: (Edge,l) -> r

  sp :: Graph l -> Node -> DP (Node,Int) r
  sp g s (v,0) = if s==v then one else zero
  sp g s (v,i) = memo (v,i-1) <+> sconcat
                  [memo (u,i-1) <.> (inj.result) e | e@((u,v'),_)<-g, v'==v]

  shortestPath :: Graph l -> Node -> Node -> r
  shortestPath g s t = runDP (sp g s) (t,noNodes g-1)
```

Fig. 2. Generic shortest path implementation.

`shortestPath` through calling `sp` and passing the number of nodes of the graph as an additional parameter (computed by the helper function `noNodes`).

To execute the `shortestPath` function for producing path lengths for graphs with non-decomposed edge labels, we need to create an instance of the `SP` type class with the corresponding type parameters. Since the functions `sp` and `shortestPath` are already defined, we only need a definition for `result`.

```
instance SP Double (Large Double) where
  result (_,l) = Finite l
```

The result of running the shortest-path algorithm on the non-decomposed graph shown on the left of Figure 3 produces the following output.

```
> shortestPath g 1 4 :: Large Double
30.0
```

Specifying the result type (`r`) to be `Large Double` selects the implementation in which the `result` function maps a labeled edge to the DP result type as shown. In addition to the length of the shortest path we may also want to know the path itself. We develop a solution based on semirings next.

3.4 Computing Shortest Paths

To compute shortest paths in addition to their lengths, we need an instance of `Semiring` for the type `(Large Double,[Edge])`. A first attempt could be to define pairs of semirings as semirings. This would require both components to

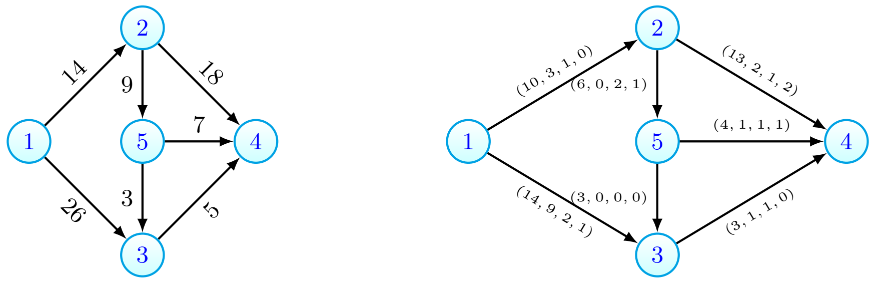

Fig. 3. An edge-labeled graph (`g`) and its version with decomposed edge-labels (`gd`).

be semirings themselves, but since there is not a straightforward instance of lists as semirings, we have to adopt a different strategy.

If we look at this example more closely, we can observe that the DP computation of a shortest path is solely driven by the first component of the pair type and that the paths are computed alongside. This means that the path type doesn't really need to support the `Semiring` structure. We can exploit this fact by defining a semiring instance for pairs that relies for its semiring semantics only on the semiring instance of its first component. To handle the combination of values in its second component, we require that the type be a monoid and use the binary operation in the instance definition for the `<.>` function. The `<+>` function acts as a selector of the two values, and the selection is controlled by a selection function that the first type parameter has to support through a corresponding instance definition for the class `Selector`. The function `first` implements a selection decision between two values; it returns `True` if the first argument is selected and `False` otherwise. The code is shown in Figure 4.

Note that `View` is *not* a semiring, because the absorption rule ($a \otimes \mathbf{0} = \mathbf{0} \otimes a = \mathbf{0}$) doesn't hold. However, that is not a problem, since the monotonicity property, which ensures the correctness of DP implementation, is not affected by that.

With the help of this view quasi-semiring structure we can now obtain a DP algorithm that computes the paths alongside the lengths. To this end, we represent a path `p` and its length `l` as a value `View l p` so that the length provides a view on the path on which the DP computation operates.

```
type Path l = View l [Edge]
```

The shortest path algorithm results from an instance of the `SP` type class for the result type `Path (Large Double)`, which again only requires the definition of the `result` function to map labeled edges to the DP result type.

```
instance SP Double (Path (Large Double)) where
  result (e,l) = View (Finite l) [e]
```

The result of running the shortest-path algorithm on the non-decomposed graph produces the following output. Again, we specify the result type of the DP computation to select the appropriate implementation of `result` and thus `sp`.

```
data View a b = View a b

class Selector a where
  first :: a -> a -> Bool

instance Ord a => Selector (Large a) where
  first = (<=)

instance (Selector a,Semiring a,Monoid b) => Semiring (View a b) where
  zero = View zero mempty
  one  = View one mempty
  l@(View x _) <+> r@(View y _) = if first x y then l else r
  (View x a)   <.> (View y b)   = View (x <.> y) (mappend a b)
```

Fig. 4. The `View` quasi-semiring.

```
> shortestPath g 1 4 :: Path (Large Double)
View 30.0 [(1,2),(2,5),(5,4)]
```

This is the correct result, but we don't get an explanation why it is faster than, say, the more direct path `[(1,3),(3,4)]`.

4 Explanations From Value Decomposition

To use the generic DP programming framework with value decompositions, we have to define a type for decomposed values and define its `Ord` and `Eq` type class instances. Both definitions use the sum of the elements of the lists contained in the `Values` constructors to perform the comparison.

```
newtype Decomposed a = Values {values :: [a]}

instance (Eq a,Num a) => Eq (Decomposed a) where
  (==) = (==) `on` sum.values

instance (Ord a,Num a) => Ord (Decomposed a) where
  (<=) = (<=) `on` sum.values
```

These definitions ensure that decomposed values are compared based on their sums.

To use value decompositions in the shortest path computation, we need a `Num` instance for the `Decomposed` data type, which is straightforward to define, except that we need a flexible interpretation of the semiring constants that depends on the number of value components. For example, in the Min-Plus semiring we expect $\mathbf{1}$ to denote $[0,0]$ in the context of $[1,2] \otimes \mathbf{1}$, while it should denote $[0,0,0]$ in the context of $[1,2,3] \otimes \mathbf{1}$. We can achieve this behavior by defining the `Num` instance to be singleton lists by default that will be padded to match the length of potentially longer arguments.

Next we can obtain two more versions of the shortest path algorithm as an instance of the `SP` type class, one for computing lengths only, and one for computing paths alongside lengths. The type of the edge labels is `[Double]` to reflect the decomposed edge labels in the input graphs. The result types for the DP computations are either the path lengths represented as decomposed edge labels or the view of paths as decomposed values. Here are the corresponding instance definitions.

```
instance SP [Double] (Large (Decomposed Double)) where
  result (_,l) = Finite (Values l)

instance SP [Double] (Path (Large (Decomposed Double))) where
  result (e,l) = View (Finite (Values l)) [e]
```

The shortest-path algorithms use the graph `gd` with decomposed edge labels.

```
> shortestPath gd 1 4 :: Large (Decomposed Double)
[20.0,4.0,4.0,2.0]

> shortestPath gd 1 4 :: Path (Large (Decomposed Double))
View [20.0,4.0,4.0,2.0] [(1,2),(2,5),(5,4)]
```

We can compute valuation differences and minimally dominating sets to compare the results with alternative solutions. For example, the decomposed length of the alternative path $(1,3),(3,4)$ between nodes 1 and 4 is `[17,10,3,1]`. Since `Decomposed` and `Large` are `Num` instances, we can compute the valuation difference with respect to the shortest path to be `[3.0,-6.0,1.0,1.0]`. To implement a function for computing minimally dominating sets, we have to extract the decomposed values (of type `Decomposed Double`) from the semiring values (of type `Large (Decomposed Double)`) produced by the shortest path function. Moreover, when sorting the components of the valuation difference into positive and negative parts, we need to decide which parts constitute the barrier and which parts are supporting components of the computed optimal value. This decision depends on the semiring on which the computation to be explained is based. In the shortest path example, we have used the Min-Plus semiring for which positive value differences constitute barriers and negative values overcome the barrier. In general, a value s is a *supporting value* (for overcoming a barrier) if $s \oplus \mathbf{1} = s$. We can realize both requirements through a (multi-parameter) type class `Decompose` that relates semiring types with the types of values used in decompositions, see Figure 5.[4]

With this type class we can directly implement the definition of $\underline{\Delta}$ from Section 2 as a function for computing the smallest sublist of supporting values whose

[4] The additional argument of type `a` for `supportive` is required to keep the types in the multi-parameter type class unambiguous. Moreover, we can't unfortunately simply give the generic definition for `supportive` indicated by the equation, since that would also lead to an ambiguous type.

```
class Semiring a => Decompose a b | a -> b where
  dec :: a -> Decomposed b
  supportive :: a -> b -> Bool

instance Decompose (Large (Decomposed Double)) Double where
  dec Infinity = Values []
  dec (Finite vs) = vs
  supportive _ x = x<0

data Labeled a = Label String a

unlabel :: Labeled a -> a
unlabel (Label _ x) = x

withCategories :: Decomposed a -> [String] -> Decomposed (Labeled a)
withCategories d cs = Values (map Label cs) <*> d

explainWith :: (Decompose a b,Ord b,Num b) =>
               [String] -> a -> a -> Decomposed (Labeled b)
explainWith cs d d' = Values $ head $ sortBy (compare `on` length) doms
    where (support,barrier) = partition sup $ values delta
          doms = [d | d <- sublists support, abs (sum d) > abs (sum barrier)]
          delta = dec d `withCategories` cs - dec d' `withCategories` cs
          sup = supportive d . unlabel
```

Fig. 5. Minimal dominators and explanations.

(absolute) sum exceeds the sum of the barrier, in this case it's the singleton list `[-6.0]`. Since the number itself doesn't tell us what category provides this dominating advantage, we assign meaning to the bare numbers through a data type `Labeled` that pairs values with strings. Creating a `Num` instance for `Labeled` allows us to assign labels to the individual numbers of a `Decomposed` value and apply the computation of dominating sets to work with labeled numbers, resulting in the function `explainWith`. If `p` is the result of the shortest path function shown above and `p'` is the corresonding value for the alternative path considered, then we can explain why `p` is better than `p'` by invoking the function.

```
> explainWith ["Distance","Traffic","Weather","Construction"] p p'
[Traffic:-6.0]
```

The result says that, considering traffic alone, `p` has an advantage over `p'`, since the traffic makes the path of `p` faster by `6`.

The generation of explanation for other DP algorithms works in much the same way: First, identify the appropriate semiring for the optimization problem. The `View` quasi-semiring facilitates variety of computations that produce results on different levels of detail. Second, implement the DP algorithm as a type class that contains the main recurrence, a wrapper to run the described computation, plus the function `result` that ties the DP computation to different result types. Finally, define a value decomposition for the result type. The function `explainWith` can then compare optimal results with alternatives and produce explanations based on value categories.

5 Proactive Generation of Explanations

At this point, a user who wants an explanation has to supply an alternative as an argument for the function `explainWith`. Sometimes such examples can be automatically generated, which means that user questions about solutions can be anticipated and proactively answered.

In the case of finding shortest paths, a result may be surprising—and therefore might prompt the user to question it—if the suggested path is not the shortest one in terms of traveled distance. This is because the travel distance retains a special status among all cost categories in that it is always a determining factor in the solution and can never be ignored. This is different for other categories, such as traffic or weather, which may be 0 and in that case play no role in deciding between different path alternatives.

In general, we can therefore distinguish between those categories that always influence the outcome of the computation and those that only *may* do so. We call the former *principal categories* and the latter *minor categories*. We can exploit knowledge about principal and minor categories to anticipate user questions by executing the program with decomposed values but keeping only the values for the principal categories. If the result is different from the one produced when using the complete value decomposition, it is an alternative result worthy of an explanation, and we can compute the minimal dominating set accordingly.

Unfortunately, this strategy doesn't work as expected, because if we remove minor categories to compute an alternative solution, the values of those categories aren't aggregated alongside the computation of the alternative and thus are not available for the computation of minimal dominating sets. Alternatively, instead of changing the underlying decomposition data, we can change the way their aggregation controls the DP algorithm. Specifically, instead of ordering decomposed values based on their sum, we can order them based on a primary category (or a sum of several primary categories). In Haskell we can achieve this by defining a new data type `Principal`, which is basically identical to `Decomposed` but has a different `Ord` instance definition.

```
instance (Ord a,Num a) => Ord (Principal a) where
  (<=) = (<=) `on` head.pvalues
```

We also need a function that can map `Principal` data into `Decomposed` data within the type of the semiring to get two `Decomposed` values that can be compared and explained by the function `explainWith`. To compute the main result (calculated using all the categories), and the alternative result (calculated using just the principal categories) simultaneously we can use a *pair* semiring, which is defined

```
newtype Principal a = PValues {pvalues :: [a]}

class FromPrincipal f where
  fromPrincipal :: f (Principal a) -> f (Decomposed a)

explain :: (FromPrincipal f,Decompose (f (Decomposed a)) b,
            Eq (f (Decomposed a)),Ord b,Num b) =>
           (i -> (f (Decomposed a), f (Principal a))) -> [String] -> i ->
           (f (Decomposed a),Maybe (f (Decomposed a),Decomposed (Labeled b)))
explain f cs i | o1==o2    = (o1,Nothing)
               | otherwise = (o1,Just (o2,explainWith cs o1 o2))
               where  (o1,o') = f i
                      o2      = fromPrincipal o'
```

Fig. 6. Generating automatic explanations.

component-wise in the obvious way. With these preparations we can define the function `explain` that takes an instance of the function to be explained. The function outputs a pair of `Decomposed` and `Principal` values whenever they differ, which can then be explained as before using the function `explainWith`.

To use `explain` in our example, we have to create another instance for the `SP` class that works with the pair of `Decomposed` and `Principal` types, captured in the type synonym `LPair`. We also have to create an instance for the function `fromPrincipal` so that we can turn `Principal` data into `Decomposed` data inside the `Large` type.

```
type LPair = (Large (Decomposed Double),Large (Principal Double))

instance SP [Double] LPair where
  result (e,l) = (Finite (Values l),Finite (PValues l))

instance FromPrincipal Large where
  fromPrincipal = fmap (Values . pvalues)
```

Finally, to be able to apply `explain` we have to normalize the argument type of the shortest path function into a tuple.

```
type SPInput = (Graph [Double],Node,Node)

spPair :: SPInput -> LPair
spPair (g,v,w) = shortestPath g v w
```

When we apply `explain`, it will in addition to computing the shortest path also automatically find an alternative path and explain why it is not a better alternative.

```
> explain spPair ["Distance","Traffic","Weather","Construction"] (gd,1,4)
([20.0,4.0,4.0,2.0],Just ([17.0,10.0,3.0,1.0],[Traffic:-6.0]))
```

Of course, the output could be printed more prettily.

6 Related Work

In [10, 11] we proposed the idea of preserving the structure of aggregated data and using it to generate explanations for reinforcement learning algorithms based on so-called *minimum sufficient explanations*. That work is less general than what we describe here and strictly situated in a machine learning context that is tied to the framework of *adaptation-based programming* [12]. Value decomposition and minimal dominating sets are a general approach to the generation of explanations for a wider range of algorithms. A different concept of "minimal sufficient explanations" was also used in related work on explanations for optimal Markov Decision Process (MDP) policies [13]. That work is focused on automated planning and on explaining the optimal decision of an optimal policy. Those explanations tend to be significantly larger than explanations for decisions to select between two alternatives. Also, that work is not based on value or reward decompositions.

Debugging can be viewed as a specific form of explanation. For example, *Delta debugging* reveals the cause-effect chain of program failures, that is, the variables and values that caused the failure [14]. Delta debugging needs two runs of a program, a successful one and an unsuccessful one. It systematically narrows down failure-inducing circumstances until a minimal set remains, that is, if there is a test case which produces a bug, then delta debugging will try to trim the code until the minimal code component which reproduces the bug is found. Delta debugging and the idea of MDSs are similar in the sense that both try to isolate minimal components responsible for a certain output. An important difference is that delta debugging produces program fragments as explanations, whereas an explanation based on value decompositions is a structured representation of program inputs.

The process of debugging is complicated by the low-level representation of data processed by programs. *Declarative debugging* aims to provide a more high-level approach, which abstracts away the evaluation order of the program and focuses on its high-level logical meaning. This style of debugging is discussed in [15] and is at the heart of, for example, the Haskell debugger Buddha. *Observational debugging* as used in the Haskell debugger Hood [16] allows the observation of intermediate values within the computation. The programmer has to annotate expressions of interest inside the source code. When the source code is recompiled and rerun, the values generated for the annotated expressions are recorded. Like value decomposition, observational debugging expects the programmers to identify and annotate parts of the programs which are relevant to generate explanations. A potential problem with the approach is that the number of intermediate values can become large and not all the intermediate values have explanatory significance.

The *Whyline* system [17] inverts the debugging process, allowing users to ask questions about program behavior and responding by pointing to parts of the code responsible for the outcomes. Although this system improves the debugging process, it can still only point to places in the program, which limits its explanatory power. In the realm of spreadsheets, the *goal-directed debugging* approach [18] goes one step further and also produces change suggestions that would fix errors. Change suggestions are a kind of counter-factual explanations.

Traces of program executions can explain how outputs are produced from inputs. While traces are often used as a basis for debugging, they can support more general forms of explanations as well. Since traces can get quite large, focusing on interesting parts poses a particular challenge. Program slicing can be used to filter out irrelevant parts of traces. Specifically, *dynamic slicing* has been employed to isolate parts of a program that potentially contribute to the value computed at a point of interest [19]. Using dynamic slicing for generating explanations of functional program execution is described in [20]. This approach has been extended to imperative functional programs in [21]. Our approach does not produce traces as explanations. Instead, value decompositions maintain a more granular representation of values that are aggregated. Our approach requires some additional work on the part of the programmers in decomposing the inputs (even though in our library we have tried to minimize the required effort). An advantage of our approach is that we only record the information relevant to an explanation in contrast to generic tracing mechanisms, which generally have to record every computation that occurs in a program, and require aggressive filtering of traces afterwards.

7 Conclusions and Future Work

We have introduced an approach to explain the execution of dynamic programs through value decompositions and minimal dominating sets: Value decompositions offer more details about how decisions are made, and minimal dominating sets minimize the amount of information a user has to absorb to understand an explanation. We have put this idea into practice by integrating it into a Haskell library for dynamic programming that requires minimal effort from a programmer to transform a traditional, value-producing program into one that can also produce explanations of its results. The explanation component is modular and allows the explanations for one DP algorithm to be specialized to different application domains independently of its implementation. In addition to producing explanations in response to user requests, we have also shown how to anticipate questions about results and produce corresponding explanations automatically.

In future work, we will investigate the applicability of our approach to more general algorithmic structures. An open question is how to deal with the aggregation of data along unrelated decisions. Our approach works well for dynamic programming algorithms because all the decisions involved in the optimization process are compositionally related through a semiring. For algorithms that don't fit into the semiring structure, the data aggregation for producing explanations must be achieved in a different way.

References

1. Roehm, T., Tiarks, R., Koschke, R., Maalej, W.: How do professional developers comprehend software? In: 34th Int. Conf. on Software Engineering. (2012) 255–265

2. Murphy, G.C., Kersten, M., Findlater, L.: How are Java Software Developers Using the Eclipse IDE? IEEE Software **23**(4) (2006) 76–83
3. Vessey, I.: Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. IEEE Trans. on Systems, Man, and Cybernetics **16**(5) (1986) 621–637
4. Parnin, C., Orso, A.: Are Automated Debugging Techniques Actually Helping Programmers? In: Int. Symp. on Software Testing and Analysis. (2011) 199–209
5. Marceau, G., Cooper, G.H., Spiro, J.P., Krishnamurthi, S., Reiss, S.P.: The Design and Implementation of a Dataflow Language for Scriptable Debugging. Automated Software Engineering **14**(1) (2007) 59–86
6. Khoo, Y.P., Foster, J.S., Hicks, M.: Expositor: Scriptable Time-travel Debugging with First-class Traces. In: ACM/IEEE Int. Conf. on Software Engineering. (2013) 352–361
7. Golan, J.S.: Semirings and their Applications. Springer, Dordrecht, Netherlands (1999)
8. Huang, L.: Advanced dynamic programming in semiring and hypergraph frameworks. In: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications - Tutorial notes. (2008) 1–18
9. Bellman, R.: On a routing problem. Quarterly of Applied Mathematics **16**(1) (1958) 87–90
10. Erwig, M., Fern, A., Murali, M., Koul, A.: Explaining Deep Adaptive Programs via Reward Decomposition. In: IJCAI/ECAI Workshop on Explainable Artificial Intelligence. (2018) 40–44
11. Juozapaitis, Z., Fern, A., Koul, A., Erwig, M., Doshi-Velez, F.: Explainable Reinforcement Learning via Reward Decomposition. In: IJCAI/ECAI Workshop on Explainable Artificial Intelligence. (2019) 47–53
12. Bauer, T., Erwig, M., Fern, A., Pinto, J.: Adaptation-Based Program Generation in Java. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. (2011) 81–90
13. Khan, O.Z., Poupart, P., Black, J.P.: Minimal sufficient explanations for factored markov decision processes. In: 19th Int. Conf. on Automated Planning and Scheduling. (2009) 194–200
14. Zeller, A.: Isolating cause-effect chains from computer programs. In: 10th ACM SIGSOFT Symposium on Foundations of Software Engineering. (2002) 1–10
15. Pope, B.: Declarative debugging with buddha. In: 5th Int. Conf. on Advanced Functional Programming. (2005) 273–308
16. Gill, A.: Debugging haskell by observing intermediate data structures. Electronic Notes in Theoretical Computer Science **41**(1) (2001) 1
17. Ko, A.J., Myers, B.A.: Finding causes of program output with the java whyline. In: SIGCHI Conf. on Human Factors in Computing Systems. (2009) 1569–1578
18. Abraham, R., Erwig, M.: GoalDebug: A Spreadsheet Debugger for End Users. In: 29th IEEE Int. Conf. on Software Engineering. (2007) 251–260
19. Ochoa, C., Silva, J., Vidal, G.: Dynamic slicing based on redex trails. In: ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. (2004) 123–134
20. Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: Functional programs that explain their work. In: 17th ACM SIGPLAN Int. Conf. on Functional Programming. (2012) 365–376
21. Ricciotti, W., Stolarek, J., Perera, R., Cheney, J.: Imperative functional programs that explain their work. Proc. ACM Program. Lang. **1** (2017) 14:1–14:28