Uncanny Valleys in Declarative Language Design

Mark S. Miller[1], Daniel Von Dincklage[2], Vuk Ercegovac[3], and Brian Chin[4]

1 **Google, Inc.** erights@google.com
2 **Google, Inc.** danielvd@google.com
3 **Google, Inc.** vuke@google.com
4 **Google, Inc.** brianchin@google.com

Abstract

When people write programs in conventional programming languages, they over-specify how to solve the problem they have in mind. Over-specification prevents the language's implementation from making many optimization decisions, leaving programmers with this burden. In more declarative languages, programmers over-specify less, enabling the implementation to make more choices for them. As these decisions improve, programmers shift more attention from implementation to their real problems. This process easily overshoots. When under-specified programs almost always work well enough, programmers rarely need to think about implementation details. As their understanding of implementation choices atrophies, the controls provided so they can override these decisions become obscure.

Our declarative language project, Yedalog, is in the midst of this dilemma. The improvements in question make our users more productive, so we cannot simply retreat back towards over-specification. To proceed forward instead, we must meet some of the expectations we prematurely provoked, and our implementation's behavior must help users learn expectations more aligned with our intended semantics.

These are general issues. Discussing their concrete manifestation in Yedalog should help other declarative systems that come to face these issues.

1998 ACM Subject Classification D.3.2 Constraint and Logic Languages

Keywords and phrases Declarative logic programming language

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Background

Kowalski famously observed[4] that "Algorithm = Logic + Control". Declarative languages enable users to ask logical questions. The "logic" of a declarative program is a description of what a correct answer looks like. The "control" explains how to compute answers that satisfy that description. Often these two components are not separate parts of a declarative program but distinct ways of reading a program. For example, the *declarative reading* of a Haskell program considers Haskell functions to be the mathematical functions they seem. The *operational reading* sees these functions as code explaining how to compute results from input arguments. In a declarative language, the computed results must be within the declarative reading's description.

In declarative language design, there is an inescapable tradeoff between expressiveness and automation. *General purpose* declarative languages such as Haskell and Prolog are very expressive but with limited automation—incrementally more complex questions can be asked for incrementally more effort; but their users are responsible for controlling the direction of execution. *Special purpose* languages like Datalog and SQL are highly automated but with


© Mark S. Miller, Daniel von Dincklage, Vuk Ercegovac, and Brian Chin;
licensed under Creative Commons License CC-BY

Listing 1 A trivial Horn-clause rule

```
aa(X,Z) :- bb(X,Y), cc(Y,Z).
```

limited expressiveness—many questions cannot be asked, but of those that can, users can leave operational concerns to the implementation.[1]

As general purpose languages improve their automation, and as special purpose languages improve their expressiveness, the gap between them will narrow but not close. These improvements create dilemmas. For which questions should users let the language figure out how to compute answers? When should users still make operational choices, and how should they express them? User uncertainty about their remaining operational responsibility is the uncanny valley of concern to this paper.

Yedalog[2] is a general-purpose Datalog-like language for scalable exploratory data analysis—for asking questions of large semi-structured data sets. Yedalog has been used in production for several years by several teams. Nevertheless, the Yedalog implementation makes many control decisions that general purpose languages normally leave to their users. To reduce user uncertainty about their remaining operational responsibility, we designed and documented an informal operational model. But people learn by experience more than explanation. From their experience using Yedalog—seeing which of their programs work or do not—our users learned a different model. The model they learned assumes some forms of automation beyond any we had planned to support.

As we change Yedalog to meet these unanticipated expectations, we must better anticipate what models users will learn from Yedalog's new behavior. These models must provide users with better clarity about how to use Yedalog well. New user expectations will in turn affect what further changes we make to Yedalog. This feedback loop shapes our trajectory through the design space. How should we steer it and where will it lead?

1.1 Logic Programming

To locate Yedalog in the declarative language design space, we rapidly zoom in from declarative languages to logic programming languages, to Horn-clause logic programming languages, to Datalog-like languages, and finally to Yedalog.

Among declarative languages, Kowalski's distinction is especially crisp for logic programming languages. Logic programs consist of *facts*, *rules*, and *queries*. Queries produce *answers*. In the operational reading, facts are data, rules are procedures, queries are calls binding input parameters, producing answers binding output parameters. In the declarative reading, facts are propositions assumed to be true, rules express how some propositions imply other propositions, queries are parameterized hypotheses, and answers are their parameterizations, to be proved from those facts and rules. Execution is the search for such proofs.

How search proceeds through this search space matters. The Horn-clause logic program-

[1] "Declarative" is sometimes used to mean what this paper calls "highly automated"—the ability of users to avoid operational concerns. In this paper, "declarative" means only the ability of users to be confident that what is computed corresponds to what they have logically described. In our terms, Haskell and Prolog are declarative. Datalog and SQL are both declarative and highly automated. All these languages, as well as Yedalog, include unsound escape hatches that compromise this confidence. But so long as these escape hatches are explicit and visibly absent from most programs, we still consider these to be declarative languages.

Listing 2 Length of shortest path

```
path(X,Z) min= edge(X,Z).
path(X,Z) min= path(X,Y) + edge(Y,Z).
```

ming languages—Prolog, Concurrent Prolog[8], Datalog—have essentially the same abstract syntax (e.g. Listing 1) with the same logical meaning, but make wildly different operational choices. To the programmer trying to get practical work done, these languages feel vastly different from each other.

Programming in Prolog resembles conventional call-return programming augmented with backtracking search, where the programmer must write the program according to the precise order execution should proceed. *When `aa` is called, call `bb` and then `cc`.* Programming in Concurrent Prolog resembles actors exchanging asynchronous messages. *When `aa`'s inputs are ready, run `bb` and `cc` in parallel, communicating on their shared variable `Y`.*

Programming in Datalog resembles database query languages like SQL, where facts are data-tables, rules create views, and queries are queries. *Join `bb` and `cc` to produce `aa`.* Datalog implementations have all the freedom of query planning that databases enjoy:

- `bb` might run first, generating `Y` values for `cc` to test
- `cc` might run first, generating `Y` values for `bb` to test
- `bb` and `cc` might both generate answers intersected on `Y`

This paper freely mixes logic and database terminology. A relational table is also a disjunction of facts. `bb(X,Y),cc(Y,Z)` is a conjunction and also a join. The multiple rules of `path` in Listing 2 are a disjunction and also a relational union. Yedalog, like many Datalog-like languages, supports aggregation. `path` expresses shortest path as a `min` aggregation over the disjunction of all path lengths. When lengths are known to be non-negative, some systems will implement it using Dijkstra's algorithm[3, 7].

1.2 Yedalog's Goals

Yedalog's focus is scalable data exploration. As our users spend less attention on how things execute, they spend more attention on asking questions and interpreting answers. We thus aim for the following goals[2]. These goals conflict, requiring us to make tradeoffs and compromises based on our sense of the costs and benefits.

Query planning freedom. Programs should not accidentally over-specify the implementation. The programmers' natural way of asking questions, crafted without attention to operational details, should leave the Yedalog implementation with enough freedom to make good choices.

Good optimization decisions. As Yedalog makes better implementation choices, users do not need to.

Query planning compatibility. In order to preserve query planning freedom, the program's observable behavior should be *compatible enough* under all decisions the implementation is allowed to make.

[2] In addition, to be more quickly understood by our audience, Yedalog has a more C-like surface syntax than we show in this paper.

Listing 3 Inferring orders and modes

```
# front has modes (in,in) and (out,in)
# reverse has modes (in,out) and (out,in)

back(E,L) :- front(E,R), reverse(R,L).
```

Usable operational controls. Automatic planning will sometimes be inadequate. We must provide users the tools they need to cope, such as operational knobs for overriding default decisions.

Section 2 explains why "Query planning freedom" needs a different understanding of "Query planning compatibility" than we expected, and how to provide it. Section 3 shows how to handle errors without violating query planning compatibility. Section 4 discusses uncanny valleys in design spaces. Section 5 concludes.

2 Inferring execution orders

In pure Datalog, predicates represent concrete data or computed views of data. These data-oriented predicates can be materialized as finite relational tables. By contrast, most computational predicates express an infinite relation among their parameters. Integer addition embodies an infinite set of triples. Factorial embodies an infinite set of pairs. To better support general-purpose use, Yedalog programs can freely mix data-oriented and computational predicates.

Each Yedalog predicate has a set of *modes*. Each mode says, for each parameter, whether the parameter is *input* or *output*. Finite data predicates support an all-out mode, where all parameters are output parameters. Infinite predicates can only support modes containing at least one input parameter. For an input parameter, the caller must provide concrete data.

An output parameter is strictly more general: It can output data for the caller to use, or use data provided by the caller as input, by comparison or indexing. For example, the `edge` predicate of Listing 2 would normally be an all-out finite table indexed (at least) on its first column. Without input `edge` will enumerate all edges. With a first node as input, `edge` will lookup and efficiently enumerate only those edges emerging from that node.

Yedalog combines mode inference with mode-based reordering. The `back` predicate of Listing 3 says that `E` is at the back of `L` if it is at the front of `L` reversed. The `front` and `reverse` predicates represent infinite relations since there are infinitely many possible lists. They support the modes stated in Listing 3. Since `front` demands a binding for `R` and `reverse` can provide one, the only feasible orders run `reverse`'s `(out,in)` mode first, generating `R` bindings for either mode of `front` to use. From the feasible orders of `back`'s bodies, we can infer the possible modes of `back`: `(in,in)` and `(out,in)`.

We avoid explosive search by inferring only the minimal set of most general modes. The modes supported by `reverse` are already its minimal set, since neither `(in,out)` nor `(out,in)` is more general than the other. For `back`, we infer only `(out,in)` since it is strictly more general than `(in,in)`.

We provide our users knobs to make operational decisions. Some are subtle: Our `+` and `-` operators are irreversible, leading users naturally to force the right choice between top-down and bottom-up when it makes a difference, as in the following example. Others are explicit: Conjunction order is unspecified by default, but we provide an `&&` operator to force left-to-right order. When should it be used? Forcing order destroys choices an implementation

Listing 4 Factorials: The bad, bad, worse, and ugly

```
factA(0) = 1.
factA(M+1) = (M+1) * factA(M).

factB(0) = 1.
factB(N) = N * factB(N-1).

factC(0) = 1.
factC(N) = N * factC(N-1) :- N >= 1.

factD(0) = 1.
factD(N) = (N >= 1 && N * factD(N-1)).
```

could have used well. Not forcing can occasionally leave programs incorrect. How do users decide?

Listing 4 shows four versions of recursive factorial that are all declaratively correct. `factA` executes bottom-up, starting at 0, and reliably fails to terminate as it enumerates larger numbers[3]. `factB` executes top-down, starting with the requested argument, and reliably fails to terminate as it enumerates ever smaller negative numbers.

By "bottom-up" we mean computing forward from known facts, like the factorial of zero, to implied facts like the factorial of one, until reaching the query. By "top-down" we mean computing backward from the initial query like the factorial of 7, to subgoals like the factorial of 6, until reaching known facts like the factorial of zero[4].

As an informal experiment, we asked our users to write the standard introductory recursive factorial function. 80% submitted variations of `factC`. On the current Yedalog implementation, `factC` always happens to execute correctly and pass any possible tests. However, Yedalog is free to make either the recursive call first or the `(N >= 1)` test first. Had `factC` recurred first, it would not have terminated. Instead, it would speculatively enumerate ever smaller negative numbers before the `(N >= 1)` test that would disqualify these speculations. By contrast, `factD` is the correct Yedalog program no one wrote, which uses `&&` to avoid this hazard. Although we have documented these issues well, *none* of our survey responses even mentioned this ordering issue as a possible concern.

We documented an operational model in which `factC` might not terminate. Our users understood a model in which `factC` always works, which is a more accurate model of what our implementation does. Which is more right? As we change Yedalog, which of these models do we start with? These questions led us to better understand query planning freedom and query planning compatibility.

2.1 How much freedom to plan badly?

No matter what operational model language designers document, *the operational model implementors implement must support those user expectations that implementors dare not break.* At the same time, to enable implementors the freedom to choose among more good

[3] Although pure Datalog programs always terminate, Datalog programs with arithmetic may not.

[4] Yodalog actually implements `factB` by magic sets[1]. Magic sets is often described as bottom-up because it reuses the machinery of bottom-up execution. But since magic sets mostly work backward from queries to facts, in this paper we do not distinguish between top-down and magic sets, using "top-down" for both.

Listing 5 Unrolling multi-recursion

```
fibA(0) = 0.
fibA(1) = 1.
fibA(N) = fibA(N-1) + fibA(N-2) :- N >= 2.

fibB(0) = 0.
fibB(1) = 1.
fibB(N) = (N >= 2 && X == fibC(N-1) && Y == fibC(N-2) && X+Y).

fibC(0) = 0.
fibC(1) = 1.
fibC(N) = (N >= 2 && Y == fibB(N-2) && X == fibB(N-1) && X+Y).
```

plans, the operational model must also allow them to choose among more bad plans. Query planning compatibility helps us navigate the conflict.

Under all allowed implementation choices, `factA` and `factB` never work and `factD` always works, which upholds query planning compatibility. But saying that `factC` may or may not terminate denies reality. The fact that many patterns like `factC` always execute well today means that we dare not break them. Put this way, query planning compatibility is not so much a goal to achieve as a way to understand what query planning freedom is already lost.

Of course, we are not concerned about breaking `factC` itself. No one wrote this particular program until we asked. Our users wrote this specific program because of expectations they learned from some larger category of programs. They form these categories by generalizing over many concrete experiences. How they generalize depends on what they find intuitive. From an HCI (human computer interaction) perspective none of this is surprising; but programming languages raise the stakes. Expectations people learn from interactive use they adjust and relearn under continued use. By contrast, widespread programmer expectations get baked into large numbers of programs.

Of two observably different outcomes X and Y, we say Y is *compatible enough* with X when the expectations users learn from X do not deter implementors from causing Y. "Compatible enough" is thus always a judgement call, weighing the costs of breaking X expectations *vs.* the benefits of Y. This applies to performance as well as correctness. As we change Yedalog's implementation to make better choices in general, we might make some previously-efficient programs somewhat slower, but we dare not impose prohibitive costs on patterns already in widespread use.

"Compatible enough" is directional: No non-malicious user minds if a previously non-terminating program starts to work or a previously expensive program become cheaper. Due to the same directionality, implementors should be aware that each improvement is also a potential commitment, cutting off their freedom to make other choices. At every stage, we should rationalize our commitments back into our language design, in order to shape what freedom usefully remains[9].

The next section explain such an improvement and evaluates it by these criteria.

2.2 Unrolling multi-recursion

Since `factC` already works in the implementation, how can we change our model so that `factC` must work in all implementations? For `factC` itself we can do so trivially. Yedalog's stratification analysis already distinguishes potentially recursive calls from statically non-recursive calls. If we require recursive calls to be scheduled as late as feasible—which a good

planner would do anyway—then `factC` becomes correct.

Any intuitive category that includes `factC` also includes `fibA` from Listing 5. However, `fibA` is multi-recursive. It makes more than one potentially recursive call. They cannot both go last. As far as the implementation knows, either conjunct, if consistently run first, might never terminate even if the other conjunct would have caused an early failure.

Computation within each conjunct of a conjunction is *speculative*—only relevant when none of the other conjuncts fails. Speculative execution in hardware works because speculation checks cannot be indefinitely postponed. We could get a similar effect by specifying fairness among conjuncts, so speculation checks cannot starve. We have found a cheap approximation of fairness: Unroll a multi-recursive predicate like `fibA` into mutually multi-recursive predicates like `fibB` and `fibC`, where we rotate among the possible orders of which recursive call comes first.

The unrolled implementation does have a real performance cost: `fibA` has one memo table, reducing the naively exponential costs to linear. The unrolled form has two memo tables, doubling the number of misses we pay for. Only multi-recursive predicates pay this cost, which is linear only in the width of the multi-recursion. Wide multi-recursion is rare, so these costs are minor.

This unrolling will not cause previously-working programs to become non-terminating. It will cause some previously non-terminating programs to become correct, which sounds good. However, such "improvements" can do more harm than good. If an intuitive general category of code reliably does not terminate today, like the categories containing `factA` or `factB`, then we would muddy the waters with an "improvement" that allows some programs in such a category to work under some implementations, unless it requires all programs in that category to work on all implementations. As far as we can tell, this unrolling technique does not muddy the waters. Every general category that previously had reliably failed will continue to reliably fail.

This unrolling technique implements only a static approximation of fairness. This raises some interesting issues.

- How do we specify the approximation of fairness that this unrolling implements? We do not want to specify the unrolling technique itself because we want to preserve the freedom to achieve the same benefit by other means.
- For what programs is this approximation inadequate? We expect the accidental occurrence of such programs to be exceedingly rare, which would make these occurrences that much more uncanny when they do occur. No matter what we specify, we should expect users to learn expectations that only true fairness could implement.
- Can we close this remaining gap—implement true fairness for those rare cases—without significant cost to other programs?

Despite these open issues, for our purposes this unrolling technique is good enough. Other projects with different tradeoffs may judge these same issues differently.

3 Errors as noisy failures

In real programs, deployed in production and interacting with a wide variety of systems, many things can go wrong. Say a filename is misspelled. The parts of the program that would process the contents of the file are, declaratively, queries about the contents of a file with that name. Since there is no file with that name, these queries have no answers, i.e, they fail. Failure is normally silent, but a surprising failure that violates programmer expectations needs to alert the programmer, so that the likely problem can be fixed.

Listing 6 Making failure noisy

```
fibE(N) = fibA(N);
fibE(N) = raise(‘Must not be negative: $N‘) :- N < 0.

qq(M,N) = fibA(M) + fibE(N);
```

The `fibA` predicate of Listing 5 fails silently on negative input. The `fibE` predicate in Listing 6 acts just like `fibA` except that, on negative input, its `raise` expression fails and reports an error. To account for this, we extend our operational model to say that a query has some number of answers and reports some number of errors. A query that has no answers, fails. A query that reports no errors is silent. On negative input `fibE` produces a noisy failure. Errors have no declarative significance, so `fibA` and `fibE` have the same logical meaning. But `fibE` also produces diagnostic information. To route this diagnostic information appropriately, we must determine how errors propagate through Yedalog's constructs. Our error design has the following goals:

Suppress error storms. In a sharded computation, such as a large map-reduce job spread out over many machines, one underlying problem might trigger a massive number of errors, although most contain no new information.

Preserve at least one diagnostic. To suppress error storms, we discard tremendous numbers of errors. But we must not discard all of them. Few things are more frustrating than a program that silently behaves badly.

Do not make non-erroneous execution significantly slower. Errors are for exceptional cases we hope happen rarely. If support for occasional errors slows down the common case, we have made a bad tradeoff.

Do not make erroneous execution explosively slower. Although we allow error handling to be expensive, this is not a blank check.

Our error design should also respect the following general efficiency goals:

Stop conjunctions early on failure. The body of `qq` in Listing 6 calls both `fibA` and `fibE` in a conjunction. Whichever goal runs first might fail, rendering the other irrelevant. Efficient execution should be able to skip such irrelevant code.

Allow disjunctions to stop early on saturation. A disjunction *saturates* when no further disjuncts could change the outcome. The efficiency of Dijkstra's algorithm requires `path` to stop examining alternative paths that cannot further reduce the minimum. It is not realistic to require such optimizations in general, but we must allow them.

To suppress error storms while preserving at least one diagnostic, we allow errors to be consolidated. When a query reports at least one error, all errors but one may be discarded, preserving the property that it reports at least one error. Different plans may result in different errors being discarded. Does this violate query planning compatibility?

By convention, Yedalog errors contain only diagnostic information meant for human interpretation. Users may learn to expect specific errors, but usually they fix the indicated problem rather than write programs that depend on errors to occur. We have not encountered Yedalog programs that depend on the content of the errors that were reported. We thus consider reporting at least one error under one plan to be compatible enough with reporting at least one error, any error, under another plan.

Listing 7 Holding speculative errors

```
  (try {
      dd(X)
   } catch (E) {
      (ee(X),ff(X),gg(X)) && raise(E)
   }) && (ee(X),ff(X),gg(X))
```

Listing 8 Folding the holding of speculative errors

```
  (try {
      dd(X)
   } catch (E) {
      eTail(X) && raise(E)
   }) && eTail(X)

  eTail(X) = (try {
      ee(X)
   } catch (E) {
      fTail(X) && raise(E)
   }) && fTail(X)).
etc...
```

3.1 Errors in conjuncts

The goal "Stop conjunctions early on failure", taken literally, conflicts with query planning compatibility. To resolve the conflict, we must split silent failures from noisy failures. Say Yedalog's static analysis conservatively assumes that either `fibA` or `fibE` may fail and that either may report an error. Yedalog certainly has the freedom to run this conjunction in either order. If `fibA` runs first and fails, stopping the conjunction early, the error that `fibE` might have reported is not noticed. The conjunction as a whole would produce a silent failure. On the other hand, if `fibE` runs first and produces a noisy failure, stopping the conjunction early, then the silent failure `fibA` might have produced would not be noticed. If `fibE`'s error propagates anyway, then the conjunction as a whole produces a noisy failure.

This violates query planning compatibility. For a conjunction, the difference between silent and noisy failure is too surprising. This violation is not just a problem in theory. We became aware of the issue when correct programs started reporting errors that "could not happen", confusing everyone. This reinforces our sense that conjuncts are seen as speculative, and conjunctive failure as a failed speculation. *What happens in a failed speculation stays in a failed speculation.*

The efficiency motivation for "Stop conjunctions early on failure" applies only to silent failure. Since our efficiency goals require silent failures to stop early, query planning compatibility demands that noisy failures cannot. After `fibE` reports an error, Yedalog must treat this error as speculative, with `fibA` as the speculation check. We must execute `fibA` just enough to determine if it would have any answers, if it had run first. We do not care what the answers are or if it would have produced any more answers. If `fibA` produces any answers, then the conjunction as a whole can fail reporting `fibE`'s errors. If `fibA` fails silently, the conjunction must as well. This "unnecessary" execution of `fibA` may be expensive, but not explosively so. It only happens when the implementation was allowed to run `fibA` first and pay those costs.

What about conjunctions with data dependencies, such as `dd(X),ee(X),ff(X),gg(X)`,

where the named predicates are out-moded? Each may be used to generate `X` values or to test them. Under normal conditions, whichever executes first would generate and the rest would test. But any may also report errors.

To ensure that speculative errors only propagate once the speculation commits, the compiler could generate code approximately like that in Listing 7, where each of the remaining three-way conjunctions must be similarly expanded. To avoid an exponential expansion, we first fold each remaining conjunction into a separate predicate as shown in Listing 8. This has no explosive costs. But it is too expensive for the completely non-erroneous case—the outermost `&&` chain. Instead, we will leave this one chain fully unfolded.

3.2 Errors in disjuncts

A disjunction saturates when no further disjuncts would change the outcome. Any disjunction under a negation immediately saturates on the first answer. This answer establishes that the disjunction succeeds, allowing the negation to immediately fail, not caring what the answer is or if there are any more. We can realize some of these optimization opportunities more easily than others, so we allow disjunctions to stop early on saturation without requiring them to do so. We wish to preserve the query planning freedom to realize more of these opportunities over time.

Allowing disjuncts to stop early on saturation, by duality, should have the same conflict between efficiency, query planning compatibility, and preserving diagnostics. The dual solution would be to hold the contributions from a noisy disjunct—both its answers and at least one error—to see if the remaining disjunction would saturate silently. If it does, the noisy disjunct could have been skipped under other possible plans.

Returning to the shortest path example of Listing 2, say that the `edge` predicate, when asked for the length of a certain edge, answers and reports an error. If this edge lies on the shortest path, and if no path without this edge is tied for shortest, then the search could not have saturated without asking about this edge. Otherwise, depending on the algorithm used and the non-deterministic order in which edges were examined, a possible plan might not ask about this edge, not notice the error it would report, saturate, and silently answer with the minimal path. For the same graph and the same program, another possible plan would ask about this edge, notice the error, take its length into account, and proceed until saturating to the same answer. If it propagates this error, then a program that was silently succeeding might start reporting errors even when run on the same data.

Were we to apply the same standard of query planning compatibility that we applied to conjunctions above, and to apply the dual solution, we would postpone consideration of this edge until everything else settles down, giving us a candidate non-erroneous shortest path length. We would then contribute back in the postponed edge length and wait for the algorithm to settle again. If it settles on a shorter length, then we propagate both this error and the shorter path length. Otherwise we would silently report the unchanged candidate path length.

Should we bother with this extra bookkeeping, to avoid this observable difference of outcomes? The duality hides an important psychological difference: Disjuncts are not speculative. The success of one disjunct does not trigger an expectation that the other disjuncts "could not happen" but merely that they "might not happen". We have not found errors from unnecessary disjuncts to cause confusion in practice. Thus, we hold disjuncts to a lower standard of compatibility than we require of conjuncts.

4 Discussion

The original uncanny valley[5], in the context of robotics and computer graphics, predicted how a pattern of confused expectations leads to a feeling of creepiness. Their valley is a transient dip in affinity along a trajectory of progress towards lifelike human portrayals. Before the valley lies the pleasantness of a cute toy. After the valley are portrayals so lifelike they continue to amaze. To progress to that achievement, one must journey through the valley, where portrayals are good enough to provoke perceptual expectations that they then disappoint. We use this as a loose metaphor; our concern is not creepiness.

In declarative language design, we start at two pleasantly stable points in the design space. The first is occupied by expressive general purpose languages, in which users can ask any question but have full responsibility for figuring out how to compute an answer. These users discharge their responsibility without confusion by programming in terms of clear operational models. These programs over-specify, foreclosing on many optimization opportunities, wasting both human attention and computational resources. The second stable point is occupied by highly automated special purpose languages whose users do not need any operational model, leaving implementations free to use a wide range of fancy optimizations that need not be explained, in order to answer a limited range of questions.

From these two stable points, we see in the distance the promise of a third: A general purpose language in which users can ask many questions without operational concern, understand when they do need to make operational decisions, and understand how to express them. To find this third point we entered the valley, where operational controls are needed so rarely that they are expected even less. Despite this mismatch, our users are already much more productive, so we proceed.

Other languages are on similar journeys. Dyna[3] in particular entered this valley ahead of us and helped us find our footing. Software engineering has many uncanny valleys. A vivid example outside of language design is refactoring IDEs.

Refactoring IDEs were first invented and used for Smalltalk, a dynamically typed language. Without static types, automated refactorings have many false hits, so refactoring interactions always involve the programmer reviewing each decision. Programmers learn by doing. From the experience using these tools, programmers rapidly learn that they need to carefully decide whether to approve or reject each individual change.

Refactoring IDEs for Java use its static types to make many decisions reliably. For example, when changing the order of a function's parameters, the IDE can correctly identify exactly the call sites of this function, with no false hits and (in the absence of reflection) no false misses. Nevertheless, when it reorders argument expressions at these call sites it still might break the program—these argument expressions might now perform their side effects in the wrong order. However, this happens so rarely that most programmers never experience it. Programmers learn by doing. From these experiences, programmers learn to assume these refactorings are correct and not to bother reviewing each individual call site[6].

Should we make these refactorings less reliable, so programmers stop learning that they are more reliable than they are? Hardly. Rather, this example illustrates that we would have knowingly proceeded into this valley anyway because the benefits are worth it, and that retreat is not an attractive option. The only way out is through.

5 Conclusions

General purpose declarative languages, at first, leave many operational decisions to their programmers; but may absorb more operational responsibility over time. Declarative lan-

guages that absorb all this responsibility start special purpose; but may become more general over time. These paths lead to a dilemma, where these systems have gotten good enough that users perceive them, and use them, as more than they are. Expectations outrun reality. This problem is also an opportunity, to use the feedback between implementation behavior and user expectations to help shape both to be more aligned and, together, more effective.

Acknowledgements. For feedback and suggestions, we thank Yedalog's users, Peter Ludemann, Terry Stanley, Kevin Reid, and anonymous SNAPL referees.

References

1 Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15. ACM, 1985.

2 Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S Miller, Franz Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring knowledge at scale. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

3 Jason Eisner and Nathaniel W Filardo. Dyna: Extending Datalog for Modern AI (full version). 2011. URL: `https://www.cs.jhu.edu/~jason/papers/eisner+filardo.datalog11-long.pdf`.

4 Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.

5 Masahiro Mori, Karl F MacDorman, and Norri Kageki. The uncanny valley [from the field]. *IEEE Robotics & Automation Magazine*, 19(2):98–100, 2012 original 1970 Energy.

6 Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program metamorphosis. In *European Conference on Object-Oriented Programming*, pages 394–418. Springer, 2009.

7 Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

8 Ehud Y Shapiro. *Concurrent prolog: collected papers*. MIT press, 1987.

9 Allen Wirfs-Brock. Programming language standardization: Patterns for participation. In *5th Asian Conference on Pattern Languages of Programs*.