Markedly

A cartographic approach for mapping eDSL implementation costs

Matthew P. Ahrens
Tufts University
Medford, Massachusetts, USA
mahrens@cs.tufts.edu

Karl Cronburg
Tufts University
Medford, Massachusetts, USA
karl@cs.tufts.edu

Jeanne-Marie Musca
Tufts University
Medford, Massachusetts, USA
jmusca@cs.tufts.edu

Abstract

The cost of implementing an embedded domain specific language (eDSL) depends on the eDSL development tools used to build it, but these costs are not readily apparent. Markedly enables developers to map elements of the eDSL implementation and attach costs to these elements. An implementation diverging from its design incurs four distinct types of costs: Preserve, Extend, Adhere, and Recognize.

To illustrate the Markedly approach, we calculate the costs of example eDSLs using a map for the Haskell language metaprogramming and host language toolsets. To evaluate this approach, we will assess how developers reason about costs while implementing an eDSL.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords programming languages, functional languages, domain specific languages, semantics

ACM Reference Format:
Matthew P. Ahrens, Karl Cronburg, and Jeanne-Marie Musca. 2017. Markedly: A cartographic approach for mapping eDSL implementation costs. Presented at *Workshop on Meta-Programming Techniques and Reflection (Meta'17)*. 5 pages.

1 Introduction

1.1 The Practice

EDSL developers using Haskell's language development toolset choose between host language constructs and metaprogramming tools when implementing their eDSLs. Host language constructs provide the eDSL developer direct *reuse* of the core Haskell semantics and GHC's extension constructs such as generics, type families, and data kinds. Haskell's metaprogramming tools include the Template Haskell Q Monad, algebraic data type AST representation, and libraries for parsing and compile time splicing[8, 10]. For simplicity, developers prefer to implement their eDSL using only one tool, but some eDSLs are better implemented by a composition of tools.

Meta'17, October 22, 2017, Vancouver, Canada

1.2 The Problem

When a tool requires extraneous effort to implement an eDSL feature, the implementation incurs an unexpected cost. Costs are difficult to track across stages of language development and make comparing the impact of implementation decisions difficult with respect to the parts of the design those decisions affect.

1.3 The Solution

EDSL developers will better construct and modify their implementation by:

- Decomposing the design into eDSL features
- Associating features with a tool implementation
- Identifying connections between implementations
- Track costs accumulated across connections

2 Methodology

Given a chosen deconstruction of an eDSL design into features, Markedly allows developers to calculate implementation costs. To illustrate, we apply Markedly techniques to a toy eDSL for *Identifiers* with the features:

- identifiers must match the regex [A-Z][a-z0-9]*
- identifiers must be unique in scope

For tracking costs during any stage of development – planning, constructing, or iterating – Markedly describes an informal algorithm for visualizing an eDSL implementation as a map.

2.1 Constructing the Map

Using Markedly, an eDSL developer constructs a map with regions representing language development tools. The developer marks points on a region when implementing part of the language in that tool. When planning, developers put one point per feature on the map and connect points with edges to represent program dataflow. In practice, developers may encounter instances where multiple tools may be needed for one feature, resulting in multiple points and edges. In iteration, developers may add to or modify the implementation, resulting in adding or moving points and edges.

For example, in Figures 1 and 2, the map partitions regions for the tools: Haskell semantics and Parsec parsing. The developers for the Figure 1 implementation mark both features with one point which represents implementing identifiers

as host language identifiers. In the Figure 2 implementation, developers mark the regex feature in the parser tool region as a parser implementation and the uniqueness feature in the Haskell semantics region as a Haskell set data structure. To share program data between the features, developers add an edge from the parser to the set data structure.

2.2 Metrics of cost

To help developers track the impact of their implementation decisions, Markedly accumulates costs along paths. To help developers focus on specific concerns of their implementation decisions, Markedly splits costs along four vectors, labeled PEAR.

2.2.1 Preserve

The heuristic for the preservation cost of a map element (points and edges) is *How many features in the design are compromised by this element?* An implementation element compromises a design feature by not enforcing a property of the design feature. In Figure 1, Haskell identifiers do not enforce the leading capital property of the regex feature, incurring a preservation cost of one.

2.2.2 Extend

The heuristic for the extension cost of a map element is *How many intermediate representations are not accessible from this element?* When the implementation of a map element does not expose transformed program data, then that element incurs an extension cost for each hidden representation. Suppose the representation of identifiers to check syntax was introduced and lost during error reporting in Figure 3. The error reporting point would incur an extension cost of one.

2.2.3 Adhere

The heuristic for the adherence cost of a map element is *How many expressions or syntactic annotations must be written in addition to simply implementing the feature?* Complexity of the function required to convert between tool representations informs this metric. For example, in Figure 3, the edge from the parser to the error correcting point must convert from the parser's representation to a string thus incurring an adherence cost of one against the regex feature.

2.2.4 Recognize

The heuristic for the recognition cost of a map element is *Is this element present in the design?* If the element is directly tied to at least one feature, it has a cost of zero; otherwise it has a cost of one. In the example in Figure 3, error reporting does not appear in the design, so the extra points and edges added to support that feature incur a cost of one, each.

2.3 Calculation and Comparison

A four-tuple of natural numbers represents the cost for a map element : (P, E, A, R). A map element is either a point

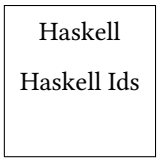

Figure 1. Haskell map for a host language embedding of *Identifiers*

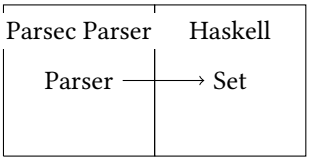

Figure 2. Haskell map for a metaprogram embedding of *Identifiers*

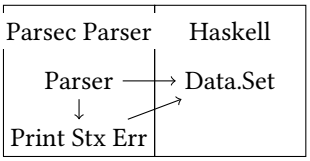

Figure 3. Haskell map for a metaprogram embedding of *Identifiers* with error reporting

or an edge. When multiple edges converge on a point, the path takes the maximum cost for each metric at that point. In the implementation of *Identifiers* in Figure 3 which adds error reporting, the accumulated cost along the parser edge and syntax edge are (0,0,0,1) and (0,1,0,2), and set edge cost is (0,0,0,0). Thus, $(argmax(0,0,1,1)(0,1,0,3)) + (0,0,0,0))$ calculates the path cost as (0,1,1,3).

Developers can now compare the costs of different implementations with proper separation of concerns. The *Identifiers* eDSL implementations have costs of (1,0,0,0) for the host language implementation, (0,0,0,1) for the metaprogram implementation, and (0,1,1,3) for the error handling implementation. Using these metrics, developers decide which implementation to maintain depending on what will be difficult to maintain. Connecting the terminal points of maps allows developers to compose maps together and track costs while iterating over their eDSLs.

3 Example eDSLs

3.1 Data Parsing

Consider an eDSL, *µPADS* [5], whose features are:

- best-effort (longest parse) left-to-right parsing
- first-fit (packrat) parsing

The implementation of (<|>) in Figure 4 takes two parser results and decides which one progresses. The map in Figure 5 defines the type signature, Nothing case, Both-Just case,

```
1 (<|>) :: Maybe [t] -> Maybe [t] -> Maybe [t]
2 (<|>) Nothing Nothing = Nothing
3 (<|>) (Just as) (Just bs)
4   | length as <= length bs  = Just as
5   | otherwise               = Just bs
6 (<|>) (Just as) _ = Just as
7 (<|>) _ (Just bs) = Just bs
```

Figure 4. Representations of the two *µPADS* features. Feature (1) is implemented by lines 4 and 5, and feature (2) is implemented by lines 4, 6, and 7.

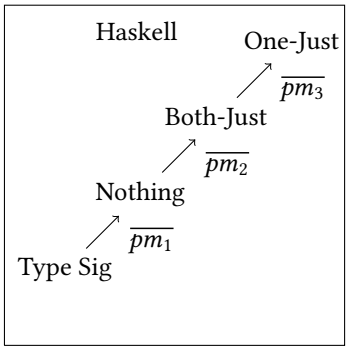

Figure 5. Haskell map for *µPADS*

TySig	P	E	A	R
Nothing	0	0	0	1
Both-Just	0	0	1	1
One-Just	0	0	0	0
$\overline{pm1}$	0	0	0	1
$\overline{pm2}$	0	0	0	1
$\overline{pm3}$	0	0	0	1
$\overline{TotalPath}$	0	0	1	5

Figure 6. The non-zero PEAR costs of *µPADS*

and One-Just case(s) as points and connects them via the standard Haskell tool of pattern matching (pm).

Line 4 in Figure 4 simply exists to adhere to the implied semantics of first-fit while implementing best-effort, incurring an adherence cost. The type representation, Nothing failure case, and evaluation order imposed by pattern matching are not reflected by features thus incurring a recognition cost.

The design could reduce the PEAR cost to zero by being more specific:

- best-effort (longest parse) left-to-right parsing preferred
- first-fit (packrat) parsing if results equivalent or only one result

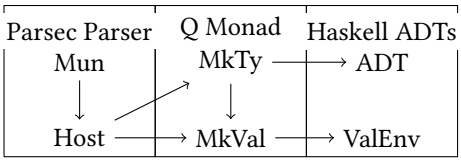

Figure 7. The Haskell map for *MunLang*

Element	P	E	A	R
Host	0	0	1	0
ADT	0	0	0	1
ValEnv	0	0	0	1
$\overline{MkTyMkVal}$	0	1	0	1
$\overline{MkTyADT}$	0	0	1	1
$\overline{TyConPath}$	0	0	2	2
$\overline{ValConPath}$	0	1	1	2

Figure 8. The non-zero PEAR costs of *MunLang*

- propagate failure if no result
- expect computation failure

3.2 Containers

Consider an eDSL *MunLang* that implements the *multi-union* data structure [4] The design of *MunLang* contains three features:

- Define a munion type with {: :} operators
- Construct a munion value with {: :} operators
- Splice types and values from host language

To define a munion type, specify a set of label-type pairs: $typeM_1 = \{: \tau_1\ l_1, \tau_2\ l_2, \tau_3\ l_3\ :\}$. Similarly, to construct munion values of type M_1, declare a subset of label-value pairs,: $V_1 = \{: l_1 = v_1, l_3 = v_2\ :\} :: M_1$

In the map in figure 7, the points *mun* and *host* enforce syntactic constraints for the munion and host language syntax, respectively. *MkTy* and *MkVal* points construct munion types and values, respectively. The *ADT* point represents Haskell type splicing and *ValEnv* represents Haskell value splicing.

The developers use the heuristics for each PEAR dimension to calculate the *PEAR* costs in Figure . From these costs, the developers document that the implementation "fills in" the underspecified host language representation apparent in the recognition cost, contains superfluous code to parse the host language apparent in the adherence cost, and hides the type environment apparent in the extension cost.

3.3 Type-Level Naturals

Consider an eDSL, *Numeric Type Annotations* whose features are:

```
data Nat = Z | S Nat
type Weighted (w :: Nat) a = a
type family (w1 :: Nat) :+: (w2 :: Nat) :: Nat
type instance Z :+: m = m
type instance (S n) :+: m = S (n :+: m)
($$) :: Weighted w1 (a -> b) -> Weighted w2 a
     -> Weighted (w1 :+: w2) b
($$) f a = f a
```

Figure 9. The deeply embedded implementation for Type Level Naturals

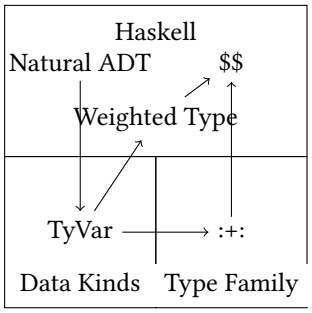

Figure 10. The Haskell map for Type Level Naturals

Element	P	E	A	R
Natural ADT	1	0	0	0
Weighted Type Alias	0	0	1	0
TyVar	0	0	0	1
:+:	0	0	1	0
$\overline{NatTyVar}$	0	0	0	1
$\overline{TyVarWeight}$	0	0	0	1
$\overline{TyVar:+:}$	0	0	0	1
$\overline{ProgPath}$	1	0	2	3

Figure 11. The non-zero PEAR costs of Type Level Naturals

- readable natural number weight types
- attach weights to arbitrary expressions
- summing weights across function application

A deep embedding represents the eDSL using the Haskell type system in Figure 9. The map in Figure 10 for this eDSL contains the regions of data kinds, type families and Haskell semantics.

In Figure 11, the implementation manifests a connection between numeric representation and weights incurring a recognition cost. Because the implementation must thread the type variable, $w :: Nat$, elements adding extra type variable annotations in addition to their features incur an adherence cost. The preservation cost occurs from the natural numbers Peano representation being difficult to read. The developers may add implementation details which make use of algebraic numerals, risking adherence costs if adding complicated parsing and printing expressions.

4 Evaluation

To evaluate our cartographic approach, developers will implement an eDSL in Haskell as part of a user study.

4.1 Given

Support for an overspecified eDSL feature evaluates preservation. The amount of code modified when adding a feature after initial implementation evaluates extension. The number of intermediate representations evaluates adherence. Abstracting the design from the implementation evaluates recognition.

4.2 Experiment

Half of participating developers will use the Haskell map that contains the maps from the eDSL examples and the costs of known paths between regions. Each developer will place eDSL features on the map and calculate the accumulated PEAR costs while iterating over their implementation.

5 Current Work and Conclusion

To bridge the gap between developer knowledge and automatic tool support, an artifact of Markedly would provide an interactive view where:

- A specification language encodes the design
- PEAR costs are automatically calculated
- A library provides common eDSL map elements
- Code templates are generated from a map instance

Language design workbenches associated with edges on a Markedly map by proving a unified representation for constructing an eDSL. Developers can apply Markedly to language design workbench eDSL implementations by representing the workbench as a large region on the map and focusing on the features that require tools outside the workbench as points and edges that leave the region.

The realities of implementing an eDSL incur various kinds of costs dependent on the stage of development. In the Markedly approach, we categorize these costs into the four PEAR metrics. Markedly describes costs for eDSL implementation tools and our Haskell eDSL examples provide insight into instances of the approach. A user study would allow us to determine if the approach and Haskell specific artifacts facilitate reasoning about implementation choices with respect to PEAR costs.

Acknowledgments

Markedly is sponsored by the Defense Advanced Research Projects Agency (contract: FA8750-15-2-0033). Markedly does not necessarily reflect the position or the policy of the US Government, and no official endorsement should be inferred.

References

[1] Walter Cazzola and Albert Shaqiri. 2016. Modularity and Optimization in Synergy. In *Proceedings of the 15th International Conference on Modularity (MODULARITY 2016)*. ACM, New York, NY, USA, 70–81. https://doi.org/10.1145/2889443.2889445

[2] Walter Cazzola and Ivan Speziale. 2009. Sectional Domain Specific Languages. In *Proceedings of the 4th Workshop on Domain-specific Aspect Languages (DSAL '09)*. ACM, New York, NY, USA, 11–14. https://doi.org/10.1145/1509307.1509311

[3] Martin Churchill, Peter D. Mosses, and Paolo Torrini. 2014. Reusable Components of Semantic Specifications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/2577080.2577099

[4] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. 2004. Hancock: A Language for Analyzing Transactional Data Streams. *ACM Trans. Program. Lang. Syst.* 26, 2 (March 2004), 301–338. https://doi.org/10.1145/973097.973100

[5] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 295–304. https://doi.org/10.1145/1065010.1065046

[6] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-yang: Concealing the Deep Embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/2658761.2658771

[7] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Conference on Software Product Line (SPLC '15)*. ACM, New York, NY, USA, 71–80. https://doi.org/10.1145/2791060.2791092

[8] Geoffrey Mainland. 2007. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/1291201.1291211

[9] Jan Midtgaard, Norman Ramsey, and Bradford Larsen. 2013. Engineering Definitional Interpreters. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (PPDP '13)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2505879.2505894

[10] Tim Sheard and Simon Peyton Jones. 2002. Template Metaprogramming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691