K3: Language Design for Building Multi-Platform, Domain-Specific Runtimes

P. C. Shyamshankar, Zachary Palmer, Yanif Ahmad
Department of Computer Science, Johns Hopkins University
{pcshyamshankar,zachary.palmer,yanif}@jhu.edu

Abstract

We pose the question: while it is natural to embed specialized systems functionality (e.g. databases) in programming languages, how can programming language techniques simplify constructing and analyzing scalable computing systems? This topic is of increasing importance with the ongoing specialization of data and computing models in databases, systems, scientific computing and AI. We introduce K3, an event-driven language to represent, compose and analyze distributed runtimes. K3's goals are twofold, first to facilitate the synthesis of scalable *domain-specific runtimes* that take holistic advantage of domain properties, and second, to coordinate existing popular runtimes and platforms in a declarative fashion under a unified optimization framework. We present K3's distributed event-driven computation model, and its annotation model for data, control, execution and optimization concerns.

1. Introduction

The emergence of domain-specific languages (DSLs) and systems is intrinsically tied to the spread of computational methods throughout business, science, engineering, medicine and numerous other disciplines. DSLs do not require general programming proficiency from users, while benefiting from restricted semantics for safety and correctness, as well as optimization and efficiency.

However, from an execution perspective, DSLs continue to utilize the host language's computation infrastructure as designed for the general-purpose setting. Consider a classical runtime: a relational database (RDBMS). RDBMS are often used to embed specialized data models, such as for stream and graph processing. At scale, the architectural rigidity [13] of monolithic "one-size-fits-all" DBMS (e.g. transactions, set-at-a-time execution, storage schemes) necessitates lightweight, specialized runtimes.

We present K3[1], an expressive language and toolchain for building scalable data- and compute-intensive runtimes that exploit domain-specific semantics throughout the software stack. Our goals complement the literature on language support for embedding (e.g. multi-stage programming [14] and language virtualization [12]), focusing on simpler DSL runtime synthesis, and scalability. K3 addresses the needs of *long-running* event and data-driven processing, a computation model that captures *services*, from systems daemons to cloud-based web applications and data-intensive scientific computing and analytics.

K3 is a *multi-platform compiler* that targets a range of parallel programming frameworks and distributed data stores (as well as a C++, OpenMP, MPI, and CUDA stack) as its execution primitives, to achieve a variety of scalability and fault-tolerance characteristics. Today, such "platforms-as-libraries" composition is achieved with scripting languages such as Python which are agnostic to the deployment and performance traits of the underlying platforms, and perform little analysis-driven or statistics-driven optimization.

The central challenges in K3 are in designing an expressive representation for diverse data and computation models, and the high-level specification of system design and deployment aspects to realize flexibility and scalability. These challenges are present alongside the gamut of language design aspects, from type system design to extensible, verified synthesis and code generation, program analyses, and optimization strategies. K3's ongoing design and approach to these challenges is influenced by compiling runtimes for three declarative languages, SQL (as the compiler for DBToaster [1]), Dyna [5] (a weighted logic programming language for NLP and AI applications), and BLOG [11] (Bayesian Logic, a programming language for probabilistic graphical models).

This paper introduces the language model of K3 (Section 2), its extensible, typed annotation system for declarative specification of program and data properties (Sections 3- 4), and finally optimization and synthesis (Section 5) based on annotations, that addresses adaptive optimization for a long-running event loop, and automatically handles runtime parameter tuning and platform integration.

2. Language Overview

Computation Model The basic unit of computation in a K3 program is a *trigger*, which is invoked with a message as its argument. Triggers can access per-process global state, and can declare local state which will not persist after the end of the trigger. The body of a trigger is a side-effecting expression performing state transformation, and contains only acyclic code defined by primitive operations (without higher-order functions) and computation with bounded iteration over collections. These restrictions guarantee the termination of a trigger. K3's core primitives are described in Figure 1.

Communication between triggers takes place through an asynchronous message passing model, with messages consisting of the arguments to the recipient trigger. This message passing model can be used to represent complex control flows such as recursion, while keeping the core computations acyclic. The separation of complex control flow from basic control flow allows several optimizations such as deforestation and fusion to be applied aggressively. All triggers are tail-calls by construction; each successive invocation of the trigger is independent and uses only the message contents sent to it and the global state.

This model is general enough that it can express the computation models of several other execution frameworks quite well, allowing for a great deal of flexibility during optimization and code generation. As a toy example of messaging, we can compute the Fibonacci sequence as follows:

```
trigger fibonacci(n:int, a:int, b:int) =
    if n == 0 then send(sink, b)
    else send(fibonacci, n - 1, b, a + b)
```

[1] The name K3 is inspired by the Kleisli [3] language and the connections drawn between complex objects, nested collections and monads.

Variables x,y,z ***Operators*** $\oplus ::= \texttt{+} \mid \texttt{-} \mid \times \mid \ldots$

Constants $c ::= 1, 2, \ldots\ '\texttt{a}', '\texttt{b}', \ldots, \langle ip : port \rangle$

Functions $f ::= x \mid \lambda x.e$

Expressions

$$\begin{array}{lll}
e ::= & x \mid c \mid e \oplus e \mid (\, e, \bar{e}\,) \mid C & \textit{Primitives, tuples, collections} \\
\mid & f(e) & \textit{Function Application} \\
\mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \textit{Conditional} \\
\mid & \texttt{do}\ \{\ \bar{e}\ \} & \textit{Sequencing} \\
\mid & \texttt{send}(t\texttt{@}c, e) & \textit{Asynchronous Messaging} \\
\mid & e\ \texttt{@}\ \{\bar{x}\} & \textit{Annotated expressions}
\end{array}$$

Collection Operations

$$\begin{array}{lll}
C ::= & x \mid \texttt{[]} \mid \texttt{[}e\texttt{]} \mid C\ \texttt{++}\ C & \textit{Constructors} \\
\mid & C\ \texttt{+=}\ e & \textit{Insert} \\
\mid & C\ \texttt{-=}\ e & \textit{Delete} \\
\mid & C[e]\ \texttt{:=}\ e & \textit{Update by Value} \\
\mid & C[e] \leftarrow e & \textit{Update by Reference} \\
\mid & \texttt{map}\ (\, f, C\,) \mid \texttt{groupby}\ (\, f, f, e, C\,) \mid \ldots & \textit{Transformers} \\
\mid & \texttt{peek}(C) \mid C[\overline{e, \_}] & \textit{Accessors}
\end{array}$$

Programs

$$\begin{array}{lll}
t ::= & \texttt{trigger}\ x\ (\overline{y : \tau}) = (\bar{d}, e) & \textit{Trigger Decl.} \\
d ::= & x : \tau = e & \textit{Function and Value Decl.} \\
\mid & x : \texttt{collection}(\bar{\tau})\ \texttt{@}\ \{\bar{x}\} & \textit{Annotated Collection Decl.} \\
\mid & \texttt{annotation}\ x\ \{\ \bar{a}\ \} & \textit{Annotation Decl.} \\
a ::= & x : \tau & \textit{Interface Requirement} \\
\mid & x : \tau = e & \textit{Implementation Provision} \\
\mid & \texttt{schema}\ x : \tau & \textit{Schema Extension Decl.} \\
p ::= & \bar{d} \mid \bar{t} \mid \texttt{stream}\ [\overline{(x, v)}] \mid \texttt{loop}\ \texttt{<}\overline{I/O}\texttt{>} & \textit{Event loop}
\end{array}$$

Figure 1. K3 trigger, annotation and expression syntax.

In this example, the trigger will send itself a message if there is more computation to be performed, or will send the result to a sink for output if it is completed.

Data Model The K3 data model consists of purely functional intermediate data structures for operations within individual expressions, along with mutable data structures for global state and persistence across triggers.

The collection model is inspired by the complex object representation [3], providing for nested collections and a small but powerful set of collection transformers. Collections can further be extended with desired properties using annotations, as discussed in Section 4. Views, which are derived data structures, can also be expressed directly. This allows for the automatic generation of the triggers necessary to perform automatic view maintenance and incremental computation, as in the DBToaster Project [1].

Mutability in K3 is expressed through a Dual-Reference model, which addresses two different concerns with the capability of references. *Contained References* (`cref`s) are read-only references used to obtain a pointer to an element in a particular collection. Since these are read-only, the collection element pointed to by a `cref` can only be modified by issuing update instructions to the collection. This ensures that runtime invariants of the collection (such as the uniqueness of its elements) are maintained, while still allowing sharing within the collection. *Isolated References* (`iref`s) are read-write references that point to explicitly allocated data, which may exist outside a collection. These allow individual data elements to persist and be mutated arbitrarily.

3. Annotations

K3 uses annotations to encode domain-specific knowledge which relates to a program or its operating circumstances. Annotations are declarative in that they are a statement of properties, giving the optimizer flexibility in choosing the most efficient implementation which satisfies the expressed intent. In general, annotations can be applied to any part of a K3 program.

The goal of the K3 annotation model is to express composability within categories of annotations. Knowledge of the interaction between annotations allows the optimizer to determine the implementation effect of the annotations at the group level, rather than at the individual annotation, leading to more desirable global system properties. This sets the K3 annotation model apart from conventional macro systems and metaprogramming models, which do not consider multiple metaprograms together.

One classification of annotations is based on what components of a program they describe. Data annotations specify expected properties, interfaces and behaviors of K3 data structures. We discuss data structure annotations in more detail in Section 4. Control annotations provide knowledge about control flow patterns and desired system behaviors. These can include concurrency properties, fault-tolerance behaviors, debugging and profiling properties, among others. An example of an annotation describing runtime logging of a collection for resilience is discussed in Section 5. Examples of annotations in other classes are provided in Figure 2.

Constraint and Hint Annotations Annotations can also be classified on the basis of whether they express a constraint on the compiler or optimizer, or provide them a hint.

A constraint annotation is a directive to the compiler that it must ensure the maintenance of a property or invariant at runtime. It is a statement that the program will not be correct if the compiler does not respect the annotation. An example of a requirement annotation is uniqueness – a collection annotated with `@unique` indicates that the collection cannot contain duplicate elements. In general, integrity constraints are expressed through constraint annotations.

A suggestion annotation provides a hint to the optimizer as to possible properties it may exploit in order to generate more efficient code. These give the optimizer more flexibility, as they provide information that cannot be inferred from the code, but can be ignored if necessary, in favor of other opportunities.

4. Declarative Data Structures

In K3, collections are declared by specifying a *content schema*, a *depth*, and a set of annotations dictating the properties of the collection. The elements of each collection conform to the collection schema, which is a concatenation of the above-mentioned content schema and the collection's *structural schema*. The structural schema is initially empty but may be extended by collection annotations as necessary. This permits, e.g., an annotation which organizes a collection into a tree structure to add schema extensions without knowledge of the content schema or of the other structural extensions provided by other annotations. Such schema extensions are inspired by classic relational data structure encodings, but K3 encourages these encodings to be encapsulated in composable parts.

Composing Annotations An annotation is free to specify functions or schema extensions that it *requires* or to specify which of its definitions it *provides* to other annotations. K3 also permits requirements to be specified in terms of the requirements of other annotations; thus, an annotation which merely declares a list of requirements can be viewed as an interface. In this fashion, the annotations used to describe a collection are similar to inherited mixins in an object-oriented system. While programmers may manually

Data		Control and execution	
Integrity (Constraint)	**Efficiency (Hint)**	**Assurances (Constraint)**	**Scalability (Hint)**
Functional dependencies	Layout, and compression	Fault tolerance, checkpointing	Degrees of parallelism
Sortedness	Indexes, views	Service-level agreements	Vectorization
Orderedness	Allocation, GC		Scheduling
Referential integrity	Data placement and replication	Auditing and compliance	Autotuning heuristics
Concurrency	Lock granularity	Access control	Profiling

Figure 2. Examples of annotation categories.

specify the annotations which satisfy these requirements, one area of future research lies in how an optimizer might automatically select satisfactory annotations from a predefined set. This decision may be made based on system configuration, runtime metrics, or other similar information.

In addition to this fairly straightforward interaction, K3 annotations may also specify *hooks*: functions which are to be invoked in response to the invocation of other collection functions, much in the style of aspect-oriented programming [10]. Hooks may be used in a number of ways; they are, for instance, convenient for collecting usage metrics or checking runtime invariants. They are also particularly helpful in maintaining supporting data structures; an annotation for defining logarithmic-time lookup on a functional dependency, for instance, may add a hook to the collection's `add` method which updates the data structure used for the lookup. Such a mixin which could generally extend a data structure with functional dependency behavior would be extraordinarily awkward to implement in object-oriented languages. In general, hooks provide a means by which behavioral extensions can be composed much as the structural schema composes data extensions.

As an example, let us consider a B+-Tree to demonstrate the composition of annotations. A B+-Tree is a generalization of a Binary Search Tree, where each tree node can hold a block of entries. It is common in applications like databases and filesystems which can optimize the input/output of the individual blocks by fixing their size to the page size. There are two components to a description of a B+-Tree in this framework: a description of the data structure itself, and that of the interface over the structure which provides the user-facing functionality.

To model the B+-Tree structure, we need to consider the behaviors at both the block and tree level. At the block level, B+-Tree nodes have a capacity that is a parameter of the B+-Tree itself, and well-defined methods for handling overflow and underflow. We can encode those properties through corresponding annotations.

```
annotation Capacity(k:int) {
    schema size: int = 0;
    schema capacity: int = k;
    requires OverflowHandler;
}
annotation Fill(f:float) {
    schema fill: float = f;
    requires Capacity;
    requires UnderflowHandler;
}
```

The `Capacity` annotation declares metadata to keep track of the current size of the block, while the `Fill` annotation keeps track of the current occupancy relative to the total capacity. They declare their requirement of the `OverflowHandler` and `UnderflowHandler` annotations to handle the violation of their respective constraints. The implementations of these annotations would depend on the structure itself, and would be provided elsewhere. We can then describe each block of the B+-Tree as a collection of elements annotated with appropriate `Capacity`, `Fill`, `OverflowHandler` and `UnderflowHandler` annotations.

To describe the tree structure over the blocks, we need to use an annotation which adds a collection of child pointers to each block, and an interface to operate over them.

```
annotation Tree {
    schema children: Collection(self);
    ...;
}
```

In the above annotation, we use the keyword `self` to refer to the entire schema type of an element of the collection, including the extensions made by other annotations. There can be other notions of *self* (with different names) that other annotations may need, including the content schema by itself, or with just the extensions provided by the current annotation.

Having described the structure of the B+-Tree, we need to describe how the user-facing functionality of the B+-Tree is provided over this interface. This is done by using a specific B+-Tree annotation.

```
annotation BPTree {
    requires Tree;
    insert = { ... };
    update = { ... };
    delete = { ... };
}
```

The complete B+-Tree therefore is a collection of blocks, with the `Tree` and `BPTree` annotations in addition to the block annotations. By abstracting the functionality required by the `BPTree` interface, it would be possible to adapt a similiar interface over a structure other than a tree – such as a DAG – which also provided that functionality.

We can also extend the B+-Tree in other directions by including other annotations. For example, if we can define a fractal layout strategy for a collection and encapsulate that functionality in an annotation, we can attach that to each block of the B+-Tree to obtain a cache-conscious B+-Tree. Alternatively, we can attach the functionality of multi-version concurrency control to the B+-Tree by incorporating a logging system that records the modifications made to the data structure. Both these sets of functionality can be implemented in a data-structure agnostic way, and used independently of each other on the same collection.

Depth and Acyclicity Typechecking the annotations themselves is fairly straightforward. We use a subtype constraint system [4] for its flexibility in inference and its capacity to model the composition of the annotation declarations (in a fashion similar to inheritance). While such systems can often produce intractably complex constraints, subtyping in K3 is quite restricted: only collections themselves have subtyping, while subtyping over other structures (e.g. primitives and tuples) is naturally homomorphic. These restrictions serve to mitigate constraint complexity.

Typechecking collections, however, is somewhat more complex. As stated above, collections must be finite in structure to guarantee termination; operations over collections must therefore also be finite. But unlike triggers, collection functions may be recursive. We solve this problem with *depth constraints*, which enforce a variant of primitive recursion (and thus termination) on collection functions. While type-based termination has been accomplished with dependent typing [9], K3 aims for a simpler approach to aid user comprehension.

Whenever storage is allocated, it is assigned a depth. K3 depths are modeled as intervals in one dimension. A depth constraint is an assertion about the relationship between two intervals; the depth of a collection, for instance, is an interval which entirely contains the depths of all of that collection's elements. Depth constraints are also accumulated through usage; a `cref` may refer to another `cref` only if the prior's depth is greater than the latter's. After all depth constraints are accumulated through type derivation, constraint closure tests for a contradiction. If no contradiction appears in closure, then no cycles appear in the reference model and thus all computation following the reference model will terminate. To our knowledge, this termination guarantee is novel in a stateful system. The `iref` references do not have a specific depth due to the fact that they cannot contain cyclic types.

5. Optimization and Synthesis

Annotation-Driven Synthesis Annotations can drive the generation of control flows that are complex, but express simple design patterns. For example, the following steps form one example of a control structure to produce a continuously maintained log of the events on a collection.

1. Construct an auxiliary log data structure `LogC`.
2. Write a trigger `logOnCUpdate` to generate a log entry given an update on `C`, and insert that entry into `LogC`.
3. Hook `logOnCUpdate` to execute when an update of `C` occurs.
4. Write a trigger `flushLogC` to flush entries in `LogC` to disk.
5. Hook `flushLogC` to execute according to the desired flushing policy.

The above process naturally fits in with K3's data and computation model, and the corresponding code can be completely generated by the compiler using a single annotation: `declare C:Collection(...) @logged`.

Multi-Platform Synthesis As mentioned in Section 2, the K3 computation model is sufficiently general to express the computation models of several execution platforms. This ability allows the optimizer to analyze the flow of control of a program, and make a decision of which one or several execution platforms to target. In particular, this generality allows the compiler to use the high level primitives provided by these platforms.

Alternatively, the optimizer may decide that it would be more efficient to synthesize a runtime from scratch, if existing frameworks proved inadequate for the control flow and data structure requirements. Similar observations have been made by the Clydesdale Project [8] and Spark [15]. This balance of platform reuse and synthesis is intended to improve the productivity of applications developers in building systems tuned to their applications.

Adaptive Optimization K3 focuses on expressing long-running programs, and one goal of the optimizer model is to be adaptive over the course of execution. The optimizer should be capable of making use of statistics gathered during execution and use principled machine learning models and statistical techniques to determine possible program transformations. Metrics may include access patterns, availability of nodes in the network with time, etc. In addition, the annotation model can make it simpler for the programmer to specify which statistics should be accumulated.

The optimizer could also be adaptive with respect to the computational resources available for execution. The addition of more – or more powerful – hardware should influence the optimizer's decision making process to favor backends that work well with the new configuration. This effectively incorporates part of the problem of scalability into the annotation and optimization layer.

6. Closing Discussion

Related work In contrast to parallel programming frameworks (e.g. Hadoop, Pregel, GraphLab, Pig/Latin, FlumeJava) that focus on few programming primitives and a single data model, or distributed data stores that focus on storage alone (e.g. complex-objects with BigTable and Accumulo [2], key-values with Dynamo, documents with MongoDB), K3 is a general language for long-running computation, leveraging PL techniques for extensible program annotations to reason about optimization opportunities and scalability. Perhaps the most relevant is Spark [15], however their focus is on resilient distributed datasets and coarse-grained parallelism compared to our broader scope.

Our goals of a language for domain-specific runtimes contextualizes our annotations differently than related work on annotations for provenance in the database theory and PL communities [6]. Our work on declarative data structures, support for related and derived data through functional dependencies, integrity constraints, and views, and the implications that *large* data structures have for scalability extends work on data structure synthesis [7].

Conclusion We have outlined K3, and introduced a subset of its language-level features, including annotations, and optimization and synthesis concerns. Our ongoing work includes developing a rich standard library of annotations to capture common data structure and execution properties, as well as both white- and black-box optimization exploiting machine learning and control theory techniques, and toolchain and use-case implementation.

References

[1] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently refreshed views. *PVLDB (to appear)*, 5(11), 2012.

[2] Apache Accumulo. http://accumulo.apache.org/.

[3] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[4] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. *SIGPLAN Not.*, 30(10):169–184, Oct. 1995.

[5] J. Eisner and N. W. Filardo. Dyna: Extending Datalog for modern AI. In *Datalog 2.0*, pages 181–220, 2010.

[6] J. N. Foster, T. J. Green, and V. Tannen. Annotated xml: queries and provenance. In *PODS*, 2008.

[7] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, 2011.

[8] T. Kaldewey, E. Shekita, and S. Tata. Clydesdale: structured data processing on mapreduce. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 15–25. ACM, 2012.

[9] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. *SIGPLAN Not.*, 44(6):304–315, June 2009.

[10] G. Kiczales and E. Hilsdale. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, 26(5):313–, Sept. 2001.

[11] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: probabilistic models with unknown objects. In *IJCAI*, 2005.

[12] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. In *PEPM*, 2012.

[13] M. Stonebraker and U. Çetintemel. "One Size Fits All": An idea whose time has come and gone (abstract). In *ICDE*, 2005.

[14] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.