Regularized Programming with the BOSQUE Language

Moving Beyond Structured Programming

Mark Marron
marron@microsoft.com

Abstract

The rise of *Structured Programming* and *Abstract Data Types* in the 1970's represented a major shift in programming languages. These methodologies represented a move away from a programming model that reflected incidental features of the underlying hardware architecture and toward a model that emphasized programmer intent more directly. This shift simultaneously made it easier and less error prone for a developer to convert their mental model of a system into code *and* led to a golden age of compiler and IDE tooling development. This paper takes another step on this path by further lifting the model for iterative processing away from low-level loop actions, enriching the language with algebraic data transformation operators, and further simplifying the problem of reasoning about program behavior by removing incidental ties to a particular computational substrate and indeterminate behaviors. We believe that, just as structured programming did years ago, this *regularized programming* model will lead to massively improved developer productivity, increased software quality, and enable a second golden age of developments in compilers and developer tooling.

1 Introduction

The introduction and widespread use of structured programming [14] and abstract data types [42] marked a major shift in how programs are developed. Fundamentally, the concepts and designs introduced by these programming methodologies simplified the reasoning about program behavior by eliminating substantial sources of, usually entirely *accidental* [8], complexity. This allowed engineers to focus on the intent and core behavior of their code more directly and, as a result, produced a drastic improvements in software quality and ability to construct large software artifacts. Just as accidental complexity is an impediment to human understanding of a program it is also an impediment to applying formal reasoning techniques. Despite the inability of structured programming to fully bridge the chasm of formal mathematical analysis, issues with loop-invariants and mutation-frames among others prevented the practical use of verification techniques, it did provide the needed simplifications to enable reasoning about limited forms of program behavior and supported a golden age of IDE tooling and compiler development [34, 54].

Drawing inspiration from these successes, this paper seeks to identify additional opportunities to eliminate complexity with the intent that these simplifications will lead to similar advances in software quality, programmer productivity, and compilers/tooling. The first contribution of this paper is an identification and categorization of *unnecessary complexity sources* which can be alleviated via thoughtful language design (Section 2).

Using the sources of complexity identified in Section 2 as a guide, the second contribution of this paper is the introduction of the BOSQUE LANGUAGE[1] (Section 3). The BOSQUE language demonstrates the feasibility of eliminating these sources of complexity while retaining the expressivity and performance needed for a practical language as well as hinting at opportunities for improved developer productivity and software quality.

Section 4 goes into detail on how the concepts used in the design of the BOSQUE language represent a larger step in the development of programming languages and models. Thus, the third contribution of this paper is the introduction of the *regularized programming* model. This model builds on the successes of structured programming and abstract data types by simplifying existing programming models into a *regularized* form that eliminates major sources of errors, simplifies code understanding and modification, and converts many automated reasoning tasks over code into trivial propositions.

To explore the opportunities that *regularized programming* enables this paper concludes with three case studies, using the ability to precisely analyze the semantics of a program to verify correctness, validating *SemVer* [67] usage, and how the regularized semantics enable SIMD [34] optimization in scenarios that would be otherwise difficult or impossible. These results demonstrate how the unique properties of regularized programming enable the implementation of previously impractical developer experiences beyond the direct benefits to software quality and developer productivity that are possible when moving beyond structured programming!

2 Complexity Sources

Based on a range of experiences and sources including developer interviews, personal experience with analysis/runtime/-compiler development, and empirical studies this section identifies five major sources of accidental complexity that can be addressed via thoughtful language design. These are sources of various of bug families, increase the effort required for a developer to reason about and implement functionality in an

[1] The BOSQUE specification, parser, type checker, reference interpreter, and IDE support are open-sourced and available at https://github.com/Microsoft/BosqueLanguage.

application, and greatly complicate (or make it infeasible to) automatically reason about a program.

Mutable State and Frames: Mutable state is a deeply complicated concept to model and reason about. The introduction of mutability into a programming language destroys the ability to reason about the application in a *monotone* [56] manner which forces the programmer (and any analysis tools) to identify which facts remain true after an operation and which are invalidated. The ability for mutable code to affect the state of the application via both return values and side affects on arguments (or other global state) also introduces the need to reason about the *logical frame* [47, 64] of every operation. Section 4.1 examines how the BOSQUE language eliminates this source of complexity through the use of immutable data representation.

Loops, Recursion, and Invariants: Loops and recursion represent a fundamental challenge to reasoning as the code describes the effects of a single step but understanding the full construct requires *generalization* to a quantified property over a set of values. Invariants [19, 27] provide the needed connection but a generalized technique for their computation is, of course, impossible in general and has proved elusive even for restricted applications. Section 4.5 examines how BOSQUE handles the invariant problem by eliminating loops and restricting recursion.

Indeterminate Behaviors: Indeterminate behaviors, including undefined, under specified, or non-deterministic or environmental behavior, require a programmer or analysis tool to reason about and account for all possible outcomes. While truly undefined behavior, *e.g.* uninitialized variables, has disappeared from most languages there is a large class of under-specified behavior, *e.g.* sort stability, map/dictionary enumeration order, etc., that remains. These increase the complexity of the development process and, as time goes on, are slowly being seen as liabilities that should be removed [9]. Less obviously the inclusion of non-deterministic and/or environmental interaction results in code that cannot be reliably tested (flakey tests), behaves differently for non-obvious reasons, and frequently mixes failure logic widely through a codebase. Section 4.2 explains how BOSQUE eliminates these sources of complexity by fully determinizing the language semantics and Section 4.4 further simplifies issues around evaluation orders.

Data Invariant Violations: Programming languages generally provide operators for accessing, and in imperative languages updating, individual elements in arrays/tuples or fields in objects/records. The fact that these accessors/updaters operate on an individual elementwise basis results in programmers updating the state of an object over multiple steps, or manually exploding and object before creating an updated copy, during this span invariants which normally hold are temporarily invalidated before being restored. During these intervals the number of details that must be tracked and restored can increase drastically increasing opportunities for mistakes and oversights to occur. Section 4.1 shows how the BOSQUE language eliminates this problem through the introduction of algebraic bulk data operators.

Equality and Aliasing: Programming languages live at the boundary of mathematics and engineering. Although language semantics are formulated as a mathematical concept there are common cases, *e.g.* reference equality, pass by-value *vs.* by-reference, or evaluation orders, that expose and favor one particular hardware substrate, generally a *Von Neumann* architecture, either intentionally for performance or accidentally by habit or history. While seemingly minor these choices have a major impact on comprehensibility – merely exposing reference equality pulls in the complexity of reasoning about aliasing relations and greatly complicates compilation on other architectures. Section 4.3 shows how the BOSQUE language eliminates reference equality and, as a consequence, eliminates the complexity of aliasing questions and by-value *vs.* by-ref argument passing issues.

3 BOSQUE Language Overview

The BOSQUE language derives from a combination of TypeScript [72] inspired syntax and types plus ML [52] and Node/JavaScript [31, 58] inspired semantics. This section provides the syntax and operators of the BOSQUE language with an emphasis on features of the language that are not widely seen in other programming languages. Section 4 focuses on how these features and design choices address various sources of complexity enumerated in Section 2.

3.1 BOSQUE Type System

The BOSQUE language supports a simple and non-opinionated type system that allows developers to use a range of structural, nominal, and combination types to best convey their intent and flexibly encode the relevant features of the problem domain.

Nominal Type System: The nominal type system is a mostly standard object-oriented design with parametric polymorphism provided by generics. Users can define abstract types, `concept` declarations, which allow both abstract definitions and inheritable implementations for *const* members, *static* functions, member *fields*, and member *methods*. The `concept` types are fully abstract and can never be instantiated concretely. The `entity` declarations create types that can instantiate concepts as well as override definitions in them and can be instantiated concretely but can never be further inherited from. Developers can also alias types or create special types using `typedef`, `enum`, and `ckey` (Section 4.3) constructs.

The BOSQUE core library defines several unique concepts/entities. The `Any` type is a uber type which all others are a subtype of, the `None` and `Some` types are for distinguishing

around the unique `none` value, and `Tuple`, `Record`, etc. exist to unify with the structural type system. The language has primitives for `Bool`, `Int`, `String`, etc. as well as the expected parametric collection types such as `List[T]` and `Map[K, V]`.

Structural Type System: The type system includes *Tuples* and *Records* for structural types. These are structurally self-describing, allow for *optional* entries with the "`?`" syntax, and can be specified as *closed* or *open* using the "..." syntax. Functions are also first class types. In the BOSQUE language functions can use *named* arguments, thus names are part of the function type with a special "_" name value for don't cares. Functions also allow for optional parameters, with the "`?`" syntax, and *rest* parameters using the "..." syntax. Examples of these types include:

```
[Int, ?:Bool] //Tuple, second index is optional
[Int, ...] //Tuple, has an Int rest is open
{f: Int, g?:Bool} //Record required f optional g
{f: Int, ...} //Record required f open other
fn(x: Int) -> Int //Function required x arg
fn(_: Int) -> Int //Function required unnamed arg
fn(x?: Int) -> Int //Function optional x arg
fn(...l: List[Int]) -> Int //Function rest List arg
```

Combination Type System: With the base structural and nominal types we also provide support for *union* and, limited, *conjunction*. The "`T1 | T2`" notation specifies a type may be either `T1` *or* `T2` while the notation "`T1?`" is shorthand for `T1 | None`. Note that in this construction `(T1?)?` is the same type as `T1?`. The type system also admits conjunction but limits it to conjunctions of nominal types. The notation "`T1 + T2`" specifies the type must provide both `T1` *and* `T2`.

3.2 BOSQUE Expressions

Expressions in BOSQUE include the expected set of function calls, global/const/local variable accesses, binary operators, comparators, and data structure accessors. However, a key observation from our interaction with Node.js developers was the extensive time spent *reshaping data* via transforms which involve copying, merging, and updating data from various sources into a new representation. To simplify these tasks, and remove several common sources of accidental complexity discussed in Section 4.1, we provide specialized *bulk algebraic* data operations and integrated support for `none` (or optional data) processing. Other notable features include atomic constructors, *pipeline* support for collection processing and the support of *if* and *match* as expressions.

Call Arguments: A major feature of the JavaScript ES6 specification [31] was the introduction of *spread* and *rest* operators. The rest operator is analogous to the well known `varargs` from other programming languages but the spread operator introduces new power by functioning as the conceptual inverse. The BOSQUE language has borrowed the rest/spread concepts and combined them with *named argument* support which can be used in situations as shown:

```
function nsum(d: Int, ...args: List[Int]): Int {
  return args.sum(default=d);
}

function np(p1: Int, p2: Int): {x: Int, y: Int} {
  return @{x=p1, y=p2};
}

//calls with explicit arguments
var x = nsum(0, 1, 2, 3);

var a = np(1, 2);
var b = np(p2=2, 1);     //same as a
var c = np(p2=2, p1=1); //also same as a

//calls with spread arguments
var t = @[1, 2, 3];
var y = nsum(0, ...t); //same as x

var r = @{p1=1, p2=2};
var d = np(...r); //same as a
```

The first of the examples show the use of rest and named arguments in call signatures. These features work similarly to implementations in existing languages. In our example the call to `nsum` takes an arbitrary number of arguments which are automatically converted into a List. The calls to `np` show how named parameters can be used and mixed with positional parameters.

The next set of examples show how *spread* arguments can be used. In the first case a tuple, `@[1, 2, 3]`, is created and assigned to the variable `t`. This tuple is then spread to provide some of the arguments to `nsum`. Semantically, the call `nsum(0, ...t)` is the same as `nsum(0, t[0], t[1], t[2])` and, as a result, the value in `y` is the same as the value computed for `x`. The spread operator also works for records and named parameters. In the example the call to `np(...r)` is semantically the same as `np(p1=r.p1, p2=r.p2)`. Although not shown here spread can also be used on any collection, List, Set, Map, based data values as well.

Atomic Constructors: BOSQUE tuples, records and lambda functions are created in the standard way. However, to reduce the amount of boilerplate code introduced by constructors, and in particular constructors that have long argument lists that are mainly passed through to super constructors, BOSQUE uses construction via direct field initialization to construct entity (object) values. For many uses this simple direct initializer approach is sufficient and there is no need for complex constructors that compute derived values as part of the execution.

```
concept Bar {
  field f: Int;

  factory default(): {f: Int} {
    return @{f=1};
  }
}

entity Baz provides Bar {
  field g: Int;
  field h: Bool = true;
```

```
Exp         :=  Const | Access | Constructor | Lambda | Call | TupleOp | NamedOp
              | Project | Merge | Apply | Invoke | Pipeline | BinOp | BinCmp | BinLogic
              | Coalesce | Select | StmtExp | (Exp)
            ...
Constructor :=  @[Args] | @{Args} | Type@{Args} | Type@Identifier(Args) | Lambda
TupleOp     :=  Exp(?)?[ Idx ] | Exp(?)?@[ (Idx) *' ] | Exp(?)?<~( (Idx=Expression) *' )
NamedOp     :=  Exp(?)?.Identifier | Exp(?)?@{ (Identifier) *' } | Exp(?)?<~( (Identifier=Expression) *' )
Project     :=  Exp(?)?#Type
Merge       :=  Exp(?)?<+(Exp)
Invoke      :=  Exp(?)?-> Identifier(Args)
Pipeline    :=  Exp|( ?|?? )?>Method([Type +' ])?(Args)
            ...
Coalesce    :=  Exp ( ?& | ?| ) Exp
StmtExp     :=  If | Match | Block
Args        :=  ( Exp | Identifier=Exp | ...Exp )*'
```

Figure 1. BOSQUE Expression Grammar (Subset)

```
factory identity(i: Int): {f: Int, g: Int} {
  return @{f=i, g=i};
}
}

var x = Baz@{f=1, g=2};
var y = Baz@{f=1, g=2, h=false};

var p = Baz@identity(1);
var q = Baz@{...Bar::default(), g=2};
```

In this code snippet two `Baz` entities are allocated via the atomic initialization constructor. In the first case the omitted `h` field is set to the provided default value of `true`. Sometimes it is useful to encapsulate initialization logic and, to accomplish this, we allow for the definition of `factory` functions which operate similar to constructors but, in some sense, are upside down. A factory function returns a record with all the fields needed for the enclosing entity/concept. So, the `identity` factory defines `f` and `g`. When invoked with the constructor syntax this is desugared to the atomic initializer with the result of factory, `Baz@{...Baz::identity(1)}`, in our example.

Bulk Algebraic Data Operations: The bulk algebraic operations in BOSQUE start with support for bulk reads and updates to data values. In addition to eliminating opportunities to forget or confuse a field the BOSQUE operators help focus the code on the overall intent, instead of being hidden in the individual steps, and allow a developer to perform algebraic reasoning on the data structure operations. We provide several flavors of these algebraic operations for various data types, tuples, records, and nominal types, and operations including projection, multi-update, and merge.

```
(@[7, 8, 9])@[0, 2];     //@[7, 9]
(@[7, 8])<~(0=5, 3=1); //@[5, 8, none, 1]
(@[7, 8])<+(@[5]);      //@[7, 8, 5]

(@{f=1, g=2})@{f, h};           //@{f=1, h=none}
(@{f=1, g=2})<~(f=5, h=1);      //@{f=5, g=2, h=1}
(@{f=1, g=2})<+(@{f=5, h=1}); //@{f=5, g=2, h=1}

Baz@identity(1)@{f, h};     //@{f=1, h=true}
Baz@identity(1)@{f, k};     //error
Baz@identity(1)<~(f=5);     //Baz@{f=5, g=1, h=true}
Baz@identity(1)<~(p=5);     //error
Baz@identity(1)<+(@{f=5}); //Baz@{f=5, g=1, h=true}
```

None Processing: Handling `none` values (or null, undefined, *etc.* in other languages) is a relatively common task that can obscure the fundamental intent of a section of code with nests of cases and conditional handling for the special case. To simplify this type of code languages have introduced various forms of *null-coalescing* or *elvis* operators. The definition of BOSQUE follows a similar approach by having both elvis operator support for all chainable actions and specific none-coalescing operations (*e.g.* as opposed to truthy based coalescing of the logical operators in JavaScript).

```
@{}.h            //none
@{}.h.k          //error
@{}.h?.k         //none
@{h={}}.h?.k     //none
@{h={k=3}}.h?.k //3

function default(x?: Int, y?: Int) : Int {
    return (x ?| 0) + (y ?| 0); //default on none
}
default(1, 1) //2
default(1)    //1
default()     //0

function check(x?: Int, y?: Int) : Int? {
    return x ?& y ?& x + y; //check none
```

```
Stmt     := VarDecl | Assign | Result | Validate | If | Match | Block
VarDecl  := var Identifier(: Type)? = Exp; | var! Identifier(: Type)?( = Exp)?; | (var | var!) Structure = Exp;
Assign   := Identifier = Exp; | Structure = Exp;
Validate := assert Exp; | check Exp;
...
```

Figure 2. BOSQUE Statement Grammar (Subset)

```
}
default(1, 1) //2
default(1)    //none
default()     //none
```

Collection Pipelining: Higher-order processing of collections is a fundamental aspect of the BOSQUE language (see Section 4.5) but often times chaining filter/map/*etc.* is not a natural way to think about a particular set of operations and can result in the creation of substantial memory allocation for intermediate collection objects. Thus, BOSQUE allows the use of both method chaining for calls on collections *and* pipelining, `|>` inspired by LINQ [6, 41], values through multiple steps. This allows a developer to specify a sequence of operations, each of which is applied to elements from a base collection sequence, that transform the input data into a final collection. As with other chaining we support none-coalescing operations, `|?>`, which propagates a none immediately to the output in the pipeline and, `|??>`, which short circuits the processing and drops the value.

```
var v: List[Int?] = List@{1, 2, none, 4};

//Chained -- List@{1, 4, 16}
var r1 = v.filter(fn(x) => x != none)
          .map[Int](fn(x) => x*x);

//Piped with none to result -- List@{1, 4, none, 16}
var r2 = v|?>map[Int](fn(x) => x*x);

//Piped with noneable filter -- List@{1, 4, 16}
var r3 = v|??>map[Int](fn(x) => x*x);
```

3.3 BOSQUE Statements

Given the rich set of expression primitives in BOSQUE there is a reduced need for a large set of statement combinators. Coming from a functional language perspective the language includes the expected *Match* and *If* which can be used as both expressions and statements as well as a *structured assignment* operator for easy destructuring of return values. As high reliability software is a key goal, BOSQUE provides an `assert`, enabled only for debug builds, and a `check`, enabled on all builds, as first class features in the language (in addition to pre/post conditions and class invariants). We also note that there are *no looping constructs* in the language.

Block SSA Local variables with block structured code is a very appealing model for developers in the cloud/IoT space. JavaScript and TypeScript being two of the most popular languages with other interesting projects, like ReasonML [62], experimenting with fusing functional programming with block scopes and "{...}" braces. The BOSQUE language follows this trend with two novel adjustments, allowing multiple assignments to a variable and supporting statement expressions, to support functional style programming in a block-scoped language. Consider the code:

```
//Multiple assignment
function abs(x: Int): Int {
  var! y = x;
  if(y < 0) {
    y = -y;
  }
  return y;
}
```

This function shows the use of multiple updates to the same variable. We distinguish between variables, `var`, that are fixed and those, `var!`, that can be updated. This ability to set/update a variable as a body executes simplifies a variety of common coding patters and, since the language is loop free, can be easily converted to a SSA [13] form that restores the purely functional nature of the semantics.

4 Regularized Programming

This section focuses on how concepts and features in the BOSQUE language interact to address various sources of complexity identified in Section 2 and how this *regularized* form eliminates major sources of errors, simplifies code understanding and modification, and converts many automated reasoning tasks over code into trivial propositions.

4.1 (Im)mutable State

Reasoning about and understanding the effect of a statement or block of code is greatly simplified when it is side-effect free. Functional languages have long benefited from this property and developers, in particular in the cloud/web space, have been migrating towards programming models that either encourage the use of immutable data [29, 61, 63] or explicitly provide immutable and functional programming as part of the language semantics [16, 62]. BOSQUE follows this trend by adopting a functional programming model with immutable data. Moving to an immutable model of computation these avoids the issues with non-monotone reasoning [56] and the need to compute logical frames [47, 64] identified previously.

```
//JavaScript Implementation
class Bar {
  constructor(f) {
    this.f = f;
  }
}

class Baz extends Bar {
    constructor(f, g) {
      super(f);
      this.g = g;
    }
}

var x = new Baz(1, 2);
var y = new Baz(3, x.g);
```

```
//Bosque Implementation
concept Bar {
  field f: Int;
}

entity Baz provides Bar {
  field g: Int;
}

var x = Baz@{f=1, g=2};
var y = x<~(f=3);
```

Figure 3. Constructors and Updates: JavaScript *vs.* Bosque

However, even with immutable objects there can be subtle challenges with constructor semantics and implementing operations which create a copy of a value with updates to some subset of the contained values. Constructor bodies where fields are initialized sequentially with, potentially other computation mixed in, can lead to issues where methods are invoked on partially initialized values. Updates to objects are often implemented by copying fields/properties individually while replacing the some subset with new values. These issues can lead to both subtle bugs during initial coding *and* also make it difficult to update data representations when refactoring code or adding a new feature at some later date.

Consider the code shown in Figure 3. The JavaScript implementation on the left has substantially more boilerplate construction code and argument propagation than the BOSQUE version on the right. Further, the use of atomic constructors prevents the partially initialized object problem [17, 18, 32] when constructing the `Baz` object. This example also shows how the *bulk algebraic* operations simplify the update/copy of the `Baz` object as well.

The value of the bulk algebraic operators is more evident when we consider what happens when a developer later decides that the base class `Bar` needs a new field, say `k`, to support a new feature request. The diff of code in JavaScript and BOSQUE required to make this change is shown in Figure 4.

As can be seen the addition of a single field in the JavaScript code requires manually threading the new value though both constructors, updating the constructor calls, and the copy operations – touching almost every line of code. On the other hand the BOSQUE code only requires an update to the constructor since the algebraic update operator, `<~`, handles the changed set of fields automatically. This is not only a reduction in the number of places that must be updated to support the change but it also eliminates whole classes of bugs when there may be multiple constructors and one update copy might match a constructor with a default value for `k` creating an incorrect update.

4.2 Indeterminate Behavior

When the behavior of a code block is under-specified the result is code that is harder to reason about and more prone to errors. As a key goal of the BOSQUE language is to eliminate sources of unneeded complexity that lead to confusion and errors we naturally want to eliminate these under-specified behaviors. To start with BOSQUE does not have any truly *undefined* behavior such as allowing uninitialized variable reads. However, we can go further by eliminating *implementation defined* behavior as well. Thus, in the language definition sorting is defined to be stable and all associative collections (sets and maps) have a stable enumeration order. As a result of these design choices there is always a single *unique* and *canonical* result for any BOSQUE program!

This means that developers will never see intermittent production failures or flakey unit-tests in BOSQUE code. Consider the following example where a list with records containing duplicate `id` properties is sorted and then, as the test, the developer asserts the first element is associated with a given value. In many languages, including JavaScript, where sort stability is not specified this test may fail or succeed depending on the version of the runtime or even between different runs on the same runtime.

```
var l = List[{id: Int, val: String}]@{
                @{id=1, val="yes"},
                @{id=1, val="no"},
                @{id=2, val="maybe"}
              }
              .sort(fn(a, b) => a.id < b.id);

assert l[0].val == "yes";
```

```
//JavaScript Implementation (Diff)
class Bar {
  constructor(f, k) {
    ...
    this.k = k;
  }
}

class Baz extends Bar {
    constructor(f, g, k) {
      super(f, k);
      ...
    }
}

var x = new Baz(1, 2, true);
var y = new Baz(3, x.g, x.k);
```

```
//Bosque Implementation (Diff)
concept Bar {
  ...
  field k: Bool;
}

var x = Baz@{f=1, g=2, k=true};
...
```

Figure 4. Diff for New Field: JavaScript *vs.* Bosque

These types of indeterminate behaviors, and the failures they cause, are a major pain point for testing and for development in general. However, the semantics of BOSQUE ensure a single *unique* and *canonical* result so these types of issues are completely eliminated.

In addition to undefined and implementation defined behaviors there are also environmental sources of nondeterminism, IO, getting dates/times, event-loop scheduling, random numbers, UUID generation, *etc.* that are also present in most languages. JavaScript took an interesting step by decoupling the core compute language in the JS specification from the IO and event loop which are provided by the host [44, 58]. The AMBROSIA [20] project pushed this idea further by fully moving all sources of environmental nondeterminism to host provided calls. We take the same approach of decoupling the core compute language, BOSQUE, from the host runtime which is responsible for managing environmental interaction.

In combination with the fully determinized semantics the movement of external interaction out of the core language enables transparent failure recovery [20], diagnostic record-replay or time-travel-debugger systems [3, 4], and serverless frameworks that rely on restartability and migration [15] to be built on BOSQUE code without any (or minimal) additional instrumentation or runtime support!

4.3 Equality and Representation

Equality is a multifaceted concept in programming [57] and ensuring consistent behavior across the many areas it can surface in a modern programming language such as `==`, `.equals`, `Set.has`, and `List.sort`, is source of subtle bugs [28]. This complexity further manifests itself in the need for developers and tooling to consider the possible aliasing relations of values, in addition to their structural data, in order to understand the behavior of a block of code. The fact that *reference equality* is chosen as a default, or is an option, is also a bit of an anachronism as reference equality heavily ties the execution semantics of the language to a hardware model in which objects are associated with a memory location.

Once reference equality, and the aliasing relation it induces, are present in a language they quickly spread with major impacts on value representation, passing semantics, and evaluation strategies. In the absence of user visible reference semantics then the choice of *pass-by-value vs. pass-by-reference* for argument and return values no longer impacts the behavior of the program. Eliminating aliasing also moves choices on value representation, inline, shared reference, sliced, and evaluation optimizations such as memoization from semantic to purely performance related concerns.

The BOSQUE language *does not* allow user visible *reference equality* in any operation including `==` or container operations. Instead equality is defined either by the core language for the primitives `Bool`, `Int`, `String`, *etc.*, or as a user defined *composite key* (**`ckey`**) type. The composite key type allows a developer to create a distinct type to represent a composite equality and order comparable value. The language also allows types to define a `key` field that will be used for equality/order by the associative containers in the language.

```
ckey MyKey {
  idx: Int;
  category: String;
}

entity Baz provides Indexable {
  field key: MyKey;
}

var a = Baz@{MyKey@{1, "yes"}};
var b = Baz@{MyKey@{1, "yes"}};
var c = Baz@{MyKey@{1, "no"}};

var set = Set[Baz]@{a};
```

```
set.has(a); //true
set.has(b); //true
set.has(c); //false
```

This sample shows how a developer can use primitive types and custom keys to define the notion of equality *e.g.* identity, primary key, equivalence, *etc.* that makes sense for their domain without the complication of a default that needs to be overridden. From a reasoning standpoint this simplification eliminates the need to track aliasing, allows all values to be reasoned about as pure terms, and eliminates the need to model implicit re-entrant `.equals` calls during container operations. As desired it also enables a developer or compiler to switch between representations and evaluation strategies without worrying about effecting semantically observable behavior.

4.4 Evaluation Strategies

Evaluation order and error behavior are the second area where details of a CPU based execution model have crept into default choices for language semantics. The sequencing operators, specifically "`;`" and "`,`", inject a notion of single-threaded in-order execution for a CPU or a step-debugger. This artificially limits the evaluation strategies that a runtime can use by adding a new and arbitrary relation between actions in a program. Beyond the natural mapping to execution on an idealized CPU this choice is also motivated by the ability to observe execution order via side effecting *logging*, *raise*, or *runtime error* semantics which leak information on execution interleaving. Our goal is to provide a model where the semantics of sequence operators are *simultaneous execution* modulo true data or control dependencies and errors are unable to leak ordering information.

The BOSQUE language takes a novel view of logging, runtime errors, and debugging. Since the semantics and design of the language ensure fully determinized execution of any code there is actually no real need to perform logging within a block of code. So, logging is not available in the compute language. However, runtime error reporting requires the inclusion of observable information, like line numbers and error messages, to support failure analysis and debugging. In this case, since BOSQUE execution is fully deterministic and repeatable, we opt for a design where the language has two execution semantics: *deployed* and *debug*. In the deployed semantics *all runtime errors* are indistinguishable while in the debug semantics errors contain full line number, call-stack, and error metadata. When an error occurs in *deployed* mode the runtime simply aborts, resets, and re-runs the execution in *debug* mode to compute the precise error!

For a given program P the *debug* mode semantics the follow a strict sequential and left-to-right evaluation of the code and preserves a simple model that a developer can step though in the debugger. The *deployed* mode applies *simultaneous execution* semantics which: respects any true data or control dependencies and for errors is only required to ensure that:

- $P_{\texttt{debug}}(x) \rightsquigarrow v \Rightarrow P_{\texttt{deployed}}(x) \rightsquigarrow v$
- $P_{\texttt{debug}}(x) \rightsquigarrow \texttt{error} \Rightarrow P_{\texttt{deployed}}(x) \rightsquigarrow \texttt{error}'$

Or informally that the deployed semantics must raise some error at some point[2] if the debug semantics would also raise an error.

```
function foo(l: List[Int], i: Int): Int {
  var y = l.min(); //runtime error
  check i != 0; //check error
  return y + i;
}

var k = foo(List[Int]{}, 0);
```

This example shows code with 2 errors under *debug* execution semantics and will always fail on the call to `l.min`. However, the *deployed* semantics can either, fail with `l.min`, fail with `check i != 0`, or statically determine the call always fails and reduce the program to `error`. This has a number of interesting implications. A compiler can reorder execution freely, parallelization restrictions are substantially relaxed, and it becomes feasible to perform larger scale semantic transforms such as switching from chained collection processing to pipelined or eager evaluation to lazy.

4.5 Loop Free and Controlled Recursion

Enforcing limited structure on iteration was a breakthrough in allowing developers to concisely express their intent, *e.g.* `for(i = 0; i < length; ++i)` *vs.* `goto Lhead`, and was a critical development in empowering compilation and verification tools based on structured dataflow analyses [54]. In this section we explore a second step in raising the level of expressive power, from structures to intents, in expressing iterative flow with the goal of achieving similar improvements in program clarity and analysis effectiveness.

4.6 Looping Constructs

A fundamental concept in a programming language is the iteration construct and a critical question is can this construct be provided as high-level functors, such as filter/map/reduce or list comprehensions, or do programmers benefit from the flexibility available with iterative, while or for, looping constructs. To answer this question in a definitive manner the authors in [1] engaged in an empirical study of the loop "idioms" found in real-world code. The categorization and coverage results of these semantic loop idioms shows that almost every loop a developer would want to write falls into a small number of idiomatic patterns which correspond to higher level concepts developers are using in the code, *e.g.*, filter, find, group, map, *etc.*

Inspired by this result BOSQUE trades structured loops for a set of high-level iterative processing constructs (functors). The elimination of loops and their replacement with functors

[2]The error raised by the deployed semantics does not need to be from the same source or related in any way to the error from the debug semantics. One could be a div by 0 while the other is an out-of-bounds access.

```
//Imperative Loop (JavaScript)
var: number[] a = [...];
var: number[] b = [];
/**
Pre: b.length == 0
**/
for(var i = 0; i < a.length; ++i) {
    /**
    Inv: b.length == i /\
         forall i in [0, i) b[i] == a[i]*2
    **/
  b.push(a[i]*2);
}
/**
Post: b.length == a.length /\
      forall i in [0, a.length) b[i] == a[i]*2
**/
```

```
//Functor (Bosque)
var a = List[Int]@{...};
/**
Pre: true
**/
var b = a.map[Int](fn(x) => x*2);
/**
Post: List[Int]::eq(fn(x, y) => y == x*2, a, b)
**/
```

Figure 5. Loops vs. Functors with Symbolic Transformers : JavaScript *vs.* Bosque

is a critical design choice. Eliminating the boilerplate of writing the same loops repeatedly eliminates whole classes of bugs, *e.g.* bounds arithmetic, and makes programmer intent clear with descriptively named functors instead of relying on a shared set of mutually known loop patterns. Critically, for enabling automated program validation and optimization, eliminating loops also eliminates the need for computing loop-invariants [19, 27]. Instead, with careful design of the collection libraries, it is possible to write precise transformers for the functors. Thus, eliminating the loop invariant generation problem when computing *strongest-postconditions*, or *weakest-preconditions*, and reducing the task to a simple and deterministic matter of formula pushing!

The idioms in [1] contain the expected set of functors seen in most functional languages, filter/map/reduce/find, as well as operators, join/take/first/every, that also appear in the *stream* oriented C# LINQ [41], Java Stream [30] APIs, or JavaScript functional libraries like underscore [73] or lodash [43]. Instead of relying solely on our intuition to select these operators we apply the data driven design suggested in [1] to ensure that our library of operators does not miss any frequent use cases.

Figure 5 contains a simple example of how the loop free nature of BOSQUE simplifies the analysis of a program. Instead of requiring sophisticated techniques to heuristically generate an invariant for each loop a programmer writes, we manually write a template (once) for each functor in the collection library, and then when analyzing a call to the functor we instantiate it with the lambda body semantics. The result is a faster and more predictable analysis as there are no heuristics which may run slowly or fail based on subtle variations in the way a loop is written. Since the collection functors are associated with high-level intents the transformers associated with them can also be written using higher-level predicates, as seen by the use of the `List[Int]::eq` predicate in the sample. The improvements in performance and results quality are not limited to *weakest-precondition/strongest-postcondition* formula computation but also apply to any other abstract interpretation [56] based analyses [46, 53] or more specialized symbolic based analysis [22].

In previous systems the semantics of container processing operations and their chaining was restricted by the underlying language providing a single "." composition operator. Thus, steps in the process were either *layered* where all elements were processed before moving to the next step or *streamed* where a single element was run through all processing steps before moving the next element. Further study of looping code indicated that developer preference for reasoning about a block of code, and efficiency of processing, was variable and that providing developers the option of using either semantics was ideal. Thus, the BOSQUE language provides a stream piping operator "`|>`" which allows a developer to stream objects through a processing pipeline in addition to the standard "." chaining operator.

4.7 Recursion

The lack of explicit looping constructs, and the presence of collection processing functors, is not unusual in functional languages. However, the result is often the replacement of complex loop structures with complex recursion structures. Complex raw control flows obfuscate the intent of the code and hinder automated analysis and tooling regardless of if the flow is a loop or recursion. Thus, BOSQUE is designed to encourage limited uses of recursion, increase the clarity of the recursive structure, and enable compilers/runtimes to avoid stack related errors [48].

To accomplish these goals we borrow from the design of the *async/await* syntax and semantics from JavaScript/TypeScript

and F# [2] which is used to add structured asynchronous execution to these languages. In this design the *async* keyword is used to explicitly identify functions that are asynchronous while the *await* keyword is placed at the continuation points of async function calls. In the background the compiler uses these markers to identify where it should convert the linear code into a *continuation passing* form.

The BOSQUE language takes a similar approach by introducing the **rec** keyword which is used at both declaration sites to indicate a function/method is recursive and again at the call site so to affirm that the caller is aware of the recursive nature of the call.

```
typedef TNode = {l: TNode?, r: TNode?, d: Int};

rec function find(t: TNode?, d: Int): TNode? {
  if(t == none) {
    return none;
  }
  elif(t.d == d) {
    return t;
  }
  else {
    return (t.d < d) ? rec find(t.l, d)
                     : rec find(t.r, d);
  }
}
```

As seen in the example the **rec** marker provides clarity to the developer on which calls may involve recursion and enables the compiler to check these as well. The **rec** keyword can also be used to power alternative compiler/runtime implementation of recursive calls. In addition to the standard unbounded call stack implementation the **rec** keyword can be used like the *await* keyword as a marker for points where conversion to a *continuation passing* version can be performed, with a worklist implementation, to enable recursion on a bounded depth stack.

The combination of explicit demarcation of recursive execution along with the ability to place strong pre/post conditions on these calls serve as limits on the complexity that recursion can introduce while still providing it as an option for when functors cannot (or cannot reasonably) be used to express a computation. We believe that further work, such as identifying *recursive idioms* [26, 49] or other fundamental algorithmic patterns [33, 55, 74], can further reduce the need for unstructured recursion – eventually limiting it to an infrequently used part of the language.

5 Case Studies

The previous sections of this paper have presented *regularized programming* as a concept and a specific realization of this model in the BOSQUE language. This section uses three examples to examine the implications of these ideas in enabling the creation of previously impractical developer experiences, improving software quality, and increasing developer productivity.

Figure 6 shows a fragment of BOSQUE code from a hypothetical coffee-shop finder application. Our focus is on the `closest` function which takes in a list of stores, a matching list of their locations, and the users current location. It is intended to output the `[Shop, Location]` tuple corresponding to the nearest, in Manhattan distance, coffee shop.

Given the argument lists and current location the `closest` code in Figure 6 will first check the **requires** clause of the function to ensure that the lists are of the same length. Assuming this test passes the first step in the actual computation is to use pipeline processing to go over the elements in the `locs` list, first doing a combination of computing the Manhattan distance for the element and checking that it does not exceed a sanity limit on distances, and then finding the index in the list with with minimum distance. Once this minimum distance index is known the code gets the corresponding store and location from the paired lists and returns a tuple containing these values.

5.1 Verification and Threshing

The BOSQUE type system provides rich structural information about the values of function arguments. In particular the non-noneability and lack of aliasing in the language semantics make it feasible to build precise symbolic models a program state from them and, as as result, it is feasible to perform modular abstract symbolic execution on each function. This approach is very effective for this code and will flag three potential errors that could be raised.

1. The **requires** `stores.size() == locs.size()` fails.
2. The **check** `mdist < sanityDist` fails for some value.
3. The call to `minIndex` on line 19 raises a runtime error.

Instead of reporting these potential errors directly, burdening the developer with potential false positives and forcing them to interpret a first-order logic formula detailing the violation, we can instead try a combination of weakest-precondition computation and *threshing* [7].

For the first error, since there is no mutation, it is simple to back prop the failure condition to line 31 where the arguments originate. Using the fact that the size of the result list from a map is the same as the input list size it is trivial to deduce that `stores.size() == opts.size() == locs.size()`. Thus, this can be ruled out as satisfied by the caller.

Similarly, the check on line 16 reduces to `manhattanDist(l, curr) >= sanityDist` when propagated back to through the lambda, reduces to $\exists$ `e` $\in$ `locs` s.t. `manhattanDist(l, e) >= sanityDist` as the precondition for the map functor, and eventually back to line 28/29 where the locations are computed. At line 29 this is trivally refuted as the list is empty. On line 28 we can use a template on the precondition semantics for the `filter` operation and check if ($\exists$ `e` $\in$ `opts` s.t. `manhattanDist(l, e[1]) >= sanityDist`) $\wedge$ ($\forall$ `e` $\in$ `opts` `manhattanDist(l, e[1]) < sanityDist`) is satisfiable. In

```
1  global sanityDist: Int = 100;
2
3  typedef Shop = ...;
4  typedef Location = {x: Int, y: Int, zip: String};
5  typedef GeoData = Map[String, List[[Shop, Location]]];
6
7  function manhattanDist(l1: Location, l2: Location): Int {
8      return (l1.x - l2.x).abs() + (l1.y - l2.y).abs());
9  }
10
11 function closest(stores: List[Shop], locs: List[Location], curr: Location): [Shop, Location]
12   requires stores.size() == locs.size()
13 {
14     var near = locs |> map[Int](fn(l) => {
15                           var mdist = manhattanDist(l, curr);
16                           check mdist < sanityDist;
17                           return mdist;
18                       })
19                       |> minIndex();
20
21     var store = stores.at(near);
22     var at = locs.at(near);
23     return @[store, at];
24 }
25
26 function getOptions(data: GeoData, curr: Location): [List[Shop], List[Location]] {
27     var opts = data.tryGet(curr.zip)
28                    ?.filter(fn(opt) => manhattanDist(opt[1], curr) < sanityDist)
29                ?| List[[Shop, Location]]@{};
30
31     return @[opts.map[Shop](fn(opt) => opt[0]), opts.map[Location](fn(opt) => opt[1])];
32 }
33
34 entrypoint function getNearby(data: GeoData, curr: Location): [Shop, Location]? {
35     var info = getOptions(data, curr);
36     return closest(...info, curr);
37 }
```

Figure 6. Example Code Closest Coffee-Shop Code

this case it is not and so we know that this error is impossible as well.

In the third case it is possible to propagate the formula, `locs.size() != 0` from line 19. Up to line 29 as a possible source of the empty list. At this point we can continue to push the formula up with the inference that `¬data.has(curr.zip)` and eventually to the **`entrypoint`** function `getNearby` on line 34.

At this point we can concretize the error trace to get at least *one* failing input for the developer to inspect and debug. In this case we may provide `data = Map[String, List[[Shop, Location]]@{}` as the failure example. From this the developer could add a check for the empty list in the `closest` function and return the sentinel **`none`** to resolve the issue.

The ideas of symbolic validation and threshing are not new but, as shown in this example, features of the regularized programming model move these tools from aspirational to a realizable part of a developers toolkit. The regularized programming model allowed the symbolic engine to avoid the complexities of reasoning about frames, havocs, loop-invariant generation, fact-retraction, alias analysis, and nullity. As a result the symbolic analysis task was done using basic formula propagation and without the use of heuristics, case splitting optimizations, or complex generalization strategies.

5.2 SemVer Check

In addition to enabling verification and validation scenarios, regularized programming also has the potential to revolutionize many aspects of the software lifecycle as developers experience it today. The topic of Semantic Versioning

(SemVer) [67] is a major concern. Our example code may ship as part of a library consumed by other applications. It is given a SemVer number, `Major.Minor.Patch`, which specify the version of the software and, informally, what may change during an upgrade. By convention `Major` may change existing APIs, `Minor` may add new functionality but existing behavior is preserved, and `Patch` may only fix bugs. There is no formal specification for what any of these mean exactly and, when updating a SemVer, the developer must use their best judgement as to what their changes are and what the appropriate SemVer change should be.

The ability to effectively reason about code in the regularized model has the potential to change this. With techniques like *symbolic diffing* [37] we can begin to formalize what each SemVer level means and validate, both on the provider and consumer side, if they can upgrade without experiencing breakage or estimate where and how much change might occur. As a very conservative approximation we can state formally that for a change from `P` → `P'` where ∀ `v` s.t. `P(v)` ≠ `error` ⇒ `P(v)` ≡ `P'(v)`, *i.e.* we have only removed possible errors, then the SemVer change is a `Patch`.

In our example the developer has fixed a previous error when the `GeoData` had no information for the current zipcode and would fail with an exception. In the new version it will instead return the sentinal **`none`** value indicating no data was found for the users query. In this case, as we saw in the verification example, the regularized programming model enables the automated analysis of this code and validation of our SemVer `Patch` condition. Thus, after fixing a bug that was (automatically) identified the developer can also fix *and* deploy it with complete confidence that it will not cause problems to downstream consumers.

5.3 SIMD Vectorization

Beyond correctness and streamlining the software lifecycle process, regularized programming also has the potential to unlock substantial developments in compiler optimization and application performance. To illustrate this we will look at the task of auto-vectorizing code using SIMD instructions. This transformation can lead to large performance increases but it has been very difficult to consistently apply these optimizations in practice [5, 45].

Suppose we want to transform the collection processing code in the `closest` function from a simple scalar implementation into a SIMD version. There are four issues that need to be resolved:

1. SIMD operations either cannot (or are poor performance) at pointer chasing. Thus, the list of `Location` records should have value, not reference, semantics.
2. The pipeline operator implies a sequential order on processing – each element in `locs` is processed by the full pipeline before the next value.
3. There is the possibility of a **`check`** failure on each pipelined element which must be considered.
4. The scalar logic needs to be converted into SIMD – existing compiler and synthesis techniques can handle this [5, 34, 45].

In any previous language the first 3 issues would realistically eliminate any hope of SIMD converting this loop. However, the regularized semantics of BOSQUE mostly or completely eliminates these issues.

As discussed in Section 4.3 the language semantics do not allow the observation of reference identity and by-value *vs.* by-reference representation are indistinguishable. Thus, the compiler can trivially transform the `locs` list into a representation where the values are stored inline. The error semantics in Section 4.3 also allow the compiler to interchange the error raise across iterations of the pipeline processing or even move out of the loop entirely and only raise upon completion if required. With the error issue resolved the rest of the operations are trivially commutative based just on the immutable value semantics and definition of `map` and `minIndex`.

Using these insights a compiler can trivially convert the pipeline into a fused version of *map* + *min*, hoist the **`check`** error out of the loop, flatten the `locs` values into a by-value representation, and perform SIMD expansion plus instruction selection on the now fully flat and scalar code.

This example shows how the simplifications of regularized semantics transformed a complex, and often impossible to verify safe, optimization into one that was almost trivial to check and perform. Many of the features that made the SIMD transformation simple including representation opacity, immutability, reorderabilty, also drastically simplify other classes of program optimizations or runtime implementations including, memory allocation/collection, expression move/elimination, stack allocation, partial evaluation, *etc.*

6 Related Work

Throughout this paper we have cited the conceptual frameworks [8, 14, 33, 42] and language constructs [1, 6, 15, 52, 55, 72] that have motivated the development of the regularized programming paradigm and the design of the BOSQUE language. Thus, we focus on topics related to the complexity issues identified and connections to other lines of research.

Invariant generation: The problem of generating loop invariants goes back to the introduction of loops as a concept [19, 27]. Despite substantial work on the topic [23, 39, 66, 68] the problem of generating precise loop invariants remains an open problem. This has severely limited the usability and adoption of formal methods in industrial development workflows. Notable successes include seL4 [35], CompCert [40], and Everest [60]. However, all of these systems required expertise in formal methods that is beyond what is available to most development teams. The BOSQUE

language seeks to sidestep this challenge entirely by avoiding the presence of unconstrained iteration.

Equality and Reference Identity: Equality is a complicated concept in programming [57]. Despite this complexity it has been under-explored in the research literature and is often defined based on historical precedent and convenience. This can result in multiple flavors of equality living in a language that may (or may not) vary in behavior and results in a range of subtle bugs [28] that surface in surprising ways.

Reference identity, and the equality relation it induces, is a particularly interesting example. Identity is often the desired version of equality for classic object-oriented programming [57] and having it as a default is quite convenient. However, in many cases a programmer desires equality based on values, or a primary key, or an equivalence relation and a default equality based on identity is, instead, a source of bugs. Further, the fact that it is based on memory addresses is a complication to pass-by-value optimizations of attempts to compile to non *Von Neumann* architectures like FPGAs [36].

Alias Analysis: The introduction of identity as an observable feature in a language semantics immediately pulls in the concept of aliasing. This is another problem that has been studied extensively over the years [24, 25, 38, 50, 51, 69] and remains an open and challenging problem. A major motivation for this work is, in a sense, to undo the introduction of reference identity and identify code where reference equality does not need to be preserved. This is critical to many compiler optimizations including classic transformations like scalar field replacement, conversion to pass-by-value, and copy-propagation [34, 54]. As discussed in Section 5 this information is also critical to compiling to accelerator architectures like SIMD hardware [5].

Frames and Ownership: The problem of aliasing is further compounded with the introduction of mutation. Once this is in the language the problem of computing frames [64] and purity [70] becomes critical. Often developers work around the problem of explicit frame reasoning by using an *ownership* [11, 12] discipline in their code. This may be a completely convention driven discipline or, more recently, may be augmented by runtime support such as smart pointers [55] and type system support [21, 65, 71, 75].

Synthesis: Program synthesis is an active topic of research but the need to reason about loops has limited the application of synthesis to mostly straight-line code. Work on code with loops has been more limited due to the challenge of reasoning about loops in code [5, 10] and the difficultly synthesizers have constructing reasonable code that includes raw loop and conditional control-flow [59]. Thus, a language like BOSQUE, that provides high-level functors as primitives and can be effectively reasoned about opens new possibilities for program synthesis.

7 Conclusion

This paper introduced and defined the concept of *regularized programming* which represents major step in the journey, started with structured programming, towards code that simple, obvious, and easy to reason about for both humans and machines. This advance was based on the identification and elimination of various sources of *accidental* complexity that have remained unaddressed in modern programming languages and insights on how they can be alleviated via thoughtful language design. Using these insights we developed the BOSQUE language[3] which demonstrates the feasibility of regularizing these sources of complexity while retaining the expressivity and performance needed for a practical language. Finally, using several case studies this paper demonstrated opportunities for improved developer productivity and software quality. As a result of these developments we believe that, just as structured programming did years ago, this *regularized programming* model will lead to massively improved developer productivity, increased software quality, and enable a second golden age of developments in compilers and developer tooling.

Acknowledgments

We would like to thank our colleagues and interested readers (both in person and via GitHub) who provided comments and found mistakes in earlier drafts of this work.

References

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, Premkumar T. Devanbu, Mark Marron, and Charles A. Sutton. 2018. Mining Semantic Loop Idioms. *IEEE Transactions on Software Engineering* 44 (2018), 651–668.

[2] Async/Await 2018. https://blogs.msdn.microsoft.com/dsyme/2007/10/10/introducing-f-asynchronous-workflows/.

[3] Earl T. Barr and Mark Marron. 2014. Tardis: Affordable Time-travel Debugging in Managed Runtimes. In *OOPSLA*.

[4] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-travel Debugging for JavaScript/Node.Js. In *FSE*.

[5] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *PPoPP*.

[6] Gavin Bierman, Erik Meijer, and Wolfram Schulte. 2005. The Essence of Data Access in Cω: The Power is in the Dot!. In *ECOOP*.

[7] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *PLDI*.

[8] Frederick P. Brooks, Jr. 1987. No Silver Bullet Essence and Accidents of Software Engineering. *Computer* 20 (1987), 10–19.

[9] Chromium 2018. V8 doesn't stable sort. https://bugs.chromium.org/p/v8/issues/detail?id=90.

[10] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. In *ASPLOS*.

[11] Dave Clarke and Sophia Drossopoulou. 2002. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*.

[3] The BOSQUE specification, parser, type checker, reference interpreter, and IDE support are open-sourced and available at https://github.com/Microsoft/BosqueLanguage.

[12] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language Systems* 13 (1991), 451–490.

[14] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (Eds.). 1972. *Structured Programming*. Academic Press Ltd., London, UK, UK.

[15] Durable Functions 2019. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[16] elm 2019. https://elm-lang.org/.

[17] Manuel Fähndrich and K. Rustan M. Leino. 2003. Declaring and Checking Non-null Types in an Object-oriented Language. In *OOPSLA*.

[18] Manuel Fahndrich and Songtao Xia. 2007. Establishing Object Invariants with Delayed Types. In *OOPSLA*.

[19] R. W. Floyd. 1967. Assigning meanings to programs. *Mathematical Aspects of Computer Science* 19 (1967), 19–32.

[20] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, and Irene Zhang. 2018. *A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications*. Technical Report.

[21] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*.

[22] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL*.

[23] Ashutosh Gupta and Andrey Rybalchenko. 2009. InvGen: An Efficient Invariant Generator. In *CAV*.

[24] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *CGO*.

[25] Michael Hind. 2001. Pointer Analysis: Haven'T We Solved This Problem Yet?. In *PASTE*.

[26] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. 2013. Unifying Structured Recursion Schemes. In *ICFP*.

[27] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12 (1969), 576–580.

[28] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Notices* 39 (2004), 92–106.

[29] immutable-js 2019. https://github.com/immutable-js/immutable-js.

[30] Java Streams 2019. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

[31] JavaScript 2015. ES6. http://www.ecma-international.org/ecma-262/6.0/index.html.

[32] Joe Duffy Blog 2010. On partially-constructed objects. http://joeduffyblog.com/2010/06/27/on-partiallyconstructed-objects/.

[33] Deepak Kapur, David R. Musser, and Alexander A. Stepanov. 1982. Tecton: A Language for Manipulating Generic Objects. In *Program Specification*.

[34] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[35] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *SOSP*.

[36] Stephen Kou and Jens Palsberg. 2010. From OO to FPGA: Fitting Round Objects into Square Hardware?. In *OOPSLA*.

[37] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential Assertion Checking. In *ESEC/FSE*.

[38] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *PLDI*.

[39] K. Rustan M. Leino and Francesco Logozzo. 2005. Loop Invariants on Demand. In *APLAS*.

[40] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal Automated Reasoning* 43 (2009), 363–446.

[41] LINQ 2019. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.

[42] Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *VHLL*.

[43] Lodash 2019. https://lodash.com/.

[44] Matthew C. Loring, Mark Marron, and Daan Leijen. 2017. Semantics of Asynchronous JavaScript. In *DLS*.

[45] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *PACT*.

[46] Mark Marron, Mario Méndez-Lojo, Manuel Hermenegildo, Darko Stefanovic, and Deepak Kapur. 2008. Sharing Analysis of Arrays, Collections, and Recursive Structures. In *PASTE*.

[47] John McCarthy and Patrick J. Hayes. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*. Edinburgh University Press, 463–502.

[48] Steve McConnell. 2004. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

[49] Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *FPCA*.

[50] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-based Points-to Analysis. In *OOPSLA*.

[51] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering and Methodology* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

[52] Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA.

[53] Antoine Miné. 2006. The Octagon Abstract Domain. *Higher Order Symbolic Computation* 19 (2006), 31–100.

[54] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[55] David R. Musser, Gilmer J. Derge, and Atul Saini. 2001. *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[56] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg.

[57] James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Mark S. Miller. 2016. The Left Hand of Equals. In *Onward!*

[58] Node.js 11 2019. https://nodejs.org/.

[59] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven Synthesis. In *PLDI*.

[60] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-level Programming Embedded in F*. In *ICFP*.

[61] React 2019. https://reactjs.org/.

[62] ReasonML 2019. https://reasonml.github.io/.

[63] Redux 2019. https://github.com/reduxjs/redux.

[64] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*.

[65] Rust 2019. https://www.rust-lang.org/.

[66] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Non-linear Loop Invariant Generation Using GröBner Bases. In *POPL*.

[67] SemVer 2018. Semantic Versioning 2.0.0. https://semver.org/.

[68] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31*. 7751–7762.
[69] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *POPL*.
[70] Alexandru Sălcianu and Martin Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *VMCAI*.
[71] Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *POPL*.
[72] TypeScript 2019. 3.3. https://www.typescriptlang.org/.
[73] Underscore.js 2019. https://underscorejs.org/.
[74] W3C 2013. Selectors API. https://www.w3.org/TR/selectors-api2/.
[75] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*.

Appendix A: Tic-Tac-Toe Code

The main body of the paper contains numerous examples of BOSQUE code which illustrate key novel and interesting features of the language. However, to provide a more organic view for how the language works and flows *in the large* this appendix contains the source code for a simple `tic-tac-toe` program that supports both updating the board with user supplied moves, making an automated computer move, and managing the various game state.

```
namespace NSMain;

entity Board {
    const playerX: String[PlayerMark] = 'x'#PlayerMark;
    const playerO: String[PlayerMark] = 'o'#PlayerMark;

    const allCellPositions: List[[Int, Int]] = List[[Int, Int]]@{
        @[ 0, 0 ], @[ 1, 0 ], @[ 2, 0 ],
        @[ 0, 1 ], @[ 1, 1 ], @[ 2, 1 ],
        @[ 0, 2 ], @[ 1, 2 ], @[ 2, 2 ]
    };

    const winPositionOptions: List[List[[Int, Int]]] = List[List[[Int, Int]]]@{
        List[[Int, Int]]@{ @[ 0, 0 ], @[ 0, 1 ], @[ 0, 2 ] },
        List[[Int, Int]]@{ @[ 0, 1 ], @[ 1, 1 ], @[ 2, 1 ] },
        List[[Int, Int]]@{ @[ 0, 2 ], @[ 1, 2 ], @[ 2, 2 ] },

        List[[Int, Int]]@{ @[ 0, 0 ], @[ 1, 0 ], @[ 2, 0 ] },
        List[[Int, Int]]@{ @[ 1, 0 ], @[ 1, 1 ], @[ 1, 2 ] },
        List[[Int, Int]]@{ @[ 2, 0 ], @[ 2, 1 ], @[ 2, 2 ] },

        List[[Int, Int]]@{ @[ 0, 0 ], @[ 1, 1 ], @[ 2, 2 ] },
        List[[Int, Int]]@{ @[ 0, 2 ], @[ 1, 1 ], @[ 2, 0 ] }
    };

    //Board is a list of marks, indexed by x,y coords
    field cells: List[String[PlayerMark]?];

    factory static createInitialBoard(): { cells: List[String[PlayerMark]?] } {
        return @{ cells=List[String[PlayerMark]?]::createOfSize(9, none) };
    }

    method getOpenCells(): List[[Int, Int]] {
        return Board::allCellPositions->filter(fn(pos) => {
            return !this->isCellOccupied(pos[0], pos[1]);
        });
    }

    method getCellContents(x: Int, y: Int): String[PlayerMark]?
        requires 0 <= x && x < 3 && 0 <= y && y < 3;
    {
        return this.cells->at(x + y * 3);
    }

    method isCellOccupied(x: Int, y: Int): Bool {
        return this->getCellContents(x, y) != none;
    }

    method isCellOccupiedWith(x: Int, y: Int, mark: String[PlayerMark]): Bool
        requires mark == Board::playerX || mark == Board::playerO;
    {
        return this->getCellContents(x, y) == mark;
    }

    method markCellWith(x: Int, y: Int, mark: String[PlayerMark]): Board
        requires mark == Board::playerX || mark == Board::playerO;
        requires 0 <= x && x < 3 && 0 <= y && y < 3;
        requires !this->isCellOccupied(x, y);
```

```
    {
        return this<~(cells=this.cells->set(x + y * 3, mark));
    }

    hidden method checkSingleWinOption(opt: List[[Int, Int]], mark: String[PlayerMark]): Bool {
        return opt->all(fn(entry) => this->isCellOccupiedWith(entry[0], entry[1], mark));
    }

    hidden method checkSingleWinner(mark: String[PlayerMark]): Bool {
        return Board::winPositionOptions->any(fn(opt) => this->checkSingleWinOption(opt, mark));
    }

    method checkForWinner(): String[PlayerMark]? {
        if(this->checkSingleWinner(Board::playerX)) {
            return Board::playerX;
        }
        elif(this->checkSingleWinner(Board::playerO)) {
            return Board::playerO;
        }
        else {
            return none;
        }
    }
}

entity Game {
    field winner: String[PlayerMark]? = none;
    field board: Board = Board@createInitialBoard();

    method hasWinner(): Bool {
        return this.winner != none;
    }

    method getWinner(): String[PlayerMark]
        requires this->hasWinner();
    {
        return this.winner->as[String[PlayerMark]]();
    }

    method makeAutoMove(mark: String[PlayerMark], rnd: Int): Game
        requires !this->hasWinner();
    {
        var! nboard: Board;
        if(!this.board->isCellOccupied(1, 1)) {
            nboard = this.board->markCellWith(1, 1, mark);
        }
        else {
            var opts = this.board->getOpenCells();
            var tup = opts->uniform(rnd);
            nboard = this.board->markCellWith(...tup, mark);
        }

        return this<~( board=nboard, winner=nboard->checkForWinner() );
    }

    method makeExplicitMove(x: Int, y: Int, mark: String[PlayerMark]): Game
        requires !this.board->isCellOccupied(x, y);
    {
```

```
        var nboard = this.board->markCellWith(x, y, mark);
        return this<~( board=nboard, winner=nboard->checkForWinner() );
    }
}

entity PlayerMark provides Parsable {
    field mark: String;

    override static tryParse(str: String): PlayerMark | None {
        return (str == "x" || str == "o") ? PlayerMark@{ mark=str } : none;
    }
}
```