High Assurance Software for Financial Regulation and Business Platforms

Stephen Goldbaum[1], Attila Mihaly[1], Tosha Ellison[2], Earl T. Barr[3], and Mark Marron[4]

[1] Morgan Stanley, USA
{stephen.goldbaum, attila.mihaly}@morganstanley.com
[2] Fintech Open Source Foundation (FINOS), USA
tosha.ellison@finos.org
[3] University College London, UK
e.barr@ucl.ac.uk
[4] Microsoft Research, USA
marron@microsoft.com

Abstract. The financial technology sector is undergoing a transformation in moving to open-source and collaborative approaches as it works to address increasing compliance and assurance needs in its software stacks. Programming languages and validation technologies are a foundational part of this change. Based on this viewpoint, a consortium of leaders from Morgan Stanley and Goldman Sachs, researchers at Microsoft Research, and University College London, with support from the Fintech Open Source Foundation (FINOS) engaged to build an open programming stack to address these challenges.
The resulting stack, Morphir, centers around a converged core intermediate representation (IR), MorphirIR, that is a suitable target for existing languages in use in major investment banks and that is amenable to analysis with formal methods technologies. This paper documents the design of the MorphirIR language and the larger Morphir ecosystem with an emphasis on how they benefit from and enable formal methods for error checking and bug finding. We also report our initial experiences working in this system, our experience using formal validation in it, and identify open issues that we believe are important to the Fintech community and relevant to the research community.

Keywords: Fintech · Intermediate Language · Software Assurance.

1 Introduction

The financial technology sector is undergoing a transformation in embracing open-source and collaborative approaches to managing their collective industry challenges. Many of those challenges involve sharing data models as well as business logic and calculations. A prime example is the focus on leveraging community initiatives around the digitization of regulatory needs to streamline industry efficiency. Managing the myriad regulations that any single organization must

comply with is a enormous task. Reuters Regulatory Intelligence tracks regulatory changes across 190 countries and reported an average of 257 daily alerts in 2020[5]. Programming languages and the tooling around them play a core role in managing the complexity and engineering effort involved in fulfilling these regulatory requirements and implementing critical business applications.

The current approach to high assurance development is based on classical process quality and provenance. Legal teams review regulations, or process descriptions, to generate a set of rules and compliance examples. Development teams use these documents to produce the actual code and build an architecture that implements the systems/regulations described in the documents. In this classic, waterfall style method, the assurance of quality is based on the documentation, workflow checklists, and conformance tests provided by the legal team. While effective, this process is a time consuming and expensive way to develop high assurance systems. Since different companies have different platforms and systems, this work is duplicated multiple times. Beyond the raw costs inherent to this approach, the increasing complexity of financial rules creates situations where the rules are interpreted differently by different systems, resulting in increased regulatory uncertainty and issues with interoperability.

These challenges require broad community engagement to overcome. Thus, a consortium of leaders from Morgan Stanley and Goldman Sachs and researchers at Microsoft Research and University College London drove this project, with support from the Fintech Open Source Foundation (FINOS). The core challenge involved creating a mechanism to share rules, calculations, and their data models in a form that spans the wide range of current and future technologies across the industry. In this paper, we describe our experience in creating a programming and validation ecosystem that can support the needs of financial services companies in developing and delivering high assurance software and regulatory compliance software artifacts. Three interlocking goals guided our work:

1. Developing a core IR and programming model that converge existing languages to leverage the hard won knowledge embedded in them and to maximise its deployability in and sharing throughout the ecosystem;
2. Setting up a baseline validation methodology to provide assurance guarantees on programs in the core IR; and
3. Creating workflows that help the wider community integrate their frontend platforms and backend validation tools into the ecosystem.

To achieve the first two goals, we developed MorphirIR, a converged, core intermediate representation (IR) with two key properties: 1) it is a suitable target for existing languages in use in major investment banks and 2) it is amenable to analysis with formal methods technologies. MorphirIR, described in Section 4, is based on a convergence of two languages — Morphir from Morgan Stanley [18] and Legend from Goldman Sachs [14] — with simplifications made to improve its amenability to analysis.

[5] According to Thomson Reuters "Cost of Compliance 2021" 78% of market participants they surveyed expect the amount of regulatory information published by regulators and exchanges to increase in 2021.

The Fintech space has many bespoke domain specific and contract languages (DSLs) that serve valuable purposes in their niche but are not large enough to justify the cost of building a full toolchain. MorphirIR provides a core set of language constructs that are sufficient to describe common business concepts, while remaining simple enough to provide an easy translation target. As described in Section 4, the converged MorphirIR language is based on a standard let-style functional core calculus with algebraic types and polymorphic collections. This core is augmented with a number of commonly useful types and operators such as decimal numbers or dispatch tables. This allows a wide range of source languages used in the community to, with a minimal investment to build a translator, gain access to the full checking and compilation tooling stack provided.

MorphirIR has many features that make it well suited for formal analysis. Its language core is purely functional, referentially transparent, fully deterministic, and utilizes a small number of collection functors (instead of recursion) for most iterative processing. To provide a baseline for the effectiveness and value that formal methods can provide beyond the current stacks, we transpiled MorphirIR code to Microsoft Research's Bosque language [3]. As described in Section 3, the focus is on providing simple ways to encode high-level intents and approaches to analyzing the code along with the intents to provide actionable results and/or increased confidence that the code successfully implements the specified properties.

Our experience with these systems, the corresponding workflows, and our work to make these systems widely available are described in the Experience Report (Section 5). Our experience translating existing languages, including Elm, Legend, Bosque, and a few small DSLs show the viability of MorphirIR as a shared intermediate language for this space. The initial experiences with validation have been similarly positive. The workflow, which supports full refutations of errors, generation of witness failure inducing inputs, and partial checking [15,19] results in an easy to use system that consistently provides actionable feedack and confidence.

Based on these experiences and community feedback to date, we believe Morphir establishes the basis for a vibrant software ecosystem in the Fintech space as well as a unique opportunity to advance the state of the art in formal methods and their practical application. Section 5 outlines where the expertise and experience of the formal methods community will be particularly useful. These areas range from the direct opportunity of demonstrating the effectiveness and utility of new techniques in the Fintech proving ground by integrating them into Morphir's validation pipeline, to insights on the design of richer specification languages for MorphirIR, to the challenge of extending Morphir's validation stack from just code to the larger ecosystem of data and process compliance, a space that calls for hybridising AI and verification techniques.

The contributions of this paper are:

- A report on our experience of building the MorphirIR core language as a backend target for regulatory modeling languages and business platforms.

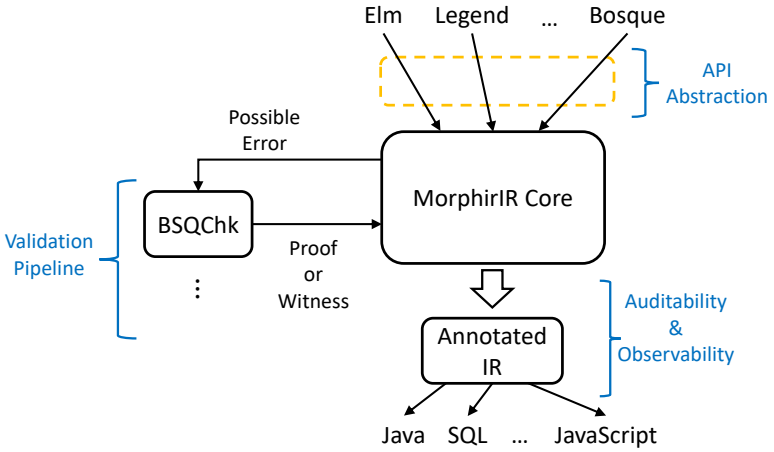

Fig. 1. MorphirIR technology stack.

- Mapping of the MorphirIR language into the Bosque language and checking ecosystem as a baseline for formal quality assurance.
- A report on our experience on using this tooling workflow with code coming from Elm and Bosque applications.
- A fully open source language and analysis stack for the community, including a suite of annotated code as an evaluation benchmark.

2 MorphirIR Stack

The converged MorphirIR core language provides a shared compiler and runtime platform. The diagram in Figure 1 shows the components of the Morphir stack and where various members of the community are interacting with the core MorphirIR language. The MorphirIR core language (Section 4) sits at the center of the diagram and is the central component that enables innovation in the rest of the stack.

2.1 Surface Languages

There are several surface languages, including Elm, Bosque, and Legend, that can target the MorphirIR language. Currently, each one uses a custom transpiler pass and interoperability requires manually projecting from the MorphirIR representation to the semantics/types of the source language. The dotted *API abstraction* component is a key open work item needed to develop a higher level vocabulary of type and operation API's to make interoperability more transparent.

2.2 Validation Pipeline

The validation pipeline takes MorphirIR code into an underlying verification language and tool for analysis. The default checker is currently BSQChk (Section 3). Errors or other information are reported back into the original source code using source maps maintained in the MorphirIR core model.

Our philosophy for validation tooling is pragmatic. In an ideal world, we would have a full, and stable, specification for a task which we use to prove that our implementation never fails and satisfies the specification. However, in the world of under-specified and evolving regulations and business application platforms, this ideal does not hold. Instead, we must deal with limited to no specifications and, since our developers do not have time to debug/resolve proof failures, we should be able to provide useful results even when proofs fail.

Thus, we consider the following hierarchy of *confidence boosting* results that ensure useful feedback that either provides assurance the code is free of errors or provides actionable information to fix a problem:

1a. Proof of infeasibility for all possible executions
1b. Feasibility witness input that reaches target state
2a. Proof of infeasibility on under-approximated executions
2b. No witness input found before search time exhausted

The 1a and 1b cases are our ideal outcomes where the system either proves that the error is infeasible for all possible executions or provides a concrete witness that can be used by the developer to debug the issue. The 2a and 2b cases represent useful *best effort* results. While they do not entirely rule out the possibility that a given error can occur, they do provide a substantial boost in a developer's confidence that the error is infeasible.

2.3 Monitoring and Compilation

Applications in the Fintech sector often run critical software, subject to extensive compliance and auditing requirements. A common regulatory requirement involves demonstrating to auditors exactly why any decision was made for up to several years in the past. The Morphir tooling takes advantage of its functional purity to reevaluate decisions to produce automated audit-quality explanations. Explanations can take a variety of forms, such as natural language explanations or flow charts. Evaluation can be injected into the code to publish explanations through observability technologies or can be executed after the fact, for example through an interactive web page that allows users to replay decision evaluation much as a debugger would. Figure 1 shows a dedicated pipeline for providing applicaton behavior observability [1], runtime safety monitoring, and explanatory logic into the final executable images.

The explanatory logic component is an interesting feature that plays an important role in audit compliance and in many business applications. Consider the (simplified) regulatory code below derived from the 173 page FR 2052a Liquidity Reporting instructions [13] for computing the category of an inflow:

```
function classify(cashflow: CashFlow): FedCode {
  if(netCashUSD(cashflow) >= 0) then
    if(isOnshore(cashflow)) then IU1 else IU2
  else
    IU4
}
```

For a given transaction, an auditor (or analyst) may need to know why a flow was categorized as `IU2`. In most systems, this would require looking into the code and manually tracing the execution flows. Most analysts at a trading desk, or, in this case, accountants, would not be comfortable with this type of error prone task. Thus, the MORPHIR backend can automatically inject automated logging code for each branch to record which are taken and the values of the arguments. Thus, if a user sees a flow categorized as `IU2` (an *Offshore Placement*), the MORPHIR system can *explain* this result by noting in the trace log that the `netCashUSD(cashflow)` was positive and the flow was offshore e.g. `!isOnshore(cashflow)`.

Finally, the MORPHIRIR stack supports emitting source code in a variety of languages for integration into the desired execution environment. The Java Virtual Machine (via Scala or Java) is the standard output; SQL and JavaScript are also supported. Each of these target languages currently requires a custom emitter implementation but, as the MORPHIRIR language has special support for types like Decimal and BigNat plus an opinionated container library, it also requires non-trivial work to ensure full runtime support in each target as well. The MORPHIR stack also provides cloud deployment and distributed execution support via integration with the Dapr [4] platform.

3 Validation Methodology

The validation workflow for MORPHIRIR programs is modular to enable a variety of tooling for either general correctness properties or specialized analyses for specific domains — *e.g.* checking for numerical stability or applying lint-style checks to specific sections of code. In this paper, we focus on our experience with the BOSQUE language's validation system [15].

3.1 BSQChk Validation Workflow

The BSQCHK checker first builds the code under analysis by translating the MORPHIRIR code to the BOSQUE representation. Given the structure of the MORPHIRIR code (Section 4), this translation is mostly a 1-1 process with bookkeeping to build source maps for error reporting. After this translation, BSQCHK loads the code and enumerates all possible error conditions it can check. For each identified error, BSQCHK follows the algorithm shown in Figure 2.

The first action is to check if the error can be refuted under various definitions of simplified models of the program – limited sizes on input values and numeric bitwidth sizes ranging from 4-16. If the error can be show to be impossible in these simplified models then the checker attempts a refutation proof with no

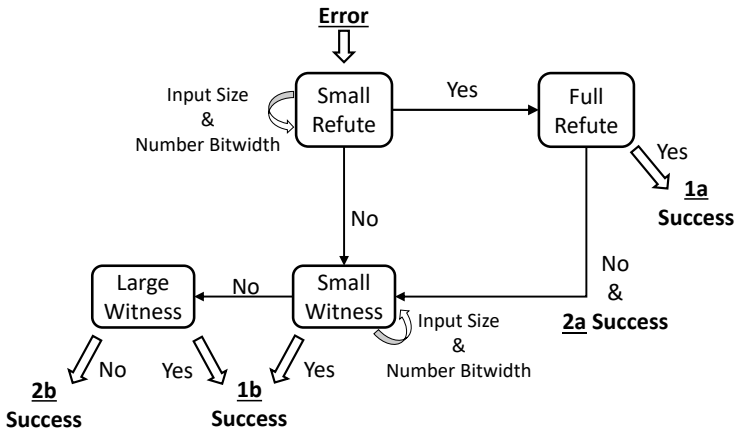

Fig. 2. BSQChk checker workflow.

limits on the size of inputs and 64 bitwidth sized numbers. If this is successful then the checker has shown that the error is infeasible on all executions and we achieved the highest quality, 1a, confidence level.

If we succeeded in proving the error infeasible for simplified models of the program, but then failed to prove the infeasibility for the full case, we still achieved the partial, 2a, confidence level.

If the refutation proofs fail then we search for a witness input for the error. The small model search incrementally expands the input sizes and bitwidths up to size 16. If we find an input that reaches the target error then we have succeeded in producing a high value actionable result for the developer, 1b, in our quality confidence level. With this result we know there is a real failure and have a small input that can be used to trigger and debug it.

In the case we cannot generate a small witness we make a final witness generation attempt without limits on the input sizes and at the full 64 bitwidth for numbers. If we find an input that reaches the target error then we have succeeded in producing a high value actionable result for the developer. Otherwise we produce our minimal success result, 2b, where we aggressively explored the input space.

The code in Figure 3 shows a MorphirIR implementation of a business application modeling example. The code snippet is focused on the `available` function. This function computes the number of items still available to sell based on the number at start of the day (`initialPosition`) and the list of buy transactions (`buys`) so far. As a precondition it asserts that the `initialPosition` is non-negative. As a postcondition it asserts that the result `$result` is bounded by the initial position value.

The code to compute the number of buy transactions that have been completed successfully and the sum of the quantities from these purchases is concisely

```
function available(initialPosition: BigInt, buys: List<Response>): BigInt
    requires initialPosition >= 0;
    ensures $result <= initialPosition;
{
    let sumOfBuys = buys
        .filterType<BuyAccepted>()
        .map(fn(x) => x.quantity)
        .sum()
    in
    initialPosition - sumOfBuys;
}

...
type Response =
  BuyAccepted of {
      productId: String;
      price: Decimal;
      quantity: BigInt; //<--- should be BigNat
    }
  | BuyRejected of {
      ...
    }
```

Fig. 3. BOSQUE implementation of order processing code.

expressed using the functor chain `buys.filterType<BuyAccepted>().map(fn(x)=> x.quantity ).sum()`. While this code is conceptually simple from a developer viewpoint, its actual strongest postcondition logic semantics are quite complex. They include a subset relation and predicate satisfaction relation on the `filter`, a quantified user defined binary relation with the `map`, and an inductively defined relation as a result of the `sum`. Thus, trying to prove that the postcondition is satisfied (or finding an input that demonstrates the error is possible) is a challenging task involving inductive reasoning, relationships between container sizes and contents, and quantified formula.

Despite these complexities, the BSQCHK checker can model this code, in strongest postcondition form, as a logical formula in a decidable fragment of first-order logic and instantaneously solve it [15]. The result is the following assignment which satisfies all the input constraints and violates the ensures condition:

$$\text{initialPosition} = 0 \wedge \text{buys} = \text{List<Response>}(\text{BuyAccepted}(\text{“}a\text{”}, 0.0, -1))$$

A developer can run the application on this witness, investigate the problem, and identify the appropriate course of action to resolve the issue. In this case, the fix uses the fact that the MORPHIRIR language supports `BigNat`, in addition to `BigInt`, numbers to ensure that the buy quantity is always non-negative.

After this simple change, rerunning BSQCHK instantaneously reports that the program state where the `ensures` clause is *false* is unreachable for all possible inputs. All of this analysis and proving is fully automated and does not require any assistance, knowledge of the underlying theorem prover, or use of specialized logical assertions by the developer.

4 MorphirIR Core Langauge

The MorphirIR language provides a unified target IR for various modeling and platform development toolchains in use in the Fintech space and leverages findings from recent work [15,21] on language design for automated reasoning, to support advanced verification, error checking, and analysis tooling.

The initial source languages targeting this IR are a dialect of Elm (used in the Morphir [18] stack) and Legend [14]. As these systems were built for modeling financial data, logic, and calculations for business critical operations, their designs already had most of the features we would want from the viewpoint of building a high assurance ecosystem. They are pure, functional, and referentially transparent. From this base, we refined the IR design based on experience with the Bosque [3] language and tooling stack — making the programming model fully deterministic, including additional primitive types, and expanding the set of collection functors in the core library. The full language type grammar is shown in Figure 4 and the expression language in Figure 5.

4.1 Types and Values

Primitives: MorphirIR provides the standard assortment of primitive types and values including `Bool`, `Int`, and `Float` values. As the language is focused on financial computation, we also provide a `Decimal` type. To support high assurance programming, MorphirIR also supports overflow free `BigInt` numbers, plus, the generally useful positive only numeric refinement types `Nat` and `BigNat`. The MorphirIR `String` type represents immutable unicode string values.

Tuples and Records: Structural *Tuple* and *Record* types provide standard forms of self describing types. MorphirIR records and tuples are always closed, *e.g.* they must explicitly include all indices/properties.

Algebraic Data Types: The primary means of organizing data in MorphirIR is classic algebraic datatypes. The members of the ADT can have named or positional members.

Parametric Containers: Following the design of principles of the Bosque language, we include `List<T>`, `Set<T>`, and `Map<K, V>` as core types in the MorphirIR language. These types support a rich set of functors that enable the majority of iterative processing tasks to be described without the use of arbitrary iteration or recursion (see Figure 6).

4.2 Expressions

Constants and Variables: MorphirIR has the usual constants for booleans, numbers, and strings. Variables are used for function parameters and `let` bindings in the usual way.

$$
\begin{array}{rcl}
\text{Primitive} & := & \texttt{Bool} \mid \texttt{Nat} \mid \texttt{Int} \mid \texttt{BigInt} \mid \texttt{BigNat} \mid \texttt{Float} \mid \texttt{Decimal} \mid \texttt{String} \\
\text{Tuple} & := & [Type_1, \ldots, Type_k] \\
\text{Record} & := & \{p_1 : Type_1, \ldots, p_k : Type_k\} \\
\text{ADT} & := & \texttt{type}\ Ty = C_1\ \texttt{of}\ CRepr_1 \mid \ldots \mid C_k\ \texttt{of}\ CRepr_k \\
\text{CRepr} & := & [Type_1, \ldots, Type_k] \mid \{f_1 : Type_1, \ldots, f_k : Type_k\} \\
\text{Container} & := & \texttt{List<T>} \mid \texttt{Set<T>} \mid \texttt{Map<K, V>} \\
\text{Function} & := & (Type_1, \ldots, Type_k)\ \texttt{->}\ Type_{result} \\
\text{Type} & := & \textit{Primitive} \mid \textit{Tuple} \mid \textit{Record} \mid \textit{ADT} \mid \textit{Container} \mid \textit{Function}
\end{array}
$$

Fig. 4. MorphirIR Types.

Primitive Operators: The language provides a standard set of operations on primitive types including, logical, arithmetic, and comparison operations. Arithmetic operations on numeric types are always checked for overflow, underflow, and div by 0. We do not allow implicit type coercions, so these operators are only defined for values of the same types and conversions for mixed types must be explicit. MorphirIR also provides the specialized `//` operator for integer division (as opposed to the `/` operator for floating point divsion).

Constructors and Destructors: The constructor operations for the tuples, records, and algebraic data types have familiar semantics. *Patterns* provide a type safe way to destruct a value and access the constituent values.

Lambda: The use of functors to process collections is a major part of MorphirIR programs. However, the widespread use of unrestricted higher order code greatly increases the complexity and computational cost of program analysis. Combined with our experiences, and the code style guidelines we have used, we opted to restrict the use of raw lambdas. Thus, syntactically, lambda constructors are only permitted in direct application positions. Consider the code:

```
function okc(l: List<Int>): Int {
    return l.filter(fn(x) => x >= 0).size(); //ok - direct position
        constructor
}

function okp(l: List<Int>, p: fn(Int) -> Bool): Int {
    return l.filter(p).size(); //ok - direct position from parameter
}

function invalid(l: List<Int>): Int {
    let fun = fn(x) => x >= 0; //error - lambda not in direct position

    return l.filter(fun).size();
}
```

In the first function, `okc`, the lambda expression is in the direct call position to the list `filter` functor. In the second function, `okp`, the lambda is a parameter to the function which must be passed in from a direct declaration. In contrast, in

the `invalid` function, the lambda expression is indirectly assigned to a variable before being passed to the `filter` functor and is an error in MorphirIR.

Function and Lambda Invocation: Function invocations are statically resolvable direct calls to the named function, or named lambda parameter, with the given arguments. Since lambda uses are syntactically restricted to the direct call positions, these uses can either be defunctionalized, so that all calls become fully static (which is done when translating to Bosque for verification), or they can be dynamically constructed as closures when compiling to a language like JavaScript.

Assert: Assertions can be explicitly added to check for user defined conditions and take a `Bool` typed condition expression along with a continuation *ok* expression. When the *assert* expression evaluates to `true` then the *ok* expression is evaluated as the result otherwise the programs fails with an error.

Control Flow: Control flow is handled by a classic `if-then-else` construct or a pattern matching and destructuring `case` operator. The case operation finds the first condition in the list that matches the type of the value that is dispatched on and binds variable names to the specified values from the constructor. The case can be used on algebraic types, records, and tuples. There is a special wildcard case "_" which matches everything and the cases must be exhaustive.

Decision Tables: Sets of rules that define business logic are a frequent occurrence in Fintech applications. These rules can be encoded as nests of *case*, *let*, and *if-then-else* statements. However, these encodings are complex and result in the loss of information about the intent of the original rule structure. The MorphirIR language includes decision tables as a first-class construct (the *Table* row in Figure 5). The argument expressions are evaluated and bound to a set of variables. Then, in this scope, the *Opt* clauses are evaluated in order. For each clause the expressions in the list are evaluated in short circuit && order. If all the expressions in the list are *true* then the result of the expression is the evaluation of the tail Exp_{result}. If the set is not exhaustive *or* any *Opt* is unreachable it is a program error.

```
function getDecision(f: Facts, env: Jurisdiction): Decision
  dispatch(docType = f.documentType, law = getGoverningLaw(env, f)) [
    [docType == DRV]                  => Yes,
    [docType == ISDA, law == England] => Yes,
    [docType == ISDA]                 => No
  ]
}
```

The `getDecision` program shows a (simplified) table for computing business rules around derivatives handling. In this code if the `docType` is `DRV` the result is always a `Yes`. In the other case the result depends on if the governing law is `England`. Note that if we accidentally switched the last 2 opt clauses, so that the opt where `law == England` was last, this would be an error.

$$
\begin{array}{rcl}
\text{Const} & := & \texttt{true} \mid \texttt{false} \mid i \mid f \mid s \\
\text{Var} & := & v \\
\text{Operator} & := & (\texttt{!}\mid\texttt{+}\mid\texttt{-})\,\mathit{Exp} \mid \mathit{Exp}\,(\texttt{+}\mid\texttt{-}\mid\texttt{*}\mid\texttt{/}\mid\texttt{//})\,\mathit{Exp} \mid \mathit{Exp}\,(\texttt{\&\&}\mid\texttt{||})\,\mathit{Exp} \\
\text{Compare} & := & \mathit{Exp}\,(\texttt{==}\mid\texttt{!=})\,\mathit{Exp} \mid \mathit{Exp}\,(\texttt{<}\mid\texttt{<=}\mid\texttt{>}\mid\texttt{>=})\,\mathit{Exp} \\
\text{Cons} & := & [\mathit{Exp}_1, \ldots, \mathit{Exp}_j] \mid \{f_1 = \mathit{Exp}_1, \ldots, f_j = \mathit{Exp}_j\} \mid \mathit{Type}(\mathit{Exp}_1, \ldots, \mathit{Exp}_j) \\
\text{Lambda} & := & \texttt{fn}(v_1, \ldots, v_k)\ \texttt{=>}\ \mathit{Exp} \\
\text{Invoke} & := & \mathit{fname}(\mathit{Exp}_1, \ldots, \mathit{Exp}_j) \mid \mathit{Exp}.\mathit{iname}(\mathit{Exp}_1, \ldots, \mathit{Exp}_j) \\
\text{Assert} & := & \texttt{assert}\ \mathit{Exp}_c\ \texttt{then}\ \mathit{Exp}_{ok} \\
\text{If} & := & \texttt{if}\ \mathit{Exp}_c\ \texttt{then}\ \mathit{Exp}_t\ (\texttt{elif}\ \mathit{Exp}_c\ \texttt{then}\ \mathit{Exp}_t)\ \texttt{*}\ \texttt{else}\ \mathit{Exp}_f \\
\text{Case} & := & \texttt{case}\ \mathit{Exp}\ \texttt{of}\ (\mathit{Pattern}\ \texttt{=>}\ \mathit{Exp} \mid \texttt{\_}\ \texttt{=>}\ \mathit{Exp})\ \texttt{*} \\
\text{Let} & := & \texttt{let}\ v = \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \mid \texttt{let}\ \mathit{Pattern} = \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \\
\text{Table} & := & \texttt{dispatch}(v_1 = \mathit{Exp}_1, \ldots, v_j = \mathit{Exp}_j)[\mathit{Opt}_1, \ldots, \mathit{Opt}_k] \\
 & & \text{where } \mathit{Opt}_i \ := \ [\mathit{Exp}_1, \ldots, \mathit{Exp}_m] \Rightarrow \mathit{Exp}_{result} \\
\text{Pattern} & := & [v_1, \ldots, v_j] \mid \{f_1 = v_1, \ldots, f_j = v_j\} \mid \mathit{Type}(v_1, \ldots, v_j) \\
 & & \text{where each } v_i \text{ is a variable } v \text{ or is the ignore match ``\_''} \\
\text{Exp} & := & \mathit{Const} \mid \mathit{Var} \mid \mathit{Operator} \mid \mathit{Compare} \mid \mathit{Cons} \mid \mathit{Lambda} \mid \mathit{Invoke} \\
 & & \mid \mathit{Assert} \mid \mathit{If} \mid \mathit{Case} \mid \mathit{Table} \mid \mathit{Let}
\end{array}
$$

Fig. 5. MorphirIR Expressions.

Let: The `let` operation binds a value to a variable name in the expected manner *or* binds a set of variables to the destructured value of a tuple/struct/datatype.

4.3 Containers and Operations

The standard collection libraries play a central role in the design and use of MorphirIR, which does not include looping constructs and where the use of recursion is discouraged. Instead, we lean heavily on the use of a rich set of collection operations to support iterative data processing. This has the advantage of aligning well with development guidelines for high assurance software and, as Marron and Kapur showed [15], allows us to reason about most container manipulating code using decidable theories that are amenable to solving using existing SMT provers. Figure 6 provides a brief summary of these operations.

List Operations: Lists can be constructed using a number of algebraic primitives, including explicit initialization with fixed values, initialization using the contents of another container, concatenation, and slicing. In addition, lists also provide the usual *size*, *get*, and *find* index operations.

The functor family of algorithms provide higher order functions that reshape lists based on user specified functions. These can *filter* subsets of elements in

$$
\begin{array}{rcl}
\text{List Cons} & := & \texttt{List}(Exp_1, \ldots, Exp_j) \mid \texttt{ListFrom}(Exp) \mid \texttt{ListRange}(Exp_{low}, Exp_{high}) \\
& & \mid \texttt{concat}(Exp_1, Exp_2) \mid \texttt{slice}(Exp, Exp_{start}, Exp_{end}) \\
\text{List Primitive} & := & \texttt{size}(Exp, Exp_{index}) \mid \texttt{get}(Exp, Exp_{index}) \mid \texttt{find}(Exp, Fn) \\
\text{List Functors} & := & \texttt{filter}(Exp, Fn) \mid \texttt{map}(Exp, Fn) \mid \texttt{join}(Exp_1, Exp_2, Fn) \\
\text{List Ops} & := & \texttt{zip}(Exp_1, Exp_2) \mid \texttt{reverse}(Exp) \mid \texttt{sort}(Exp, Fn) \mid \texttt{unique}(Exp, Fn) \\
\text{List Reduce} & := & \texttt{sum}(Exp) \mid \texttt{min}(Exp) \mid \texttt{max}(Exp) \mid \texttt{reduce}(Exp, Exp_{init}, Fn) \\
\text{Set Cons} & := & \texttt{Set}(Exp_1, \ldots, Exp_j) \mid \texttt{SetFrom}(Exp) \\
& & \mid \texttt{union}(Exp_1, Exp_2) \mid \texttt{intersect}(Exp_1, Exp_2) \\
\text{Set Primitive} & := & \texttt{empty}(Exp) \mid \texttt{has}(Exp, Exp_{key}) \mid \texttt{isSubset}(Exp, Exp_{of}) \\
\text{Set Functors} & := & \texttt{subset}(Exp, Fn) \\
\text{Map Cons} & := & \texttt{Map}(Exp_1, \ldots, Exp_j) \mid \texttt{MapFrom}(Exp) \\
& & \mid \texttt{merge}(Exp_1, Exp_2) \mid \texttt{restrict}(Exp, Exp_{elems}) \\
\text{Map Primitive} & := & \texttt{empty}(Exp) \mid \texttt{has}(Exp, Exp_{key}) \mid \texttt{get}(Exp, Exp_{key}) \\
& & \mid \texttt{keys}(Exp) \mid \texttt{isSubDom}(Exp, Exp_{of}) \\
\text{Map Functors} & := & \texttt{project}(Exp, Fn) \mid \texttt{remap}(Exp, Fn)
\end{array}
$$

Fig. 6. MorphirIR Container Operations.

a list, *map* functions over all elements in a list, or *join* two lists. We also provide operations for reorganizing lists, including the usual *zip*, *reverse*, *sort*, and *unique*. In contrast to many languages which leave the algorithm used for these operations under-specified, MorphirIR ensures these operations are always order stable on the input lists.

The reduce family of algorithms is important as, following Mark and Kapur [15], we do not have a general decidable logical specification for these operations. Thus, we explicitly provide *sum*, *min*, *max* as special common cases of reduction that we can axiomatize fairly effectively and a generic *reduce* that involves heuristic inductive and/or unrolling to encode.

Set and Map Operations: The set and map datatypes are defined only for keys that are numeric or string typed. Further, the map/set enumeration order is defined to be the order of the underlying keys. These restrictions ensure that the key based comparisons are decidable and the behavior of the operations is always full deterministic.

In addition to simplifying the analysis of Sets/Maps via the semantics of the allowable key types and ensuring ordering, we also explicitly limit some parts of the API to reduce the introduction of difficult-to-reason-about constraints. Notably, there is no direct *size* operation, as cardinality and set operations are problematic to reason about simultaneously. With this design, we focus the Set/Map operations on the core contains, lookup, and set theoretic operations they can

provide while encouraging to use of the rich, and simpler to reason about, list operations as the default way to organize data.

5 Experience Report

The initial outcomes of this project have been very positive. The community is already benefiting from the network effects of sharing a core language and runtime. The validation capabilities add an additional value proposition: our initial success with the BSQChk pipeline shows the potential for formal methods in this space. Based on these experiences, we anticipate growing investment in and adoption of the MorphirIR language, platform, and ecosystem throughout the financial services community. Despite (or perhaps because of) these successes, more work needs to be done. We discuss scenarios that we encountered where we believe the Morphir stack can be improved and have begun investigating approaches for realizing those improvements.

5.1 Languages Targeting MorphirIR

To date, the main users of the MorphirIR stack are Morgan Stanley and Goldman Sachs. The bulk of models have been written in the Elm programming language. Elm proved to be a natural match with most constructs directly mapping from Elm to the MorphirIR. Elm support for a small number of data types, such as Decimal and LocalDate, was added via the MorphirIR SDK.

Elm and MorphirIR have fundamentally similar language principles and design. They are both functional and aim for the simplest language without sacrificing expressiveness. The result is a lambda calculus with a few, well-known extensions like if-then-else, let expressions, and pattern-matching. The transpiler code is *ca.* 3Kloc and is mostly a one-to-one mapping with a few exceptions where the Elm code uses constructs that Morphir does not directly support.

The Legend platform uses its own programming language called Pure to implement many features. One of those features is model-to-model mapping. The translation from Pure to MorphirIR first runs these mappings to produce a simplified Pure AST. This core AST is side-effect free and declarative; a subset of MorphirIR's semantics directly expresses it. Thus, the AST into MorphirIR transpilation step is a simple rewrite/rename process.

We are also seeing interest and usage from other members of the Fintech community. The most common use cases come from other entities that have a bespoke domain specific modeling language, often encoding business logic rules, models of financial instruments, or regulatory information, that they use internally or have developed as part of a product offering. Further afield there may also be benefit for smart contract languages like Solidity [25,27] or legal formalization languages like Catala [17].

For these types of DSLs, the MorphirIR platform is very appealing. It eliminates the cost of maintaining the compiler/toolchain/runtime system for the DSL. The network effect of the MorphirIR ecosystem also increases the

value of any DSL ported to it, as it gives them a simple, standard way to interoperate with the wider range of definitions and computations available in MorphirIR. This is particularly valuable in the Fintech space where systems frequently involve codified rules or regulations, which can be large and costly to implement. The ability to reuse, instead of re-implementing, them for every specialized stack has tremendous value. The community interest in expanding the set of surface languages that target the MorphirIR stack also introduces a number of (currently) open challenges.

DSL Translation: The current model for adding a new source language (or DSL) to the MorphirIR stack involves manually translating the source semantics into the MorphirIR semantics and syntax. This is both time consuming and error prone.

Interestingly, when compared to scenarios dealing with full fledged programming languages, these DSLs are often fairly simple and resemble macro systems for concisely encoding business or regulatory rules. This suggests the possibility of partially automating this translation process via the addition of a *macro* system or even a specialized *structured data* transformation language. In particular, if this language included the capability to connect logical assertions from the DSL into the MorphirIR code, this would enable us to generate (partially) verified translations [11].

API Abstraction: As more source languages and DSLs are added to the Morphir system, we believe there will be an increasing need for transparent interoperability support. Given the diversity of concepts in the source languages, *e.g.* Legend includes multiplicity constraints in its model/type language, and the desire for flexibility in the stack, we do not believe it is practical to build a shared universal type language that captures all of these variations.

Instead, we are looking to the world of RESTful systems [7] and the success of integrating polyglot systems there. Simplified systems, such as AWS Smithy [24], have shown great success for building distributed cloud computing systems. Starting from this perspective, we are very interested in constructing a layer that combines types, service calls, and logical constraint specifications. This layer would provide a common interoperation language to components written in different systems, encoding common information in the type system and expressing specialized information, such as the multiplicity data in some Legend constructs, in an expressive constraint language.

5.2 Validation Pipeline

Our experience with the validation pipeline has focused on using BSQChk (Section 2). Our work has focused on *ca.* 4Kloc of regulation code in a dialect of Elm that implements a portion of the U.S. Liquidity Coverage Ratio rules and *ca.* 2Kloc of code implementing a sample trading application. These applications have very few explicit assertions, so error checking is primarily of runtime errors such as invalid casts, div-by-zero, *etc.*

In our experience to date with the trading application code, the checker has found proofs of infeasibility for most errors it analyzes, result 1a in the outcomes list (Section 3). In the remaining cases, the checker has not found any witness failure inputs and has completed with result 2b from our outcome list. Our inspections indicate these situations involve the use of *reduction*, which is not contained in the BSQChk decidable fragment, or intensive bitvector operations, such as converting 64bit ints to/from a Real representation of floating point numbers, so the errors are very likely infeasible although not yet provably so by the checker.

This experience led us to rewrite samples into the Bosque source language, which has a richer type system than Elm and more support for adding pre/post conditions, asserts, and data invariants. The example in Figure 3 comes from one of these experiments and shows how the addition of specifications capturing, even partial, higher-level intents can expose code issues that the checker can successfully analyze. Thus, the major takeaway from our initial work here is the need to find ways to increase the scope of checkable properties.

Enriched MorphirIR Language: In the example code (Figure 3), the fix involves using a refined numeric type. This example is a simple case of the wide range of ways numbers are heavily used in specific, and semantically distinct, ways in these financially focused applications. In practice, base numeric types, like Int and Decimal, are *typedef*'d into many other conceptually distinct types like currency, quantities, conversion rates, *etc.* This simple typedef is insufficient, as the typedef mechanism maps to underlying types before checking, and can result in errors with confused types. Conversely, creating a full, new nominal type for each concept generates an unwieldy amount of boiler plate code to provide the needed operations on each numerical value. It is unclear if there is a compact *unit-of-measure* [10] algebra, as for physical quantities, that can model these types. In our experience, the ontology of numeric types present in financial software systems does not fit into a simple system that depends on a small number of base units. Instead, this may be an opportunity to introduce a novel, language-level typed numeric feature.

The example code in Figure 3 illustrates the utility of first class support for including specifications in the language and the need for simple ways to specify properties of interest. Many interesting properties, like the strict reduction in the `initialPosition` value, can be easily expressed in code directly as part of an assertion. For such properties, there is a need to provide language support to ease the insertion of conditions, like first class pre/post conditions, data invariants, *etc.*, and we also are working to provide a library of commonly used predicates for properties like primary key uniqueness, domain/range subset relations for maps, *etc.*. However, other properties are not so easily expressed in code, such as implicit global quantification like the multiplicity constraints in Legend. An open question here is "Do we need to introduce a single (or perhaps dialects of) specialized domain modeling languages for expressing assertions?".

Lifting Checkers to the Data Layer: The semantic information that is added to the MorphirIR code often contains information about data types (shapes) and invariants on them. These implicit data invariants present a rich source of information that can be used in *data quality* [23] assurance tasks. At the basic level, we can look at data flows and type information to extract core type and structure checks including numeric, string, enum values, and record or tuple structures. To the extent that ADT constructors contain validation rules (or invariants) and functions have pre/post conditions, we would also like to use a weakest-precondition style analysis to infer other checks.

For example, a `Trade` record might have `tradeDate` and `settlementDate` with a check in the constructor that `tradeDate < settlementDate`. We can use this check both for analysis that the code does not construct any invalid objects internally but if we push this condition to the interface with the data sources, say a SQL database, we can also generate and check this assertion on the appropriate tables. This ensures that any data flowing into the system, even if entered manually, will be validated.

Alternative and Specialized Checkers: The focus of our experience in the validation pipeline has been on checking language level assertions, like invalid casts or div-by-0, and user defined assert conditions. Many applications and DSLs have richer sets of conditions that are of interest. In some cases, these are additional checks that should be applied to all code in a certain domain and look like linter rules [9] and could be checked with the same underlying approaches as for other semantic errors. Other conditions may need to be addressed with specialized checking methodologies. One specific example that we explored was numerical stability checking [2,20], as we noticed that our application makes extensive use of float and decimal types. Interestingly, the outcome of this investigation for our target applications was that the combination of a true *Decimal* type combined with a business rule specified rounding and computation ordering resulted in numerical stability being a very low priority concern. As other users of the MorphirIR stack emerge, *e.g.* in the algorithmic trading space, this may become a property of substantial interest.

Outside of the need for checkers for specialized properties, we are also interested in supporting a range of checkers in the validation pipeline. The BSQChk checker we currently use is SMT based (using Z3 [5]) so, as our experience with inductive code illustrated, it is limited when dealing with certain scenarios and, at some point, we will experience scalability issues. Many of the features of the MorphirIR language that enable BSQChk to perform well should also boost the performance and effectiveness of other verification and error detection techniques. The elimination of mutation and aliasing alone eliminate two of the major causes of information loss and scalability problems for automated reasoning systems. Combined with the additional benefits of specialized code for common loop patterns [16,15,6], we expect the MorphirIR stack to be a place where formal methods are able to showcase [21] the value they can have in software development.

5.3 Injection of Compliance and Audit Logs

Centralizing the injection of cross-cutting auditing and observability logic at a single point in the stack has a major benefit in ensuring compliance requirements and business needs. An example is code that is part of a regulated system that takes in data from various upstream sources. The lineage of this data, including the origin, the decisions made using it, and the outcome are all subject to compliance checks and audits. This data is usually stored, and when needed, processed to produce flow and provenance graphs. Those same tools can be used at modeling time to provide quick interaction with domain experts to ensure that the model reflects their ideas. In a report, users might want to look up the associated definition for a field and what data sources are used in the calculation. As these tasks become more complex, tools can navigate the call path on the fly and display relevant information to help users understand how a particular value was calculated.

These problems have many interesting flavors from the topics of taint analysis [22], program question answering [12], and logging management [26]. The ability to prove that a given set of values recorded in the audit (or observability) pipeline are sufficient to answer specific questions or demonstrate the reasoning for a given decision will have massive value. This type of proof would ensure that the application satisfies the relevant regulations, which today is often done by verbose logging, and would position us to confidently optimize the logging and data retention code to remove redundant output.

Our experience with program and flow visualization to understand data lineage and computation flows indicates that it is very effective for smaller applications or small numbers of data sources. However, the output becomes noisy and too complex to be reasonably understood [8,12] as system size increases. Developing heuristic or analytic techniques that abstract, organize, and visualize the most relevant aspects of these flows and lineages are of great interest.

6 Conclusion

This paper outlines our thoughts on the development of and initial experiences with the Morphir stack. This open-source platform is a collaboration across the Fintech community, academic researchers, and partners in the technology space with the goal of building a standard platform for implementing, executing, and validating regulatory compliance code as well as financial business platform applications. In these domains, building high assurance code is a foundational requirement for the system and the Morphir stack is explicitly designed to support the use of formal methods. Our experiences with the system have validated these designs and are already showing the value of this collaborative and assurance focused approach to the wider Fintech community. These experiences have also highlighted areas where we believe the system can be further improved or where innovation in verification and error checking can happen. Our hope with this experience report paper is to start a wider collaboration that will fuel the development of a vibrant software ecosystem in the Fintech space as well as

create a unique opportunity to advance the state of the art in formal methods and their practical application.

Acknowledgments

We would like to thank Beeke-Marie Nelke, Pierre De Belen, and Jianglai (Teddy) Zhang at Goldman Sachs for their technical contributions and feedback on this work. Thanks to our reviewers and numerous colleagues for their constructive comments and insights.

References

1. AppInsights, 2021. `https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview`.
2. Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. POPL, 2013.
3. Bosque repository, 2021. `https://github.com/microsoft/BosqueLanguage`.
4. Dapr, 2021. `https://dapr.io/`.
5. Leonardo de Moura, Nikolaj Bjørner, and et. al. Z3 SMT Theorem Prover. `https://github.com/Z3Prover/z3`, 2021.
6. Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. POPL '11, 2011.
7. Roy Thomas Fielding and Richard N. Taylor. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, 2000.
8. K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE TSE*, 17, 1991.
9. David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39:92–106, 2004.
10. Lingxiao Jiang and Zhendong Su. Osprey: A practical type system for validating dimensional unit correctness of C programs. In *ICSE*, 2006.
11. C. Kirkegaard, A. Moller, and M.I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE TSE*, 30, 2004.
12. Amy J. Ko and Brad A. Myers. Designing the Whyline: A debugging interface for asking questions about program behavior. CHI, 2004.
13. Complex Institution Liquidity Monitoring Report, 2019. `https://www.federalreserve.gov/reportforms/forms/FR_2052a20190331_f.pdf`.
14. Legend repository, 2021. `https://github.com/finos/legend`.
15. Mark Marron and Deepak Kapur. Comprehensive reachability refutation and witnesses generation via language and tooling co-design. Technical Report MSR-TR-2021-17, 2021.
16. Mark Marron, Darko Stefanovic, Manuel Hermenegildo, and Deepak Kapur. Heap analysis in the presence of collection libraries. PASTE, 2007.
17. Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala: A programming language for the law. In *ICFP*, 2021.
18. Morphir repository, 2021. `https://github.com/finos/morphir`.
19. Peter W. O'Hearn. Incorrectness logic. In *POPL*, 2019.
20. Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. PLDI, 2015.
21. Grant Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. The Imandra automated reasoning system (system description). IJCAR, 2020.
22. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, 2003.
23. Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. Automating large-scale data quality verification. *Proc. VLDB Endow.*, 11, 2018.
24. smithy, 2021. `https://awslabs.github.io/smithy/`.
25. Solidity repository, 2021. `https://docs.soliditylang.org/`.
26. Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. ASPLOS, 2011.
27. Jakub Zakrzewski. Towards verification of ethereum smart contracts: a formalization of core of solidity. In *VSTTE*, 2018.