Implementing Rabin's Fingerprint Scheme for the Mojo Language

Pranav Pativada

Rabin's fingerprint scheme is an algorithm that is used to generate unique identifiers, otherwise known as fingerprints, for a variety of structures such as graphs, strings and trees. It does so by leveraging polynomials in $GF(2)$ and a series of functions to emit a close-to-unique byte stream. We implement this algorithm for the Mojo programming language, extending the semantic analysis stage to provide fingerprints for types in a given Mojo program. We follow closely a Modula-3 reference for our implementation.

1. INTRODUCTION

Fingerprinting is a cryptographic algorithm used to create unique identifiers, or fingerprints, for various data structures such as graphs, strings, and trees. Rabin's fingerprinting scheme is one such technique amongst the various fingerprinting algorithms that is used to do this. We use Rabin's fingerprint scheme due to its mathematical robustness and probabilistic guarantees against *collisions* - where a collision is when two fingerprints are equal when their underlying structures are different [1, 3]. We first go through some relevant definitions for understanding the algorithm.

1.1 Definitions and Properties

The core of the fingerprinting scheme resides in using polynomial arithmetic defined over the finite field $GF(2)$ [1, 3]. The goal of this algorithm is to essentially utilize a series of functions to produce nearly unique byte streams for inputs [1]. Given a bit string $A = (a_1, ..., a_n)$ with each $a_i \in \mathbf{Z}_2 \; \forall i \in [1, n]$, we define a polynomial $A(x)$ as

$$\begin{aligned} A(x) &= \sum_{i=1}^{n} a_{n-i+1} \cdot x^{i-1} \\ &= a_1 x^{n-1} + ... + a_n x^0 \end{aligned}$$

A polynomial $P(x)$ is *irreducible* if it cannot be factored into two polynomials. This is similar to the notion of a prime number - where the factors of the polynomial is simply itself and the multiplicative identity (usually 1). With this notion of irreducibility, a *fingerprint* $f(A)$ is defined as follows:

$$f(A) = A(x) \mod P(x)$$

where $P(x)$ is some irreducible polynomial and $A(x)$ is defined by the bitstring of A.

Given this, we can programmatically define fingerprinting. We define a *nest* as either a text string or an ordered pair of two nests [2]. The fingerprint of a nest t, denoted as $\mathrm{FP}(x)$, is computed using the following rules:

$$\mathrm{FP}(t) = \begin{cases} \mathrm{FromText}(t) & \text{if } t \text{ is a text} \\ \mathrm{Combine}(\mathrm{FP}(y), \mathrm{FP}(z)) & \text{if } t \text{ is a pair } (y, z). \end{cases}$$

This is analogous to creating a fingerprint based on the bitstring representation of a text t if t is a text, or combining two fingerprints $f(A)$ and $f(B)$ if t is a pair of nests. We reconcile these two definitions by seeing that in the implementation of FromText and Combine, we implicitly will use the fact that $f(A) = A(x) \mod P(x)$ to do the fingerprint computation.

A nest t is a subnest of another nest r if it is equivalent to r or is one of the components of r if r is in the form of an ordered pair. This allows us to define two key functions - *length* and *size*. The *length* of a nest is the sum of distinct texts, whereas the *size* is the number of distinct subnests. See that for a nest N defined as:

$$((''n1'', ''n2''), ''n1'')$$

the length is 2 as we have $|\{''n1'', ''n2''\}|$. However, the size is 4 as we have the following distinct subnests: $\{''n1'', ''n2'', (''n1'', ''n2''), ((''n1'', ''n2''), ''n1'')\}$.

We note that there is a strong probabilistic guarantee for this fingerprinting algorithm. Here, two nests t and r collide if $t \neq r$ but $\text{FP}(t) = \text{FP}(r)$ [2]. This is done by using a 128-bit magic number, which guards against collisions even if there exists an adversary that knows how the algorithm works [1, 2]. This probabilistic guarantee ensures that for any nest S, the probability of collision is at most

$$\frac{\text{length}(S) \times \text{size}(S)}{2^{62}}$$

See that for large lengths and sizes, we still find that the chance of collision is very small. This guarantee allows for the computation of an upper bound on the probability of collisions in various applications. Though, note that this guarantee only holds if the underlying nest is truly independent of the 128-bit number, as we can construct subnests that violate this principle if we knew what the number was [2]. Given the chance of guessing a 128-bit number is $\frac{1}{2^{128}}$, we can assume that the probabilistic guarantee holds given no information or dependence on the number. As such, this allows us to use this scheme for many applications and create fingerprints for many structures.

1.2 Applications of Fingerprinting

Fingerprinting applications include reducing data transfer in distributed systems by comparing fingerprints instead of entire files. This ensures type safety in distributed programming, where values are transmitted as bit sequences over a network, and so fingerprints can verify type consistency efficiently. We can also use it to detect common sub-expressions in computational graphs.

Furthermore, we can fingerprint data structures like directed acyclic graphs (DAG's), binary trees, or subsets by using defining rules for a textual representation that captures the underlying structure. For example, we can fingerprint a node of a DAG by associating each of it's children with a unique label and building up a textual representation recursively.

$$f(n) = f(l_1 : f(c_1)||...||l_n : f(c_n))$$

Here, l_i is a unique label associated with the ith child of a node n. We use $||$ as the delimiter to build the textual representation. While this is a simple example, more complex representations can be used that can better convey the underlying structure. We now proceed to detail our implementation of Rabin's fingerprint scheme for the Mojo language.

2. IMPLEMENTATION

We split our implementation in the following three parts.

- $GF(2)$ Polynomial Arithmetic
- Generic Fingerprinting
- Implementation of Fingerprinting Module

We detail our implementation throughout the above stages.

2.1 $GF(2)$ Polynomial Arithmetic

The `Polynomial` class contains the implementation of $GF(2)$ polynomials. Each polynomial in $GF(2)$ is represented by its coefficients. We use a `TreeSet` of `BigInteger` to represent this. The `TreeSet` is populated with the coefficients of the polynomial, and is ordered such that the highest degree is the first entry (using a reverse comparator). We use the `BigInteger` class as it allows us to to take advantage of the `BigInteger` utility methods such as `shiftLeft`, `shiftRight`, `pow` and more - which streamline the process of implementing polynomial arithmetic. The `Polynomial` class extends an `Arithmetic` interface which consists of bitwise and arithmetic operations on $GF(2)$. We implement these by utilising the aforementioned utility methods. We also choose this representation as we can forgo overflow errors as `BigInteger`s are unbounded.

Polynomials can also be created using methods from the `Utils` static utility class. This allows for translation from the Modula-3 reference, which represents its polynomials as two 32-bit integers. The `fromInts` method encapsulates this and allows us to create polynomials via the same representation. This eases the process of translating the Modula-3 code. In addition to this, we provide `fromLong` and `fromBytes` - where we can create a polynomial from a long and a `ByteBuffer` of size 8. Utility methods `toInts`, `toLong`, and `toBytes` are also provided to translate the polynomial into the relevant datatypes/structures. With this, we provide the following polynomial constants that are matched with the Modula-3 reference.

- `ZERO`: The zero polynomial
- `ONE`: A polynomial of highest degree 63
- `X`: The X polynomial from `Poly.i3`
- `P`: The irreducible polynomial P used for polynomial modulo from `PolyBasis.i3`

2.2 Generic Fingerprinting

We implement a generic fingerprinting algorithm in the `Fingerprint` class. A fingerprint consists of a `ByteBuffer` that is allocated with size 8 and cleared upon use. We provide the following methods for fingerprinting taken faithfully from the Modula-3 reference.

- `fromText`: Creates a fingerprint from a textual representation.
- `fromChars`: Creates a fingerprint from a character buffer. We note that this can be used with a `ByteBuffer` too, since it can be converted to a character buffer.
- `combine`: Combines a series of fingerprints by calling `computeMod` on a `ByteBuffer` with initial polynomial `ONE`. We permute the resulting polynomial with the magic number and the permutation array defined in the `Constants` class.

We utilise the `computeMod` method from the `Polynomial` class, which given an initial polynomial p and a series of bytes b, outputs a polynomial k as the combined result of p and b. This is used in all of the fingerprinting methods mentioned above. We note that the inclusion of the magic number and permuting the bytes of the polynomial help keep the probabilistic guarantee of the fingerprint scheme. As such, this can be used as a generic fingerprinting algorithm for any text or sequence of characters. We provide an example usage where we fingerprint the `Queens.mj` and `QueensStatic.mj` programs as texts as follows.

Fingerprinting Mojo programs as texts.

```
// Instantiate mojo programs
File t1 = new File(T1);
File t2 = new File(T2);

// Get fingerprints
Fingerprint fp1 = Fingerprint.fromText(readText(t1));
Fingerprint fp2 = Fingerprint.fromText(readText(t2));
Fingerprint combined = Fingerprint.combine(fp1, fp2);

print(fp1, fp2, combined);
```

2.3 Implementation of Fingerprinting Module

The `FPModule` class implements the majority of the Mojo fingerprinting. It extends `Semant` to provide fingerprinting of a mojo program after semantic analysis. We first make changes to `Absyn.java`, where we augment each `Type` with the following fields:

- `fpId`: A unique fingerprinting id for a particular type. This is equivalent to converting a `Fingerprint` to a sign extended 32-bit integer via the `toInt` function in `MojoFP`.
- `fp`: The `Fingerprint` for a particular type.
- `sccId`: Used to track the strongly connected components for a particular type.
- `repId`: Used to track a unique representation of a particular type.

This allows us to traverse the type graph and mutate these fields with the right values to ensure consistent fingerprinting. Alongside this, we also create the `MojoFP` static utility class which wraps the methods in Section 2.2 as per `M3FP.m3`.

The `FPModule` class provides functionality for traversing the type graph and building up a textual representation for a particular type. We closely follow `TypeFP.m3` from the Modula-3 reference. Given a particular type, we can construct the textual representation by finding the strongly connected component of that type in the type graph. We outline the following methods which do this.

- `fromType`: Computes the fingerprint by finding the SCC for a given type.
- `visitSCC`: Finds the SCC for a given type and builds the textual representation. It does so by performing DFS and keeping track of the nodes and the textual representations for each component.

- `finishSCC`: Finalizes the fingerprint computation for a given SCC and resets the SCC. It combines the fingerprints of all nodes in the SCC and updates their unique identifiers.
- `getRep`: Retrieves the representative node for a given type, ensuring that each unique type has a single, consistent representation.

Note that utility methods `compareNode` and `compareInfo` are also used to recursively update and compare nodes and their information. The actual specifics of textual representation are implemented in the `FPrinter` for the `Type`'s. For `Value`'s, the textual representation is via the `addFPTag` method. We detail the textual representations as follows, starting with the different `Type`'s. In places where Value's are added (such as `Proc`, `Object` and `Record`), we refer to the `Value` textual representation in Table 1.

- `Int`: `$int`.
- `Proc`: Starting with `PROC`, add the `Formal` values. Add ` => ?` if the result exists. Then if there exists children add them as well.
- `Enum`: Case bash to `$boolean` or `$char` if equal to `Bool` or `Char`. Otherwise, add all the names in the current scope to `ENUM` separated by a space.
- `Object`: Starting with `OBJECT`, add all the fields first. Do the same with methods prepended by a `METHOD` tag. Then traverse the parent recursively.
- `Ref`: Case bash to `$refany`, `$address` or `$null` if `Refany`, `Addr` or `Null`. Otherwise append `REF` and traverse the target of the ref type.
- `Record`: Starting with `RECORD`, add all the fields.
- `Array`: Prepend `OPENARRAY` if open or `ARRAY` if fixed size. Then traverse through the array.
- `Err`: `$ErrType`.
- `Named`: Unsupported as named types should be stripped after semantic analysis.
- `Subrange`: Prepend with `SUBRANGE` *min max* where *min* is the min subrange value and *max* is the max subrange value. Then traverse the base type of the subrange.

A `Value`'s textual representation is similar, with the `addFPTag` and `FPStart` methods, in which a `Value` is encoded as follows. Note that *name* refers to the name of the value. Additionally, if the value has a certain offset *o*, we add `@`*o*. If the value is external, a `$` followed by the external name. These are appended before the final closing `<` tag for all of the below. We write this as [@o][$extName] where the brackets denote an existence condition (offset is not 0, and the value is external). Thus, each `Value` can take on the following core textual representation *t*.

$$t = name[@o][\$\text{extName}]$$

As such, we use this for each of the different `Value`'s in our language to yield the following textual representation.

Value	Textual Representation
`EnumElt`	`<ENUM-ELT:`t`>`
`Tipe`	`<TYPE:`t`>`
`Variable`	`<VAR:`t`>`
`Formal`	<*mode*:t> where *mode* is either `VALUE, VAR, READONLY`
`Method`	`<OVERRIDE:`t`>` if the method is an override or `<METHOD:`t`>`
`Const`	`<CONST:`$t = e$`>` where e is the constant value of the `Expr`
`Field`	`<FIELD:`t`>`
`Procedure`	`<PROCEDURE:`t`>`
`Unit`	`<UNIT:`t`>`

Table 1: Textual representation of `Value` in the Mojo language.

Note that for the above, the offsets are all 0. Also note that many `Type`'s have `Value`'s within them, such as `Proc` and `Record` and so these also appear in the textual representation of a particular `Type`. The combination of these three stages allows us to instantiate the `FPModule` for fingerprinting. An example usage of the program is as follows, where we perform semantic analysis first followed by fingerprinting.

Usage of fingerprinting module

```
FPModule module = new FPModule(wordSize);

// Semantic Analysis
module.Check(unit);
module.print(Scope.Top());
if (Error.nErrors() > 0) return;

// Fingerprinting
module.fingerprint(unit);
module.print();
```

3. DISCUSSION

3.1 Example Fingerprints

We showcase example fingerprints from our implementation. For the generic case as mentioned in Section 2.2, fingerprinting `Queens.mj` and `QueensStatic.mj` as texts and combining their fingerprints yields the following. This is provided in the `Main` class of the `Fingerprint` package.

Result of fingerprinting Queens programs

```
// Queens Fingerprint
0000001110001001011100111000111010100001111101000111100110100000
// QueensStatic Fingerprint
0000101100011111011111000111011101010111010110000110011101100011
// Combining Both Fingerprints
0111011101110000111000111110100001010010100010101101110000011100
```

To see that our implementation is concrete, we reconcile with both `fromText` and `fromChar` methods and confirm that the fingerprints are equivalent for both (and that the combination of both is also the same). We also fingerprint according to the actual textual representation as seen in Section 2.3. We do this for `RecordRecord.mj`, `Enum.mj` and `Override3-ok.mj` and find that we get the following fingerprints for the `Value`'s as according to the full textual representation.

Value Fingerprints as per actual textual representation

```
// RecordRecord Fingerprint (Record Type)
0000001111111011001010000010010100010010110111101110101010011110
// Enum Fingerprint (Enum Type)
0000100001011110010011001010101011100110100101101101011111001001
// Override Fingerprint (Object Type)
0000111111111010011010101000010100010110100010010100111110110111
```

The full textual representations of each of the values and the sub-components can be found by running `mojo.Main`.

3.2 Key Differences with Modula-3 Reference

We note key differences between the reference implementation for future work. Primarily, there are two parts where there are differences - polynomial arithmetic and the textual representation. For starters, we forgo the use of generating polynomial and power tables as defined in `PolyBasis.m3`. This is because these tables are only used in the `Power` and bit extension functions in `Poly.m3`. Given that the `Power` function effectively computes x^d `MOD P` - we can use our `TreeSet` representation to do this reasonably efficiently.

The other key difference is in the textual representation of the types. Though inspired by the Modula-3 reference, the Mojo language doesn't have a brand or a notion of tracing - and so we forgo using these in the textual representation. Examples of these in the Modula-3 reference are as follows:

Brand and Tracing in Modula-3 reference

```
 IF (NOT p.isTraced) THEN M3Buf.PutText (x.buf, "-UNTRACED") END;
Brand.GenFPrint (p.brand, x);
```

Additionally, global values aren't dealt with in our implementation as this requires checking if import and export flags and also requires messing with the scope. Specifically, we note that the implementation adds the following to the textual representation given a global `Value`

Global value implementation in Modula-3 reference

```
 IF (global) THEN
      s.top := 0;
      Scope.NameToPrefix (t, s, dots := TRUE);
      Scope.PutStack (x.buf, s);
```

where we find `NameToPrefix` and `PutStack` in `Scope.m3`.

The reference implementation also has default expressions (`Expr`) for `Formals` and `Methods`. It encodes these default expressions as well. We note that our implementation does not encode expressions, though this would be a future work. However, when we encounter a `Value.Const`, we add to the textual representation when the `Expr` is either an `Int`, `Bool.True` or `Bool.False`. This is because we know it's a constant value. Furthermore, in `addFPTag` and `addFPEdges`, we note that there is a `base` method that should be invoked which gets the base type of a given `Value`. This is so the the type can be added to the `Info` struct to traverse the children later.

Another difference is that we use modern data structures such as `HashMap` and an `ArrayList`, and so we don't need to maintain our own hash table or dynamic array. This makes the code more streamlined and as a result methods such as `ExpandHash` and `ExpandReps` in the Modula-3 reference are not implemented.

4. CONCLUSION

We implement a close version of Rabin's fingerprint scheme with inspiration taken from the Modula-3 reference. This was done by first implementing polynomials in $GF(2)$, a generic fingerprinting class, and integrating into the primary Mojo codebase by extending `Semant` to include fingerprinting by traversing the SCC's and building a textual representation. We provide a unique textual representation for the different `Type`'s and `Value`'s. Key changes in comparison to the reference implementation include polynomial arithmetic and the textual representation, which stem from design choices and language differences. Future work would involve also fingerprinting `Expr`, writing concrete tests, and reconciling with the Modula-3 reference.

References

[1] Andrei Z. Broder. *Some applications of Rabin's fingerprinting method.* Ed. by Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro. New York, NY, 1993.

[2] *Modula-3 Documentation.* URL: `http://modula3.github.io/cm3/`.

[3] M.O. Rabin. *Fingerprinting by Random Polynomials.* Center for Research in Computing Technology: Center for Research in Computing Technology. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981. URL: `https://books.google.com.au/books?id=Emu_tgAACAAJ`.