Code extraction from a dependently typed language

Under which conditions is Java a suitable target language for code extraction from Agda?

Lukas Zimmerhackl
L.K.Zimmerhackl@student.tudelft.nl
Supervised by:
J.G.H. Cockx and L.F.B. Escot

1. Agda

- Proof assistant and functional programming language
- Dependently typed
- Total language

2. Java

- Programming language
- Object-oriented
- Build-in garbage collector
- Large ecosystem
- Quite efficient

3. Motivation

- Allow more languages to use Agda's features
- Use Java's efficiency
- Target the vast Java ecosystem
- Target a type of language Agda currently does not compile to, object-oriented languages

4. Implementation

```
data Nat : Set where
  zero : Nat
  suc : Nat -> Nat

consume : Nat → Nat
consume zero = zero
consume (suc n) = consume n
```

Fig. 1: Nat Datatype and consume function in Agda

```
consume = (AgdaLambda) k -> {
    return ((AgdaData) k).match(new NatVisitor()
    {
        public Agda zero ()
        {
            return k;
        }
        public Agda suc (Agda l)
        {
            return runFunction(l, ((AgdaLambda) consume));
        }
    });
};
```

Fig. 3: Consume function in Java

```
interface NatVisitor extends Visitor {
    Agda suc (Agda arg1);
    Agda zero ();
}
abstract static class Nat implements AgdaData
{
}
static class zero extends Nat
{
    public zero ()
    {
    }
    public Agda match (Visitor visitor)
    {
        return ((NatVisitor) visitor).zero();
    }
}
static class suc extends Nat
{
    private final Agda arg1;
    public suc (Agda arg1)
    {
        this.arg1 = arg1;
    }
    public Agda match (Visitor visitor)
    {
        return ((NatVisitor) visitor).suc(arg1);
    }
}
```

Fig. 2: Nat Datatype in Java

5. Results

- Haskell outperforms Java
- Java throws a stack overflow for large inputs
- Java has same performance as interpreted Scheme

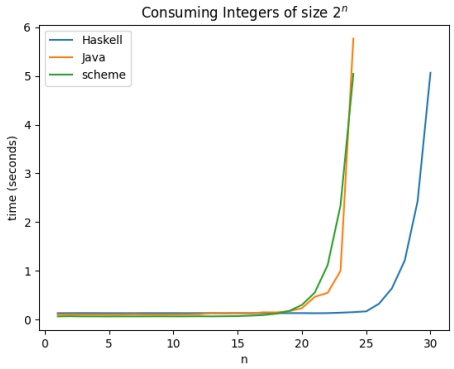

Fig. 4: runtimes for consume function

6. Limitations

- Java has no tail-call optimization
- High memory usage
- No laziness implemented
- Pattern matching is buggy for current implementation
- No natural number optimization
- Java's type system is counterproductive

7. Conclusion

- Java is not recommended as a target language
- Java is constantly evolving and there might be better support in the future
- If JVM is desirable, target Clojure or Scala

8. Future work

- Laziness
- Native natural number support
- Bug fixes
- Variable name sanitization

Code extraction from a dependently typed language

Under which conditions is Java a suitable target language for code extraction from Agda?

Lukas Zimmerhackl
L.K.Zimmerhackl@student.tudelft.nl
Supervised by:
J.G.H. Cockx and L.F.B. Escot

1. Agda

- Proof assistant and functional programming language
- Dependently typed
- Total language

2. Java

- Programming language
- Object-oriented
- Build-in garbage collector
- Large ecosystem
- Quite efficient

3. Motivation

- Allow more languages to use Agda's features
- Use Java's efficiency
- Target the vast Java ecosystem
- Target a type of language Agda currently does not compile to, object-oriented languages

4. Implementation

```
data Nat : Set where
  zero : Nat
  suc : Nat -> Nat

consume : Nat → Nat
consume zero = zero
consume (suc n) = consume n
```

Fig. 1: Nat Datatype and consume function in Agda

```
consume = (AgdaLambda) k -> {
    return ((AgdaData) k).match(new NatVisitor()
    {
        public Agda zero ()
        {
            return k;
        }
        public Agda suc (Agda l)
        {
            return runFunction(l, ((AgdaLambda) consume));
        }
    });
};
```

Fig. 3: Consume function in Java

```
interface NatVisitor extends Visitor {
    Agda suc (Agda arg1);
    Agda zero ();
}
abstract static class Nat implements AgdaData
{
}
static class zero extends Nat
{
    public zero ()
    {
    }
    public Agda match (Visitor visitor)
    {
        return ((NatVisitor) visitor).zero();
    }
}
static class suc extends Nat
{
    private final Agda arg1;
    public suc (Agda arg1)
    {
        this.arg1 = arg1;
    }
    public Agda match (Visitor visitor)
    {
        return ((NatVisitor) visitor).suc(arg1);
    }
}
```

Fig. 2: Nat Datatype in Java

5. Results

- Haskell outperforms Java
- Java throws a stack overflow for large inputs
- Java has same performance as interpreted Scheme

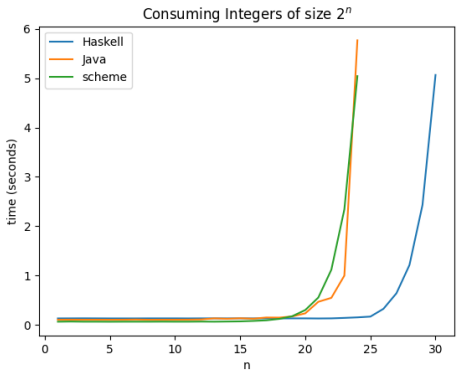

Fig. 4: runtimes for consume function

6. Limitations

- Java has no tail-call optimization
- High memory usage
- No laziness implemented
- Pattern matching is buggy for current implementation
- No natural number optimization
- Java's type system is counterproductive

7. Conclusion

- Java is not recommended as a target language
- Java is constantly evolving and there might be better support in the future
- If JVM is desirable, target Clojure or Scala

8. Future work

- Laziness
- Native natural number support
- Bug fixes
- Variable name sanitization