SEAL: Some Extensions for the Amy Language

Computer Language Processing and Compiler Design 2021

redacted

HKUST

redacted

1. Introduction

SEAL extends the Amy language by adding support for tuples and higher order functions.

These extensions build upon a compiler of a previously implemented (vanilla) Amy language. The compiler is broken down into multiple stages: lexing, parsing, name analysis, type checking, and finally interpretation or WebAssembly code generation. The *SEAL* extensions is focused on interpretation.[1]

The Amy language lacks the expressiveness found in most modern programming languages such as Python, C++, and Scala. These languages generally have tuples and higher order functions, allowing for product types to be namelessly typed and constructed, and for functions to be treated as first-class citizens and used as expressions, allowing us to program functionally with the typical `map` and function composition operations.

2. Examples and Semantics

2.1 Tuples

```
object Tuples {
  def foo(v: (Int, Boolean)): (Int, Boolean) = { // Type.
      v match {
          case (i, false) ⇒ // Pattern matching.
              (i, false) // Literal.
          case (i, true) ⇒
              (−i, false)
      }
  }

  val a: (Int, Boolean) = foo((1, true));
  val b: (Int, Boolean) = foo((−2, false));
  val a0: Int = a(0); // Variable access.

  Std.printString("a: ("
    ++ Std.intToString(a0) ++ ", "
    ++ Std.booleanToString(a(1))
    ++ ")"); // a: (−1, false)

  // Error: only int literals allowed as tuple accessors.
  // val a1: Int = a(0 + 1);

  // Error: can't access index out of range.
  // val a1: Int = a(2);

  val c: String = ("hello", "there")(0); // Literal access.
  Std.printString(c)
}
```

See `extension-examples/Tuples.scala` for a fuller demonstration.

There are three primary tuple operations:

1. Construction: `(x, y)`
2. Access: `t(i)` (for integer literal `i`)
3. Pattern matching:
   `t match { case (x, y) ⇒ ... }`

Tuples were introduced both on the value level and on the type level, and have the usual one-to-one correspondence between a field's type and value. For example, `("a", 1)` has type `(String, Int)`.

Further semantic rules are as follows:

- Tuples may hold an unlimited number of fields (within hardware limits).
- Tuple access is similar to calls, but there must only be one argument and it must be an integer literal. This is checked statically.
- Tuples in *SEAL* are heterogenous, as is usual in most programming languages.

[1] The author has decided to throw code generation and optimisations for extensions out the window mainly because his code generation for the vanilla Amy language wasn't complete (darn you match expressions!).

2.2 Higher Order Functions

```
object HigherOrderFunctions {
  def map(f: Int ⇒ Int, xs: L.List): L.List = {
      xs match {
          case L.Nil() ⇒ L.Nil()
          case L.Cons(h, t) ⇒ L.Cons(f(h), map(f, t))
      }
  }

  def add(a: Int): Int ⇒ Int = {
      \(b: Int) → a + b
  }

  val i: Int = 2;
  val xs0: L.List =
    L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
  val xs1: L.List = map(\(x: Int) → x + 1, xs0);
  val xs2: L.List = map(add(10), xs0);
  Std.printString(L.toString(xs1)); // List(2, 3, 4).
  Std.printString(L.toString(xs2)); // List(11, 12, 13).

  // \(x: Int) → val y: Int = 1; x + y; // Error.
  \(x: Int) → (val y: Int = 1; x + y); // OK.

  (\(x: Int): Int → x + 1)(1); // Return type optional.

  val f: Int ⇒ Int ⇒ Int = add;
  Std.printInt(add(40)(2)); // Currying.
}
```

See `extension-examples/HigherOrderFunctions.scala` for a fuller demonstration.

Functions are now supreme beings and have earned their rightful place as first-class citizens. The following features are included in the extension:

- Function types: `A ⇒ B`
- Lambdas (anonymous functions): `\(x: Int) → x + 1`
- Currying: `f(x)(y)`

The syntax and semantics are as follows:

(a) Function types

(i) Multiple parameters are specified as tuples, e.g. `(Int, Boolean, String) ⇒ Unit` represents a function of three arguments mapping to `Unit`.

(ii) The type `()` is a synonym for `Unit`, e.g. `() ⇒ ()` and `Unit ⇒ Unit` are the same type.

(iii) Similar to Scala, `((A, B)) ⇒ C` represents a function of one argument (here, a 2-tuple) mapping to `C`. This is to avoid confusion with `(A, B) ⇒ C`, which represents a function of *two* arguments of type `A`, `B`, mapping to `C`.

Note that the extra parentheses here is for the special case of functions with one tuple argument. For any other type of argument, the parentheses are superfluous (e.g. `(Int) ⇒ String` is the same as `Int ⇒ String`).

(iv) Unlike Scala, it is invalid to cast a function from `(A1, ..., An) ⇒ B` to `((A1, ..., An)) ⇒ B` (i.e. from a function of n arguments to a function of one n-tuple argument).

(v) Also unlike Scala, it is invalid to call `f(a1, ..., an)` where `f` has type `((A1, ..., An)) ⇒ B` and `ai` has type `Ai`. In other words, parameters won't be auto-squished into a tuple.

(b) Lambdas

(i) Any free variables in a lambda (i.e. variables that reference names outside the lambda) are captured when the lambda is instantiated. Due to the immutable nature of Amy/*SEAL*, variables are captured by value, so that there is no cause for alarm about dangling references.

(ii) The return type of lambdas can be optionally specified: `\(x: Int): Int → x + 1`. If no return type is specified for a lambda, it will be automatically deduced.

(iii) The body of a lambda is not allowed to begin with a let-expression. Should there be a need for declaring local variables within a lambda, use a subexpression.

(c) Calls and currying

(i) Any number of calls may follow a valid identifier or parenthesised value.

(ii) Calls in *SEAL* and Amy have the same precedence.

3. Implementation

How does all this work? It all begins with an idea...

3.1 Theoretical Background

Since the author has decided to throw code generation and optimisation out the window, there are few theoretical concepts involved.

However, one noteworthy concept is of closure conversions. The primary challenge with implementing lambdas is deciding how to deal with free variables (i.e. variables in the lambda expression that are scoped outside).

3.1.1 Closure Conversion

Take this simple program:

```
object Foo {
  def foo(): Int ⇒ Int = {
    val i: Int = 1;
    val res: Int ⇒ Int = \(x: Int) → x + i;
      // Problem! How do we access i from within res?
    res
  }
  foo()(42) // Should return 43.
}
```

The idea of closure conversion is to add a parameter to lambda containing the environment at the point the lambda is constructed [Might].

```
def foo(): ((Env, Int) ⇒ Int, Env) = {
  val i: Int = 1;
  val res: ((Env, Int) ⇒ Int, Env)
    = (\(env: Env, x: Int) → x + env(i),
       Env(i → 1));
  res
}
```

Imagine `Env` as a map from an identifier to a value and that `env(i)` accesses the value of identifier `i`.

`res` is now a closure: a pair containing the modified lambda procedure (`(Env, Int) ⇒ Int`) along with the captured environment (`Env`).

```
def foo(): ((Env, Int) ⇒ Int, Env) = {
  val i: Int = 1;
  val res: ((Env, Int) ⇒ Int, Env)
    = (\(env: Env, x: Int) → x + env(i),
       Env(i → 1)); // Create closure.
  res
}
```

Now we can lift the lambda up to a function (not done in *SEAL*).

```
def lambda_0(env: Env, x: Int): Int = {
  x + env(i)
}

def foo(): ((Env, Int) ⇒ Int, Env) = {
  val i: Int = 1;
  val res: ((Env, Int) ⇒ Int, Env)
    = (lambda_0, Env(i → 1));
  res
}
```

Finally, when calling the lambda, we expand the call to apply the environment as well. `foo()(42)` gets translated to:

```
foo() match {
  case (proc, env) ⇒
    proc(env, 42)
}
```

3.2 Implementation Details

SEAL is the product of multiple refactors, rollbacks, and intensive hours of toiling.[2] Below we attempt to explain a few major head-scratching dilemmas faced and why the end result was chosen. Details and sections are not necessarily presented in the order of implementation.

3.2.1 Tuples: Construction and Pattern Matching

The implementation of tuples is relatively straightforward. We add a new expression, type, and match pattern to our tree:

```
trait TreeModule { self ⇒
  sealed trait Expr extends Tree
  // ...
  case class Tuple(values: List[Expr]) // New!
    extends Expr

enum Type:
  case IntType
  // ...
  case TupleType(types: List[TypeTree]) // New!

enum Pattern extends Tree:
  case WildcardPattern()
  // ...
  case TuplePattern(args: List[Pattern]) // New!
```

Parsing and name analysis are relatively straight forward. Type checking is slightly more involved.

When generating constraints for type checking, since the expected type might not be a tuple, we create type variables for each field. However, `TupleType` doesn't accept type variables. We'll need a proxy to represent a type tree that can contain type variables.

[2] By the way, make sure to check out `extension-examples/SealSim.scala`. You won't be disappointed.

```
type TypeOrVar = ConcreteType // Updated!
               | TypeVariable


// New:
sealed trait ConcreteType
case object IntCType extends ConcreteType
// ...
case class ClassCType(qname: QualifiedName)
  extends ConcreteType
case class TupleCType(xs: List[TypeOrVar]) // !
  extends ConcreteType
```

After refactoring the old code to use the new proxy type, all that's left is to substitute type variables recursively in the substitution step of our unification algorithm.

In the interpreter, we natively support tuples by introducing `TupleValue` and handle them accordingly.

```
case class TupleValue(xs: List[Value]) extends Value
```

3.2.2 Higher Order Functions: Typing

For a first attempt at introducing function types, we take a hint from the lecture notes on Type Inference[3] and represent functions as `A ⇒ B`.

```
enum Type:
  // ...
  case FunctionType(args: TypeTree, // New!
                    ret: TypeTree)
```

Here functions of more than one parameter have a tuple type `A`. For example, `(Int, String) ⇒ Unit` is represented as

```
FunctionType(
  TypeTree(
    TupleType(
      List(TypeTree(IntType),
           TypeTree(StringType))
    )),
  TypeTree(UnitType))
```

However, it becomes convoluted when dealing with functions of one tuple argument. For `Function(Tuple(A, B), C)`, we need to distinguish whether this is a function of one tuple argument of type `(A, B)`, or a function of two arguments of type `A`, `B`.

To ease the implementation burden, we rewrite `FunctionType` so that it inherently supports multiple arguments and can unambiguously distinguish between one argument and multiple arguments.

```
case FunctionType(args: List[TypeTree], ret: TypeTree)
```

3.2.3 Functional Refactor for `Variable` and `Call` Expressions

An early consideration when implementing higher order functions is the ability to write expressive code such as:

```
def foo(x: Int): Int = { x + 1 }
val addOne: Int ⇒ Int
  = foo; // Functions as first-class citizens!
val add: Int ⇒ Int ⇒ Int
  = (\(x: Int) → \(y: Int) → x + y);
addOne(41);
add(20)(22); // Multiple calls!
(1, "a")(1); // Tuple access! (Parsed as a call.)
```

We would like to treat `foo` as a first-class citizen and be able to use the identifier in an expression just like any other variable or literal.

We would also like to perform calls on values such as lambdas, tuples (in case of tuple access), or even arbitrary expressions: `(if (b) { add } else { times })(x, y)`.

So instead of treating variables and calls separately, we treat calls as an operation that can be called on any expression. This means that the responsibility of looking up functions and constructors falls on `Variable`.[4]

```
case class Variable(name: QualifiedName) extends Expr
case class Call(expr: Expr, args: List[Expr]) extends Expr
```

This also allows us to chain multiple calls together for currying.

In parsing, we modify the grammar so that we accept `many(call)` instead of just one call. We also need to take care to disallow calling the unit literal: `()()`.

```
Expr4 := Error
       | IdAndCalls
       | BrkExprAndCalls
       | OtherLiteral

IdAndCalls := QualId Call*
QualId := Id ['.' Id]?
BrkExprAndCalls := '(' [ Unit | TupleOrSubExpr ]
Unit := ')' // No calls on unit literal!
TupleOrSubExpr := Expr [',' Expr]* ')' Call*
```

[3] Slides 15 and 16 out of 38.

[4] At this point, "variable" is probably a misnomer since it encapsulates so much more.

```
Call := '(' Expr [',' Expr]* ')'
```

In type checking, we add a new `FunctionCType` which inherits from `ConcreteType`. Each time we encounter a `Call`, we generate a `FunctionCType` constraint indicating that the called expression is expected to have a function type. The arguments of the call are type variables (to be constrained with the actual parameters). The return type of the call is the expected type.

```
case Call(expr, args) ⇒
  val typeVars = args map (_ ⇒ TypeVariable.fresh())
  val funcType = FunctionCType(typeVars, expected)
  ((args zip typeVars) flatMap genConstraints)
    ++ genConstraints(expr, funcType)
```

On the interpretive side, we introduce `FunctionValues` to encapsulate named or nameless (c.f. §3.2.5) values.

```
sealed trait FunctionValue extends Value
case class BuiltInFunctionValue(f: List[Value] ⇒ Value)
  extends FunctionValue
case class FunctionPtrValue(f: FunDef)
  extends FunctionValue
case class ConstructorPtrValue(qname: Identifier)
  extends FunctionValue
```

We distinguish between different functions (built-ins, references, and constructors) on the type level to handle the different representations.

3.2.4 Tuples: Access

Here, the objective is to allow tuple access similar to Scala. We accomplish two things: bounds checking of the index and type checking of a particular field.

To perform bounds checking, we make a language design choice to only accept integer literals. We modify `IntCType` so that we can store the value of the literal.[5]

```
sealed trait IntCType extends ConcreteType
case object IntAnyValueType extends IntCType
case class IntConstantType(n: Int) extends IntCType
```

We also need to modify the constraint solver. As mentioned in §3.2.3, all calls expect the called expression to have function type. We need to handle the special case where the called expression is a tuple instead of a function.

[5] The idea is to have something akin to C++'s `std::get<N>` for tuple access, where `N` is a constant expression `int` (i.e. can be compile-time evaluated) so that bounds-checking could be done in compile time. For *SEAL* however, we won't evaluate expressions during compilation and simply force usage of integer literals.

```
def solveConstraints(constraints: List[Constraint]) = {
  constraints match {
    case Nil ⇒ ()
    case Constraint(TupleCType(ts), // New case!
                    FunctionCType(args, expected))
                    ::more ⇒
      if (args.length != 1)
        error("can't call tuple with 0 or 2+ args")
      else
        args.head match
          case IntConstantType(i) ⇒
            if (i ≥ ts.length) {
              error("out of bounds")
            } else {
          // Constrain i-th field with expected type.
              solveConstraints(
                Constraint(ts(i), expected)::more)
            }
          case _ ⇒
            error("non-constant integer access")
```

Finally, in the interpreter, we handle `TupleValues` in `Call` expressions.

3.2.5 Lambdas

We first introduce a new expression:

```
case class Lambda(params: List[ParamDef],
                  retType: Option[TypeTree],
                  body: Expr) extends Expr
```

This is similar to a `FunDef`, but without a `name` and with the return type optional.

In parsing, we parse lambdas on the same precedence level as `if` and `match` expressions. This is because we want to avoid lambda bodies beginning immediately with let-expressions (`val`) but still be able to begin with `if` and `match` expressions.[6]

In name analysis, we introduce the lambda's formal parameters as new parameters and remove locals with clashing names. This shadows outer parameters/locals and gives the formal parameter preference during name lookup.

No desugaring or lambda lifting is done. Mainly because the interpreter implementation is fairly straightforward once we use closure conversion, and *SEAL*

[6] Another note about parsing: to avoid the LL(1) nuisance with parsing the opening parenthesis and ⇒, the author opted to use backslash and → as delimiters, reminiscent of Haskell.

doesn't have a native, generic `Map` type to lookup free variables.[7]

To implement lambdas in the interpreter, we use closure conversion. As described in §3.1.1, we start by creating a new `FunctionValue` type which encapsulates the lambda and environment. Since this is interpreted, we diverge a bit from the theory: no `env` parameter is added to the lambda procedure itself.[8]

```
// New type!
case class ClosureValue(proc: Lambda,
                        env: Map[Identifier, Value])
                        extends FunctionValue
// ...
def interpret(expr: Expr)
    (implicit locals: Map[Identifier, Value]): Value = {
  expr match {
    // ...
    case proc@Lambda(_, _, _) ⇒ // New!
      ClosureValue(proc, locals)
    case Call(e, args) ⇒
      interpret(e) match
        case TupleValue(xs) ⇒ // ...
        case BuiltInFunctionValue(func) ⇒ // ...
        // ...
        case ClosureValue(proc, env) ⇒ // New!
          interpret(proc.body)(env
            ++ (proc.params zip (args map interpret))
              .map((p, v) ⇒ p.name → v))
```

And we're done!

4. Possible Extensions

Amidst the 250-plus test cases, the implementation of the SEAL interpreter provided is very much likely riddled with horrendous bugs and diseases. If the reader comes across an unintelligible error message, do not be astonished! The error messages and source positioning may be wonky.

Had the author more time, he would extend SEAL with the following extensions:

- Implicit conversions (or type-casting): because nobody likes to write `Std.intToString` everytime you want to want to concatenate something to print to output.
- Import machinery (or cross-module name analysis): because nobody likes to run their code, get a "Oh `L.List` doesn't exist", and modify the command arguments to compile `List.scala` *every single time*!
- Generics/templates: because nobody likes to implement ten different `Lists`, `map`, and `compose` methods with the exact same function bodies.

The resulting extensions of *SEAL* is foretold to lead to the creation of *MEAL*, More Extensions for the Amy Language, or *MEASLes*, More Extensions for the Amy and *SEAL* Languages.

With desirable extensions out of the way, it is time to address the elephant in the room. *SEAL* can very much improve with code generation[9]and optimisation. The interpreter, though easy to work with, provides a limited environment for memory management. One of the examples, `extension-examples/AOC2021D01.scala` contains a function (`part2`) that crashes the program due to a stack overflow on a large input set. With code gen and optimisation this problem could hopefully be solved.

One optimisation that may help here is tail call optimisation, where the result value is passed along as a function parameter and returned in the base case. Tail call optimisation helps reduce stack usage by reusing the current stack frame for storing return value [Cronin 2008].

Another difficulty encountered was desugaring and substituting expressions (e.g. for implicit conversions). The current structure and flow of the program does not make this trivial. Desugaring appears to be a different stage in the pipeline all together, following name analysis but before type checking [Rathi 2020].[10]

So far, we've managed to avoid desugaring of lambdas by directly interpreting its expression. However, it may be useful to be able lift the lambda as a function in

[7] Technically, it should be possible to instead use a tuple to represent the environment, and assign each free variable an index in the tuple, but this is just much more tedious than the direct interpreter implementation. And remember, codegen and optimisations were thrown out the window.

[8] Also note that the entire environment is copied, not just the free variables. For the interpreter, this is fine since we're reusing an otherwise immutable map. However, this may be a slight inefficiency for codegen and should be optimised to use a minimal mapping of free variables.

[9] To be frank, the author has not given much thought about codegen.

[10] For desugaring lambdas, it makes sense for name analysis to come first, since lambdas need to deal with closures and scoping. But with simpler syntactic sugar (e.g. `enum-case` ADTs), it may be better to desugar it early on to avoid repetitive code in later stages.

order to avoid repetitive code for both functions definitions and lambdas.[11]

With any luck, perhaps *MEAL* or *MEASLes* will successfully deal with all these issues and include codegen and optimisations.

References

K. Cronin. What is tail call optimisation?, 2008. URL https://stackoverflow.com/questions/310974/what-is-tail-call-optimization.

M. Might. Closure conversion: How to compile lambda. URL https://matt.might.net/articles/closure-conversion/.

M. Rathi. Desugaring - taking our high-level language and simplifying it!, 2020. URL https://mukulrathi.com/create-your-own-programming-language/lower-language-constructs-to-llvm/.

[11] Another issue with lambda lifting is that more serious name analysis would be needed to find free variables in order to create a minimal environment.