The power of dark silicon

Darth Vader

the research facility,
darth.vader@hs-augsburg.de

Abstract. Sit amet mauris. Curabitur a quam. Aliquam neque. Nam nunc nunc, lacinia sed, varius quis, iaculis eget, ante. Nulla dictum justo eu lacus. Phasellus sit amet quam. Nullam sodales. Cras non magna eu est consectetuer faucibus. Donec tempor lobortis turpis. Sed tellus velit, ullamcorper ac, fringilla vitae, sodales nec, purus. Morbi aliquet risus in mi.

Keywords: hope, luke, ewoks

1 Foundations

We set out to implement the language, types and semantics given above. The end goal is to implement the two stepping functions

```
forward :: Location -> Participant -> Session Value ()
backward :: Location -> Participant -> Session Value ()
```

Where the `Session` type contains an `ExecutionState` holding a store of variables and other things, and the function can fail producing an `Error`.

```
type Session value a = StateT (ExecutionState value) (Except Error) a
```

1.1 The Monitor

Every participant has a monitor. This monitor stores variables, the current state of the type and some other information to be able to move backward.

```
data Monitor value tipe =
    Monitor
        { _localType :: LocalTypeState (Program value) value tipe
        , _recursiveVariableNumber :: Int
        , _recursionPoints :: List (LocalType tipe)
        , _store :: Map Identifier value
        , _applicationHistory :: Map Identifier (Identifier, value)
        }
        deriving (Show, Eq)
```

As mentioned, we have two kinds of session types: Global and Local. The Global type describes transactions and choices.

```haskell
a = "Alice"
b = "Bob"

data MyType
    = Address
    | ZipCode

globalType :: GlobalType MyType
globalType = do
    transaction a b ZipCode
    transaction b a Address
```

The Global type can then be projected onto a participant, resulting in a local type containing sends and receives, and offers and selects. The projection of `globalType` onto `a` and `b` would be equivalent to:

```haskell
aType :: LocalType MyType
aType = do
    send b ZipCode
    receive b Address

bType :: LocalType MyType
bType = do
    receive a ZipCode
    send a Address
```

TODO: unsure whether to include the full definition below, but I feel like the recursion should be mentioned

The definition of global types is given by

```haskell
type GlobalType u = Fix (GlobalTypeF u)

data GlobalTypeF u next
    = Transaction
        { from :: Participant, to :: Participant, tipe :: u, continuation ::  next }
    | OneOf { from :: Participant, to :: Participant, options :: Map String next }
    | R next
    | V
    | Wk next
    | End
    deriving (Show, Functor)
```

The recursive constructors are taken from 1. `R` introduces a recursion point, `V` jumps back to a recursion point and `Wk` weakens the recursion, making it possible to jump to a less tightly-binding `R`.

1.2 A Language

To build a prototype for the session types described previously, we also need a value language.

```haskell
type Participant = String
type Identifier = String

data ProgramF value next
    -- communication primitives
    = Send { owner :: Participant, value :: value, continuation :: next }
    | Receive { owner :: Participant, variableName :: Identifier, continuation :: next  }

    -- choice primitives
    | Offer Participant (List (String, next))
    | Select Participant (List (String, value, next))

    -- other constructors to make interesting examples
    | Parallel next next
    | Application Identifier value
    | Let Identifier value next
    | IfThenElse value next next
    | Literal value
    | NoOp
    deriving (Eq, Show, Functor)

-- TODO check this definition
type Program = Fix (ProgramF Value)


data Value
    = VBool Bool
    | VInt Int
    | VString String
    | VUnit
    | VIntOperator Value IntOperator Value
    | VComparison Value Ordering Value
    | VFunction Identifier (Program Value)
    | VReference Identifier
    | VLabel String
    deriving (Eq, Show)
```

Given a `LocalType` and a `Program`, we can now step through the program. For each instruction, we check the session type to see whether the instruction is allowed.

TODO: explain the need for `owner`

In the definition of `ProgramF`, the recursion is factored out and replaced by a type parameter. We then use `Fix` to give us back arbitrarily deep trees of instructions. The advantage of this transformation is that we can use recursion schemes - like folds - on the structure.

1.3 An eDSL with the free monad

Writing programs with `Fix` everywhere is tedious, and we can do better. A common method in haskell is to use the free monad to construct a monad out of a functor. With this monad we can use do-notation, which is much more pleasant to write.

The idea then is to use the free monad on our `ProgramF` data type to be able to build a nice DSL. For the transformation back, we also need some state: a variable counter that allows us to produce new unique variable names.

```haskell
newtype HighLevelProgram a =
    HighLevelProgram (StateT (Location, Participant, Int) (Free (ProgramF Value)) a)
        deriving (Functor, Applicative, Monad, MonadState (Location, Participant, Int))

uniqueVariableName :: HighLevelProgram Identifier
uniqueVariableName = do
    (location, participant, n) <- State.get
    State.put (location, participant, n + 1)
    return $ "var" ++ show n

send :: Value -> HighLevelProgram ()
send value = do
    (_, participant, _) <- State.get
    HighLevelProgram $ lift $ liftFree (Send participant value ())

receive :: HighLevelProgram Value
receive = do
    (_, participant, _) <- State.get
    variableName <- uniqueVariableName
    HighLevelProgram $ lift $ liftFree (Receive participant variableName ())
    return (VReference variableName)

terminate :: HighLevelProgram a
terminate = HighLevelProgram (lift $ Free NoOp)
```

We can now give correct implementations to the local types given above.

```
aType = do
    send B ZipCode
    receive B Address

alice = do
    let zipcode = VString "4242AB"
    send zipcode
    address <- receive
    terminate

bType = do
    receive A ZipCode
    send A Address

bob = do
    zipcode <- receive
    let address = "mûnewei 42"
    send address
```

And then transform them into a `Program` with

```
freeToFix :: Free (ProgramF value) a -> Program
freeToFix (Pure n) = Fix NoOp
freeToFix (Free x) = Fix (fmap freeToFix x)

compile :: Location -> Participant -> HighLevelProgram a -> Program
compile location participant (HighLevelProgram program) =
    freeToFix $ runStateT program (location, participant, 0)
```

2 Reversibility

Every local type case needs an "inverse" that contains enough information to undo the action. Additionally, the language's instructions (variable binding, if statements, function application, etc.) also need to be reversible.

```
type TypeContext program value a = Fix (TypeContextF program value a)

data TypeContextF program value a f
    = Hole
    | SendOrReceive (LocalTypeF a ()) f
    | Selected
```

```haskell
        { owner :: Participant
        , offerer :: Participant
        , selection :: Zipper (String, value, program, LocalType a)
        , continuation :: f
        }
    | Offered
        { owner :: Participant
        , selector :: Participant
        , picked :: Zipper (String, program, LocalType a)
        , continuation :: f
        }
    | Branched
        { condition :: value
        , verdict :: Bool
        , otherBranch :: program
        , continuation :: f
        }
    | Application Identifier Identifier f
    | Spawning Location Location Location f
    | Assignment
        { visibleName :: Identifier
        , internalName :: Identifier
        , continuation :: f
        }
    | Literal a f
    deriving (Eq, Show, Generic, Functor)
```

Given a `LocalType` and a `Program` we can now move forward whilst producing a trace through the execution. At any point, we can move back to a previous state.

3 Running everything

```haskell
type Session value a = StateT (ExecutionState value) (Except Error) a
```

3.1 Error generation

There are many failure conditions. accurate error reporting is very convenient whilst developing, but also to guide future users when something doesn't work.

```haskell
data Error
    = SessionNotInSync
    | UndefinedParticipant Participant
    | UndefinedVariable Participant Identifier
    | SynchronizationError String
    | LabelError String
    | QueueError String Queue.QueueError
    deriving (Eq, Show)
```

3.2 ExecutionState

```
data ExecutionState value =
    ExecutionState
        { _variableCount :: Int
        , _applicationCount :: Int
        , _participants :: Map Participant (Monitor value String)
        , _locations :: Map Location (Map Participant (Program value))
        , _queue :: Queue value
        , _isFunction :: value -> Maybe (Identifier, Program value)
        }
```

We can then finally define two stepping functions

```
forward :: Location -> Participant -> Session ()
backward :: Location -> Participant -> Session ()
```

4 future work

- also store state of the global protocol and use it to step
- make informed decisions when a branch of a choice fails

5 Conclusion

Bibliography

[1] F. van Walree, “Session types in cloud haskell,” 2017.