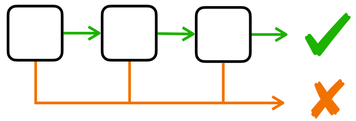
Notions of Stack-Manipulating Computation and Relative Monads

OOPSLA' 25

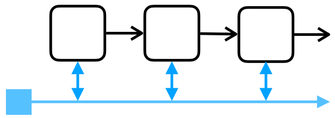
Yuchen Jiang, Runze Xue, Max S. New

Effects

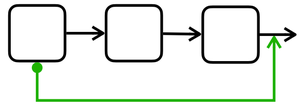
Exception



State



Continuations



How do we abstract over effects in high-level programming languages?

Monads

Moggi, Wadler,...

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathrm{ret}\ V : T\ A} \qquad \frac{\Gamma \vdash M : T\ A \quad \Gamma, x : A \vdash N : T\ A'}{\Gamma \vdash \mathrm{do}\ x \leftarrow M; N : T\ A'}$$

$T\ A$ types an effectful program that returns value of type A

Monads

Moggi, Wadler,...

Chain Effectful Programs via *monads*

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathrm{ret}\ V : T\,A} \qquad \frac{\Gamma \vdash M : T\,A \quad \Gamma, x : A \vdash N : T\,A'}{\Gamma \vdash \mathrm{do}\ x \leftarrow M; N : T\,A'}$$

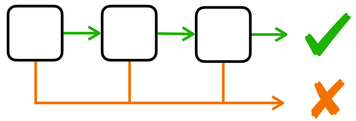
+

Interfaces for Specific Effects

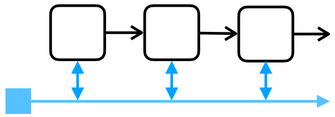
Exception E	State S	Continuation R
$\mathrm{raise} : E \to T\,A$	$\mathrm{get} : 1 \to T\,S$	$\mathrm{callcc} : ((A \to T\,A') \to T\,A) \to T\,A$
$\mathrm{Exn}\,A = A + E$	$\mathrm{set} : S \to T\,1$	

Effects

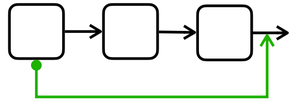
Exception



State



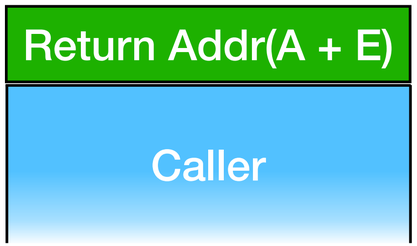
Continuation



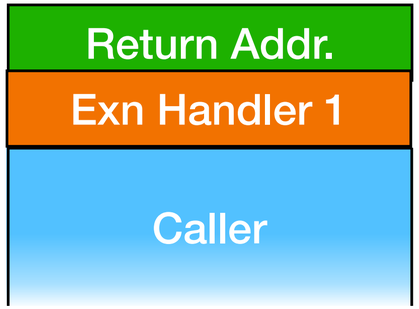
How do we implement effects in programming language implementation?

3 Implementations of Exceptions

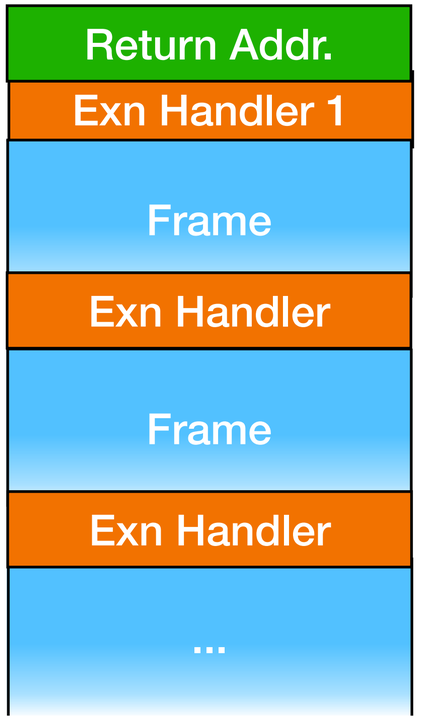
One continuation



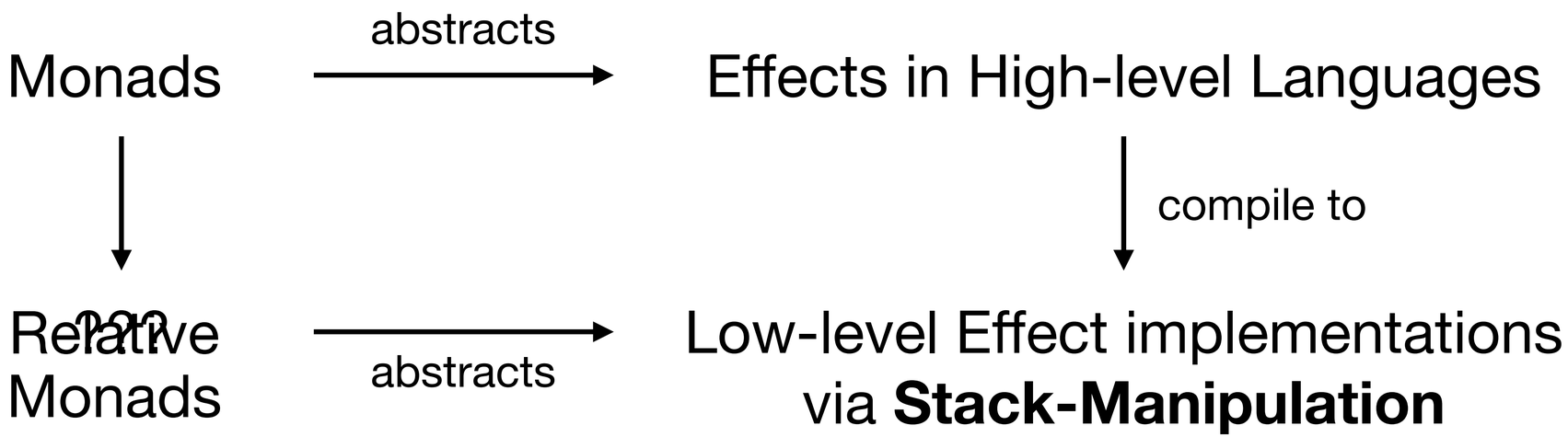
Two continuations



stack walking +
exception handlers



Monads in My Compiler?



Contribution

- Adapt *relative monads* to CBPV to abstract over low-level effects
- Introduce *monadic blocks*, an extension of do-notation for CBPV
- Automatic derivation of *relative monad transformers* to compose effects

Call-by-push-value

ValueType
classifies data

Basic Data	`Int`
Tagged Unions	$A + A'$
Tuples	$A \times A'$
Closures	Closure$B$
Existential	$\exists X.A$

```
data List (A : VType)
  | +Nil : Unit
  | +Cons : A * List A
```

ComputationType
classifies stacks

Stack-passed Argument	$A \to B$
Destructor Tag	$B \,\&\, B'$
Continuation	Return A
Polymorphism	$\forall X.B$

```
codata VarArg (A : VType) (C : CType)
| .more : A -> VarArg A B
| .done : B
```

Monads for CBPV

Altenkirch, Chapman and Uustalu
Monads Need Not Be Endofunctors

- The natural notion of monad for CBPV isn't quite a monad
  - Similar to monads but change the **kinds** of types

Monad T : `ValueType → ValueType`

Relative Monad T : `ValueType → ComputationType`

TA types **the stack** for a computation performing effects and returning A

CBPV Relative Monad Interface

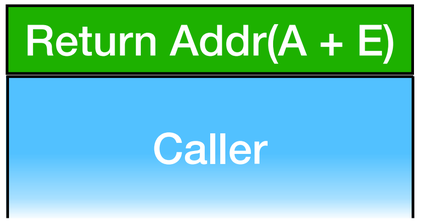
Kinds change, but the monad interface is otherwise unchanged!

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathrm{ret}\ V : T\ A} \qquad \frac{\Gamma \vdash M : T\ A \qquad \Gamma, x : A \vdash N : T\ A'}{\Gamma \vdash \mathrm{do}\ x \leftarrow M; N : T\ A'}$$

state, reader, writer, error, continuation, free monads, etc

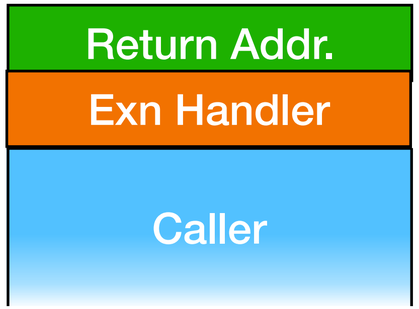
3 Implementations of Exceptions

One continuation



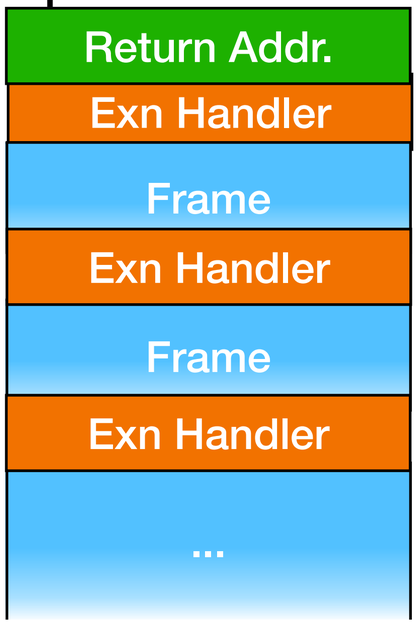

```
Ret (A + E)
```

Two continuations




```
forall R.
  Closure (E -> R) ->
  Closure (A -> R) ->
  R
```

Stack Walking + Exception Handlers

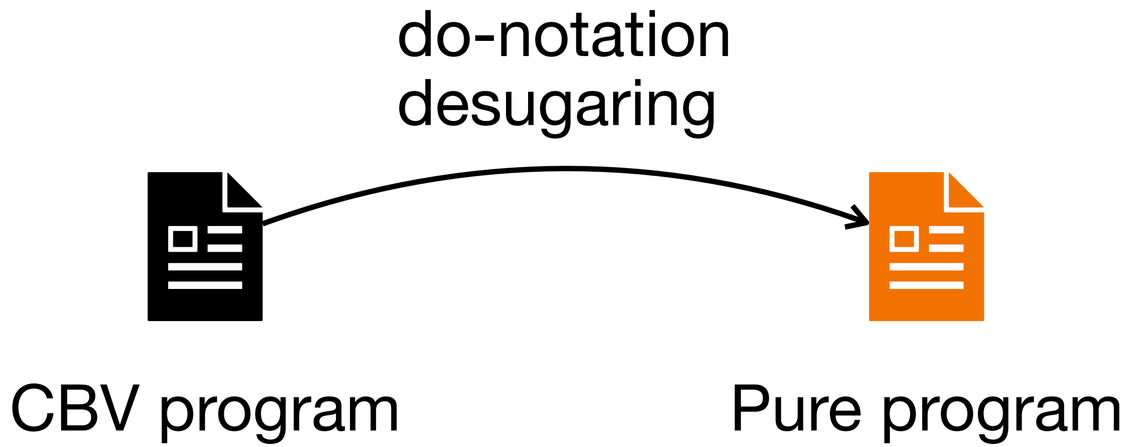



```
codata Exn3 E A where
  | .try : forall E'.
      Closure (E -> Exn3 E' A) -> Exn3 E' A
  | .kont : forall A'.
      Closure (A -> Exn3 E A') -> Exn3 E A'
  | .done : Ret (Either E A)
end
```

Contribution

- Adapt *relative monads* to CBPV to abstract over low-level effects ✓
- Introduce *monadic blocks*, an extension of do-notation for CBPV
- Automatic derivation of *relative monad transformers* to compose effects

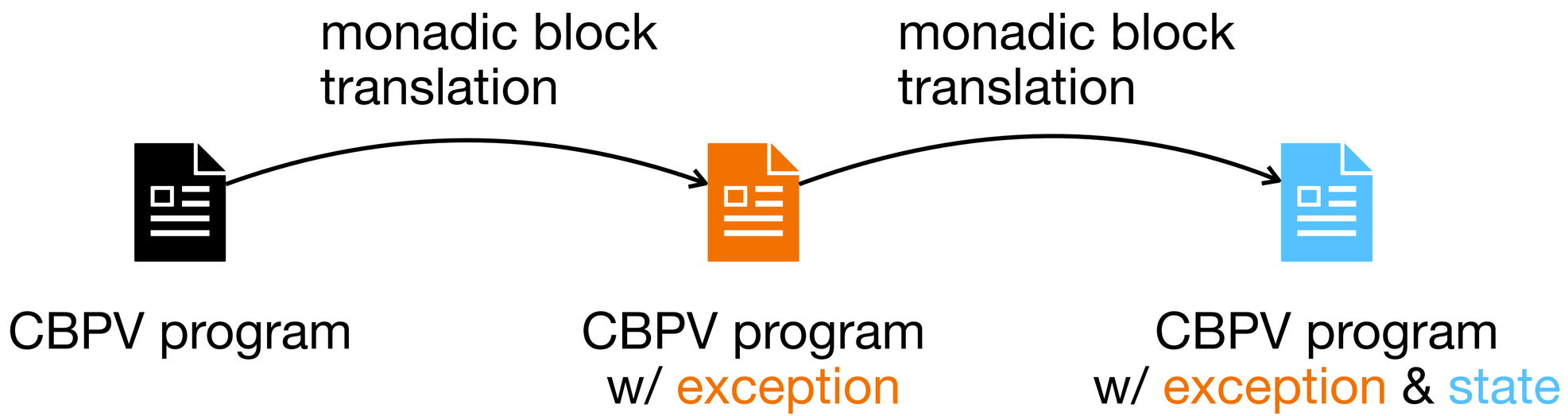
Embedded Effectful Programming



Moggi:

Given a monad on a model of pure lambda calculus, we can construct a model of call-by-vaue lambda calculus

Embedded Effectful Programming in CBPV

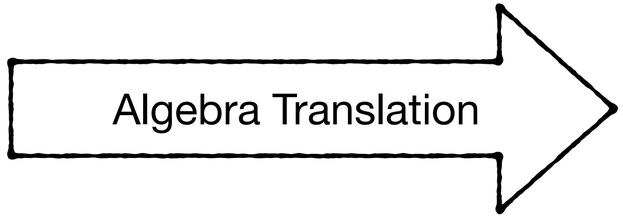


Fundamental Theorem of CBPV Relative Monads

Given a CBPV *relative monad T*, we construct a new CBPV model s.t. all closed CBPV program can be reinterpreted w.r.t. *relative monad T*.

Monadic Blocks

```
monadic fn raise ->
  do a <- 1 - 1;
  do b <- if a = 0
    then ! raise "..."
    else 42 / a;
  ret b
end
```




```
fn mo -> fn raise ->
  ! mo.bind { 1 - 1 }
    { fn a ->
      ! mo.bind
        { if a = 0
            then ! raise "..."
            else 42 / a }
        { fn b -> ! mo.return b }}
```

Contribution

Embedded Effectful Programing in CBPV

- Adapt *relative monads* to CBPV to abstract over low-level effects ✓
- Introduce *monadic blocks*, an extension of do-notation for CBPV ✓
- Automatic derivation of *relative monad transformers* to compose effects

Monad Transformers for Effect Composition

Compose Effects via *monad transformers*

:) Monad $T : \star \to \star$:(

Monad Transformer $F : (\star \to \star) \to (\star \to \star)$

+

Chain Effectful Programs via *monads*

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathrm{ret}\ V : T\,A} \qquad \frac{\Gamma \vdash M : T\,A \quad \Gamma, x : A \vdash N : T\,A'}{\Gamma \vdash \mathrm{do}\ x \leftarrow M; N : T\,A'}$$

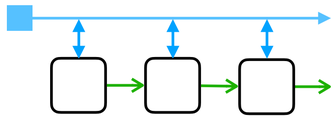
+

Interfaces for Specific Effects

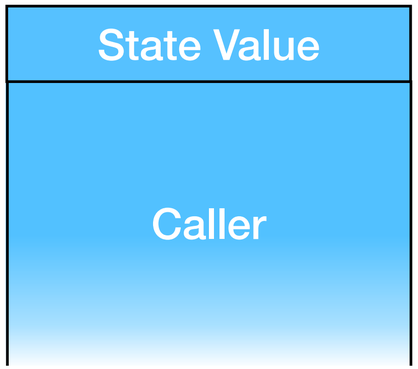
Exception E	State S	Continuation R
$\mathrm{raise} : E \to T\,A$	$\mathrm{get} : 1 \to T\,S$ $\mathrm{set} : S \to T\,1$	$\mathrm{callcc} : ((A \to T\,A') \to T\,A) \to T\,A$

Relative Monads Compose for Free!

State

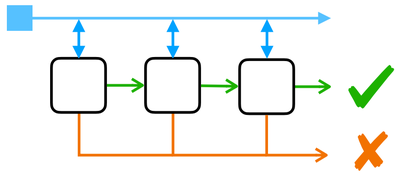


Runtime Stack

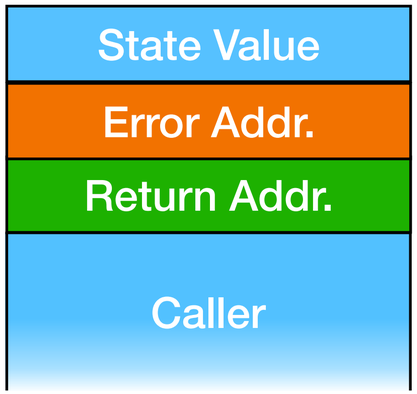


Relative Monads Compose for Free!

Exception & State



Runtime Stack

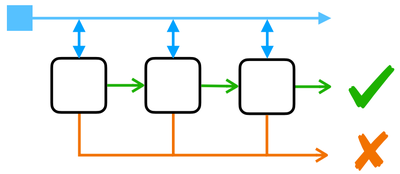


Relative Monads are always polymorphic over **tail** of the stack

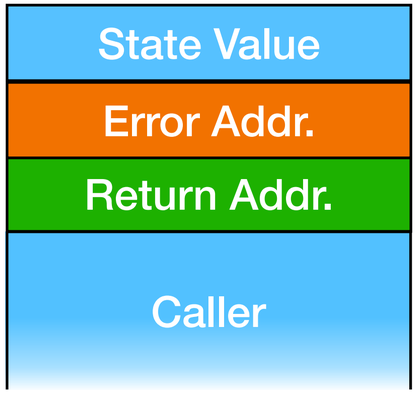
Every Relative Monad comes from a Monad Transformer

Monadic Block in CBPV

Exception & State



Runtime Stack



Relative monad transformers can be automatically derived via *monadic blocks*

Discussion

- More in paper
  - Common Relative Monad Instances and Stack-Walking Exceptions
  - Relative Monad, Algebras and their Laws
  - Algebra Translation implementing Monadic Blocks, correctness proofs
- Implementation in our research language, Zydeco 

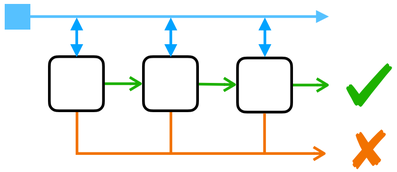
- Future work
  - CBPV as a stack-based IR in compilers
  - Relative Comonads?

Conclusion

Relative Monads for Stack-based Effect Implementations

- Adapt *relative monads* to CBPV to program with low-level effects

- Introduce *monadic blocks* to parameterize CBPV program w/ effects



- Automatically derive *relative monad transformers* to compose effects