Inferring Types and Effects via Static Single Assignment

Leonardo Rigon, **Paulo Torrens**, Cristiano Vasconcellos

Department of Computer Science
Santa Catarina State University

SAC - 2020/03/24

Motivation

- While we could argue that functional languages are based on expressions, imperative languages make a syntactic distinction between expressions and statements
  - Expressions **are**, statements **do**
  - Even impure functional languages that support loops and sequencing treat those as expressions (e.g., OCaml, Scheme)
- Most functional languages will offer some features to write more imperative-like code, but may still miss some features such as early return and arbitrary control flow
- Given an algorithm capable of translating statements into equivalent expressions, it should be possible to use control structures in a functional language
  - This algorithm exists: the algorithm for turning structured code into SSA

Motivation

- While we could argue that functional languages are based on expressions, imperative languages make a syntactic distinction between expressions and statements
  - Expressions **are**, statements **do**
  - Even impure functional languages that support loops and sequencing treat those as expressions (e.g., OCaml, Scheme)
- Most functional languages will offer some features to write more imperative-like code, but may still miss some features such as early return and arbitrary control flow
- Given an algorithm capable of translating statements into equivalent expressions, it should be possible to use control structures in a functional language
  - This algorithm exists: the algorithm for turning structured code into SSA

Motivation

- While we could argue that functional languages are based on expressions, imperative languages make a syntactic distinction between expressions and statements
  - Expressions **are**, statements **do**
  - Even impure functional languages that support loops and sequencing treat those as expressions (e.g., OCaml, Scheme)
- Most functional languages will offer some features to write more imperative-like code, but may still miss some features such as early return and arbitrary control flow
- Given an algorithm capable of translating statements into equivalent expressions, it should be possible to use control structures in a functional language
  - This algorithm exists: the algorithm for turning structured code into SSA

Motivation

- While we could argue that functional languages are based on expressions, imperative languages make a syntactic distinction between expressions and statements
  - Expressions **are**, statements **do**
  - Even impure functional languages that support loops and sequencing treat those as expressions (e.g., OCaml, Scheme)
- Most functional languages will offer some features to write more imperative-like code, but may still miss some features such as early return and arbitrary control flow
- Given an algorithm capable of translating statements into equivalent expressions, it should be possible to use control structures in a functional language
  - This algorithm exists: the algorithm for turning structured code into SSA

Motivation

- While we could argue that functional languages are based on expressions, imperative languages make a syntactic distinction between expressions and statements
    - Expressions **are**, statements **do**
    - Even impure functional languages that support loops and sequencing treat those as expressions (e.g., OCaml, Scheme)
- Most functional languages will offer some features to write more imperative-like code, but may still miss some features such as early return and arbitrary control flow
- Given an algorithm capable of translating statements into equivalent expressions, it should be possible to use control structures in a functional language
    - This algorithm exists: the algorithm for turning structured code into SSA

Motivation

- While we could argue that functional languages are based on expressions, imperative languages make a syntactic distinction between expressions and statements
  - Expressions **are**, statements **do**
  - Even impure functional languages that support loops and sequencing treat those as expressions (e.g., OCaml, Scheme)
- Most functional languages will offer some features to write more imperative-like code, but may still miss some features such as early return and arbitrary control flow
- Given an algorithm capable of translating statements into equivalent expressions, it should be possible to use control structures in a functional language
  - This algorithm exists: the algorithm for turning structured code into SSA

Static Single Assignment

- SSA is a common form used by intermediate languages for imperative programs, introduced by Cytron et al.
- Source programs are split into basic blocks, a basic unit of flow composed of atomic operations (e.g., assignments) that are necessarily sequential
- Each assignment to a local variable makes a new copy of it, and, upon control flow, phi functions are inserted to chose the live version of a variable
- SSA is a functional form described by a graph: blocks are mutually recursive abstractions, phi functions are parameters, and dominance information describes the lexical scope

Static Single Assignment

- SSA is a common form used by intermediate languages for imperative programs, introduced by Cytron et al.
- Source programs are split into basic blocks, a basic unit of flow composed of atomic operations (e.g., assignments) that are necessarily sequential
- Each assignment to a local variable makes a new copy of it, and, upon control flow, phi functions are inserted to chose the live version of a variable
- SSA is a functional form described by a graph: blocks are mutually recursive abstractions, phi functions are parameters, and dominance information describes the lexical scope

Static Single Assignment

- SSA is a common form used by intermediate languages for imperative programs, introduced by Cytron et al.
- Source programs are split into basic blocks, a basic unit of flow composed of atomic operations (e.g., assignments) that are necessarily sequential
- Each assignment to a local variable makes a new copy of it, and, upon control flow, phi functions are inserted to chose the live version of a variable
- SSA is a functional form described by a graph: blocks are mutually recursive abstractions, phi functions are parameters, and dominance information describes the lexical scope

Static Single Assignment

- SSA is a common form used by intermediate languages for imperative programs, introduced by Cytron et al.
- Source programs are split into basic blocks, a basic unit of flow composed of atomic operations (e.g., assignments) that are necessarily sequential
- Each assignment to a local variable makes a new copy of it, and, upon control flow, phi functions are inserted to chose the live version of a variable
- SSA is a functional form described by a graph: blocks are mutually recursive abstractions, phi functions are parameters, and dominance information describes the lexical scope

Static Single Assignment

```
algorithm fib(var n) {
  var a = 0;
  var b = 1;
  for(var i = 0; i < n; i++) {
    var c = b;
    b = a + b;
    a = c;
  }
  return a;
}
```

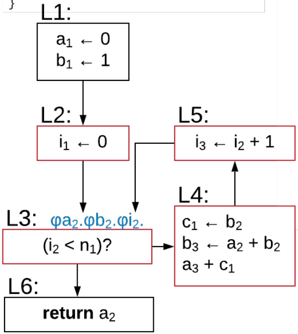
Static Single Assignment

```
algorithm fib(var n) {
  var a = 0;
  var b = 1;
  for(var i = 0; i < n; i++) {
    var c = b;
    b = a + b;
    a = c;
  }
  return a;
}
```

```
proc fib(n₁) {
     goto L1;
L1:  a₁ ← 0;
     b₁ ← 1;
     goto L2;
L2:  i₁ ← 0;
     goto L3;
L3:  a₂ ← φ(L2: a₁, L5: a₃);
     b₂ ← φ(L2: b₁, L5: b₃);
     i₂ ← φ(L2: i₁, L5: i₃);
     if i₂ < n₁ then
         goto L4
     else
         goto L6;
L4:  c₁ ← b₂;
     b₃ ← a₂ + b₂;
     a₃ ← c₁;
     goto L5;
L5:  i₃ ← i₂ + 1;
     goto L3;
L6:  return a₂
}
```

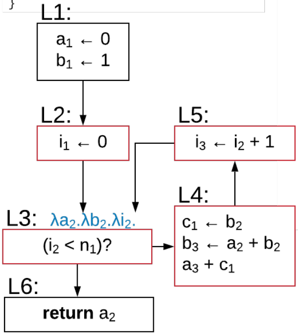
Static Single Assignment

```
algorithm fib(var n) {
  var a = 0;
  var b = 1;
  for(var i = 0; i < n; i++) {
    var c = b;
    b = a + b;
    a = c;
  }
  return a;
}
```




```
proc fib(n_1) {
    goto L1;
L1: a_1 ← 0;
    b_1 ← 1;
    goto L2;
L2: i_1 ← 0;
    goto L3;
L3: a_2 ← ϕ(L2: a_1, L5: a_3);
    b_2 ← ϕ(L2: b_1, L5: b_3);
    i_2 ← ϕ(L2: i_1, L5: i_3);
    if i_2 < n_1 then
        goto L4
    else
        goto L6;
L4: c_1 ← b_2;
    b_3 ← a_2 + b_2;
    a_3 ← c_1;
    goto L5;
L5: i_3 ← i_2 + 1;
    goto L3;
L6: return a_2
}
```

```
algorithm fib(var n) {
  var a = 0;
  var b = 1;
  for(var i = 0; i < n; i++) {
    var c = b;
    b = a + b;
    a = c;
  }
  return a;
}
```




```
proc fib(n₁) {
    goto L1;
L1: a₁ ← 0;
    b₁ ← 1;
    goto L2;
L2: i₁ ← 0;
    goto L3;
L3: a₂ ← ϕ(L2: a₁, L5: a₃);
    b₂ ← ϕ(L2: b₁, L5: b₃);
    i₂ ← ϕ(L2: i₁, L5: i₃);
    if i₂ < n₁ then
        goto L4
    else
        goto L6;
L4: c₁ ← b₂;
    b₃ ← a₂ + b₂;
    a₃ ← c₁;
    goto L5;
L5: i₃ ← i₂ + 1;
    goto L3;
L6: return a₂
}
```

Functional programs with control flow

- As it's possible to apply a syntactic translation from the SSA form to the lambda calculus, it's possible to use control flow statements in a functional setting
- If we consider a source language with just local assignment and local control flow (including `goto`), the resulting lambda term remains *purely functional*
  - As the SSA algorithm removes local mutability and control flow, these should not be viewed as computational effects, but rather as "syntactic sugar"
- Could a type and effect system, such as the one used by Koka, be used on the resulting term? If so, such a language could use restricted side effects akin to an imperative language

Functional programs with control flow

- As it's possible to apply a syntactic translation from the SSA form to the lambda calculus, it's possible to use control flow statements in a functional setting
- If we consider a source language with just local assignment and local control flow (including `goto`), the resulting lambda term remains *purely functional*
  - As the SSA algorithm removes local mutability and control flow, these should not be viewed as computational effects, but rather as "syntactic sugar"
- Could a type and effect system, such as the one used by Koka, be used on the resulting term? If so, such a language could use restricted side effects akin to an imperative language

Functional programs with control flow

- As it's possible to apply a syntactic translation from the SSA form to the lambda calculus, it's possible to use control flow statements in a functional setting
- If we consider a source language with just local assignment and local control flow (including `goto`), the resulting lambda term remains *purely functional*
  - As the SSA algorithm removes local mutability and control flow, these should not be viewed as computational effects, but rather as "syntactic sugar"
- Could a type and effect system, such as the one used by Koka, be used on the resulting term? If so, such a language could use restricted side effects akin to an imperative language

Functional programs with control flow

- As it's possible to apply a syntactic translation from the SSA form to the lambda calculus, it's possible to use control flow statements in a functional setting
- If we consider a source language with just local assignment and local control flow (including `goto`), the resulting lambda term remains *purely functional*
    - As the SSA algorithm removes local mutability and control flow, these should not be viewed as computational effects, but rather as "syntactic sugar"
- Could a type and effect system, such as the one used by Koka, be used on the resulting term? If so, such a language could use restricted side effects akin to an imperative language

- As exploratory research, a prototype has been implemented to apply the SSA translation in an imperative-like source syntax, followed by the syntactic translation into the lambda calculus

e ::=		terms
	$\mid\ x$	variables
	$\mid\ ()$	unit value
	$\mid\ n$	numbers
	$\mid\ b$	booleans
	$\mid\ \lambda x.e$	functions
	$\mid\ e_1\ e_2$	application
	$\mid\ \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$	conditional
	$\mid\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$	let bind
	$\mid\ e\ \mathtt{where}\ \{\ x_1 = e_1; \ldots; x_n = e_n\ \}$	where bind
τ ::=		types
	$\mid\ \alpha$	variable
	$\mid\ \mathit{unit}$	unit type
	$\mid\ \mathit{int}$	integer
	$\mid\ \mathit{bool}$	boolean
	$\mid\ \tau_1 \rightarrow \epsilon\ \tau_2$	function

- The target calculus has a specialized `where` clause, meant to represent basic blocks: nested, mutually recursive abstractions

Experiments

- As exploratory research, a prototype has been implemented to apply the SSA translation in an imperative-like source syntax, followed by the syntactic translation into the lambda calculus

e	$::=$		terms
	$\mid$	x	variables
	$\mid$	$()$	unit value
	$\mid$	n	numbers
	$\mid$	b	booleans
	$\mid$	$\lambda x.e$	functions
	$\mid$	$e_1\ e_2$	application
	$\mid$	$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	conditional
	$\mid$	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	let bind
	$\mid$	$e\ \mathbf{where}\ \{\ x_1 = e_1; \ldots; x_n = e_n\ \}$	where bind
τ	$::=$		types
	$\mid$	α	variable
	$\mid$	$unit$	unit type
	$\mid$	int	integer
	$\mid$	$bool$	boolean
	$\mid$	$\tau_1 \longrightarrow \epsilon\ \tau_2$	function

- The target calculus has a specialized `where` clause, meant to represent basic blocks: nested, mutually recursive abstractions

- On the resulting lambda term, a type and effect inference algorithm based on effect rows is applied
- Most rules are standard and similar to those of the Hindley-Milner type system; effect rows inspired by those of Leijen's Koka language
- A special rule is adapted for the `where` rule, which binds mutually recursive terms that are monomorphic and don't perform any effects

$$\frac{\Gamma \vdash e_1 \colon bool \mathrel{!} \epsilon \qquad \Gamma \vdash e_2 \colon \tau \mathrel{!} \epsilon \qquad \Gamma \vdash e_3 \colon \tau \mathrel{!} \epsilon}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \colon \tau \mathrel{!} \epsilon}\ (\text{IF})$$

$$\frac{\Gamma \vdash e_1 \colon \tau_1 \mathrel{!} \langle\rangle \qquad \sigma = gen(\tau_1, \Gamma) \qquad \Gamma, x \colon \sigma \vdash e_2 \colon \tau_2 \mathrel{!} \epsilon}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \colon \tau_2 \mathrel{!} \epsilon}\ (\text{LET})$$

$$\frac{\forall i \in 1..n.\ \Gamma,\ x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash e_i \colon \tau_i \mathrel{!} \langle\rangle \qquad \Gamma,\ x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash e \colon \tau \mathrel{!} \epsilon}{\Gamma \vdash e\ \textbf{where}\ \{\ x_1 = e_1; \ldots; x_n = e_n\ \} \colon \tau \mathrel{!} \epsilon}\ (\text{WHERE})$$

- On the resulting lambda term, a type and effect inference algorithm based on effect rows is applied
- Most rules are standard and similar to those of the Hindley-Milner type system; effect rows inspired by those of Leijen's Koka language
- A special rule is adapted for the `where` rule, which binds mutually recursive terms that are monomorphic and don't perform any effects

$$\frac{\Gamma \vdash e_1 \colon bool \ ! \ \epsilon \qquad \Gamma \vdash e_2 \colon \tau \ ! \ \epsilon \qquad \Gamma \vdash e_3 \colon \tau \ ! \ \epsilon}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \colon \tau \ ! \ \epsilon} \text{(IF)}$$

$$\frac{\Gamma \vdash e_1 \colon \tau_1 \ ! \ \langle\rangle \qquad \sigma = gen(\tau_1, \Gamma) \qquad \Gamma, x \colon \sigma \vdash e_2 \colon \tau_2 \ ! \ \epsilon}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \colon \tau_2 \ ! \ \epsilon} \text{(LET)}$$

$$\frac{\forall i \in 1..n.\ \Gamma,\ x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash e_i \colon \tau_i \ ! \ \langle\rangle \qquad \Gamma,\ x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash e \colon \tau \ ! \ \epsilon}{\Gamma \vdash e \textbf{ where } \{\ x_1 = e_1; \ldots; x_n = e_n\ \} \colon \tau \ ! \ \epsilon} \text{(WHERE)}$$

- On the resulting lambda term, a type and effect inference algorithm based on effect rows is applied
- Most rules are standard and similar to those of the Hindley-Milner type system; effect rows inspired by those of Leijen's Koka language
- A special rule is adapted for the `where` rule, which binds mutually recursive terms that are monomorphic and don't perform any effects

$$\frac{\Gamma \vdash e_1 : \mathit{bool} \;!\; \epsilon \qquad \Gamma \vdash e_2 : \tau \;!\; \epsilon \qquad \Gamma \vdash e_3 : \tau \;!\; \epsilon}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau \;!\; \epsilon}\ (\mathrm{IF})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \;!\; \langle\rangle \qquad \sigma = \mathit{gen}(\tau_1, \Gamma) \qquad \Gamma, x : \sigma \vdash e_2 : \tau_2 \;!\; \epsilon}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2 \;!\; \epsilon}\ (\mathrm{LET})$$

$$\frac{\forall i \in 1..n.\ \Gamma,\ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_i : \tau_i \;!\; \langle\rangle \qquad \Gamma,\ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \;!\; \epsilon}{\Gamma \vdash e\ \textbf{where}\ \{\ x_1 = e_1; \ldots; x_n = e_n\ \} : \tau \;!\; \epsilon}\ (\mathrm{WHERE})$$

Results

- Some small-scale programs were written in an imperative-like source syntax and run through the type inference algorithm
- The source syntax contained local mutability, control flow (including `goto`), and effect and handler declaration
- Preliminary results show functions to receive the expected types; basic blocks are well behaved in the sense that each block becomes a lambda abstraction with the same returning type and same effect
  - Phi functions guarantee that each block has and is called with the correct arity
- Algebraic effects freely compose inside the source program, giving an imperative look and feel, while still working on a purely functional setting (i.e., arbitrary side effects and mutability still forbidden!)

Results

- Some small-scale programs were written in an imperative-like source syntax and run through the type inference algorithm
- The source syntax contained local mutability, control flow (including `goto`), and effect and handler declaration
- Preliminary results show functions to receive the expected types; basic blocks are well behaved in the sense that each block becomes a lambda abstraction with the same returning type and same effect
  - Phi functions guarantee that each block has and is called with the correct arity
- Algebraic effects freely compose inside the source program, giving an imperative look and feel, while still working on a purely functional setting (i.e., arbitrary side effects and mutability still forbidden!)

- Some small-scale programs were written in an imperative-like source syntax and run through the type inference algorithm
- The source syntax contained local mutability, control flow (including `goto`), and effect and handler declaration
- Preliminary results show functions to receive the expected types; basic blocks are well behaved in the sense that each block becomes a lambda abstraction with the same returning type and same effect
  - Phi functions guarantee that each block has and is called with the correct arity
- Algebraic effects freely compose inside the source program, giving an imperative look and feel, while still working on a purely functional setting (i.e., arbitrary side effects and mutability still forbidden!)

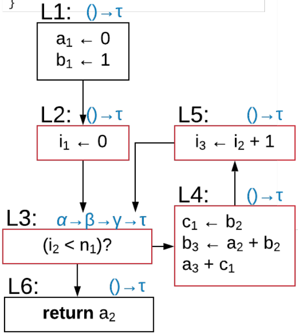
- Some small-scale programs were written in an imperative-like source syntax and run through the type inference algorithm
- The source syntax contained local mutability, control flow (including `goto`), and effect and handler declaration
- Preliminary results show functions to receive the expected types; basic blocks are well behaved in the sense that each block becomes a lambda abstraction with the same returning type and same effect
  - Phi functions guarantee that each block has and is called with the correct arity
- Algebraic effects freely compose inside the source program, giving an imperative look and feel, while still working on a purely functional setting (i.e., arbitrary side effects and mutability still forbidden!)

Results

- Some small-scale programs were written in an imperative-like source syntax and run through the type inference algorithm
- The source syntax contained local mutability, control flow (including `goto`), and effect and handler declaration
- Preliminary results show functions to receive the expected types; basic blocks are well behaved in the sense that each block becomes a lambda abstraction with the same returning type and same effect
  - Phi functions guarantee that each block has and is called with the correct arity
- Algebraic effects freely compose inside the source program, giving an imperative look and feel, while still working on a purely functional setting (i.e., arbitrary side effects and mutability still forbidden!)

Results

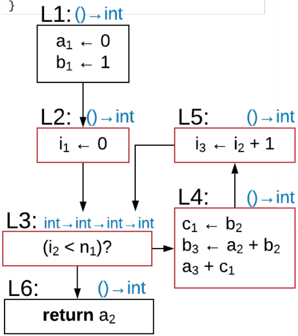
```
algorithm fib(var n) {
  var a = 0;
  var b = 1;
  for(var i = 0; i < n; i++) {
    var c = b;
    b = a + b;
    a = c;
  }
  return a;
}
```




```
proc fib(n1) {
    goto L1;
L1: a1 ← 0;
    b1 ← 1;
    goto L2;
L2: i1 ← 0;
    goto L3;
L3: a2 ← ϕ(L2: a1, L5: a3);
    b2 ← ϕ(L2: b1, L5: b3);
    i2 ← ϕ(L2: i1, L5: i3);
    if i2 < n1 then
        goto L4
    else
        goto L6;
L4: c1 ← b2;
    b3 ← a2 + b2;
    a3 ← c1;
    goto L5;
L5: i3 ← i2 + 1;
    goto L3;
L6: return a2
}
```

Results

```
algorithm fib(var n) {
  var a = 0;
  var b = 1;
  for(var i = 0; i < n; i++) {
    var c = b;
    b = a + b;
    a = c;
  }
  return a;
}
```




```
proc fib(n₁) {
    goto L1;
L1: a₁ ← 0;
    b₁ ← 1;
    goto L2;
L2: i₁ ← 0;
    goto L3;
L3: a₂ ← ϕ(L2: a₁, L5: a₃);
    b₂ ← ϕ(L2: b₁, L5: b₃);
    i₂ ← ϕ(L2: i₁, L5: i₃);
    if i₂ < n₁ then
        goto L4
    else
        goto L6;
L4: c₁ ← b₂;
    b₃ ← a₂ + b₂;
    a₃ ← c₁;
    goto L5;
L5: i₃ ← i₂ + 1;
    goto L3;
L6: return a₂
}
```

Results

```
// Context:
//   print: string -> <console | e> unit
//   raise: unit -> <exception | e> unit
//   catch: (unit -> <exception | e> a) -> e maybe<a>
```

Results

```
// Context:
//   print: string -> <console | e> unit
//   raise: unit -> <exception | e> unit
//   catch: (unit -> <exception | e> a) -> e maybe<a>

algorithm safediv(var a, var b) {
    if(b == 0) {
        print("Can't divide by zero!")
        raise();
    }
    return a / b;
}
```

Results

```
// Context:
//   print: string -> <console | e> unit
//   raise: unit -> <exception | e> unit
//   catch: (unit -> <exception | e> a) -> e maybe<a>

algorithm safediv(var a, var b) {
    if(b == 0) {
        print("Can't divide by zero!")
        raise();
    }
    return a / b;
}

// safediv: int -> int -> <exception, console | e> int
```

Results

```
// Context:
//   print: string -> <console | e> unit
//   raise: unit -> <exception | e> unit
//   catch: (unit -> <exception | e> a) -> e maybe<a>

algorithm safediv(var a, var b) {
    if(b == 0) {
        print("Can't divide by zero!")
        raise();
    }
    return a / b;
}

// safediv: int -> int -> <exception, console | e> int

algorithm safe1(var a, var b) {
    return catch([] {
        return safediv(a, b);
    });
}
```

Results

```
// Context:
//   print: string -> <console | e> unit
//   raise: unit -> <exception | e> unit
//   catch: (unit -> <exception | e> a) -> e maybe<a>

algorithm safediv(var a, var b) {
    if(b == 0) {
        print("Can't divide by zero!")
        raise();
    }
    return a / b;
}

// safediv: int -> int -> <exception, console | e> int

algorithm safe1(var a, var b) {
    return catch([] {
        return safediv(a, b);
    });
}

// safe1: int -> int -> <console | e> maybe<int>
```

Results

```
// Context:
//   print: string -> <console | e> unit
//   raise: unit -> <exception | e> unit
//   catch: (unit -> <exception | e> a) -> e maybe<a>

algorithm safediv(var a, var b) {
    if(b == 0) {
        print("Can't divide by zero!")
        raise();
    }
    return a / b;
}

// safediv: int -> int -> <exception, console | e> int

algorithm safe1(var a, var b) {
    return catch([] {
        return safediv(a, b);
    });
}

// safe1: int -> int -> <console | e> maybe<int>

algorithm safe2(var a, var b) {
    inject catch();
    return safediv(a, b);
}
```

Results

```
// Context:
//   print: string -> <console | e> unit
//   raise: unit -> <exception | e> unit
//   catch: (unit -> <exception | e> a) -> e maybe<a>

algorithm safediv(var a, var b) {
    if(b == 0) {
        print("Can't divide by zero!")
        raise();
    }
    return a / b;
}

// safediv: int -> int -> <exception, console | e> int

algorithm safe1(var a, var b) {
    return catch([] {
        return safediv(a, b);
    });
}

// safe1: int -> int -> <console | e> maybe<int>

algorithm safe2(var a, var b) {
    inject catch();
    return safediv(a, b);
}

// safe2: int -> int -> <console | e> maybe<int>
```

Conclusions & Future Work

- It's possible to write functional programs with local mutability and control flow; the SSA conversion algorithm translates statements into equivalent pure expressions
- The resulting lambda term may have its type inferred by a standard type inference algorithm (such as Hindley-Milner's)
- Algebraic effect inference, when used with the SSA algorithm, compose freely giving an imperative look-and-feel in a still functional language, hopefully helping in writing algorithms described in an imperative way
- As the translation from SSA form to the lambda calculus is syntactical, it should be possible to adapt a type inference system to SSA syntax with basic blocks, skipping a step
- Could inference algorithms applied in SSA form help in type inference for actual imperative languages?

Conclusions & Future Work

- It's possible to write functional programs with local mutability and control flow; the SSA conversion algorithm translates statements into equivalent pure expressions
- The resulting lambda term may have its type inferred by a standard type inference algorithm (such as Hindley-Milner's)
- Algebraic effect inference, when used with the SSA algorithm, compose freely giving an imperative look-and-feel in a still functional language, hopefully helping in writing algorithms described in an imperative way
- As the translation from SSA form to the lambda calculus is syntactical, it should be possible to adapt a type inference system to SSA syntax with basic blocks, skipping a step
- Could inference algorithms applied in SSA form help in type inference for actual imperative languages?

Conclusions & Future Work

- It's possible to write functional programs with local mutability and control flow; the SSA conversion algorithm translates statements into equivalent pure expressions
- The resulting lambda term may have its type inferred by a standard type inference algorithm (such as Hindley-Milner's)
- Algebraic effect inference, when used with the SSA algorithm, compose freely giving an imperative look-and-feel in a still functional language, hopefully helping in writing algorithms described in an imperative way
- As the translation from SSA form to the lambda calculus is syntactical, it should be possible to adapt a type inference system to SSA syntax with basic blocks, skipping a step
- Could inference algorithms applied in SSA form help in type inference for actual imperative languages?

Conclusions & Future Work

- It's possible to write functional programs with local mutability and control flow; the SSA conversion algorithm translates statements into equivalent pure expressions
- The resulting lambda term may have its type inferred by a standard type inference algorithm (such as Hindley-Milner's)
- Algebraic effect inference, when used with the SSA algorithm, compose freely giving an imperative look-and-feel in a still functional language, hopefully helping in writing algorithms described in an imperative way
- As the translation from SSA form to the lambda calculus is syntactical, it should be possible to adapt a type inference system to SSA syntax with basic blocks, skipping a step
- Could inference algorithms applied in SSA form help in type inference for actual imperative languages?

Conclusions & Future Work

- It's possible to write functional programs with local mutability and control flow; the SSA conversion algorithm translates statements into equivalent pure expressions
- The resulting lambda term may have its type inferred by a standard type inference algorithm (such as Hindley-Milner's)
- Algebraic effect inference, when used with the SSA algorithm, compose freely giving an imperative look-and-feel in a still functional language, hopefully helping in writing algorithms described in an imperative way
- As the translation from SSA form to the lambda calculus is syntactical, it should be possible to adapt a type inference system to SSA syntax with basic blocks, skipping a step
- Could inference algorithms applied in SSA form help in type inference for actual imperative languages?