Modular I/O Reasoning in DimSum

Alex Loitzl[1]

[1] Institute of Science and Technology Austria (ISTA)

March, 2025

Modular I/O Reasoning

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

reads from increasing sequence

```
{...}
echo
{...}
```

Modular I/O Reasoning

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

reads from increasing sequence

$$\{\lambda\ \mathsf{es},\ \ulcorner \mathsf{es} = [\ ] \urcorner\}$$

$$\texttt{echo}$$

$$\{\lambda\ \mathsf{v},\ \ulcorner \mathsf{v} = 0 \urcorner\}$$

Modular I/O Reasoning

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

reads from increasing sequence

$$\Big\{\lambda\ \mathsf{es},\ \ulcorner \mathsf{es} = [\ ]\urcorner * \{\lambda\ \mathsf{es},\ \ulcorner \mathsf{es} = [\ ]\urcorner\}\texttt{getc}\{\lambda\ \mathsf{v},\ \{\lambda\ \mathsf{es},\ \ulcorner \mathsf{es} = \mathsf{v}\urcorner\}\texttt{putc}\{\_\}\ \}\Big\}$$

$$\texttt{echo}$$

$$\{\lambda\ \mathsf{v},\ \ulcorner \mathsf{v} = 0\urcorner\}$$

Modular I/O Reasoning

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

reads from increasing sequence

$$\{\lambda\ \mathsf{es},\ \ulcorner \mathsf{es} = [\ ] \urcorner * \exists\ \mathsf{v},\ \mathsf{P\ v} * (\mathtt{getc\_spec}\ \mathsf{P}) * (\mathtt{putc\_spec}\ \mathsf{P})\}$$

$$\mathtt{echo}$$

$$\{\lambda\ \mathsf{v},\ \ulcorner \mathsf{v} = 0 \urcorner\}$$

Modular I/O Reasoning

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

reads from increasing sequence

$$\{\lambda\ \mathsf{es},\ \ulcorner \mathsf{es} = [\,]\urcorner * \exists\ \mathsf{v},\ \mathsf{P}\ \mathsf{v}\}\ \texttt{getc}\ \{\lambda\ \mathsf{ret},\ \ulcorner \mathsf{ret} = \mathsf{v}\urcorner * \mathsf{P}\ (\mathsf{v} + 1)\}$$

$$\{\lambda\ \mathsf{es},\ \exists\ \mathsf{v},\ \ulcorner \mathsf{es} = \mathsf{v}\urcorner * \mathsf{P}\ (\mathsf{v} + 1)\}\ \texttt{putc}\ \{\lambda\ \mathsf{ret},\ \mathsf{P}\ (\mathsf{v} + 1)\}$$

$$\{\lambda\ \mathsf{es},\ \ulcorner \mathsf{es} = [\,]\urcorner * \exists\ \mathsf{v},\ \mathsf{P}\ \mathsf{v} * (\texttt{getc\_spec}\ \mathsf{P}) * (\texttt{putc\_spec}\ \mathsf{P})\}$$

$$\texttt{echo}$$

$$\{\lambda\ \mathsf{v},\ \ulcorner \mathsf{v} = 0\urcorner\}$$

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

DimSum

- Decentralized/language-agnostic multi-language semantics
  - No fixed source
  - No fixed set of languages
  - No fixed memory model
  - No fixed notion of linking
- Notion of semantic linking: $\oplus$
  - Link semantic components (modules) rather than syntactic
  - Link programs with specifications (abstract program)
- Program semantics as LTS, interaction via synchronization
- Reason locally in terms of interaction of two modules

Proofs in DimSum

$$\boxed{[\![\mathsf{echo}_{\mathsf{rec}} \quad \oplus \quad \mathsf{getc}_{\mathsf{spec}}]\!]} \preceq \quad [\![\mathsf{echo}_{\mathsf{spec}}]\!]$$

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

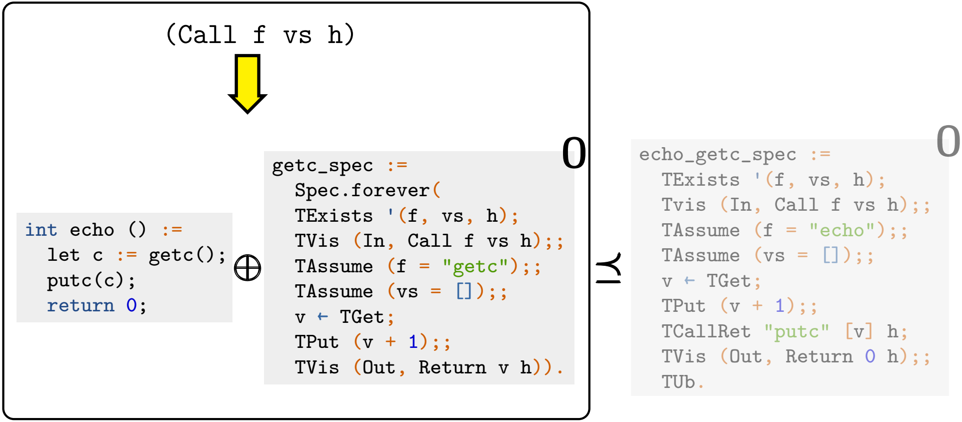
0

≼

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

0

Proofs in DimSum



(Call f vs h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

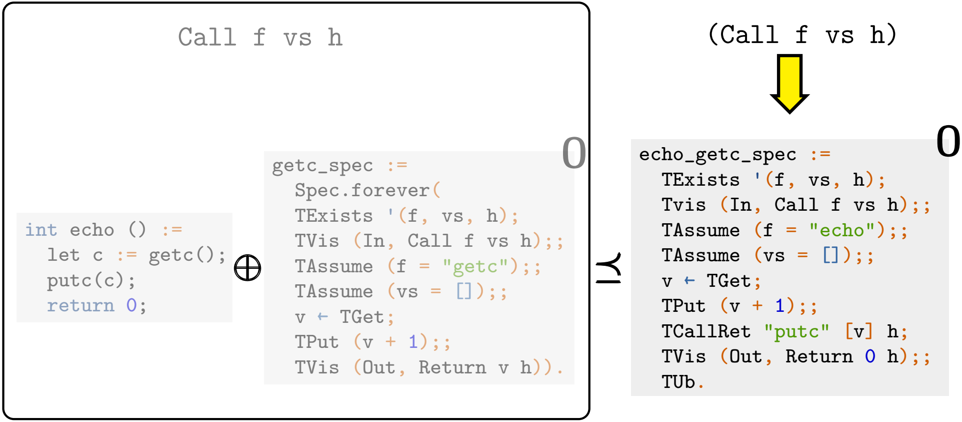
0

$\preceq$

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

0

Proofs in DimSum



```
Call f vs h
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

$\oplus$

0

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

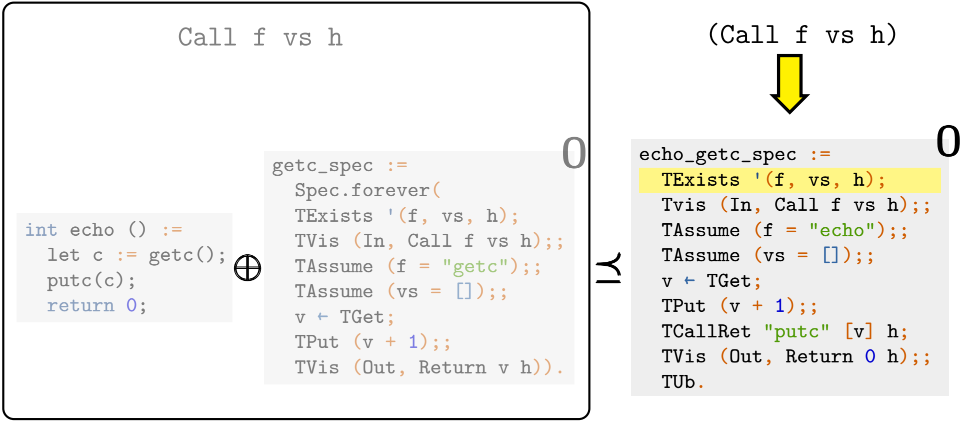
$\preceq$

```
(Call f vs h)
```

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



Call f vs h

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

0

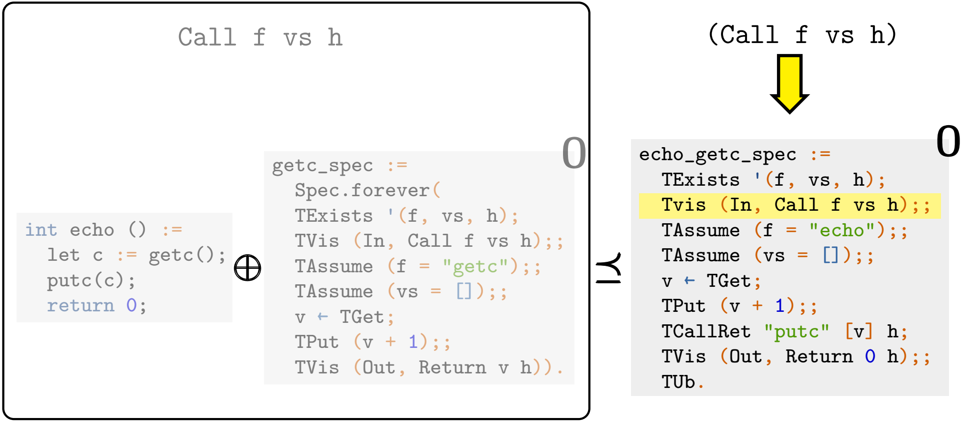
≼

(Call f vs h)

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

0

Proofs in DimSum



```
Call f vs h
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

```
(Call f vs h)
```

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
(Call "echo" vs h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

0

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
(Call "echo" [] h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

$\oplus$

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

^{0}

$\preceq$

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

^{0}

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

0

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

0

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
(Call "getc" [] h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

0

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

$\preceq$

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

(Call "getc" [] h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

0

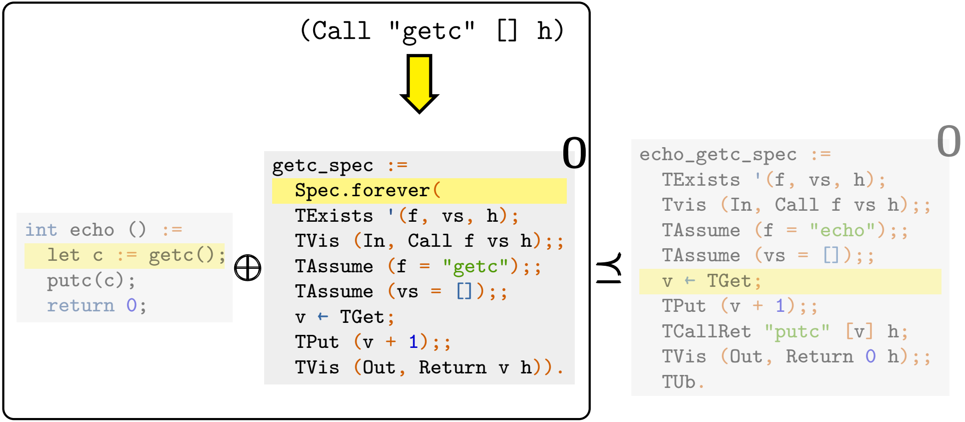
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



(Call "getc" [] h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

0

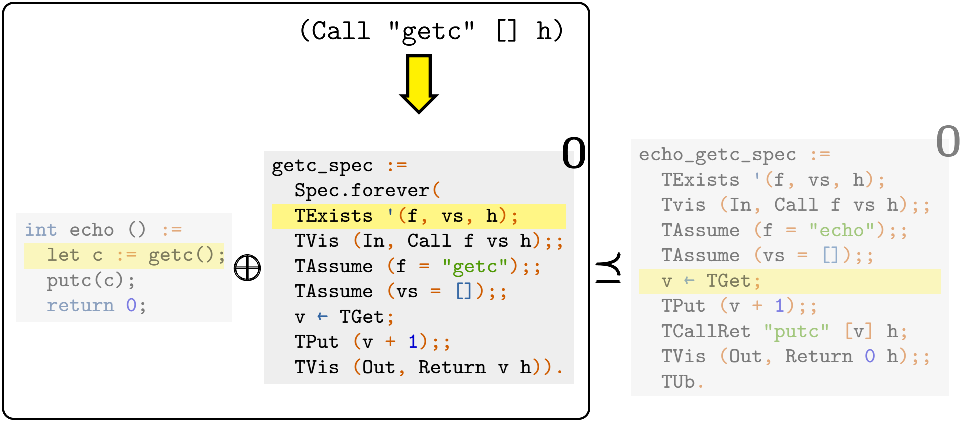
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

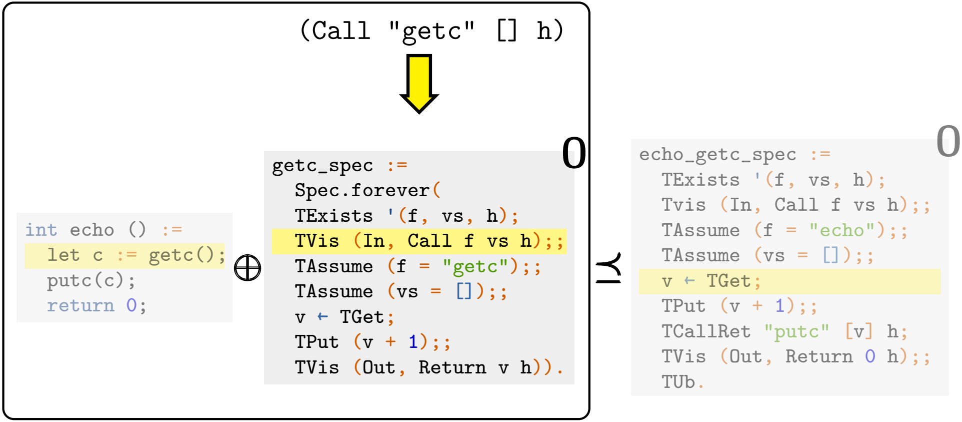
⊕

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



(Call "getc" [] h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

0

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
(Call "getc" [] h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

0

⪯

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

0

Proofs in DimSum

```
(Call "getc" [] h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

0

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

$\oplus$

0

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

$\preceq$

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

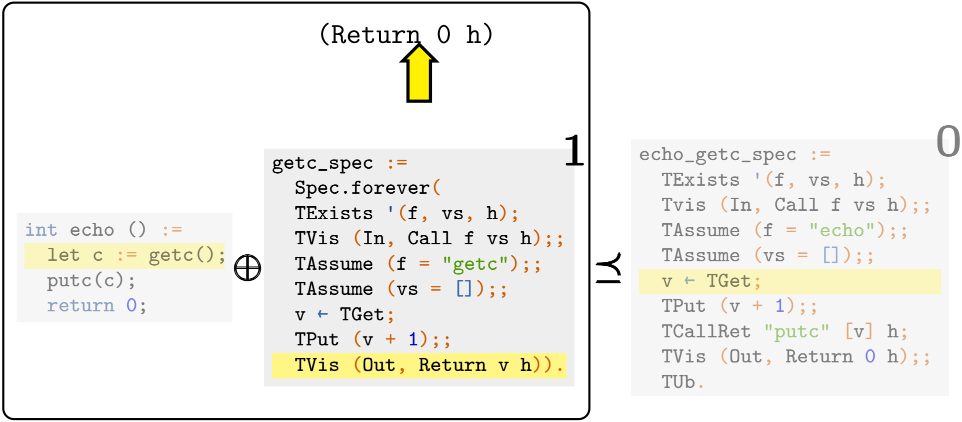
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



```
(Return 0 h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

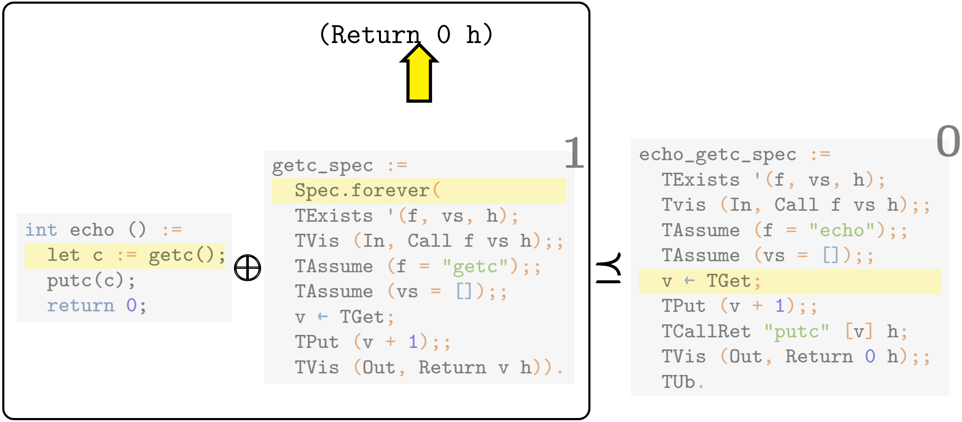
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



(Return 0 h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

$\oplus$

1

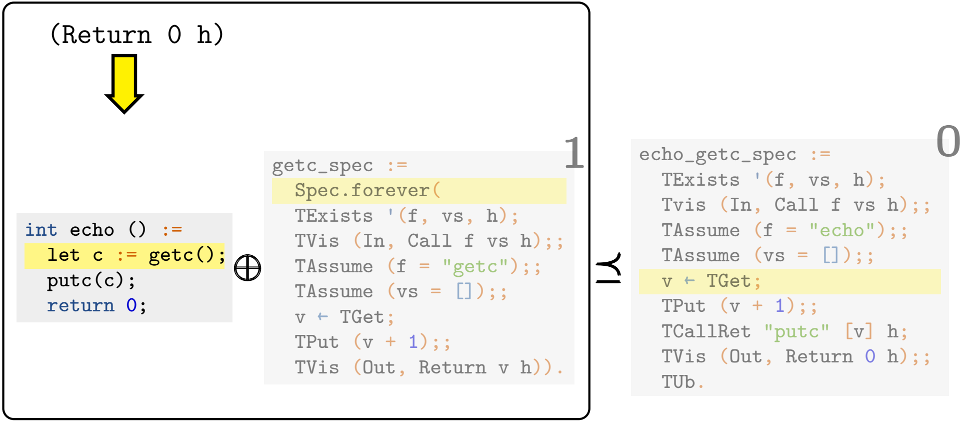
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

$\preceq$

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



(Return 0 h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

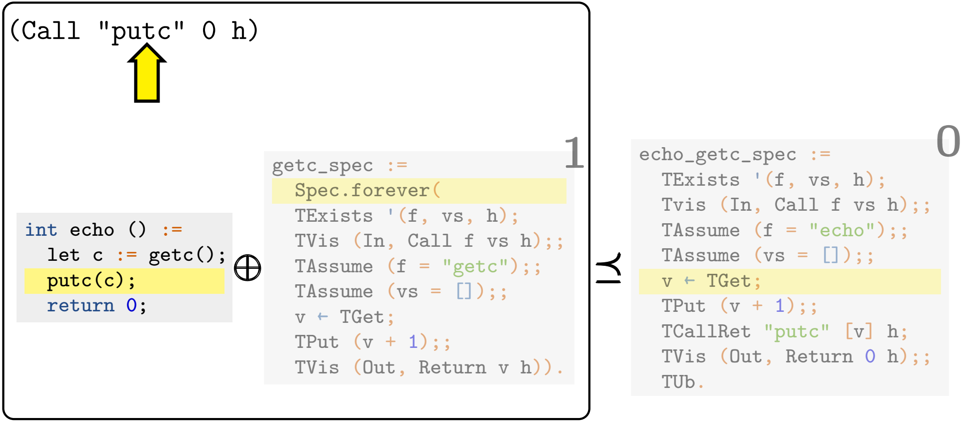
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



```
(Call "putc" 0 h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

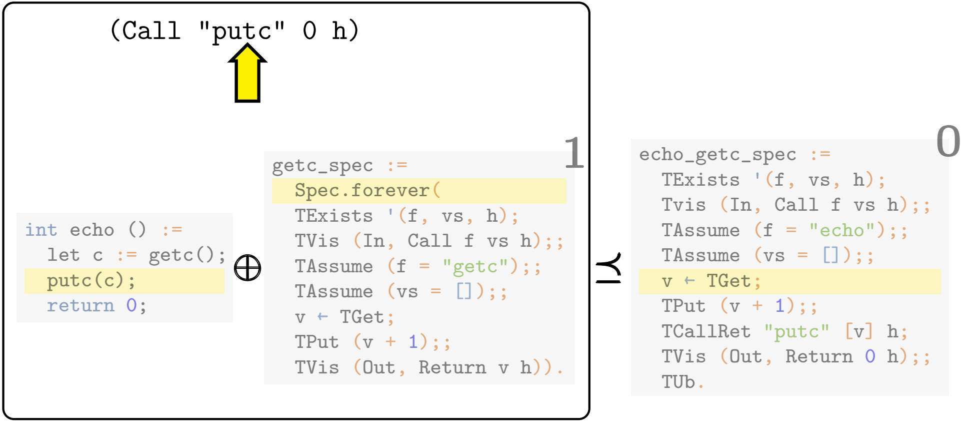
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



```
(Call "putc" 0 h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
(Call "putc" 0 h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

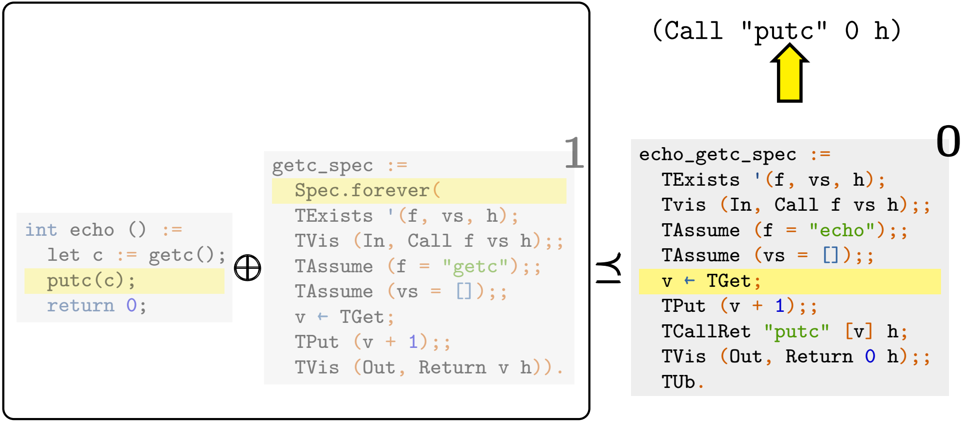
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



(Call "putc" 0 h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

0

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

(Call "putc" 0 h)

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

(Call "putc" 0 h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

(Call "putc" 0 h)

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

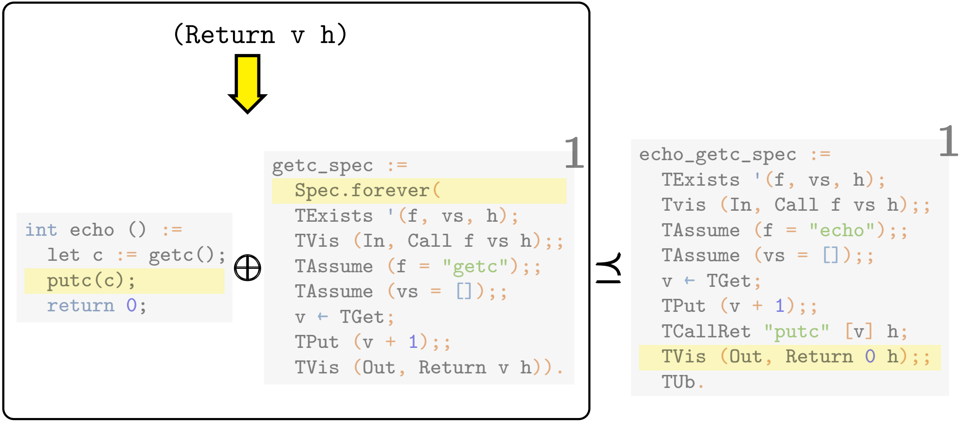
⪯

(Return v h)

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



(Return v h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

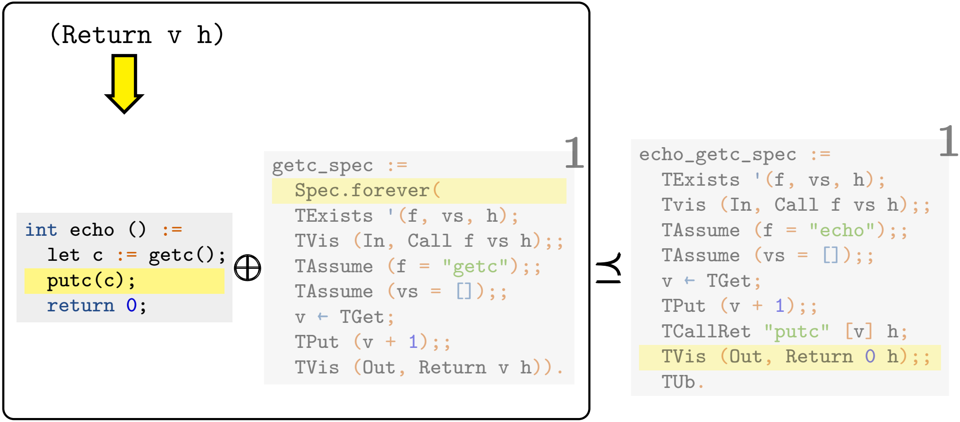
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



(Return v h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

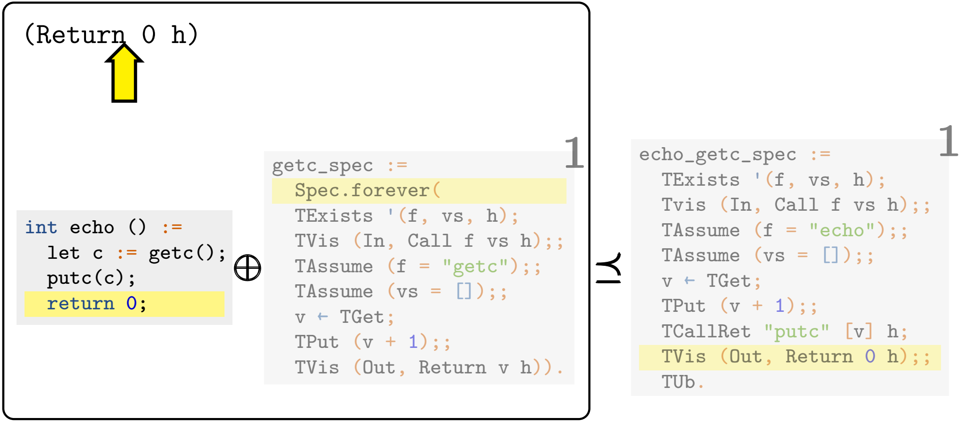
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



```
(Return 0 h)
```

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

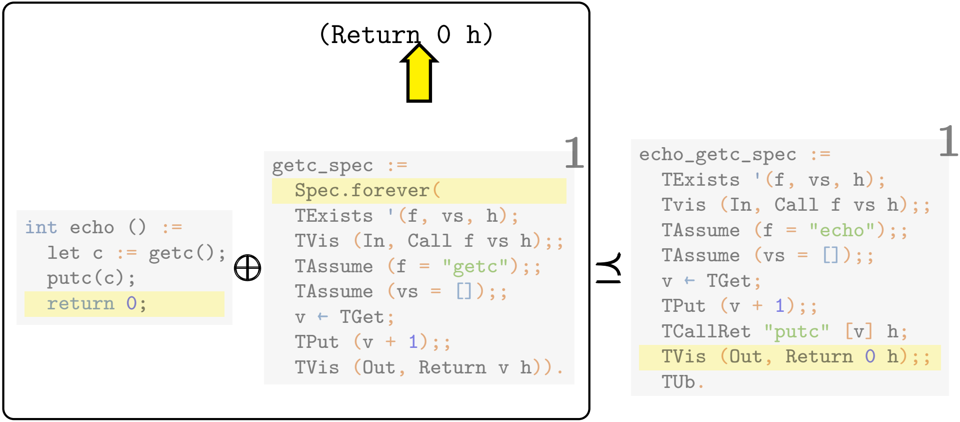
```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum



(Return 0 h)

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

≼

(Return 0 h)

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Proofs in DimSum

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

⊕

1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

⪯

1

```
echo_getc_spec :=
  TExists '(f, vs, h);
  Tvis (In, Call f vs h);;
  TAssume (f = "echo");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TCallRet "putc" [v] h;
  TVis (Out, Return 0 h);;
  TUb.
```

Towards Hoare-style Reasoning

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \quad \preceq \quad [\![\mathsf{echo}_{\mathsf{spec}}]\!]$$

Towards Hoare-style Reasoning

$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \approx\!> (\boldsymbol{\lambda} \kappa_t \, \sigma_t,$

Towards Hoare-style Reasoning

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_t \sigma_t, \llbracket \mathsf{echo}_{\mathsf{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_s \sigma_s,$$

Towards Hoare-style Reasoning

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \approx\!> (\boldsymbol{\lambda} \kappa_t \sigma_t, [\![\mathsf{echo}_{\mathsf{spec}}]\!] \approx\!> (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

Towards Hoare-style Reasoning

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_t \sigma_t, \llbracket \mathsf{echo}_{\mathsf{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> \prod_s$$

Towards Hoare-style Reasoning

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \approx> (\boldsymbol{\lambda} \kappa_t \sigma_t, [\![\mathsf{echo}_{\mathsf{spec}}]\!] \approx> (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \approx> \prod_s$$

$$[\![\mathsf{echo}_{\mathsf{rec}}]\!] \approx> \prod_{\oplus}(\Pi_s)$$

Towards Hoare-style Reasoning

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \approx\!> (\boldsymbol{\lambda} \kappa_t \sigma_t, [\![\mathsf{echo}_{\mathsf{spec}}]\!] \approx\!> (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \approx\!> \prod_s$$

$$[\![\mathsf{echo}_{\mathsf{rec}}]\!] \approx\!> (\boldsymbol{\lambda} \kappa_l \sigma_l, \texttt{if \_ then } [\![\mathsf{getc}_{\mathsf{spec}}]\!] \approx\!> \ldots \texttt{ else } \Pi_s \kappa_l \sigma)$$

$$[\![\mathsf{echo}_{\mathsf{rec}}]\!] \approx\!> \prod_{\oplus} (\Pi_s)$$

Towards Hoare-style Reasoning

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_t \sigma_t, \llbracket \mathsf{echo}_{\mathsf{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> \prod_s$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_l \sigma_l, \texttt{if} \_ \texttt{then} \; \llbracket \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> \ldots \; \texttt{else} \; \Pi_s \kappa_l \sigma)$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \rrbracket \approx\!> \prod_{\bigoplus} \big(\sigma_1, \sigma_2, \sigma_{\oplus}, \Pi_s(\sigma_{spec}) \big)$$

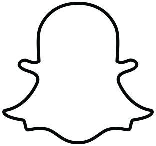
Towards Hoare-style Reasoning

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \mathrel{\approx\!>} (\boldsymbol{\lambda} \kappa_t \sigma_t, \llbracket \mathsf{echo}_{\mathsf{spec}} \rrbracket \mathrel{\approx\!>} (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \mathrel{\approx\!>} \prod_s$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \rrbracket \mathrel{\approx\!>} (\boldsymbol{\lambda} \kappa_l \sigma_l, \texttt{if}\_\texttt{then}\ \llbracket \mathsf{getc}_{\mathsf{spec}} \rrbracket \mathrel{\approx\!>} \ldots\ \texttt{else}\ \Pi_s \kappa_l \sigma)$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \rrbracket \mathrel{\approx\!>} \prod_{\bigoplus} \left(\sigma_1, \sigma_2, \sigma_{\oplus}, \Pi_s(\sigma_{spec}) \right)$$



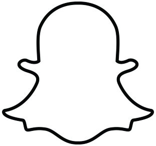
Towards Hoare-style Reasoning

$$[\![\text{echo}_{\text{rec}} \oplus \text{getc}_{\text{spec}}]\!] \approx> (\boldsymbol{\lambda} \kappa_t \sigma_t, [\![\text{echo}_{\text{spec}}]\!] \approx> (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$[\![\text{echo}_{\text{rec}} \oplus \text{getc}_{\text{spec}}]\!] \approx> \prod_s$$

$$[\![\text{echo}_{\text{rec}}]\!] \approx> (\boldsymbol{\lambda} \kappa_l \sigma_l, \texttt{if\_then}\ [\![\text{getc}_{\text{spec}}]\!] \approx> \ldots\ \texttt{else}\ \prod_s \kappa_l \sigma)$$

$$[\![\text{echo}_{\text{rec}}]\!] \approx> \prod_{\oplus}\left(\gamma_{\sigma_1}, \gamma_{\sigma_2}, \gamma_{\sigma_\oplus}, \prod_s(\gamma_{\sigma_{spec}})\right)$$



Towards Hoare-style Reasoning

$$[\![\text{echo}_{\text{rec}} \oplus \text{getc}_{\text{spec}}]\!] \approx\!> (\boldsymbol{\lambda}\kappa_t \sigma_t, [\![\text{echo}_{\text{spec}}]\!] \approx\!> (\boldsymbol{\lambda}\kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$[\![\text{echo}_{\text{rec}} \oplus \text{getc}_{\text{spec}}]\!] \approx\!> \prod_s$$

$$[\![\text{echo}_{\text{rec}}]\!] \approx\!> (\boldsymbol{\lambda}\kappa_l \sigma_l, \texttt{if \_ then } [\![\text{getc}_{\text{spec}}]\!] \approx\!> \ldots \texttt{ else } \Pi_s \kappa_l \sigma)$$

$$[\![\text{echo}_{\text{rec}}]\!] \approx\!> \prod_{\oplus} \left(\gamma_{\sigma_1}, \gamma_{\sigma_2}, \gamma_{\sigma_\oplus}, \Pi_s(\gamma_{\sigma_{spec}})\right)$$

$$[\![\text{echo}_{\text{rec}}]\!] \approx\!> \prod$$

Towards Hoare-style Reasoning

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \approx> (\boldsymbol{\lambda} \kappa_t \sigma_t, [\![\mathsf{echo}_{\mathsf{spec}}]\!] \approx> (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \approx> \prod_s$$

$$[\![\mathsf{echo}_{\mathsf{rec}}]\!] \approx> (\boldsymbol{\lambda} \kappa_l \sigma_l, \texttt{if}\,\_\,\texttt{then}\ [\![\mathsf{getc}_{\mathsf{spec}}]\!] \approx> \ldots\ \texttt{else}\ \Pi_s \kappa_l \sigma)$$

$$[\![\mathsf{echo}_{\mathsf{rec}}]\!] \approx> \prod_{\bigoplus} \left(\gamma_{\sigma_1}, \gamma_{\sigma_2}, \gamma_{\sigma_\oplus}, \Pi_s(\gamma_{\sigma_{spec}})\right)$$

$$[\![\mathsf{echo}_{\mathsf{rec}}]\!] \approx> \prod$$

$$\mathsf{TGT\ Call}\ \text{“echo”}\ \mathsf{es}\ @\ \Pi\ \{\{\Phi\}\}$$

Towards Hoare-style Reasoning

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_t \sigma_t, \llbracket \mathsf{echo}_{\mathsf{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_s \sigma_s, \kappa_t = \kappa_s * \sigma_t \preceq \sigma_s))$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> \prod_s$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \rrbracket \approx\!> (\boldsymbol{\lambda} \kappa_l \sigma_l, \texttt{if\_then}\ \llbracket \mathsf{getc}_{\mathsf{spec}} \rrbracket \approx\!> \ldots\ \texttt{else}\ \Pi_s \kappa_l \sigma)$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \rrbracket \approx\!> \prod_{\bigoplus} \left(\gamma_{\sigma_1}, \gamma_{\sigma_2}, \gamma_{\sigma_\oplus}, \Pi_s(\gamma_{\sigma_{spec}}) \right)$$

$$\llbracket \mathsf{echo}_{\mathsf{rec}} \rrbracket \approx\!> \prod$$

TGT Call “echo” es @ Π $\{\{\Phi\}\}$

$$\gamma_{\sigma_1} \rightsquigarrow \sigma_1, \gamma_{\sigma_2} \rightsquigarrow \sigma_2, \gamma_{\sigma_\oplus} \rightsquigarrow \sigma_\oplus, \gamma_{\sigma_{spec}} \rightsquigarrow \sigma_{spec}$$

Rotation Project

- Get familiar with DimSum (and Iris)
- Experiment with new reasoning style
- Proof refinements for multiple example programs

Rotation Project

- Get familiar with DimSum (and Iris)
- Experiment with new reasoning style
- Proof refinements for multiple example programs

Rotation Project

- Get familiar with DimSum (and Iris)
- Experiment with new reasoning style
- Proof refinements for multiple example programs

```
int echo () :=
  let c := getc();
  putc(c);
  return 0;
```

```
int getc (l) :=
  let _ := read(l);
  return *l;
```

```
void echo () :=
  let c := getc();
  putc(c);
  echo ();
```

```
global pos = 0;
read (l, c) :=
  if (c <= 0) {
    return 0;
  } else {
    l <- *pos;
    pos <- *pos + 1;
    ret = read(l + 1, c -1);
    return ret + 1;
  }
```

```
int echo () :=
  let c := getc();
  putc(c);
  let c := getc();
  putc(c);
  return 0;
```

(Modular) Hoare-style Reasoning

```
TGT Call "echo" es @ Π
  PRE |-∗: es POST_e, ⌜es = []⌝ ∗
    TGT Call "getc" es @ Π
      PRE |-∗: es POST, ⌜es = []⌝ ∗
      POST |∗: ret,
        TGT Call "putc" es @ Π
          PRE |-∗: es POST, ⌜es = [v]⌝ ∗
          POST |∗: _,
  POST_e |∗: ret, ⌜ret = 0⌝.
```

(Modular) Hoare-style Reasoning

```
TGT Call "getc" es @ Π
  PRE |-∗: es POST, ∃ v, P v ∗ ⌜es = []⌝ ∗
  POST |∗: ret, ⌜ret = v⌝ ∗ P (v + 1).
```

```
TGT Call "echo" es @ Π
  PRE |-∗: es POST_e, ⌜es = []⌝ ∗
    TGT Call "getc" es @ Π
      PRE |-∗: es POST, ⌜es = []⌝ ∗
      POST |∗: ret,
        TGT Call "putc" es @ Π
          PRE |-∗: es POST, ⌜es = [v]⌝ ∗
          POST |∗: _,
  POST_e |∗: ret, ⌜ret = 0⌝.
```

(Modular) Hoare-style Reasoning

```
TGT Call "getc" es @ Π
  PRE |-∗: es POST, ∃ v, P v ∗ ⌜es = []⌝ ∗
  POST |∗: ret, ⌜ret = v⌝ ∗ P (v + 1).
```

```
TGT Call "echo" es @ Π
  PRE |-∗: es POST_e, ∃ v, P v ∗ ⌜es = []⌝ ∗
    TGT Call "putc" es @ Π
      PRE |-∗: es POST, P (v + 1) ∗ ⌜es = [v]⌝ ∗
      POST |∗: _, P (v + 1) (∗)
  POST_e |∗: ret, ⌜ret = 0⌝ ∗ P (v + 1).
```

Example 1

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \preceq [\![\mathsf{echo}_{\mathsf{spec}}]\!]$$

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

```
Lemma sim_getc_spec `{!specGS} Π Φ :
  switch Π
    PRE |-∗: κ σ1 POST,
      ∃ f es h, ⌜κ = Some (Incoming, ERCall f es h)⌝ ∗
    POST Tgt _ _ |∗: σ' Π',
      ∃ v, ⌜f = "getc"⌝ ∗ ⌜es = []⌝ ∗ spec_state v ∗ ⌜σ' = σ1⌝ (∗)
  switch Π'
    PRE |-∗: κ σ POST,
      ⌜κ = Some (Outgoing, ERReturn (ValNum v) h)⌝ ∗ spec_state (v + 1) ∗
    POST Tgt _ _ |∗: σ' Π'',
      ⌜σ' = σ⌝ ∗ ⌜Π'' = Π⌝ ∗ TGT getc_spec @ Π {{ Φ }} -∗
  TGT getc_spec @ Π {{ Φ }}.
```

Example 1

$$\llbracket \mathrm{echo}_{\mathrm{rec}} \oplus \mathrm{getc}_{\mathrm{spec}} \rrbracket \approx\!> (\boldsymbol{\lambda}\kappa\sigma, \Pi\kappa\sigma)$$

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

```
Lemma sim_getc_spec `{!specGS} Π Φ :
  switch Π
    PRE |-∗: κ σ1 POST,
      ∃ f es h, ⌜κ = Some (Incoming, ERCall f es h)⌝ ∗
    POST Tgt _ _ |∗: σ' Π',
      ∃ v, ⌜f = "getc"⌝ ∗ ⌜es = []⌝ ∗ spec_state v ∗ ⌜σ' = σ1⌝ (∗)
  switch Π'
    PRE |-∗: κ σ POST,
      ⌜κ = Some (Outgoing, ERReturn (ValNum v) h)⌝ ∗ spec_state (v + 1) ∗
    POST Tgt _ _ |∗: σ' Π'',
      ⌜σ' = σ⌝ ∗ ⌜Π'' = Π⌝ ∗ TGT getc_spec @ Π {{ Φ }} -∗
  TGT getc_spec @ Π {{ Φ }}.
```

Example 1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

```
Lemma sim_getc_spec `{!specGS} Π Φ :
  switch Π
    PRE |-∗: κ σ1 POST,
      ∃ f es h, ⌜κ = Some (Incoming, ERCall f es h)⌝ ∗
    POST Tgt _ _ |∗: σ' Π',
      ∃ v, ⌜f = "getc"⌝ ∗ ⌜es = []⌝ ∗ spec_state v ∗ ⌜σ' = σ1⌝ (∗)
  switch Π'
    PRE |-∗: κ σ POST,
      ⌜κ = Some (Outgoing, ERReturn (ValNum v) h)⌝ ∗ spec_state (v + 1) ∗
    POST Tgt _ _ |∗: σ' Π'',
      ⌜σ' = σ⌝ ∗ ⌜Π'' = Π⌝ ∗ TGT getc_spec @ Π {{ Φ }} -∗
  TGT getc_spec @ Π {{ Φ }}.
```

Example 1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

```
Lemma sim_getc_spec `{!specGS} Π Φ :
  switch Π
    PRE |-∗: κ σ1 POST,
      ∃ f es h, ⌜κ = Some (Incoming, ERCall f es h)⌝ ∗
    POST Tgt _ _ |∗: σ' Π',
      ∃ v, ⌜f = "getc"⌝ ∗ ⌜es = []⌝ ∗ spec_state v ∗ ⌜σ' = σ1⌝ (∗)
  switch Π'
    PRE |-∗: κ σ POST,
      ⌜κ = Some (Outgoing, ERReturn (ValNum v) h)⌝ ∗ spec_state (v + 1) ∗
    POST Tgt _ _ |∗: σ' Π'',
      ⌜σ' = σ⌝ ∗ ⌜Π'' = Π⌝ ∗ TGT getc_spec @ Π {{ Φ }} -∗
  TGT getc_spec @ Π {{ Φ }}.
```

Example 1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

```
Lemma sim_getc_spec `{!specGS} Π Φ :
  switch Π
    PRE |-∗: κ σ1 POST,
      ∃ f es h, ⌜κ = Some (Incoming, ERCall f es h)⌝ ∗
    POST Tgt _ _ |∗: σ' Π',
      ∃ v, ⌜f = "getc"⌝ ∗ ⌜es = []⌝ ∗ spec_state v ∗ ⌜σ' = σ1⌝ (∗)
  switch Π'
    PRE |-∗: κ σ POST,
      ⌜κ = Some (Outgoing, ERReturn (ValNum v) h)⌝ ∗ spec_state (v + 1) ∗
    POST Tgt _ _ |∗: σ' Π'',
      ⌜σ' = σ⌝ ∗ ⌜Π'' = Π⌝ ∗ TGT getc_spec @ Π {{ Φ }} -∗
  TGT getc_spec @ Π {{ Φ }}.
```

Example 1

```
getc_spec :=
  Spec.forever(
  TExists '(f, vs, h);
  TVis (In, Call f vs h);;
  TAssume (f = "getc");;
  TAssume (vs = []);;
  v ← TGet;
  TPut (v + 1);;
  TVis (Out, Return v h)).
```

```
Lemma sim_getc_spec `{!specGS} Π Φ :
  switch Π
    PRE |-∗: κ σ1 POST,
      ∃ f es h, ⌜κ = Some (Incoming, ERCall f es h)⌝ ∗
    POST Tgt _ _ |∗: σ' Π',
      ∃ v, ⌜f = "getc"⌝ ∗ ⌜es = []⌝ ∗ spec_state v ∗ ⌜σ' = σ1⌝ (∗)
  switch Π'
    PRE |-∗: κ σ POST,
      ⌜κ = Some (Outgoing, ERReturn (ValNum v) h)⌝ ∗ spec_state (v + 1) ∗
    POST Tgt _ _ |∗: σ' Π'',
      ⌜σ' = σ⌝ ∗ ⌜Π'' = Π⌝ ∗ TGT getc_spec @ Π {{ Φ }} -∗
  TGT getc_spec @ Π {{ Φ }}.
```

Example 2

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \preceq [\![\mathsf{echo}_{\mathsf{spec}}]\!]$$

```
Lemma sim_getc fns Π_l Π_r PL σi :
  "getc" ↪ None -∗
  PL σi -∗
  ⌜σi.1 ≡ getc_spec⌝ -∗
  ⌜σi.2 = 0⌝ -∗
  □ switch_linked_fixed Tgt Π_l Π_r
      PRE |-∗: σ_l POST, ∃ h v σg, PL σg ∗
      POST (ERCall "getc" [] h) σg |∗: σr Π_r',
    switch_link Tgt Π_r'
      Pre |-∗: σ_r' POST, ∃ h'
      POST (ERReturn (ValNum v) h') _ σ_l |∗: _ Π_l',
        ⌜Π_l' = Π_l⌝ ∗ PL σ_r' ==∗
  ∃ P, P 0 ∗ □ rec_fn_spec_hoare Tgt Π_l "getc" (getc_fn_spec P).
```

Example 2

$$[\![\mathrm{echo}_{\mathrm{rec}} \oplus \mathrm{getc}_{\mathrm{spec}}]\!] \preceq [\![\mathrm{echo}_{\mathrm{spec}}]\!]$$

```
Lemma sim_getc fns Π_l Π_r PL σi :
  "getc" ↪ None -∗
  PL σi -∗
  ⌜σi.1 ≡ getc_spec⌝ -∗
  ⌜σi.2 = 0⌝ -∗
  □ switch_linked_fixed Tgt Π_l Π_r
      PRE |-∗: σ_l POST, ∃ h v σg, PL σg ∗
      POST (ERCall "getc" [] h) σg |∗: σr Π_r',
    switch_link Tgt Π_r'
      Pre |-∗: σ_r' POST, ∃ h'
      POST (ERReturn (ValNum v) h') _ σ_l |∗: _ Π_l',
        ⌜Π_l' = Π_l⌝ ∗ PL σ_r' ==∗
  ∃ P, P 0 ∗ □ rec_fn_spec_hoare Tgt Π_l "getc" (getc_fn_spec P).
```

Example 2

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \preceq [\![\mathsf{echo}_{\mathsf{spec}}]\!]$$

```
Lemma sim_getc fns Π_l Π_r PL σi :
  "getc" ↪ None -∗
  PL σi -∗
  ⌜σi.1 ≡ getc_spec⌝ -∗
  ⌜σi.2 = 0⌝ -∗
  □ switch_linked_fixed Tgt Π_l Π_r
      PRE |-∗: σ_l POST, ∃ h v σg, PL σg ∗
      POST (ERCall "getc" [] h) σg |∗: σr Π_r',
    switch_link Tgt Π_r'
      Pre |-∗: σ_r' POST, ∃ h'
      POST (ERReturn (ValNum v) h') _ σ_l |∗: _ Π_l',
        ⌜Π_l' = Π_l⌝ ∗ PL σ_r' ==∗
  ∃ P, P 0 ∗ □ rec_fn_spec_hoare Tgt Π_l "getc" (getc_fn_spec P).
```

Example 2

$$[\![\mathsf{echo}_{\mathsf{rec}} \oplus \mathsf{getc}_{\mathsf{spec}}]\!] \preceq [\![\mathsf{echo}_{\mathsf{spec}}]\!]$$

```
Lemma sim_getc fns Π_l Π_r PL σi :
  "getc" ↪ None -∗
  PL σi -∗
  ⌜σi.1 ≡ getc_spec⌝ -∗
  ⌜σi.2 = 0⌝ -∗
  □ switch_linked_fixed Tgt Π_l Π_r
      PRE |-∗: σ_l POST, ∃ h v σg, PL σg ∗
      POST (ERCall "getc" [] h) σg |∗: σr Π_r',
    switch_link Tgt Π_r'
      Pre |-∗: σ_r' POST, ∃ h'
      POST (ERReturn (ValNum v) h') _ σ_l |∗: _ Π_l',
        ⌜Π_l' = Π_l⌝ ∗ PL σ_r' ==∗
  ∃ P, P 0 ∗ □ rec_fn_spec_hoare Tgt Π_l "getc" (getc_fn_spec P).
```

Example 2

```
Lemma sim_getc fns Π_l Π_r PL σi :
  "getc" ↪ None -∗
  PL σi -∗
  ⌜σi.1 ≡ getc_spec⌝ -∗
  ⌜σi.2 = 0⌝ -∗
  □ switch Π_l
      PRE |-∗: κ σ0 POST, ∃ h v σg, PL σg ∗
      POST Tgt _ _ |∗:  σi0 Πi, ⌜σi0 = σg⌝ ∗ ⌜Πi = Π_r⌝ ∗
    switch Πi
      PRE |-∗: κ' σ POST0, ∃ e' : rec_ev, ⌜κ' = Some (Incoming, e')⌝ ∗
      POST0 Tgt _ _ |∗: σr Πr, ⌜σr = σ⌝ ∗ ⌜e' = ERCall "getc" [] h⌝ ∗
    switch Πr
      PRE |-∗: κ0 σ1 POST1, ∃ h', ⌜κ0 = Some (Outgoing, ERReturn v h')⌝ ∗
      POST1 Tgt _ _ |∗: σi1 Πi0, ⌜σi1 = σ0⌝ ∗
    switch Πi0
      PRE |-∗: κ'0 σ2 POST2, ∃ e'0, ⌜κ'0 = Some (Incoming, e'0)⌝ ∗
      POST2 Tgt _ _ |∗: σr0 Πr0,
        ⌜σr0 = σ2⌝ ∗ ⌜e'0 = ERReturn v h'⌝ ∗ ⌜Πr0 = Π_l⌝ ∗ PL σ1 ==∗
  ∃ P, P 0 ∗ □ rec_fn_spec_hoare Tgt Π_l "getc" (getc_fn_spec P).
```

Outcome & Takeaways

- Lemma for `TCallRet`
- Keep Πs the same - new lemmas for linking
- Balance between Abstraction and Information
- Balance between Hacking and Thinking

ToC

1. Background

2. DimSum Proof

3. Hoare Style

4. Examples