Polymorphic Types with Polynomial Sizes

Size tracking without dependent types

Jean-Louis Colaço, **Baptiste Pauget**, Marc Pouzet

ARRAY 2023, Orlando, Florida, United States

June 18, 2023

Context: Safe arrays in Scade

Context: Safe arrays in Scade

```
-- Generic scalar product [valid only if n=p]
function dot «n,p» (u: 'T^n, v: 'T^p)
returns (w:'T) where 'T numeric
  w = (fold $+$ «n») (0, (map $*$ «p») (u, v));

-- Monomorphic instantiation
function main (u: int32^6) return (s: int32)
  s = (dot «6,5») (u, 1^5);
```

Context: Safe arrays in SCADE

```
-- Generic scalar product [valid only if n=p]
function dot «n,p» (u: 'T^n, v: 'T^p)
returns (w:'T) where 'T numeric
  w = (fold $+$ «n») (0, (map $*$ «p») (u, v));

-- Monomorphic instantiation
function main (u: int32^6) return (s: int32)
  s = (dot «6,5») (u, 1^5);
```

Current limitations

- Redundant size annotations

Context: Safe arrays in SCADE

```
-- Generic scalar product [valid only if n=p]
function dot «n,p» (u: 'T^n, v: 'T^p)
returns (w:'T) where 'T numeric
  w = (fold $+$ «n») (0, (map $*$ «p») (u, v));
```

✔

```
-- Monomorphic instantiation
function main (u: int32^6) return (s: int32)
  s = (dot «6,5») (u, 1^5);
```

✘

Current limitations

- Redundant size annotations
- Late and non-modular size error detection

Context: Safe arrays in Scade

```
-- Generic scalar product [valid only if n=p]
function dot «n,p» (u: 'T^n, v: 'T^p)
returns (w:'T) where 'T numeric
  w = (fold $+$ «n») (0, (map $*$ «p») (u, v));
```

✔

```
-- Monomorphic instantiation
function main (u: int32^6) return (s: int32)
  s = (dot «6,5») (u, 1^5);
```

✘

Current limitations

- Redundant size annotations
- Late and non-modular size error detection
- Limited expressiveness (`concat`, `transpose`, `reverse`)

Context: Safe arrays in SCADE

```
-- Generic scalar product [valid only if n=p]
function dot «n,p» (u: 'T^n, v: 'T^p)
returns (w:'T) where 'T numeric
  w = (fold $+$ «n») (0, (map $*$ «p») (u, v));

-- Monomorphic instantiation
function main (u: int32^6) return (s: int32)
  s = (dot «6,5») (u, 1^5);
```

✔ (first example) ✘ (second example)

Current limitations

- Redundant size annotations
- Late and non-modular size error detection
- Limited expressiveness (`concat`, `transpose`, `reverse`)

Infer & check sizes together with types

Agenda

1. A sized type system

2. Inference

3. Perspectives and conclusion

Agenda

1. A sized type system

2. Inference

3. Perspectives and conclusion

Type safety (ML)

'***Well-typed programs cannot go wrong***'

R. Milner[1]

[1] Robin Milner. "A Theory of Type Polymorphism in Programming". (1978)

Type safety (ML)

‘***Well-typed programs cannot go wrong***’ *

R. Milner[1]

* unless partial operations exist (division, ...)

[1] Robin Milner. “A Theory of Type Polymorphism in Programming”. (1978)

Type safety (ML)

'***Well-typed programs cannot go wrong***' ⋆

R. Milner[1]

⋆ unless partial operations exist (division, ...)

Array access: indexes must respect bounds

[1] Robin Milner. "A Theory of Type Polymorphism in Programming". (1978)

Type safety (ML)

'***Well-typed programs cannot go wrong***' ⋆

R. Milner[1]

⋆ unless partial operations exist (division, ...)

Array access: indexes must respect bounds

Array combinators[2]: predefined valid access schemes

- First orders (FOACs): `reverse`, `transpose`, ...
- Second orders (SOACs): `map`, `fold`, ...

[1] Robin Milner. "A Theory of Type Polymorphism in Programming". (1978)
[2] Troels Henriksen et al. "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". (2017)

Type safety (ML)

‘***Well-typed programs cannot go wrong***’ ⋆

R. Milner[1]

⋆ unless partial operations exist (division, ...)

Array access: indexes must respect bounds

Array combinators[2]: predefined valid access schemes ⋆⋆

- First orders (FOACs): `reverse`, `transpose`, ...
- Second orders (SOACs): `map`, `fold`, ...

⋆⋆ under *size consistency* assumptions (`map2`, ...)

[1] Robin Milner. “A Theory of Type Polymorphism in Programming”. (1978)
[2] Troels Henriksen et al. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”. (2017)

Extensions of the ML type system

Type families		Refinement types
$\tau\ i_1\ \ldots\ i_k$	**Description**	$\{\ x : \tau \mid P(x)\ \}$
Polymorphism	**Genericity**	Dependent types
✗	**Sub-typing**	✓
✓	**Type checking decidability**	?
✓	**Type erasable semantics**	✗

Dimension types (1994) [5]
Sized types (1996) [3]
Indexed Types (1997) [11]

DML (1999) [10] λ^H (2006) [1]
Liquid Types (2008) [9]
Futhark (2017) [2] Dex (2021) [8]

Extensions of the ML type system

Type families		Refinement types
$\tau \; i_1 \; \ldots \; i_k$	**Description**	$\{ \; x : \tau \mid P(x) \; \}$
Polymorphism	**Genericity**	Dependent types
✗	**Sub-typing**	✓
✓	**Type checking decidability**	?
✓	**Type erasable semantics**	✗

Dimension types (1994) [5]
Sized types (1996) [3]
Indexed Types (1997) [11]

DML (1999) [10] λ^H (2006) [1]
LIQUID TYPES (2008) [9]
FUTHARK (2017) [2] DEX (2021) [8]

ML

$$e \ ::= \ x \mid n \mid e\ e \mid \lambda x.\ e \mid \texttt{let}\ x \texttt{=} e\ \texttt{in}\ e$$ ***Expressions***

ML + Types

$\tau \ ::= \ \alpha \mid \texttt{int} \mid \tau \to \tau$ ***Types***

$e \ ::= \ x \mid n \mid e\ e \mid \lambda x : \tau .\ e \mid \texttt{let}\ x : \tau \texttt{=} e\ \texttt{in}\ e \mid e \triangleright \tau$ ***Expressions***

ML + Types + Polymorphism

$\tau ::= \alpha \mid \texttt{int} \mid \tau \to \tau$ ***Types***

$e ::= {}^{\boldsymbol{S}}x \mid n \mid e\ e \mid \lambda x{:}\tau.\ e \mid \texttt{let}_{\boldsymbol{V}}\ x{:}\tau \,\texttt{=}\, e\ \texttt{in}\ e \mid e \triangleright \tau$ ***Expressions***

$\boldsymbol{V} ::= \varepsilon \mid \alpha \cdot \boldsymbol{V}$ ***Generalization***

$\boldsymbol{S} ::= \varepsilon \mid \tau \cdot \boldsymbol{S}$ ***Instantiation***

ML + Types + Polymorphism + Sizes

$\eta \ ::= \ \iota \mid n \mid \eta + \eta \mid \eta * \eta$ ***Sizes***

$\tau \ ::= \ \alpha \mid \texttt{int} \mid \tau \to \tau \mid \texttt{<}\eta\texttt{>} \mid [\eta]$ ***Types***

$e \ ::= \ {}^{S}x \mid n \mid e\ e \mid \lambda x{:}\tau.\ e \mid \texttt{let}_{V}\ x{:}\tau = e\ \texttt{in}\ e \mid e \rhd \tau \mid \texttt{<}\eta\texttt{>}$ ***Expressions***

$V \ ::= \ \varepsilon \mid \alpha \cdot V \mid \iota \cdot V$ ***Generalization***

$S \ ::= \ \varepsilon \mid \tau \cdot S \mid \eta \cdot S$ ***Instantiation***

Integer refinements

- $\texttt{<}\eta\texttt{>}$: singleton type $\{\ x : \texttt{int} \mid x = \eta\ \}$ (*size* η)
- $[\eta]$: interval type $\{\ x : \texttt{int} \mid 0 \leq x < \eta\ \}$ (*index* η)

ML + Types + Polymorphism + Sizes

η ::= ι … **Sizes**

τ ::= α … **Types**

e ::= ***s*** … ***…ressions***

V ::= ε … ***…alization***

S ::= ε … ***…ntiation***

Where are arrays?

Integer…

- $\mathtt{<}\eta\mathtt{>}$: singleton type $\{ x : \mathtt{int} \mid x = \eta \}$ (*size* η)
- $\mathtt{[}\eta\mathtt{]}$: interval type $\{ x : \mathtt{int} \mid 0 \leq x < \eta \}$ (*index* η)

ML + Types + Polymorphism + Sizes

η ::= ι **_Sizes_**

τ ::= α **_Types_**

e ::= s **_ressions_**

V ::= ε **_alization_**

S ::= ε **_ntiation_**

Where are arrays?

For typing purposes, arrays are functions on bounded domains:

$$[\eta]\tau \equiv [\eta] \to \tau$$

Array access: application

Array definition: abstraction

Integer

- $<\eta>$: singleton type $\{ x : \texttt{int} \mid x = \eta \}$ (*size* η)
- $[\eta]$: interval type $\{ x : \texttt{int} \mid 0 \leq x < \eta \}$ (*index* η)

ML + Types + Polymorphism + Sizes

η ::= ι … ***Sizes***

τ ::= α … ***Types***

e ::= *s* … ***…ressions***

V ::= ε … ***…alization***

S ::= ε … ***…ntiation***

Where are arrays?

For typing purposes, arrays are functions on bounded domains:

$$[\eta]\tau \equiv [\eta] \to \tau$$

Array access: application

Array definition: abstraction

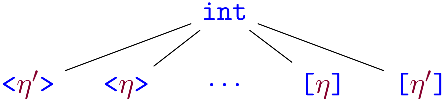
$$\texttt{let length}_{\iota\cdot\alpha} : [\iota]\alpha \to \texttt{<}\iota\texttt{>} \;=\; \lambda x : [\iota]\alpha.\ \texttt{<}\iota\texttt{>}$$

Integer …

- $[\eta]$: interval type $\{\ x : \texttt{int} \mid 0 \le x < \eta\ \}$ (*index* η)

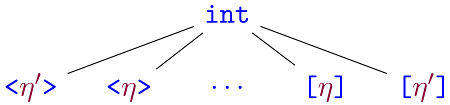
A flat hierarchy of types

Rudimentary sub-typing. No comparison between refinements



A flat hierarchy of types

Rudimentary sub-typing. No comparison between refinements

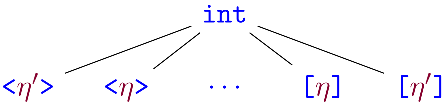


In particular

$$[\eta] \not<: [\eta + 1]$$

A flat hierarchy of types

Rudimentary sub-typing. No comparison between refinements



In particular

$$[\eta] \not<: [\eta + 1]$$

No size inequalities

- Checking is decidable and simple (normalized polynomials)
- Inference is practical (although incomplete)

Second Order Array Combinators

$$\texttt{val map} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta$$

$$\texttt{val fold} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\beta \to \alpha$$

$$\texttt{val map2} : \forall \iota \cdot \alpha \cdot \beta \cdot \gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta \to \texttt{[}\iota\texttt{]}\gamma$$

Second Order Array Combinators

$$\texttt{val map} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta$$

$$\texttt{val fold} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\beta \to \alpha$$

$$\texttt{val map2} : \forall \iota \cdot \alpha \cdot \beta \cdot \gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta \to \texttt{[}\iota\texttt{]}\gamma$$

Second Order Array Combinators

$\texttt{val map} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \rightarrow \beta) \rightarrow \texttt{<}\iota\texttt{>} \rightarrow \texttt{[}\iota\texttt{]}\alpha \rightarrow \texttt{[}\iota\texttt{]}\beta$

$\texttt{val fold} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \texttt{<}\iota\texttt{>} \rightarrow \alpha \rightarrow \texttt{[}\iota\texttt{]}\beta \rightarrow \alpha$

$\texttt{val map2} : \forall \iota \cdot \alpha \cdot \beta \cdot \gamma.\ (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \texttt{<}\iota\texttt{>} \rightarrow \texttt{[}\iota\texttt{]}\alpha \rightarrow \texttt{[}\iota\texttt{]}\beta \rightarrow \texttt{[}\iota\texttt{]}\gamma$

$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc}\ \texttt{<6>}\ x$

Second Order Array Combinators

$$\texttt{val map} : \forall\iota\cdot\alpha\cdot\beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta$$
$$\texttt{val fold} : \forall\iota\cdot\alpha\cdot\beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\beta \to \alpha$$
$$\texttt{val map2} : \forall\iota\cdot\alpha\cdot\beta\cdot\gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta \to \texttt{[}\iota\texttt{]}\gamma$$

$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <6>}\ x \qquad [\ \texttt{[6]int} \to \texttt{[6]int}\ ]$$

Second Order Array Combinators

$\texttt{val map} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta$

$\texttt{val fold} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\beta \to \alpha$

$\texttt{val map2} : \forall \iota \cdot \alpha \cdot \beta \cdot \gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta \to \texttt{[}\iota\texttt{]}\gamma$

$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <6>}\ x$ $\quad [\ \texttt{[6]int} \to \texttt{[6]int}\ ]$

$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <\_>}\ x$

Second Order Array Combinators

$$\texttt{val map} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta$$
$$\texttt{val fold} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\beta \to \alpha$$
$$\texttt{val map2} : \forall \iota \cdot \alpha \cdot \beta \cdot \gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta \to \texttt{[}\iota\texttt{]}\gamma$$

$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <6>}\ x \qquad [\ \texttt{[6]int} \to \texttt{[6]int}\ ]$$
$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <\_>}\ x \qquad [\ \forall \iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int}\ ]$$

Second Order Array Combinators

$$\texttt{val map} : \forall\iota\cdot\alpha\cdot\beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta$$
$$\texttt{val fold} : \forall\iota\cdot\alpha\cdot\beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\beta \to \alpha$$
$$\texttt{val map2} : \forall\iota\cdot\alpha\cdot\beta\cdot\gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta \to \texttt{[}\iota\texttt{]}\gamma$$

$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <6>}\ x \qquad [\ \texttt{[6]int} \to \texttt{[6]int}\ ]$$
$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <\_>}\ x \qquad [\ \forall\iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int}\ ]$$

$$\texttt{let dot} = \lambda u.\ \lambda v.$$
$$\quad \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\texttt{map2}\ (*)\ \texttt{<\_>}\ u\ v)$$

Second Order Array Combinators

$$\texttt{val map} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta$$

$$\texttt{val fold} : \forall \iota \cdot \alpha \cdot \beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\beta \to \alpha$$

$$\texttt{val map2} : \forall \iota \cdot \alpha \cdot \beta \cdot \gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\alpha \to \texttt{[}\iota\texttt{]}\beta \to \texttt{[}\iota\texttt{]}\gamma$$

$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <6>}\ x \qquad [\ \texttt{[6]int} \to \texttt{[6]int}\ ]$$

$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <\_>}\ x \qquad [\ \forall \iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int}\ ]$$

$$\texttt{let dot} = \lambda u.\ \lambda v.$$

$$\quad \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\texttt{map2}\ (*)\ \texttt{<\_>}\ u\ v) \qquad [\ \forall \iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int} \to \texttt{int}\ ]$$

Second Order Array Combinators

$$\texttt{val map} : \forall\iota\cdot\alpha\cdot\beta.\ (\alpha\to\beta)\to\texttt{<}\iota\texttt{>}\to\texttt{[}\iota\texttt{]}\alpha\to\texttt{[}\iota\texttt{]}\beta$$

$$\texttt{val fold} : \forall\iota\cdot\alpha\cdot\beta.\ (\alpha\to\beta\to\alpha)\to\texttt{<}\iota\texttt{>}\to\alpha\to\texttt{[}\iota\texttt{]}\beta\to\alpha$$

$$\texttt{val map2} : \forall\iota\cdot\alpha\cdot\beta\cdot\gamma.\ (\alpha\to\beta\to\gamma)\to\texttt{<}\iota\texttt{>}\to\texttt{[}\iota\texttt{]}\alpha\to\texttt{[}\iota\texttt{]}\beta\to\texttt{[}\iota\texttt{]}\gamma$$

$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <6>}\ x \qquad [\ \texttt{[6]int}\to\texttt{[6]int}\ ]$$

$$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <\_>}\ x \qquad [\ \forall\iota.\ \texttt{[}\iota\texttt{]int}\to\texttt{[}\iota\texttt{]int}\ ]$$

$$\texttt{let dot} = \lambda u.\ \lambda v.\ \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\texttt{map2}\ (*)\ \texttt{<\_>}\ u\ v) \qquad [\ \forall\iota.\ \texttt{[}\iota\texttt{]int}\to\texttt{[}\iota\texttt{]int}\to\texttt{int}\ ]$$

$$\texttt{let dot} = \lambda n.\ \lambda u.\ \lambda v.\ \texttt{fold}\ (+)\ n\ 0\ (\texttt{map2}\ (*)\ n\ u\ v)$$

Second Order Array Combinators

$\texttt{val map} : \forall\iota\cdot\alpha\cdot\beta.\ (\alpha\to\beta)\to\texttt{<}\iota\texttt{>}\to\texttt{[}\iota\texttt{]}\alpha\to\texttt{[}\iota\texttt{]}\beta$

$\texttt{val fold} : \forall\iota\cdot\alpha\cdot\beta.\ (\alpha\to\beta\to\alpha)\to\texttt{<}\iota\texttt{>}\to\alpha\to\texttt{[}\iota\texttt{]}\beta\to\alpha$

$\texttt{val map2} : \forall\iota\cdot\alpha\cdot\beta\cdot\gamma.\ (\alpha\to\beta\to\gamma)\to\texttt{<}\iota\texttt{>}\to\texttt{[}\iota\texttt{]}\alpha\to\texttt{[}\iota\texttt{]}\beta\to\texttt{[}\iota\texttt{]}\gamma$

$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <6>}\ x$ $\qquad [\ \texttt{[6]int}\to\texttt{[6]int}\ ]$

$\texttt{let inc\_array} = \lambda x.\ \texttt{map inc <\_>}\ x$ $\qquad [\ \forall\iota.\ \texttt{[}\iota\texttt{]int}\to\texttt{[}\iota\texttt{]int}\ ]$

$\texttt{let dot} = \lambda u.\ \lambda v.$
$\quad \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\texttt{map2}\ (*)\ \texttt{<\_>}\ u\ v)$ $\qquad [\ \forall\iota.\ \texttt{[}\iota\texttt{]int}\to\texttt{[}\iota\texttt{]int}\to\texttt{int}\ ]$

$\texttt{let dot} = \lambda n.\ \lambda u.\ \lambda v.$
$\quad \texttt{fold}\ (+)\ n\ 0\ (\texttt{map2}\ (*)\ n\ u\ v)$ $\qquad [\ \forall\iota.\ \texttt{<}\iota\texttt{>}\to\texttt{[}\iota\texttt{]int}\to\texttt{[}\iota\texttt{]int}\to\texttt{int}\ ]$

Type erasable semantics?

$\texttt{let repeat} = \lambda n{:}\texttt{<}\iota\texttt{>}.\ \lambda a.\ (\lambda i{:}\texttt{[}\iota\texttt{]}.\ a)$

Type erasable semantics?

$\texttt{let repeat} = \lambda n{:}\texttt{<}\iota\texttt{>}.\ \lambda a.\ (\lambda i{:}\texttt{[}\iota\texttt{]}.\ a)$ $\qquad [\ \forall \iota \cdot \alpha.\ \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\alpha\ ]$

Type erasable semantics?

```
let repeat = λn:<ι>. λa. (λi:[ι]. a)        [ ∀ι·α. <ι> → α → [ι]α ]

let inc_array = λa.
    map2 (+) <_> a (repeat <_> 1)
```

Type erasable semantics?

$\texttt{let repeat} = \lambda n : \texttt{<}\iota\texttt{>}.\ \lambda a.\ (\lambda i : \texttt{[}\iota\texttt{]}.\ a)$ $\quad [\ \forall \iota \cdot \alpha.\ \texttt{<}\iota\texttt{>} \rightarrow \alpha \rightarrow \texttt{[}\iota\texttt{]}\alpha\ ]$

$\texttt{let inc\_array} = \lambda a.$
$\quad \texttt{map2}\ (+)\ \texttt{<\_>}\ a\ (\texttt{repeat <\_>}\ 1)$ $\quad [\ \forall \iota.\ \texttt{[}\iota\texttt{]int} \rightarrow \texttt{[}\iota\texttt{]int}\ ]$

Type erasable semantics?

$\texttt{let repeat} = \lambda n{:}\texttt{<}\iota\texttt{>}.\ \lambda a.\ (\lambda i{:}\texttt{[}\iota\texttt{]}.\ a)$ $[\ \forall \iota \cdot \alpha.\ \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\alpha\ ]$

$\texttt{let inc\_array} = \lambda a.$
$\quad \texttt{map2}\ (+)\ \texttt{<\_>}\ a\ (\texttt{repeat}\ \texttt{<\_>}\ 1)$ $[\ \forall \iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int}\ ]$

$\texttt{let even} =$
$\quad \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\texttt{repeat}\ \texttt{<\_>}\ 2)$

Type erasable semantics?

$\texttt{let repeat} = \lambda n{:}\texttt{<}\iota\texttt{>}.\ \lambda a.\ (\lambda i{:}\texttt{[}\iota\texttt{]}.\ a)$ $\quad [\ \forall \iota \cdot \alpha.\ \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\alpha\ ]$

$\texttt{let inc\_array} = \lambda a.$
$\quad \texttt{map2}\ (+)\ \texttt{<\_>}\ a\ (\texttt{repeat}\ \texttt{<\_>}\ 1)$ $\quad [\ \forall \iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int}\ ]$

$\texttt{let even} =$
$\quad \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\texttt{repeat}\ \texttt{<\_>}\ 2)$ $\quad$ (Error: *Unconstrained size*)

Type erasable semantics?

$\texttt{let repeat} = \lambda n : \texttt{<}\iota\texttt{>}.\ \lambda a.\ (\lambda i : \texttt{[}\iota\texttt{]}.\ a)$ $\qquad [\ \forall \iota \cdot \alpha.\ \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\alpha\ ]$

$\texttt{let inc\_array} = \lambda a.$
$\quad \texttt{map2}\ (+)\ \texttt{<\_>}\ a\ (\texttt{repeat}\ \texttt{<\_>}\ 1)$ $\qquad [\ \forall \iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int}\ ]$

$\texttt{let even} =$
$\quad \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\texttt{repeat}\ \texttt{<\_>}\ 2)$ $\qquad$ (Error: *Unconstrained size*)

Remark 1. Non type-erasable semantics

- Refinements[3] $\qquad (\lambda x : \texttt{[}\eta\texttt{]}.\ e)\ 8 \leadsto e\{8 \triangleright \texttt{[}\eta\texttt{]} / x\}$ $\qquad \Rightarrow$ stops reduction
- Size expressions $\qquad \texttt{<}\eta\texttt{>} \leadsto [\![\eta]\!]_\rho$ $\qquad \Rightarrow$ defines result

[3] Cormac Flanagan. "Hybrid type checking". (2006)

Type erasable semantics?

$\mathtt{let}\ \mathtt{repeat}\ =\ \lambda n{:}\mathtt{<}\iota\mathtt{>}.\ \lambda a.\ (\lambda i : \mathtt{[}\iota\mathtt{]}.\ a)$ $\qquad$ $[\ \forall \iota \cdot \alpha.\ \mathtt{<}\iota\mathtt{>} \rightarrow \alpha \rightarrow \mathtt{[}\iota\mathtt{]}\alpha\ ]$

$\mathtt{let}\ \mathtt{inc\_array}\ =\ \lambda a.$
$\quad \mathtt{map2}\ (+)\ \mathtt{<\_>}\ a\ (\mathtt{repeat}\ \mathtt{<\_>}\ 1)$ $\qquad$ $[\ \forall \iota.\ \mathtt{[}\iota\mathtt{]int} \rightarrow \mathtt{[}\iota\mathtt{]int}\ ]$

$\mathtt{let}\ \mathtt{even}\ =$
$\quad \mathtt{fold}\ (+)\ \mathtt{<\_>}\ 0\ (\mathtt{repeat}\ \mathtt{<\_>}\ 2)$ $\qquad$ (Error: *Unconstrained size*)

Remark 1. Non type-erasable semantics

- Refinements[3] $\quad (\lambda x : \mathtt{[}\eta\mathtt{]}.\ e)\ 8 \ \rightsquigarrow\ e\{8 \triangleright \mathtt{[}\eta\mathtt{]}\ / x\}$ $\quad \Rightarrow$ stops reduction
- Size expressions $\quad \mathtt{<}\eta\mathtt{>} \ \rightsquigarrow\ [\![\eta]\!]_\rho$ $\quad \Rightarrow$ defines result

Observational semantics does not depend on type variables

[3] Cormac Flanagan. "Hybrid type checking". (2006)

First Order Array Combinators

Simple combinators (available in SCADE)

Remark 1. Non type-erasable semantics

- Refinements[3] $(\lambda x : [\eta].\ e)\ 8 \rightsquigarrow e\{8 \triangleright [\eta] / x\}$ $\Rightarrow$ stops reduction
- Size expressions $<\eta> \rightsquigarrow [\![\eta]\!]_\rho$ $\Rightarrow$ defines result

Non type-erasable semantics

First Order Array Combinators

Simple combinators (available in Scade)

$\texttt{val reverse} : \forall \iota \cdot \alpha.\ \texttt{[}\iota\texttt{]}\alpha \rightarrow \texttt{[}\iota\texttt{]}\alpha$

Non type-erasable semantics

First Order Array Combinators

Simple combinators (available in SCADE)

$\texttt{val reverse} : \forall \iota \cdot \alpha.\ [\iota]\alpha \to [\iota]\alpha$

$\texttt{val transpose} : \forall \iota \cdot \kappa \cdot \alpha.\ [\iota][\kappa]\alpha \to [\kappa][\iota]\alpha$

First Order Array Combinators

Non type-erasable semantics

Simple combinators (available in Scade)

$\texttt{val reverse} : \forall \iota \cdot \alpha.\ [\iota]\alpha \rightarrow [\iota]\alpha$

$\texttt{val transpose} : \forall \iota \cdot \kappa \cdot \alpha.\ [\iota][\kappa]\alpha \rightarrow [\kappa][\iota]\alpha$

$\texttt{val concat} : \forall \iota \cdot \kappa \cdot \alpha.\ [\iota]\alpha \rightarrow [\kappa]\alpha \rightarrow [\iota + \kappa]\alpha$

Non type-erasable semantics

First Order Array Combinators

Simple combinators (available in SCADE)

$\texttt{val reverse} : \forall \iota \cdot \alpha.\ [\iota]\alpha \to [\iota]\alpha$

$\texttt{val transpose} : \forall \iota \cdot \kappa \cdot \alpha.\ [\iota][\kappa]\alpha \to [\kappa][\iota]\alpha$

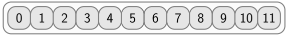
$\texttt{val concat} : \forall \iota \cdot \kappa \cdot \alpha.\ [\iota]\alpha \to [\kappa]\alpha \to [\iota + \kappa]\alpha$

Equivalent type schemes

$\texttt{val concat} : \forall \iota \cdot \kappa \cdot \alpha.\ [\iota]\alpha \to [\kappa - \iota]\alpha \to [\kappa]\alpha$
$\texttt{val concat} : \forall \iota \cdot \kappa \cdot \alpha.\ [\iota - \kappa]\alpha \to [\kappa]\alpha \to [\iota]\alpha$

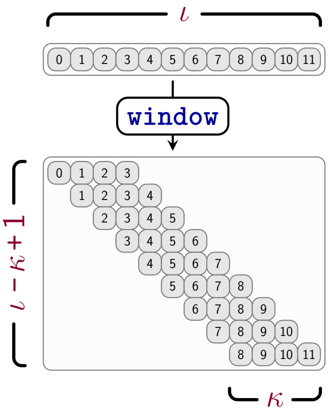
First Order Array Combinators

Non type-erasable semantics



Non type-erasable semantics

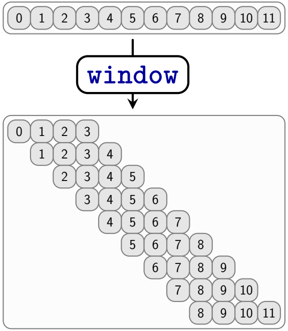
First Order Array Combinators



`val window` : $\forall \iota \cdot \kappa \cdot \alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota]\alpha \to [\iota - \kappa + 1][\kappa]\alpha$

Non type-erasable semantics

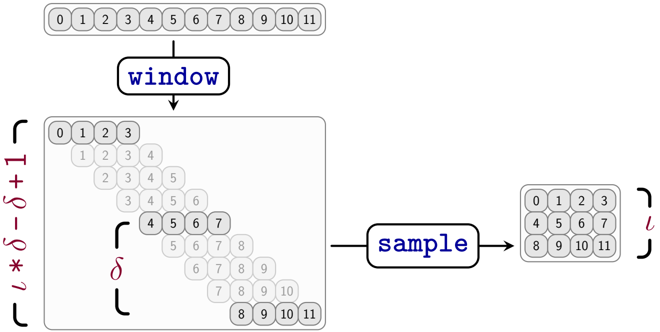
First Order Array Combinators




```
let convolution = λk. λi.
    map (dot <_> k) <_> (window <_> i)      [ ∀ι·κ. [κ]int → [ι + κ − 1]int → [ι]int ]
```

First Order Array Combinators

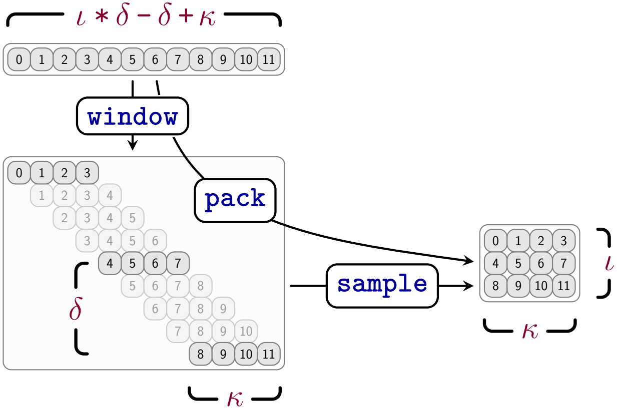
Non type-erasable semantics



`val sample` : $\forall\iota\cdot\delta\cdot\alpha.\ \langle\delta\rangle \to [\iota * \delta - \delta + 1]\alpha \to [\iota]\alpha$

Non type-erasable semantics

First Order Array Combinators

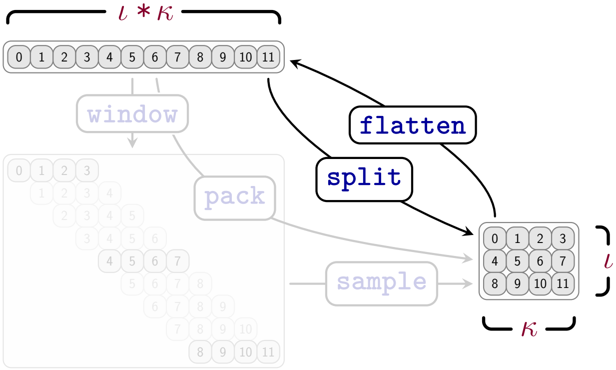



```
let pack = λs. λx.
    sample s (window <_> x)
```

$[\ \forall \iota \cdot \kappa \cdot \delta \cdot \alpha.\ \langle\delta\rangle \rightarrow [\iota * \delta - \delta + \kappa]\alpha \rightarrow [\iota]\,[\kappa]\alpha\ ]$

Non type-erasable semantics

First Order Array Combinators

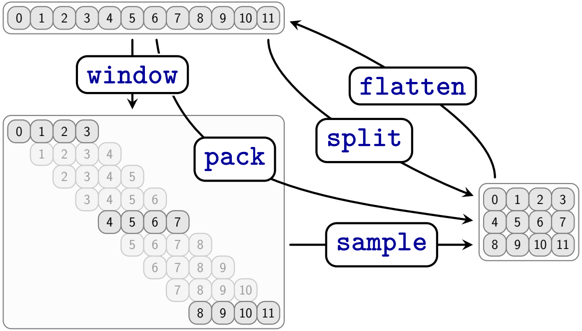


val split : $\forall \iota \cdot \kappa \cdot \alpha.\ \langle\kappa\rangle \to [\iota * \kappa]\alpha \to [\iota][\kappa]\alpha$

val flatten : $\forall \iota \cdot \kappa \cdot \alpha.\ \langle\kappa\rangle \to [\iota][\kappa]\alpha \to [\iota * \kappa]\alpha$

Non type-erasable semantics

First Order Array Combinators



Principal types?

```
let mat : [_][_]int =
    split <_> (λi : [4]. 0)
```

Non type-erasable semantics

Principal types?

```
let mat : [_][_]int =
    split <_> (λi : [4]. 0)
```

$$\left\{ \begin{array}{l} \texttt{[1][4]int} \\ \texttt{[2][2]int} \\ \texttt{[4][1]int} \end{array} \right.$$

Principal types?

Non type-erasable semantics

```
let mat : [_][_]int =
    split <_> (λi : [4]. 0)
```

$\left\{\begin{array}{l}\texttt{[1][4]int}\\ \texttt{[2][2]int}\\ \texttt{[4][1]int}\end{array}\right.$

```
let slope = λf. λi : [_]. λj : [_].
    (f i - f j) / (i - j)
```

Non type-erasable semantics

Principal types?

```
let mat : [_][_]int =
    split <_> (λi : [4]. 0)
```

$$\left\{ \begin{array}{l} [1]\,[4]\,\texttt{int} \\ [2]\,[2]\,\texttt{int} \\ [4]\,[1]\,\texttt{int} \end{array} \right.$$

```
let slope = λf. λi : [_]. λj : [_].
    (f i - f j) / (i - j)
```

$$\left\{ \begin{array}{r} \forall \iota.\ ([\iota] \to \texttt{int}) \to [\iota] \to [\iota] \to \texttt{int} \\ \forall \iota \cdot \kappa.\ (\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int} \end{array} \right.$$

Non type-erasable semantics

Principal types?

```
let mat : [_] [_] int =
    split <_> (λi : [4]. 0)
```

$$\left\{\begin{array}{l}\texttt{[1] [4] int}\\ \texttt{[2] [2] int}\\ \texttt{[4] [1] int}\end{array}\right.$$

```
let slope = λf. λi : [_]. λj : [_].
    (f i - f j) / (i - j)
```

$$\left\{\begin{array}{r}\forall\iota.\ (\texttt{[}\iota\texttt{]} \to \texttt{int}) \to \texttt{[}\iota\texttt{]} \to \texttt{[}\iota\texttt{]} \to \texttt{int}\\ \forall\iota\cdot\kappa.\ (\texttt{int} \to \texttt{int}) \to \texttt{[}\iota\texttt{]} \to \texttt{[}\kappa\texttt{]} \to \texttt{int}\end{array}\right.$$

Remark 2. No principal types

- Polynomial sizes constraints — $\iota * \kappa - 4 = 0$
- Simple polymorphism — Extra refinements add size constraints

Non type-erasable semantics

Principal types?

$$\texttt{let mat} : [\_]\,[\_]\,\texttt{int} = \texttt{split} \mathrel{<\_>} (\lambda i : [4].\ 0)$$

$$\begin{cases} [1]\,[4]\,\texttt{int} \\ [2]\,[2]\,\texttt{int} \\ [4]\,[1]\,\texttt{int} \end{cases}$$

$$\texttt{let slope} = \lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i - f\ j)\ /\ (i - j)$$

$$\begin{cases} \forall \iota.\ ([\iota] \to \texttt{int}) \to [\iota] \to [\iota] \to \texttt{int} \\ \forall \iota \cdot \kappa.\ (\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int} \end{cases}$$

Remark 2. No principal types

- Polynomial sizes constraints — $\iota * \kappa - 4 = 0$
- Simple polymorphism — Extra refinements add size constraints

Constrained polymorphism[4] $\Rightarrow$ delayed size resolution

$$\forall \iota \cdot \kappa \cdot \alpha \mid [\iota], [\kappa] <: \alpha.\ (\alpha \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

4 John C. Mitchell. “Coercion and Type Inference”. (1984)

Non type-erasable semantics

Agenda

1. A sized type system

2. Inference

3

Remark 2. No principal types

- Polynomial sizes constraints $\iota * \kappa - 4 = 0$
- Simple polymorphism Extra refinements add size constraints

Inference

Non type-erasable semantics

No principal types

Purpose. Fill missing *redundant* information

Inference

Non type-erasable semantics

No principal types

Purpose. Fill missing *redundant* information

t^*

Untyped term

Inference

💡 Non type-erasable semantics

💡 No principal types

Purpose. Fill missing *redundant* information

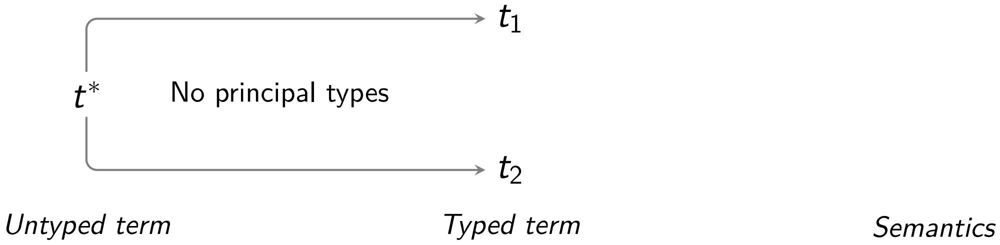
t^*

Untyped term

Typed term

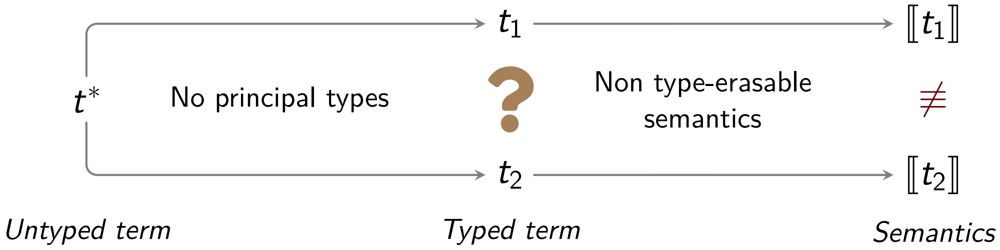
Inference

Purpose. Fill missing *redundant* information



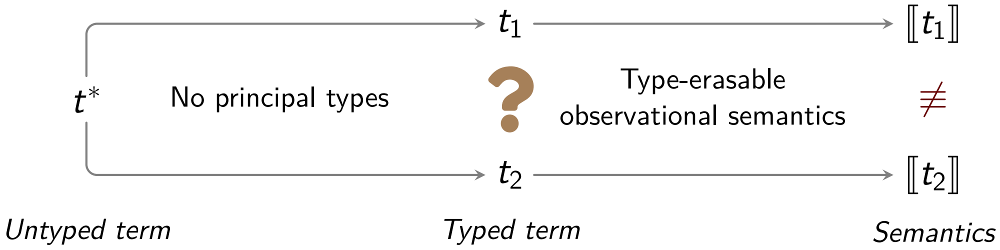
Inference

Purpose. Fill missing *redundant* information



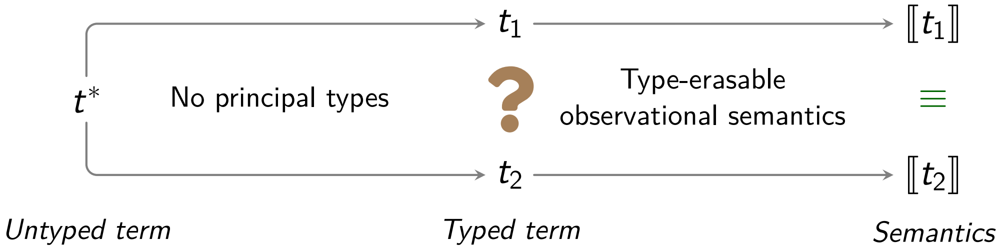
Inference

Purpose. Fill missing *redundant* information



Inference

Purpose. Fill missing *redundant* information, i.e. **without *specializing* terms**



No arbitrary size substitutions

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. At `let` declarations, solve them and generalize size and type variables

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. At `let` declarations, solve them and generalize size and type variables

Constraint solving. Build type & size substitution

Type unification	Refinement propagation	Size resolution	Refinement saturation

```
let slope : α =
    λf. λi : [_]. λj : [_]. (f i - f j) / (i - j)
```

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. ...

Cons

First order unification (Hindley-Milner) $\tau = \tau$

Type unification | *Refinement propagation* | *Size resolution* | *Refinement saturation*

```
let slope : (int → int) → int → int → int =
    λf. λi:[_]. λj:[_]. (f i - f j) / (i - j)
```

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. .

Cons

Select needed refinements (regardless sizes) $r <: r$

Type unification

Refinement propagation

Size resolution

Refinement saturation

```
let slope : (α⁺ → α⁻) → [_] → [_] → int =
    λf. λi : [_]. λj : [_]. (f i - f j) / (i - j)
```

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. .

Simplify polynomial system (keeping all solutions) $\eta = \eta$

Cons

Type unification | *Refinement propagation* | *Size resolution* | *Refinement saturation*

$\texttt{let slope} : (\alpha^+ \to \alpha^-) \to [\iota] \to [\kappa] \to \texttt{int} =$

$\quad \lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i\ -\ f\ j)\ /\ (i\ -\ j)$

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2.

Cons

Propagate refinements if sizes match $\tau <: \tau$

Type unification | *Refinement propagation* | *Size resolution* | ***Refinement saturation***

$\texttt{let}\ \texttt{slope} : (\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}\ =$
$\quad \lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i\ \texttt{-}\ f\ j)\ /\ (i\ \texttt{-}\ j)$

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. At `let` declarations, solve them and generalize size and type variables

Constraint solving. Build type & size substitution

Type unification	Refinement propagation	Size resolution	Refinement saturation

```
let slope : α =
    λf. λi:[6]. λj:[6]. (f i - f j) / (i - j)
```

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. At `let` declarations, solve them and generalize size and type variables

Constraint solving. Build type & size substitution

Type unification	*Refinement propagation*	*Size resolution*	*Refinement saturation*

$$\texttt{let slope} : (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int} =$$
$$\lambda f.\ \lambda i : \texttt{[6]}.\ \lambda j : \texttt{[6]}.\ (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$$

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. At `let` declarations, solve them and generalize size and type variables

Constraint solving. Build type & size substitution

Type unification | *Refinement propagation* | *Size resolution* | *Refinement saturation*

```
let slope : (α⁺ → α⁻) → [_] → [_] → int =
    λf. λi : [6]. λj : [6]. (f i - f j) / (i - j)
```

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. At `let` declarations, solve them and generalize size and type variables

Constraint solving. Build type & size substitution

Type unification	Refinement propagation	Size resolution	Refinement saturation

$$\texttt{let slope} : (\alpha^+ \rightarrow \alpha^-) \rightarrow \texttt{[6]} \rightarrow \texttt{[6]} \rightarrow \texttt{int} =$$
$$\lambda f.\ \lambda i : \texttt{[6]}.\ \lambda j : \texttt{[6]}.\ (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$$

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. At `let` declarations, solve them and generalize size and type variables

Constraint solving. Build type & size substitution

Type unification | *Refinement propagation* | *Size resolution* | *Refinement saturation*

$\texttt{let slope} : (\texttt{[6]} \rightarrow \texttt{int}) \rightarrow \texttt{[6]} \rightarrow \texttt{[6]} \rightarrow \texttt{int} =$
$\quad \lambda f.\, \lambda i : \texttt{[6]}.\, \lambda j : \texttt{[6]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

Algorithm

One-pass resolution.

1. Collect sub-typing constraints: $\{ \tau_1 <: \tau_2 \}$
2. At **`let`** declarations, solve them and generalize size and type variables

Constraint solving. Build type & size substitution

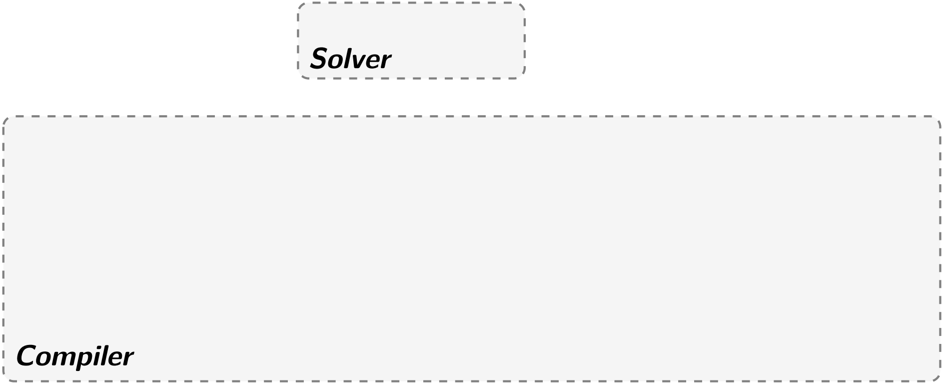
Type unification	*Refinement propagation*	*Size resolution*	*Refinement saturation*

$$\texttt{let}\ \texttt{slope} : (\texttt{[6]} \to \texttt{int}) \to \texttt{[6]} \to \texttt{[6]} \to \texttt{int} \ =$$
$$\lambda f.\ \lambda i : \texttt{[6]}.\ \lambda j : \texttt{[6]}.\ (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$$

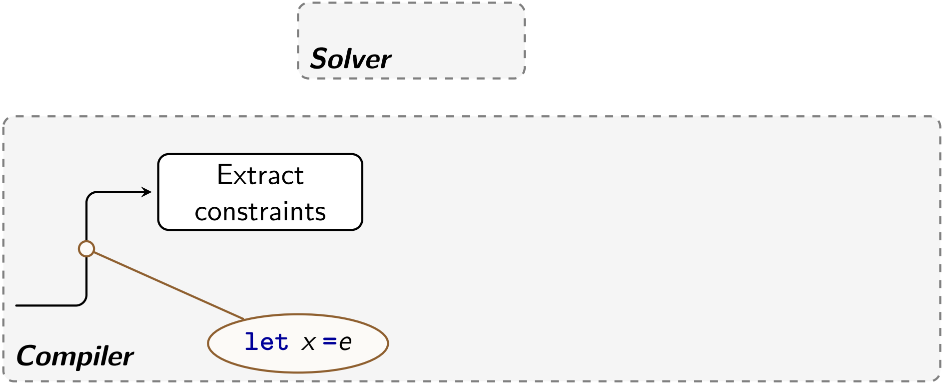
Generalization. Variables must...

- ... not appear in any remaining constraints (simple polymorphism)
- ... appear in the type of the declaration (implicit instantiation)

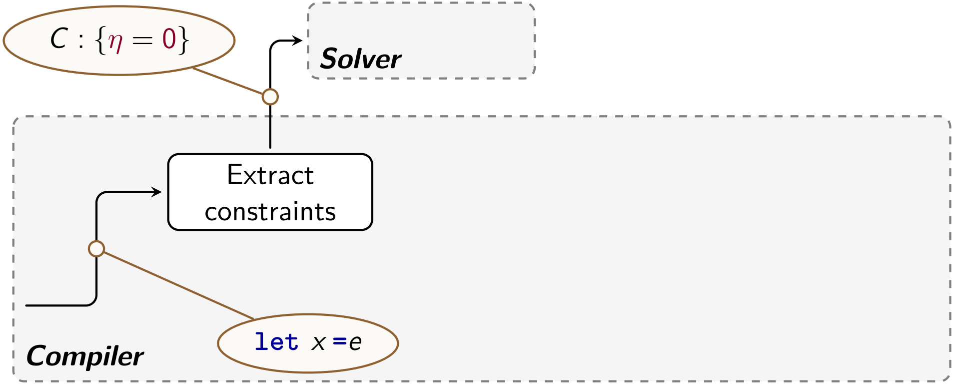
Using solvers



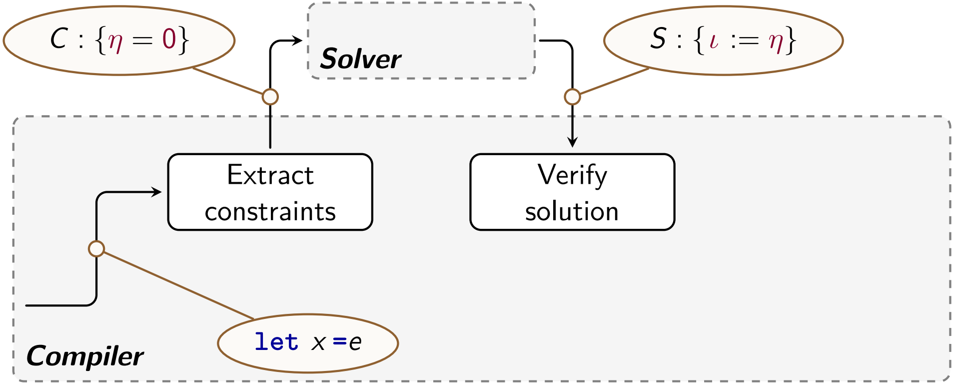
Using solvers



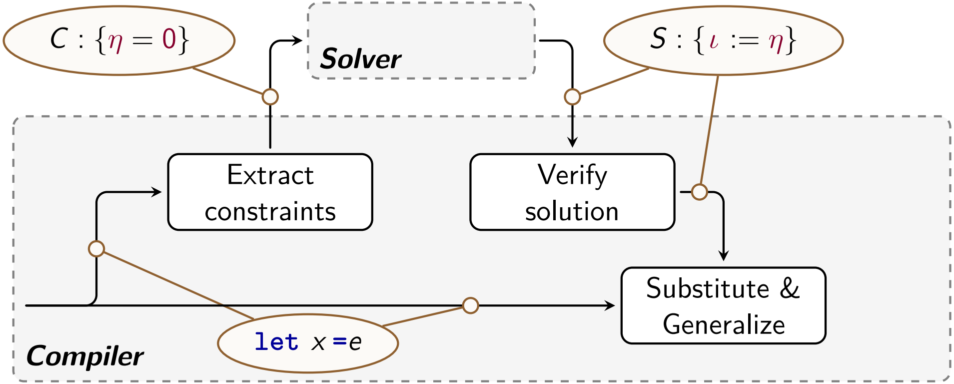
Using solvers



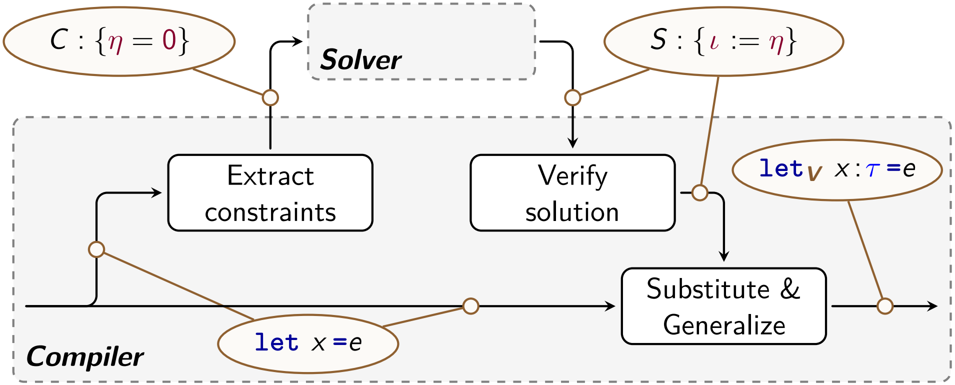
Using solvers



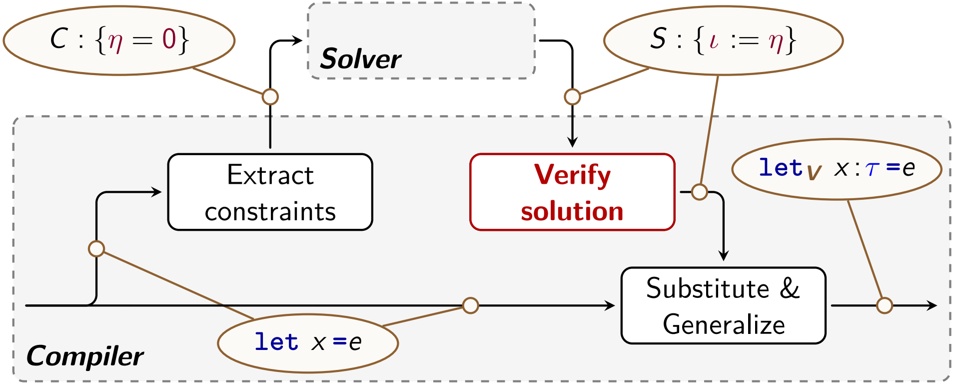
Using solvers



Using solvers



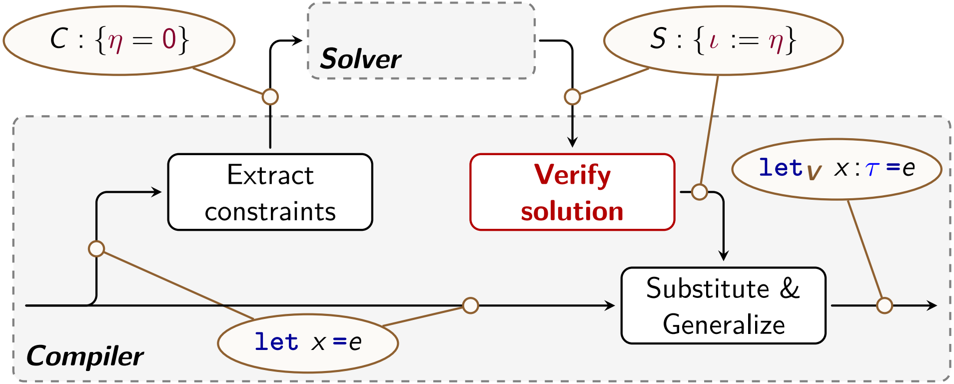
Using solvers



Solution checks

- *Soundness* $\vdash C\{S\}$ S solves C ✔ (Correction)

Using solvers



Solution checks

- *Soundness* $\vdash C\{S\}$ — S solves C ✔ (Correction)
- *Completeness* $C \implies S$ — S is necessary ✘ (Determinism)

Agenda

1. A sized type system

2. Inference

3. Perspectives and conclusion

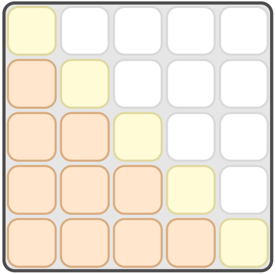
Locally abstract sizes (or types)

Locally abstract sizes (or types)

Local existential sizes.

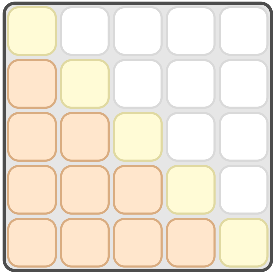
Construct matrix lines by assembling arrays of varying size

Example: The Cholesky decomposition



Locally abstract sizes (or types)

Local existential sizes.
Construct matrix lines by assembling arrays of varying size
Example: The Cholesky decomposition

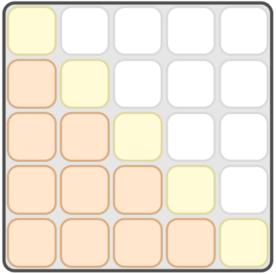


First class polymorphism. [5]
Handle dynamically sized arrays with existential sizes.

[5] Simon L. Peyton Jones et al. "Practical type inference for arbitrary-rank types". (2007)

Locally abstract sizes (or types)

Local existential sizes.
Construct matrix lines by assembling arrays of varying size
Example: The Cholesky decomposition



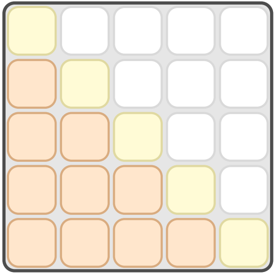
First class polymorphism.[5]
Handle dynamically sized arrays with existential sizes.

Polymorphic recursion.
Call recursively with varying size

[5] Simon L. Peyton Jones et al. “Practical type inference for arbitrary-rank types”. (2007)

Locally abstract sizes (or types)

Local existential sizes.
Construct matrix lines by assembling arrays of varying size
Example: The Cholesky decomposition



First class polymorphism.[5]
Handle dynamically sized arrays with existential sizes.

Polymorphic recursion.
Call recursively with varying size

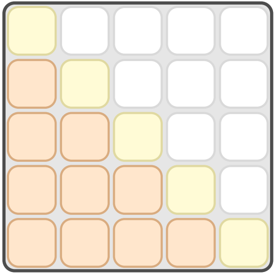
Size resolution.
Polynomial decomposition

$$\eta_1 * \kappa + \eta_2 = 0 \overset{?}{\iff} \begin{cases} \eta_1 = 0 \\ \eta_2 = 0 \end{cases}$$

[5] Simon L. Peyton Jones et al. “Practical type inference for arbitrary-rank types”. (2007)

Locally abstract sizes (or types)

Local existential sizes.
Construct matrix lines by assembling arrays of varying size
Example: The Cholesky decomposition



First class polymorphism.[5]
Handle dynamically sized arrays with existential sizes.

Polymorphic recursion.
Call recursively with varying size

Size resolution.
Polynomial decomposition

$$\begin{cases} \eta_1 * \kappa + \eta_2 = 0 \\ \text{if } \eta_2 \text{ cannot capture } \kappa \end{cases} \overset{\checkmark}{\iff} \begin{cases} \eta_1 = 0 \\ \eta_2 = 0 \end{cases}$$

[5] Simon L. Peyton Jones et al. "Practical type inference for arbitrary-rank types". (2007)

Take-away

A modest extension of ML

- A restricted form of sub-typing
- Polymorphism is extended to sizes
- Polynomial sizes: trade-off between expressiveness, verification and inference

Take-away

A modest extension of ML

- A restricted form of sub-typing
- Polymorphism is extended to sizes
- Polynomial sizes: trade-off between expressiveness, verification and inference

Compilation perspectives

- Unguarded dynamic array accesses `mapi`
- Improved memory placement `concat (A,B)`

Take-away

A modest extension of ML

- A restricted form of sub-typing
- Polymorphism is extended to sizes
- Polynomial sizes: trade-off between expressiveness, verification and inference

Compilation perspectives

- Unguarded dynamic array accesses `mapi`
- Improved memory placement `concat (A,B)`

Thanks for your attention

Size checking?

Negative sizes. Type $[\eta]$ is empty if $\eta \leq 0$

Size checking?

Negative sizes. Type $[\eta]$ is empty if $\eta \leq 0$

Primitives (SOACs)

```
let C = concat A B
```

Size checking?

Negative sizes. Type $[\eta]$ is empty if $\eta \leq 0$

Primitives (SOACs)

`let C = concat A B`

$$\forall i,\ 0 \leq i < \iota + \kappa \implies \mathtt{C}[i] = \begin{cases} \mathtt{A}[i] & \text{if } 0 \leq i < \iota \\ \mathtt{A}[i + \iota] & \text{if } \iota \leq i < \iota + \kappa \end{cases}$$

Size checking?

Negative sizes. Type $[\eta]$ is empty if $\eta \leq 0$

Primitives (SOACs)

`let C = concat A B`

$$\forall i,\ 0 \leq i < \iota + \kappa \implies \texttt{C}[i] = \begin{cases} \texttt{A}[i] & \text{if } 0 \leq i < \iota \\ \texttt{A}[i + \iota] & \text{if } \iota \leq i < \iota + \kappa \end{cases}$$

```
let magic = λa.
    flatten <−1> (map (λe. λ_: [−1]. e) <_> a)
```

Size checking?

Negative sizes. Type $[\eta]$ is empty if $\eta \leq 0$

Primitives (SOACs)

`let C = concat A B` $\qquad \forall i,\ 0 \leq i < \iota + \kappa \implies \mathtt{C}[i] = \begin{cases} \mathtt{A}[i] & \text{if } 0 \leq i < \iota \\ \mathtt{A}[i + \iota] & \text{if } \iota \leq i < \iota + \kappa \end{cases}$

```
let magic = λa.
    flatten <−1> (map (λe. λ_: [−1]. e) <_> a)
```

$[\ \forall \iota \cdot \alpha.\ [\iota]\alpha \to [-\iota]\alpha\ ]$

Size checking?

Negative sizes. Type $[\eta]$ is empty if $\eta \leq 0$

Primitives (SOACs)

`let C = concat A B` $\quad \forall i,\ 0 \leq i < \iota + \kappa \implies \texttt{C}[i] = \begin{cases} \texttt{A}[i] & \text{if } 0 \leq i < \iota \\ \texttt{A}[i + \iota] & \text{if } \iota \leq i < \iota + \kappa \end{cases}$

```
let magic = λa.
    flatten <−1> (map (λe. λ_: [−1]. e) <_> a)
```

$[\ \forall \iota \cdot \alpha.\ [\iota]\alpha \to [-\iota]\alpha\ ]$

Add post-typing constraints. Check...

- ... at run-time (exceptions)
- ... at instantiation, when sizes get constant values
- ... at declaration, with advanced formal tools

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\, \lambda i : \texttt{[\_]}.\, \lambda j : \texttt{[\_]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

Constraint solving

`let slope` : $\alpha = \lambda f.\, \lambda i : [\_].\, \lambda j : [\_].\, (f\ i - f\ j)\ /\ (i - j)$

1. Simple type unification $\tau = \tau$

ML simple types (without refinements):

$\tau ::= \alpha \mid \texttt{int} \mid \tau \rightarrow \tau$

First order unification (Hindley-Milner)

Possible failures

- ✗ Incompatible types $\texttt{int} = \alpha \rightarrow \alpha$
- ✗ Recursive substitution $\alpha = \alpha \rightarrow \alpha$

Constraint solving

`let` slope : α $=$ $\lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i\ -\ f\ j)\ /\ (i\ -\ j)$

1. Simple type unification — $\tau = \tau$ α

Constraint solving

$\texttt{let slope} : \alpha \; = \; \lambda f.\, \lambda i : \texttt{[\_]}.\, \lambda j : \texttt{[\_]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. Simple type unification — $\tau = \tau$

α

$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\, \lambda i : \texttt{[\_]}.\, \lambda j : \texttt{[\_]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. Sim

α

$; \rightarrow \texttt{int}$

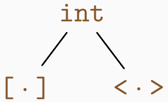
2. Refinement propagation

$r <: r$

Select needed refinements (regardless sizes)

$r ::= x \mid \texttt{int} \mid \texttt{<}\cdot\texttt{>} \mid \texttt{[}\cdot\texttt{]}$

$$
\begin{array}{lcl}
x^- <: \texttt{[}\cdot\texttt{]} & \rightarrow & x^- := \texttt{[}\cdot\texttt{]} \\
\texttt{int} <: x^+ & \rightarrow & x^+ := \texttt{int}
\end{array}
$$



Saturation is deferred after size resolution

Possible failures

✘ Incompatible refinements $\texttt{<}\cdot\texttt{>} <: \texttt{[}\cdot\texttt{]}$

Constraint solving

`let slope` : α = $\lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i\ -\ f\ j)\ /\ (i\ -\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\, \lambda i : \texttt{[\_]}.\, \lambda j : \texttt{[\_]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to \texttt{[\_]} \to \texttt{[\_]} \to \texttt{int}$$

Constraint solving

```
let slope : α = λf. λi:[_]. λj:[_]. (f i - f j) / (i - j)
```

1. Simp

2. Refi

3. Size resolution

$\eta = \eta$

Simplification of a polynomial system: $\{\ \eta_i = 0\ \}_i$

Isolated variable elimination:

$$\iota - \eta = 0 \ \rightarrow\ \iota := \eta \qquad \text{where } \iota \notin FV(\eta)$$

More advanced strategy using *locally abstract variables*

Possible failures

✗ Incompatible sizes $6 = 0$

α

$; \rightarrow$ `int`

$\rightarrow \alpha_r^+$

$\rightarrow$ `int`

Constraint solving

`let slope` : α = $\lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i\ -\ f\ j)\ /\ (i\ -\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution** — $\eta = \eta$

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\, \lambda i : \texttt{[\_]}.\, \lambda j : \texttt{[\_]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification — $\tau = \tau$**

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation — $r <: r$**

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to \texttt{[\_]} \to \texttt{[\_]} \to \texttt{int}$$

3. **Size resolution — $\eta = \eta$**

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

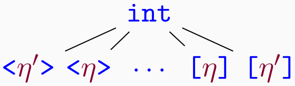
`let slope` : α = $\lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i\ -\ f\ j)\ /\ (i\ -\ j)$

4. Refinement saturation $\tau <: \tau$

Propagate refinements to positive variables, if sizes match

$$\langle\eta\rangle <: \alpha^+ \ \rightarrow\ \alpha^+ := \langle\eta\rangle$$

$$\left\{\begin{array}{l}[\eta_1] <: \alpha^+ \\ [\eta_2] <: \alpha^+\end{array}\right. \ \rightarrow\ \alpha^+ := \texttt{int}$$



Possible failures

1. Sim

2. Refi

3. Size

α

; $\rightarrow$ `int`

$\rightarrow \alpha_r^+$

$\rightarrow$ `int`

$\rightarrow$ `int`

$\rightarrow$ `int`

Constraint solving

`let` slope : α $=$ $\lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i\ -\ f\ j)\ /\ (i\ -\ j)$

1. **Simple type unification — $\tau = \tau$**

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation — $r <: r$**

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution — $\eta = \eta$**

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

4. **Refinement saturation — $\tau <: \tau$**

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\, \lambda i : \texttt{[\_]}.\, \lambda j : \texttt{[\_]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to \texttt{[\_]} \to \texttt{[\_]} \to \texttt{int}$$

3. **Size resolution** — $\eta = \eta$

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

4. **Refinement saturation** — $\tau <: \tau$

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

`let` slope : α $=$ $\lambda f.\ \lambda i : [6].\ \lambda j : [6].\ (f\ i\ -\ f\ j)\ /\ (i\ -\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution** — $\eta = \eta$

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

4. **Refinement saturation** — $\tau <: \tau$

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

`let slope` : α = $\lambda f.\ \lambda i : [6].\ \lambda j : [6].\ (f\ i\ -\ f\ j)\ /\ (i\ -\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution** — $\eta = \eta$

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

4. **Refinement saturation** — $\tau <: \tau$

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\ \lambda i : \texttt{[6]}.\ \lambda j : \texttt{[6]}.\ (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification — $\tau = \tau$**

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation — $r <: r$**

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution — $\eta = \eta$**

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

4. **Refinement saturation — $\tau <: \tau$**

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \; = \; \lambda f.\, \lambda i : [6].\, \lambda j : [6].\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution** — $\eta = \eta$

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

4. **Refinement saturation** — $\tau <: \tau$

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\, \lambda i : \texttt{[6]}.\, \lambda j : \texttt{[6]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification — $\tau = \tau$**

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation — $r <: r$**

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to \texttt{[\_]} \to \texttt{[\_]} \to \texttt{int}$$

3. **Size resolution — $\eta = \eta$**

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

4. **Refinement saturation — $\tau <: \tau$**

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \ = \ \lambda f.\ \lambda i : \texttt{[6]}.\ \lambda j : \texttt{[6]}.\ (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution** — $\eta = \eta$

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$

4. **Refinement saturation** — $\tau <: \tau$

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\, \lambda i : [6].\, \lambda j : [6].\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification — $\tau = \tau$**

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation — $r <: r$**

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution — $\eta = \eta$**

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [6] \to [6] \to \texttt{int}$$

4. **Refinement saturation — $\tau <: \tau$**

$$(\texttt{int} \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\ \lambda i : [6].\ \lambda j : [6].\ (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to [\_] \to [\_] \to \texttt{int}$$

3. **Size resolution** — $\eta = \eta$

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to [6] \to [6] \to \texttt{int}$$

4. **Refinement saturation** — $\tau <: \tau$

$$([6] \to \alpha_c^-) \to [6] \to [6] \to \texttt{int}$$
$$(\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

Constraint solving

$\texttt{let slope} : \alpha \;=\; \lambda f.\, \lambda i : \texttt{[6]}.\, \lambda j : \texttt{[6]}.\, (f\ i\ \texttt{-}\ f\ j)\ \texttt{/}\ (i\ \texttt{-}\ j)$

1. **Simple type unification** — $\tau = \tau$

$$\alpha$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2. **Refinement propagation** — $r <: r$

$$(\alpha_d^+ \to \alpha_c^-) \to \alpha_i^- \to \alpha_j^- \to \alpha_r^+$$
$$(\alpha_d^+ \to \alpha_c^-) \to \texttt{[\_]} \to \texttt{[\_]} \to \texttt{int}$$

3. **Size resolution** — $\eta = \eta$

$$(\alpha_d^+ \to \alpha_c^-) \to [\iota] \to [\kappa] \to \texttt{int}$$
$$(\alpha_d^+ \to \alpha_c^-) \to \texttt{[6]} \to \texttt{[6]} \to \texttt{int}$$

4. **Refinement saturation** — $\tau <: \tau$

$$(\texttt{[6]} \to \alpha_c^-) \to \texttt{[6]} \to \texttt{[6]} \to \texttt{int}$$
$$(\texttt{[6]} \to \texttt{int}) \to \texttt{[6]} \to \texttt{[6]} \to \texttt{int}$$

Issue 1: Size-dependent Semantics

Non type-erasable semantics

Coerced substitutions

$(\lambda x : [\eta].\ e)\ 8 \rightsquigarrow e\{8 \triangleright [\eta] / x\}$

Sizes may stop the reduction

Size expressions

$<\eta> \rightsquigarrow [\![\eta]\!]_\rho$

Sizes define the reduction result

Observational semantics does not depend on type variables:

$$e\{\overline{\tau}_1/\overline{\alpha}\} \equiv e\{\overline{\tau}_2/\overline{\alpha}\} \qquad \text{where } \overline{\alpha} = FV(e)$$

Intuition: Type variables...

- ... have no *computational content*
- ... may only impact the domain of definition

Issue 2: Principal types?

$$\texttt{let slope} = \lambda f.\ \lambda i : [\_].\ \lambda j : [\_].\ (f\ i\ \texttt{-}\ f\ j)\ /\ (i\ \texttt{-}\ j)$$

1. Simple polymorphism

$$\forall \iota.\ ([\iota] \to \texttt{int}) \to [\iota] \to [\iota] \to \texttt{int}$$

$$\forall \iota \cdot \kappa.\ (\texttt{int} \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

⇒ no principal types

Constrained polymorphism[6]

$$\forall \iota \cdot \kappa \cdot \alpha \mid [\iota], [\kappa] <: \alpha.\ (\alpha \to \texttt{int}) \to [\iota] \to [\kappa] \to \texttt{int}$$

⇒ delayed size resolution

2. Polynomial size constraints:

$$\iota * \kappa - 4 = 0$$

[6] John C. Mitchell. "Coercion and Type Inference". (1984)

References I

[1] Cormac Flanagan. "Hybrid type checking". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 245–256. DOI: 10.1145/1111037.1111059.

[2] Troels Henriksen et al. "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 556–571. DOI: 10.1145/3062341.3062354.

[3] John Hughes, Lars Pareto, and Amr Sabry. "Proving the Correctness of Reactive Systems Using Sized Types". In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, 1996, pp. 410–423. DOI: 10.1145/237721.240882.

[4] Simon L. Peyton Jones et al. "Practical type inference for arbitrary-rank types". In: *Journal of functional programming* 17.1 (2007), pp. 1–82. DOI: 10.1017/S0956796806006034.

[5] Andrew Kennedy. "Dimension Types". In: *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*. Ed. by Donald Sannella. Vol. 788. Lecture Notes in Computer Science. Springer, 1994, pp. 348–362. DOI: 10.1007/3-540-57880-3_23.

References II

[6] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[7] John C. Mitchell. “Coercion and Type Inference”. In: *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*. Ed. by Ken Kennedy, Mary S. Van Deusen, and Larry Landweber. ACM Press, 1984, pp. 175–185. DOI: 10.1145/800017.800529.

[8] Adam Paszke et al. “Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming”. In: *Proceedings of the ACM on Programming Languages* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473593.

[9] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Liquid types”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602.

[10] Hongwei Xi and Frank Pfenning. “Dependent Types in Practical Programming”. In: *POPL ’99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 214–227. DOI: 10.1145/292540.292560.

References III

[11] Christoph Zenger. “Indexed Types”. In: *Theoretical computer science* 187.1-2 (1997), pp. 147–165. DOI: 10.1016/S0304-3975(97)00062-5.