Size polymorphism

Ensuring correction of array accesses with typing

Jean-Louis Colaço, Baptiste Pauget, Marc Pouzet

Synchron 2021 - La Rochette

November 22, 2021

Requirements and context

Requirements and context

Safety critical embedded systems

- No errors at run-time
- Graphical specification (inference)
- Statically bounded memory

Requirements and context

Safety critical embedded systems

- No errors at run-time
- Graphical specification (inference)
- Statically bounded memory

Targeted array applications

- Signal processing, AI
- Non linear size relations, recursion
- Polymorphism (on size, on shape)

Arrays in programming languages

Arrays in programming languages

Extensional arrays: collections of elements

- Out of bounds accesses
- Incomplete definitions

Arrays in programming languages

Extensional arrays: collections of elements

- Out of bounds accesses
- Incomplete definitions

Intensional arrays: indivisible objects

- Used with *iterators* (`map`, `fold`, ...)
- Correct accesses by construction (but limited expressiveness)
- Size inconsistencies (`zip`, `map2`, ...)

Agenda

1. Bringing sizes in types
    1. Refinements
    2. Size language
    3. Polymorphism
2. Size Inference
    1. Inference steps
    2. Size constraint resolution
3. The language
    1. Examples
    2. Coercions
    3. Binding time analysis

Agenda

1. Bringing sizes in types
   1. Refinements
   2. Size language
   3. Polymorphism

2. Size Inference
   1. Inference steps
   2. Size constraint resolution

3. The language
   1. Examples
   2. Coercions
   3. Binding time analysis

Sizes in types

Size Polymorphism

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

Dependent Types

$\texttt{val map} : \forall\alpha,\beta.\ (\alpha \to \beta) \to \Pi n{:}\texttt{int}.\ \texttt{[}n\texttt{]}\alpha \to \texttt{[}n\texttt{]}\beta$

Sizes in types

1. Refinements

Size Polymorphism

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

Dependent Types

val map : $\forall\alpha,\beta.\ (\alpha \to \beta) \to \Pi n{:}\texttt{int}.\ \texttt{[}n\texttt{]}\alpha \to \texttt{[}n\texttt{]}\beta$

1. Refinements

Sizes in types

❃ Refinements

General form [XP98, Fla06]

- Predicates over base type: $\{x : \mathtt{B} \mid P(x)\}$
- Sub-typing: predicate implication

Sizes in types

❃ Refinements

General form [XP98, Fla06]

- Predicates over base type: $\{x : \mathtt{B} \mid P(x)\}$
- Sub-typing: predicate implication

  $\star$ **Undecidable type checking** $\star$

Sizes in types

❃ Refinements

General form [XP98, Fla06]

- Predicates over base type: $\{x : \mathtt{B} \mid P(x)\}$
- Sub-typing: predicate implication

  ⋆ **Undecidable type checking** ⋆

τ	$::=$		*Types*
	$\mid$	α, β, γ	variable
	$\mid$	`int`	integer
	$\mid$	`bool`	boolean
	$\mid$	$\tau \to \tau$	function

Sizes in types

❃ Refinements

General form [XP98, Fla06]

- Predicates over base type: $\{x : \texttt{B} \mid P(x)\}$
- Sub-typing: predicate implication

  ⋆ **Undecidable type checking** ⋆

τ	$::=$		*Types*
	$\mid$	α, β, γ	variable
	$\mid$	$\texttt{<}\eta\texttt{>}$	singleton
	$\mid$	$\texttt{[}\eta\texttt{]}$	interval
	$\mid$	$\texttt{int}$	integer
	$\mid$	$\texttt{bool}$	boolean
	$\mid$	$\tau \to \tau$	function

Integer refinements

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

Sizes in types

❃ Refinements

General form [XP98, Fla06]

- Predicates over base type: $\{x : \mathtt{B} \mid P(x)\}$
- Sub-typing: predicate implication

  ⋆ **Undecidable type checking** ⋆

τ	::=		*Types*
	|	α, β, γ	variable
	|	$\texttt{<}\eta\texttt{>}$	singleton
	|	$\texttt{[}\eta\texttt{]}$	interval
	|	$\texttt{int}$	integer
	|	$\texttt{bool}$	boolean
	|	$\tau \to \tau$	function

Integer refinements

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

Sizes in types

❃ Refinements

τ	$::=$		*Types*
	$\mid$	α, β, γ	variable
	$\mid$	$\texttt{<}\eta\texttt{>}$	singleton
	$\mid$	$\texttt{[}\eta\texttt{]}$	interval
	$\mid$	$\texttt{int}$	integer
	$\mid$	$\texttt{bool}$	boolean
	$\mid$	$\tau \to \tau$	function

General form [XP98, Fla06]

- Predicates over base type: $\{x : \mathtt{B} \mid P(x)\}$
- Sub-typing: predicate implication

  $\star$ **Undecidable type checking** $\star$

Integer refinements

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

Arrays as functions[1] with bounded domain: $\texttt{[}\eta\texttt{]}\tau \equiv \texttt{[}\eta\texttt{]} \to \tau$

- Correctness of accesses ensured by typing

[1] for typing purposes only

Sizes in types

1. Refinements

Size Polymorphism

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

Dependent Types

$\texttt{val map} : \forall \alpha, \beta.\ (\alpha \to \beta) \to \Pi n{:}\texttt{int}.\ \texttt{[}n\texttt{]}\alpha \to \texttt{[}n\texttt{]}\beta$

1. Refinements

Sizes in types

1. Refinements
2. Size language

Size Polymorphism

val map : $\forall\iota.\ \forall\alpha,\beta.\ (\alpha \to \beta) \to \langle\iota\rangle \to [\iota]\,\alpha \to [\iota]\,\beta$

Dependent Types

val map : $\forall\alpha,\beta.\ (\alpha \to \beta) \to \Pi n{:}\mathtt{int}.\ [n]\,\alpha \to [n]\,\beta$

1. Refinements
2. Expressions

Sizes in types

✻ Size language

Multivariate polynomials $\eta \in \mathbb{Z}[\mathcal{V}_\eta]$

η	$::=$		*Sizes*
	$\mid$	ι, δ, κ	variable
	$\mid$	n	constant
	$\mid$	$\eta + \eta$	sum
	$\mid$	$\eta * \eta$	product

Sizes in types

❃ Size language

Multivariate polynomials $\eta \in \mathbb{Z}[\mathcal{V}_\eta]$

- Formal handling: normal form

η	$::=$		*Sizes*
	$\mid$	ι, δ, κ	variable
	$\mid$	n	constant
	$\mid$	$\eta + \eta$	sum
	$\mid$	$\eta * \eta$	product

Sizes in types

❃ Size language

Multivariate polynomials $\eta \in \mathbb{Z}[\mathcal{V}_\eta]$

- Formal handling: normal form
- Expressiveness: non-linear expressions

η	$::=$		*Sizes*
	$\mid$	ι, δ, κ	variable
	$\mid$	n	constant
	$\mid$	$\eta + \eta$	sum
	$\mid$	$\eta * \eta$	product

Sizes in types

❃ Size language

Multivariate polynomials $\eta \in \mathbb{Z}[\mathcal{V}_\eta]$

- Formal handling: normal form
- Expressiveness: non-linear expressions

η	$::=$		*Sizes*
	$\mid$	ι, δ, κ	variable
	$\mid$	n	constant
	$\mid$	η + η	sum
	$\mid$	η * η	product

Equivalent types schemes

$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \rightarrow \texttt{[}\iota * \delta - \delta + 1\texttt{]}\alpha \rightarrow \texttt{[}\iota\texttt{]}\alpha$

$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \rightarrow \texttt{[}\iota * \delta + 1\texttt{]}\alpha \rightarrow \texttt{[}\iota + 1\texttt{]}\alpha$

Sizes in types

✻ Size language

η	$::=$		*Sizes*
	$\mid$	ι, δ, κ	variable
	$\mid$	n	constant
	$\mid$	$\eta + \eta$	sum
	$\mid$	$\eta * \eta$	product

Multivariate polynomials $\eta \in \mathbb{Z}[\mathcal{V}_\eta]$

- Formal handling: normal form
- Expressiveness: non-linear expressions

Equivalent types schemes

`val sample` : $\forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta - \delta + 1]\,\alpha \to [\iota]\,\alpha$

`val sample` : $\forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta + 1]\,\alpha \to [\iota + 1]\,\alpha$

No most general types schemes

`let zero` : $\texttt{<}\iota\texttt{>} \to \texttt{<}0\texttt{>} = \lambda n : \texttt{<}\iota\texttt{>}.\ (n - 1) * (n - 2)$

Sizes in types

❃ Size language

η	$::=$		*Sizes*
	$\mid$	ι, δ, κ	variable
	$\mid$	n	constant
	$\mid$	$\eta + \eta$	sum
	$\mid$	$\eta * \eta$	product

Multivariate polynomials $\eta \in \mathbb{Z}[\mathcal{V}_\eta]$

- Formal handling: normal form
- Expressiveness: non-linear expressions

Equivalent types schemes

`val sample` : $\forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta - \delta + 1]\,\alpha \to [\iota]\,\alpha$

`val sample` : $\forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta + 1]\,\alpha \to [\iota + 1]\,\alpha$

No most general types schemes

`let zero` : $\texttt{<}\iota\texttt{>} \to \texttt{<}0\texttt{>} = \lambda n{:}\texttt{<}\iota\texttt{>}.\ (n - 1) * (n - 2)$

$\Longrightarrow$ Well-typed if $\iota = 1$ or $\iota = 2$

$$\text{Incompatible types} \begin{cases} \texttt{<}1\texttt{>} \to \texttt{<}0\texttt{>} \\ \texttt{<}2\texttt{>} \to \texttt{<}0\texttt{>} \end{cases}$$

Sizes in types

1. Refinements
2. Size language

Size Polymorphism

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

Dependent Types

$\texttt{val map} : \forall \alpha, \beta.\ (\alpha \to \beta) \to \Pi n : \texttt{int}.\ \mathtt{[}n\mathtt{]}\,\alpha \to \mathtt{[}n\mathtt{]}\,\beta$

1. Refinements
2. Expressions

Sizes in types

1. Refinements
2. Size language
3. Polymorphism

Size Polymorphism

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

Dependent Types

$\texttt{val map} : \forall\alpha,\beta.\ (\alpha \to \beta) \to \Pi n\texttt{:int}.\ \texttt{[}n\texttt{]}\,\alpha \to \texttt{[}n\texttt{]}\,\beta$

1. Refinements
2. Expressions
3. Dependency

Sizes in types

✻ Polymorphism

σ	$::=$		*Type scheme*
	$\mid$	τ	simple type
	$\mid$	$\forall \iota .\ \sigma$	size quantif.
	$\mid$	$\forall \alpha .\ \sigma$	type quantif.

Handling sizes as types

- Static sizes only
- Constraint based definitions
- Implicit instantiation and generalization, inference

Sizes in types

❃ Polymorphism

σ	$::=$		*Type scheme*
	$\mid$	τ	simple type
	$\mid$	$\forall\iota.\ \sigma$	size quantif.
	$\mid$	$\forall\alpha.\ \sigma$	type quantif.

Handling sizes as types

- Static sizes only
- Constraint based definitions
- Implicit instantiation and generalization, inference

Example: scalar product

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

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

Sizes in types

❃ Polymorphism

σ	$::=$		*Type scheme*
	$\mid$	τ	simple type
	$\mid$	$\forall\iota.\ \sigma$	size quantif.
	$\mid$	$\forall\alpha.\ \sigma$	type quantif.

Handling sizes as types

- Static sizes only
- Constraint based definitions
- Implicit instantiation and generalization, inference

Example: scalar product

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

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

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

Sizes in types

❃ Polymorphism

$$\begin{array}{llll} \sigma & ::= & & \textit{Type scheme} \\ & \mid & \tau & \text{simple type} \\ & \mid & \forall\iota.\ \sigma & \text{size quantif.} \\ & \mid & \forall\alpha.\ \sigma & \text{type quantif.} \end{array}$$

Handling sizes as types

- Static sizes only
- Constraint based definitions
- Implicit instantiation and generalization, inference

Example: scalar product

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

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

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

$$\begin{array}{llll} e & ::= & \ldots & \textit{Expressions} \\ & \mid & \texttt{<}\eta\texttt{>} & \text{size} \end{array}$$

Sizes in types

❃ Polymorphism

σ	$::=$		*Type scheme*
	$\mid$	τ	simple type
	$\mid$	$\forall\iota.\ \sigma$	size quantif.
	$\mid$	$\forall\alpha.\ \sigma$	type quantif.

Handling sizes as types

- Static sizes only
- Constraint based definitions
- Implicit instantiation and generalization, inference

Example: scalar product

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

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

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

$\texttt{val dot\_product} : \forall\iota.\ [\iota]\texttt{int} \to [\iota]\texttt{int} \to \texttt{int}$

Agenda

Inference

The `pack` operator

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to \texttt{[}\iota + \kappa - 1\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\texttt{[}\kappa\texttt{]}\,\alpha$$
$$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to \texttt{[}\iota * \delta - \delta + 1\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\alpha$$

Inference

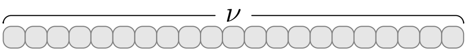
The `pack` operator

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

$\texttt{val sample} : \forall\iota, \delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \to \texttt{[}\iota * \delta - \delta + 1\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\alpha$

$\texttt{let pack} : \_ = \lambda x : \_ .$

$\quad \texttt{sample <\_> (window <\_>}\ x\texttt{)}$



Inference

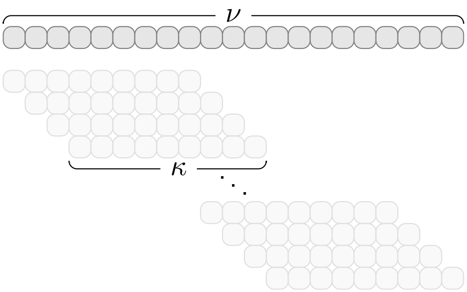
The `pack` operator

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

$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \rightarrow [\iota * \delta - \delta + 1]\,\alpha \rightarrow [\iota]\,\alpha$

$\texttt{let pack} : \_ = \lambda x : \_ .$

$\quad \texttt{sample <\_> (window <\_>}\ x\texttt{)}$

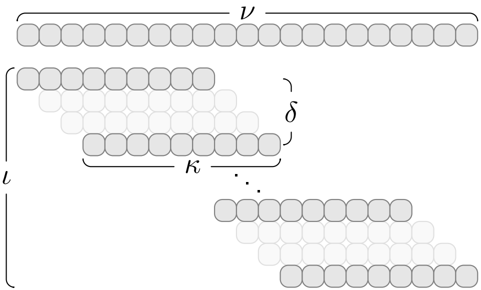


The `pack` operator

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

$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \rightarrow [\iota*\delta-\delta+1]\alpha \rightarrow [\iota]\alpha$

$\texttt{let pack} : \_ = \lambda x : \_.$
$\quad \texttt{sample <\_> (window <\_>}\ x)$

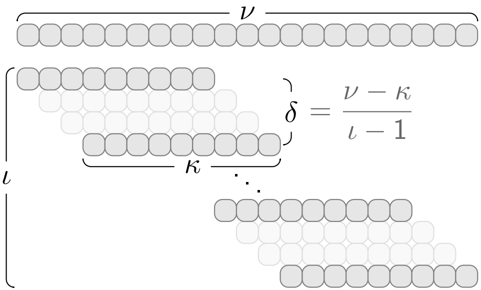


The `pack` operator

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

$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \to [\iota*\delta-\delta+1]\alpha \to [\iota]\alpha$

$\texttt{let pack} : \_ = \lambda x : \_.$
$\quad \texttt{sample <\_>}\ (\texttt{window <\_>}\ x)$



Inference

Typing `pack`

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

$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \rightarrow [\iota * \delta - \delta + 1]\alpha \rightarrow [\iota]\alpha$

$$\delta = \frac{\nu - \kappa}{\iota - 1}$$

$\texttt{let pack} : \_ =$
$\quad \lambda x : \_.$
$\quad \texttt{sample <\_>}$
$\qquad (\texttt{window <\_>}\ x)$

▷ Type inference

▷ Integer refinement

▷ Size inference

Inference

Typing `pack`

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota + \kappa - 1]\alpha \to [\iota]\,[\kappa]\alpha \qquad \delta = \frac{\nu - \kappa}{\iota - 1}$$

$$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta - \delta + 1]\alpha \to [\iota]\alpha$$

$$\begin{aligned}&\texttt{let pack} : \_ =\\ &\quad \lambda x : \_.\\ &\quad \texttt{sample <\_>}\\ &\qquad (\texttt{window <\_>}\ x)\end{aligned}$$

▷ **Type inference** ▷ Integer refinement ▷ Size inference

Inference

Typing `pack`

$$\texttt{val window} : \forall\iota,\kappa.\ \forall\alpha.\ \texttt{<}\kappa\texttt{>} \rightarrow [\iota+\kappa-1]\alpha \rightarrow [\iota][\kappa]\alpha$$
$$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \rightarrow [\iota*\delta-\delta+1]\alpha \rightarrow [\iota]\alpha$$

$$\delta = \frac{\nu-\kappa}{\iota-1}$$

$$\begin{aligned}&\texttt{let pack} : \alpha_1 = \\ &\quad \lambda x{:}\alpha_2. \\ &\quad \texttt{sample}_{\beta_1}\ \texttt{<\_>} \\ &\qquad (\texttt{window}_{\beta_2}\ \texttt{<\_>}\ x)\end{aligned}$$

▷ **Type inference** ▷ Integer refinement ▷ Size inference

- Explicit instantiation

Inference

Typing `pack`

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

$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \rightarrow [\iota * \delta - \delta + 1]\,\alpha \rightarrow [\iota]\,\alpha$

$$\delta = \frac{\nu - \kappa}{\iota - 1}$$

$\texttt{let pack} : \forall \alpha.\ (\overline{\texttt{int}} \rightarrow \alpha) \rightarrow \overline{\texttt{int}} \rightarrow \overline{\texttt{int}} \rightarrow \alpha =$
$\quad \lambda x : \overline{\texttt{int}} \rightarrow \alpha.$
$\quad \texttt{sample}_{\overline{\texttt{int}} \rightarrow \alpha}\ \texttt{<\_>}$
$\quad\quad (\texttt{window}_{\alpha}\ \texttt{<\_>}\ x)$

▷ **Type inference** ▷ Integer refinement ▷ Size inference

- Explicit instantiation
- Unrefined type $\overline{\texttt{int}}$
- Structural unification, generalization

Inference

Typing `pack`

$$\texttt{val window} : \forall\iota,\kappa.\ \forall\alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota+\kappa-1]\,\alpha \to [\iota]\,[\kappa]\,\alpha \qquad \delta = \frac{\nu-\kappa}{\iota-1}$$

$$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \to [\iota*\delta-\delta+1]\,\alpha \to [\iota]\,\alpha$$

$$\begin{array}{l} \texttt{let pack} : \forall\alpha.\ (\overline{\texttt{int}} \to \alpha) \to \overline{\texttt{int}} \to \overline{\texttt{int}} \to \alpha = \\ \quad \lambda x : \overline{\texttt{int}} \to \alpha. \\ \quad \texttt{sample}_{\overline{\texttt{int}}\to\alpha}\ \texttt{<\_>} \\ \qquad (\texttt{window}_{\alpha}\ \texttt{<\_>}\ x) \end{array}$$

▷ Type inference ▷ **Integer refinement** ▷ Size inference

Inference

Typing `pack`

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

$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \to [\iota*\delta-\delta+1]\alpha \to [\iota]\alpha$

$$\delta = \frac{\nu-\kappa}{\iota-1}$$

$\texttt{let pack} : \forall\alpha.\ [\_]\alpha \to [\_][\_]\alpha =$

$\quad \lambda x : [\_]\alpha.$

$\quad \texttt{sample}_{[\_]\alpha}\ \texttt{<\_>}$

$\quad\quad (\texttt{window}_{\alpha}\ \texttt{<\_>}\ x)$

▷ Type inference

▷ **Integer refinement**

▷ Size inference

- $\overline{\texttt{int}}$ occurrence refinement
- Local propagation

Inference

Typing pack

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

$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta - \delta + 1]\alpha \to [\iota]\alpha$

$$\delta = \frac{\nu - \kappa}{\iota - 1}$$

$\texttt{let pack} : \forall \alpha.\ [\_]\alpha \to [\_]\,[\_]\alpha =$
$\quad \lambda x : [\_]\alpha.$
$\quad \texttt{sample}_{[\_]\alpha}\ \texttt{<}\_\texttt{>}$
$\qquad (\texttt{window}_{\alpha}\ \texttt{<}\_\texttt{>}\ x)$

- ▷ Type inference
- ▷ Integer refinement
- ▷ **Size inference**

Inference

Instantiating `pack`

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \rightarrow [\iota + \kappa - 1]\alpha \rightarrow [\iota][\kappa]\alpha$$
$$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \rightarrow [\iota * \delta - \delta + 1]\alpha \rightarrow [\iota]\alpha$$

$$\delta = \frac{\nu - \kappa}{\iota - 1}$$

$$\texttt{let pack} : \forall \alpha.\ [\nu]\alpha \rightarrow [\iota][\kappa]\alpha =$$
$$\quad \lambda x : [\nu_i]\alpha.$$
$$\quad \texttt{sample}_{\ \iota_s\ \delta_s\ [\kappa'_w]\alpha}\ \texttt{<}\kappa_1\texttt{>}$$
$$\quad\quad (\texttt{window}_{\ \iota_w\ \kappa_w\ \alpha}\ \texttt{<}\kappa_2\texttt{>}\ x)$$

▷ Type inference ▷ Integer refinement ▷ **Size inference**

- Explicit instantiation
- Collect constraints of the form $\eta = 0$

Inference

Instantiating `pack`

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

$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \rightarrow [\iota*\delta-\delta+1]\alpha \rightarrow [\iota]\alpha$

$$\delta = \frac{\nu-\kappa}{\iota-1}$$

$\texttt{let pack} : \forall\iota,\kappa,\delta.\ \forall\alpha.\ [\iota*\delta-\delta+\kappa]\alpha \rightarrow [\iota][\kappa]\alpha =$

$\quad \lambda x : [\iota*\delta-\delta+\kappa]\alpha.$

$\quad \texttt{sample}_{\,\iota\ \delta\ [\kappa]\alpha}\ \texttt{<}\delta\texttt{>}$

$\qquad (\texttt{window}_{\,(\iota*\delta-\delta+1)\ \kappa\ \alpha}\ \texttt{<}\kappa\texttt{>}\ x)$

▷ Type inference ▷ Integer refinement ▷ **Size inference**

- Explicit instantiation
- Collect constraints of the form $\eta = 0$
- Resolve system at generalization points
- Isolated variable elimination: $\iota - \eta = 0,\ \iota \notin \mathit{Vars}(\eta)$

Inference

Instantiating `pack`

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota + \kappa - 1]\alpha \to [\iota]\,[\kappa]\alpha$$
$$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta - \delta + 1]\alpha \to [\iota]\alpha$$

$$\delta = \frac{\nu - \kappa}{\iota - 1}$$

$$\texttt{val pack} : \forall \iota, \kappa, \delta.\ \forall \alpha.\ [\iota * \delta - \delta + \kappa]\alpha \to [\iota]\,[\kappa]\alpha$$

Inference

Instantiating `pack`

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

$$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \rightarrow [\iota * \delta - \delta + 1]\alpha \rightarrow [\iota]\alpha$$

$$\texttt{val pack} : \forall \iota, \kappa, \delta.\ \forall \alpha.\ [\iota * \delta - \delta + \kappa]\alpha \rightarrow [\iota][\kappa]\alpha$$

$$\texttt{let split} : [\iota * \kappa]\_ \rightarrow [\iota][\kappa]\_ = \texttt{pack}$$

Inference

Instantiating `pack`

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

$$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \to [\iota*\delta-\delta+1]\alpha \to [\iota]\alpha$$

$$\texttt{val pack} : \forall\iota,\kappa,\delta.\ \forall\alpha.\ [\iota*\delta-\delta+\kappa]\alpha \to [\iota][\kappa]\alpha$$

$$\texttt{let split} : \forall\alpha.\ [\iota*\kappa]\alpha \to [\iota][\kappa]\alpha = \texttt{pack}_{\ \iota'\ \kappa'\ \delta\ \alpha}$$

▷ **Size inference**

Inference

Instantiating `pack`

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota + \kappa - 1]\alpha \to [\iota]\,[\kappa]\alpha$$
$$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [\iota * \delta - \delta + 1]\alpha \to [\iota]\alpha$$

$$\texttt{val pack} : \forall \iota, \kappa, \delta.\ \forall \alpha.\ [\iota * \delta - \delta + \kappa]\alpha \to [\iota]\,[\kappa]\alpha$$

$$\texttt{let split} : \forall \alpha.\ [\iota * \kappa]\alpha \to [\iota]\,[\kappa]\alpha = \texttt{pack}_{\iota\ \kappa\ \delta\ \alpha}$$

▷ **Size inference**

1. Variable elimination (equivalent substitution)

$$\iota * \delta - \delta + \kappa = \iota * \kappa$$

Inference

Instantiating `pack`

$$\texttt{val window} : \forall\iota,\kappa.\ \forall\alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota+\kappa-1]\,\alpha \to [\iota]\,[\kappa]\,\alpha$$
$$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \to [\iota*\delta-\delta+1]\,\alpha \to [\iota]\,\alpha$$

$$\texttt{val pack} : \forall\iota,\kappa,\delta.\ \forall\alpha.\ [\iota*\delta-\delta+\kappa]\,\alpha \to [\iota]\,[\kappa]\,\alpha$$

$$\texttt{let split} : \forall\alpha.\ [\iota*\kappa]\,\alpha \to [\iota]\,[\kappa]\,\alpha = \texttt{pack}_{\iota\ \kappa\ \delta\ \alpha}$$

- **Size inference**
  1. Variable elimination (equivalent substitution)

$$\iota*\delta-\delta+\kappa = \iota*\kappa$$

  2. Structural unification (nonequivalent substitution)

$$(\iota-1)*(\delta-\kappa) = 0$$

Inference

Instantiating `pack`

$$\texttt{val window} : \forall\iota,\kappa.\ \forall\alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota+\kappa-1]\alpha \to [\iota][\kappa]\alpha$$
$$\texttt{val sample} : \forall\iota,\delta.\ \forall\alpha.\ \texttt{<}\delta\texttt{>} \to [\iota*\delta-\delta+1]\alpha \to [\iota]\alpha$$

$$\texttt{val pack} : \forall\iota,\kappa,\delta.\ \forall\alpha.\ [\iota*\delta-\delta+\kappa]\alpha \to [\iota][\kappa]\alpha$$

$$\texttt{let split} : \forall\iota,\kappa.\ \forall\alpha.\ [\iota*\kappa]\alpha \to [\iota][\kappa]\alpha = \texttt{pack}_{\ \iota\ \kappa\ \kappa\ \alpha}$$

▷ **Size inference**

1. Variable elimination (equivalent substitution)

$$\iota*\delta-\delta+\kappa = \iota*\kappa$$

2. Structural unification (nonequivalent substitution)

$$(\iota-1)*(\delta-\kappa) = 0$$

Agenda

1. Bringing sizes in types
    1. Refinements
    2. Size language
    3. Polymorphism

2. Size Inference
    1. Inference steps
    2. Size constraint resolution

3. The language
    1. Examples
    2. Coercions
    3. Binding time analysis

Expressions (I)

Extensional array use [SSSV17]

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$<\eta>$	size

Expressions (I)

Extensional array use [SSSV17]

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

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size

Expressions (I)

Extensional array use [SSSV17]

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

let map : $\_ = \lambda f : \_.\ \lambda n : \texttt{<}\iota\texttt{>}.\ \lambda X : \_.\ \lambda i : \texttt{[}\iota\texttt{]}.\ f\ (X\ i)$

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	true $\mid$ false	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size

Expressions (I)

Extensional array use [SSSV17]

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

$\texttt{let map} : \_ = \lambda f : \_.\ \lambda n : \texttt{<}\iota\texttt{>}.\ \lambda X : \_.\ \lambda i : [\iota].\ f\ (X\ i)$

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

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size

Expressions (I)

Extensional array use [SSSV17]

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

$\texttt{let map} : \_ = \lambda f : \_.\ \lambda n : \texttt{<}\iota\texttt{>}.\ \lambda X : \_.\ \lambda i : \texttt{[}\iota\texttt{]}.\ f\ (X\ i)$

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

$\texttt{let window} : \_ = \lambda k : \texttt{<}\kappa\texttt{>}.\ \lambda X : \texttt{[}\iota + \kappa - 1\texttt{]}\_.\ \lambda i : \texttt{[}\iota\texttt{]}.\ \lambda j : \texttt{[}\kappa\texttt{]}.\ X\ (i + j \triangleright \texttt{[}\_\texttt{]})$

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size

Expressions (I)

Extensional array use [SSSV17]

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

$\texttt{let map} : \_ = \lambda f : \_.\ \lambda n : \texttt{<}\iota\texttt{>}.\ \lambda X : \_.\ \lambda i : [\iota].\ f\ (X\ i)$

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

$\texttt{let window} : \_ = \lambda k : \texttt{<}\kappa\texttt{>}.\ \lambda X : [\iota + \kappa - 1]\_.\ \lambda i : [\iota].\ \lambda j : [\kappa].\ X\ (i + j \triangleright [\_])$

Coercions: post-typing checks [Fla06, HE21]

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size
	$\mid$	$e \triangleright \tau$	coercion

Expressions (I)

Extensional array use [SSSV17]

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

$\texttt{let map} : \_ = \lambda f : \_.\ \lambda n : \texttt{<}\iota\texttt{>}.\ \lambda X : \_.\ \lambda i : \texttt{[}\iota\texttt{]}.\ f\ (X\ i)$

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

$\texttt{let window} : \_ = \lambda k : \texttt{<}\kappa\texttt{>}.\ \lambda X : \texttt{[}\iota + \kappa - 1\texttt{]}\_.\ \lambda i : \texttt{[}\iota\texttt{]}.\ \lambda j : \texttt{[}\kappa\texttt{]}.\ X\ (i + j \triangleright \texttt{[}\_\texttt{]})$

Coercions: post-typing checks [Fla06, HE21]

- Integer upcasting

$$(e : \texttt{int}) \triangleright \texttt{<}\eta\texttt{>}$$
$$(e : \texttt{int}) \triangleright \texttt{[}\eta\texttt{]}$$

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size
	$\mid$	$e \triangleright \tau$	coercion

Expressions (I)

Extensional array use [SSSV17]

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

$\texttt{let map} : \_ = \lambda f:\_.\ \lambda n:\texttt{<}\iota\texttt{>}.\ \lambda X:\_.\ \lambda i:\texttt{[}\iota\texttt{]}.\ f\ (X\ i)$

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

$\texttt{let window} : \_ = \lambda k:\texttt{<}\kappa\texttt{>}.\ \lambda X:\texttt{[}\iota+\kappa-1\texttt{]}\_.\ \lambda i:\texttt{[}\iota\texttt{]}.\ \lambda j:\texttt{[}\kappa\texttt{]}.\ X\ (i+j \triangleright \texttt{[}\_\texttt{]})$

Coercions: post-typing checks [Fla06, HE21]

- Integer upcasting

$$(e : \texttt{int}) \triangleright \texttt{<}\eta\texttt{>}$$
$$(e : \texttt{int}) \triangleright \texttt{[}\eta\texttt{]}$$

- Size conversion

$$(e : \tau) \triangleright \tau', \text{ if } \tau \approx \tau'$$

$\approx$: Size ignoring comparison

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x:\tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size
	$\mid$	$e \triangleright \tau$	coercion

Expressions (II)

Fast Fourier Transform

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$<\eta>$	size
	$\mid$	$e \triangleright \tau$	coercion

Expressions (II)

Fast Fourier Transform

$\texttt{val gft} : \forall\iota.\ [\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = \iota^2$

$\texttt{val fft} : \forall\iota.\ ([\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}) \to [2\iota]\,\texttt{cpx} \to [2\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

e	::=		*Expressions*
	|	x	variable
	|	$e\ e$	application
	|	$\lambda x : \tau .\ e$	abstraction
	|	`true` | `false`	boolean
	|	n	integer
	|	$\langle \eta \rangle$	size
	|	$e \triangleright \tau$	coercion

Expressions (II)

Fast Fourier Transform

$\texttt{val gft} : \forall\iota.\ [\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = \iota^2$

$\texttt{val fft} : \forall\iota.\ ([\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}) \to [2\iota]\,\texttt{cpx} \to [2\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

$\texttt{let dft} : \underline{\ \ } = \ldots$

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$<\eta>$	size
	$\mid$	$e \triangleright \tau$	coercion

Expressions (II)

Fast Fourier Transform

$\texttt{val gft} : \forall\iota.\ [\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = \iota^2$

$\texttt{val fft} : \forall\iota.\ ([\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}) \to [2\iota]\,\texttt{cpx} \to [2\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

$\texttt{let dft} : \_ = \texttt{fix}\ f : \forall\iota.\ [\iota]\,\_ \to [\iota]\,\_ = \lambda X : \_.$

$\quad \ldots$

Polymorphic recursion [Mee83, Myc84]

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$<\eta>$	size
	$\mid$	$e \triangleright \tau$	coercion
	$\mid$	$\texttt{fix}\ x : \sigma = e$	fix-point

Expressions (II)

Fast Fourier Transform

$\texttt{val gft} : \forall\iota.\ [\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = \iota^2$

$\texttt{val fft} : \forall\iota.\ ([\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}) \to [2\iota]\,\texttt{cpx} \to [2\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

$\texttt{let dft} : \_ = \texttt{fix}\ f : \forall\iota.\ [\iota]\,\_ \to [\iota]\,\_ = \lambda X : \_.$

$\quad \ldots$

Polymorphic recursion [Mee83, Myc84]

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$<\eta>$	size
	$\mid$	$e \triangleright \tau$	coercion
	$\mid$	$\texttt{fix}\ x : \sigma = e$	fix-point
	$\mid$	$\texttt{let}\ x : \sigma = e\ \texttt{in}\ e$	local def

Fast Fourier Transform

val gft : $\forall\iota.\ [\iota]\,\mathrm{cpx} \to [\iota]\,\mathrm{cpx}$ $\mathcal{O}(\iota) = \iota^2$

val fft : $\forall\iota.\ ([\iota]\,\mathrm{cpx} \to [\iota]\,\mathrm{cpx}) \to [2\iota]\,\mathrm{cpx} \to [2\iota]\,\mathrm{cpx}$ $\mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

```
let dft : _ = fix f:∀ι. [ι]_ → [ι]_ = λX:_.
    let size ν = <ι>/2 in
    ...
```

Polymorphic recursion [Mee83, Myc84]

Local size existential quantification

e	::=		*Expressions*
	|	x	variable
	|	$e\ e$	application
	|	$\lambda x : \tau.\ e$	abstraction
	|	true | false	boolean
	|	n	integer
	|	$<\eta>$	size
	|	$e \triangleright \tau$	coercion
	|	fix $x : \sigma = e$	fix-point
	|	let $x : \sigma = e$ in e	local def
	|	let size $\iota = e$ in e	size def

Expressions (II)

Fast Fourier Transform

```
val gft : ∀ι. [ι] cpx → [ι] cpx                                  𝒪(ι) = ι²
val fft : ∀ι. ([ι] cpx → [ι] cpx) → [2ι] cpx → [2ι] cpx          𝒪(ι) = 2𝒪_f + 2ι
```

```
let dft : _ = fix f : ∀ι. [ι]_ → [ι]_ = λX : _.
    let size ν = <ι>/2 in
    case <ι> ≠ <2ν>
    then ...
    else ...
```

Polymorphic recursion [Mee83, Myc84]

Local size existential quantification

Statically resolved branching

e ::=		*Expressions*
		x	variable
		$e\ e$	application
		$\lambda x : \tau.\ e$	abstraction
		`true` | `false`	boolean
		n	integer
		$<\eta>$	size
		$e \triangleright \tau$	coercion
		`fix` $x : \sigma = e$	fix-point
		`let` $x : \sigma = e$ `in` e	local def
		`let size` $\iota = e$ `in` e	size def
		`case` e `then` e `else` e	cases

Expressions (II)

Fast Fourier Transform

$\texttt{val gft} : \forall\iota.\ [\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = \iota^2$

$\texttt{val fft} : \forall\iota.\ ([\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}) \to [2\iota]\,\texttt{cpx} \to [2\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

$\texttt{let dft} : \_ = \texttt{fix}\ f : \forall\iota.\ [\iota]\,\_ \to [\iota]\,\_ = \lambda X : \_.$
$\quad \texttt{let size}\ \nu = \texttt{<}\iota\texttt{>}/2\ \texttt{in}$
$\quad \texttt{case}\ \texttt{<}\iota\texttt{>} \neq \texttt{<}2\nu\texttt{>}$
$\quad \texttt{then gft}\ X$
$\quad \texttt{else}\ \ldots$

Polymorphic recursion [Mee83, Myc84]

Local size existential quantification

Statically resolved branching

e	::=		*Expressions*
	|	x	variable
	|	$e\ e$	application
	|	$\lambda x : \tau.\ e$	abstraction
	|	true | false	boolean
	|	n	integer
	|	$\texttt{<}\eta\texttt{>}$	size
	|	$e \triangleright \tau$	coercion
	|	$\texttt{fix}\ x : \sigma = e$	fix-point
	|	$\texttt{let}\ x : \sigma = e\ \texttt{in}\ e$	local def
	|	$\texttt{let size}\ \iota = e\ \texttt{in}\ e$	size def
	|	$\texttt{case}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	cases

Expressions (II)

Fast Fourier Transform

$\texttt{val gft} : \forall\iota.\ [\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = \iota^2$

$\texttt{val fft} : \forall\iota.\ ([\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}) \to [2\iota]\,\texttt{cpx} \to [2\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

$$\begin{array}{l}
\texttt{let dft} : _ = \texttt{fix}\ f : \forall\iota.\ [\iota]_ \to [\iota]_ = \lambda X : _. \\
\quad \texttt{let size}\ \nu = \texttt{<}\iota\texttt{>}/2\ \texttt{in} \\
\quad \texttt{case}\ \texttt{<}\iota\texttt{>} \neq \texttt{<}2\nu\texttt{>} \\
\quad \texttt{then gft}\ X \\
\quad \texttt{else fft}\ f\ X
\end{array}$$

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size
	$\mid$	$e \triangleright \tau$	coercion
	$\mid$	$\texttt{fix}\ x : \sigma = e$	fix-point
	$\mid$	$\texttt{let}\ x : \sigma = e\ \texttt{in}\ e$	local def
	$\mid$	$\texttt{let size}\ \iota = e\ \texttt{in}\ e$	size def
	$\mid$	$\texttt{case}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	cases

Polymorphic recursion [Mee83, Myc84]

Local size existential quantification

Statically resolved branching

Expressions (II)

Fast Fourier Transform

val gft : $\forall\iota.\ [\iota]\,$cpx $\to [\iota]\,$cpx $\qquad \mathcal{O}(\iota) = \iota^2$

val fft : $\forall\iota.\ ([\iota]\,$cpx $\to [\iota]\,$cpx$) \to [2\iota]\,$cpx $\to [2\iota]\,$cpx $\qquad \mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

```
let dft : _ = fix f:∀ι. [ι]_ → [ι]_ = λX:_.
    let size ν = <ι>/2 in
    case <ι> ≠ <2ν>
    then gft X
    else fft f (X ▷ [2ν]_)
```

Polymorphic recursion [Mee83, Myc84]

Local size existential quantification

Statically resolved branching

e	::=		*Expressions*
	|	x	variable
	|	$e\ e$	application
	|	$\lambda x : \tau.\ e$	abstraction
	|	true | false	boolean
	|	n	integer
	|	$<\eta>$	size
	|	$e \triangleright \tau$	coercion
	|	fix $x : \sigma = e$	fix-point
	|	let $x : \sigma = e$ in e	local def
	|	let size $\iota = e$ in e	size def
	|	case e then e else e	cases

Expressions (II)

Fast Fourier Transform

$\texttt{val gft} : \forall\iota.\ [\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = \iota^2$

$\texttt{val fft} : \forall\iota.\ ([\iota]\,\texttt{cpx} \to [\iota]\,\texttt{cpx}) \to [2\iota]\,\texttt{cpx} \to [2\iota]\,\texttt{cpx}$ $\qquad \mathcal{O}(\iota) = 2\mathcal{O}_f + 2\iota$

$\texttt{let dft} : \_ = \texttt{fix}\ f : \forall\iota.\ [\iota]\_ \to [\iota]\_ = \lambda X : \_.$
$\quad \texttt{let size}\ \nu = \texttt{<}\iota\texttt{>}/2\ \texttt{in}$
$\quad \texttt{case}\ \texttt{<}\iota\texttt{>} \neq \texttt{<}2\nu\texttt{>}$
$\quad \texttt{then gft}\ X$
$\quad \texttt{else fft}\ f\ (X \triangleright [2\nu]\_) \triangleright [\iota]\_$

Polymorphic recursion [Mee83, Myc84]

Local size existential quantification

Statically resolved branching

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x : \tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	$\texttt{<}\eta\texttt{>}$	size
	$\mid$	$e \triangleright \tau$	coercion
	$\mid$	$\texttt{fix}\ x : \sigma = e$	fix-point
	$\mid$	$\texttt{let}\ x : \sigma = e\ \texttt{in}\ e$	local def
	$\mid$	$\texttt{let size}\ \iota = e\ \texttt{in}\ e$	size def
	$\mid$	$\texttt{case}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	cases

Coercions

Purposes

- Separating array accesses from property checking (bounds, ...)
- Bypass size language expressiveness limitations
- Simplify typing (avoid *ad-hoc* rules)
- While rarely needed (if some primitives are available: `window`, `sample`, ...)

Coercions

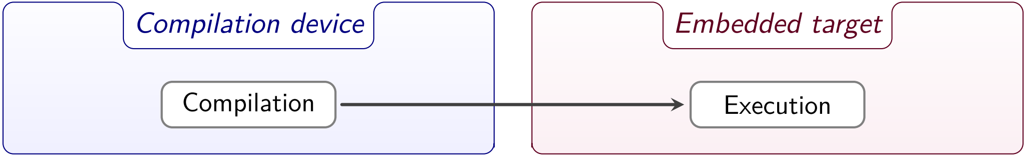
Purposes

- Separating array accesses from property checking (bounds, ...)
- Bypass size language expressiveness limitations
- Simplify typing (avoid *ad-hoc* rules)
- While rarely needed (if some primitives are available: `window`, `sample`, ...)

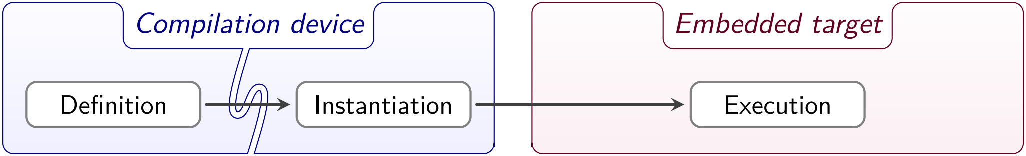
Handling coercion errors

- Defensive code generation
- Advanced formal analysis (abstract interpretation, SMT solvers)
- Binding time restriction (static sizes)

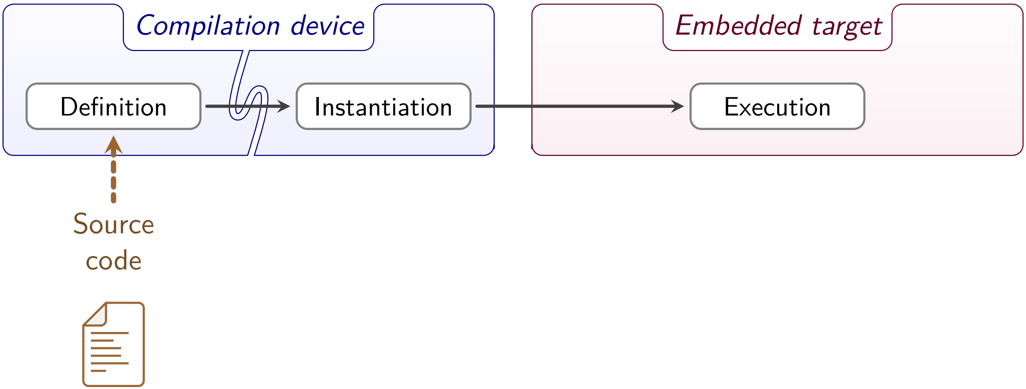
Embedded Synchronous Program Timeline



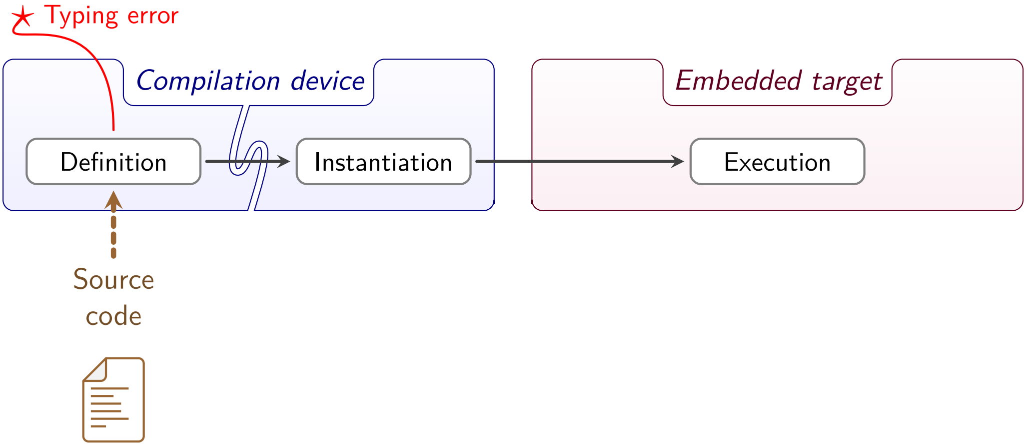
Embedded Synchronous Program Timeline



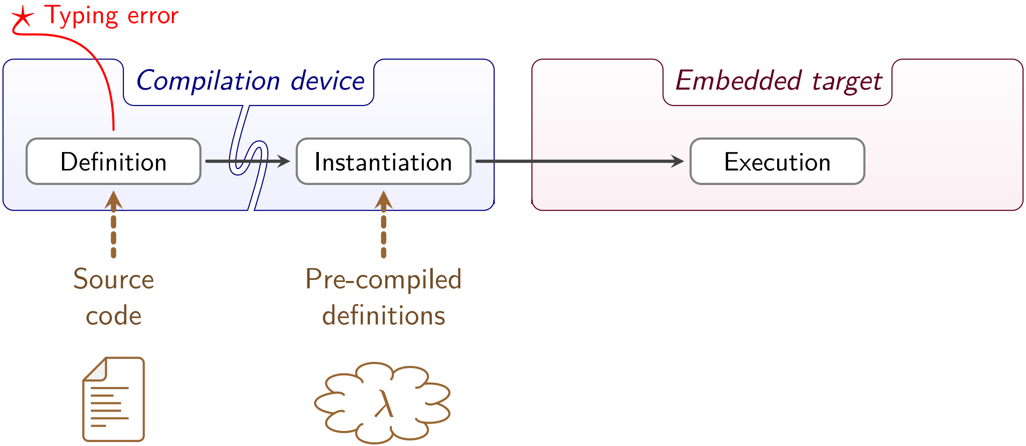
Embedded Synchronous Program Timeline



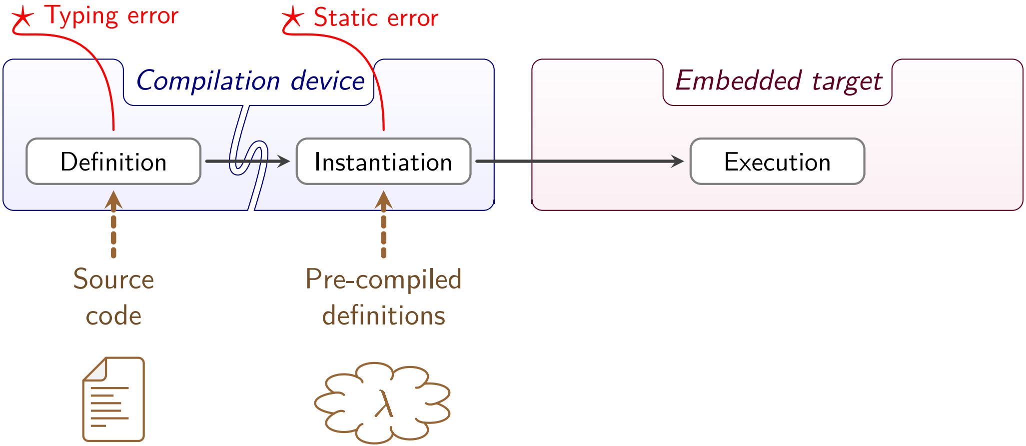
Embedded Synchronous Program Timeline



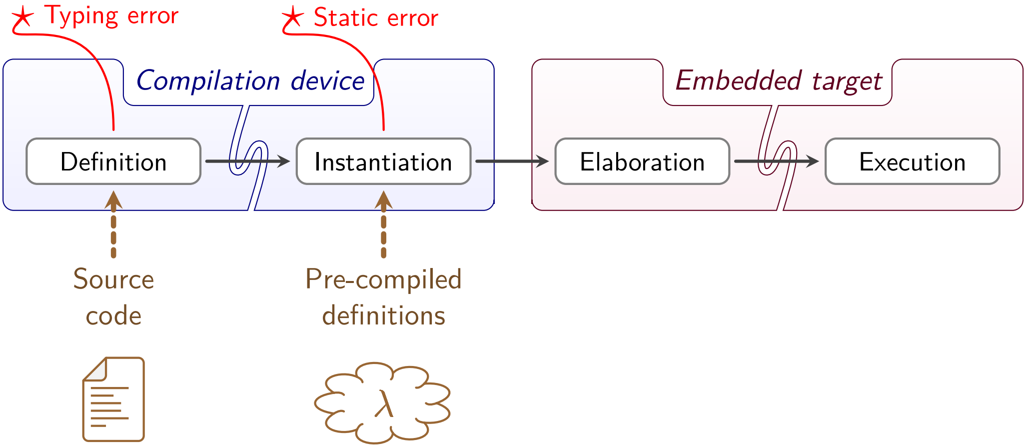
Embedded Synchronous Program Timeline



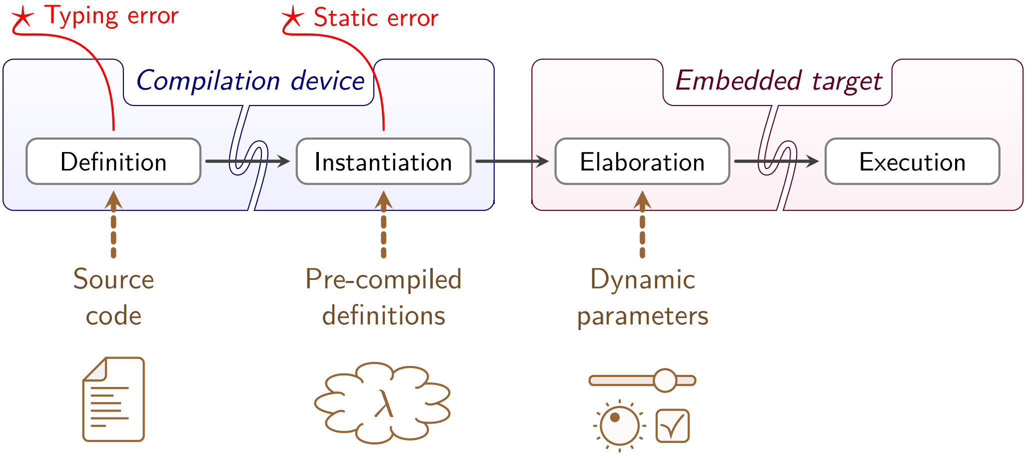
Embedded Synchronous Program Timeline



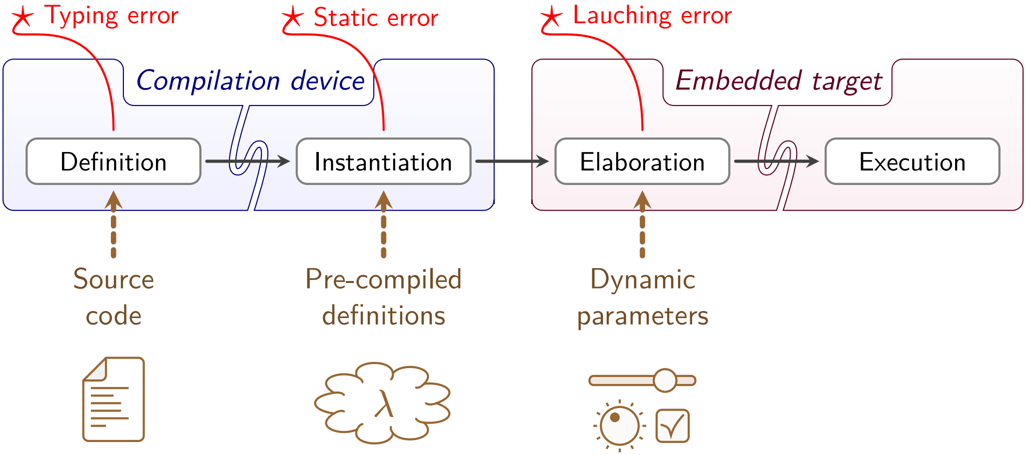
Embedded Synchronous Program Timeline



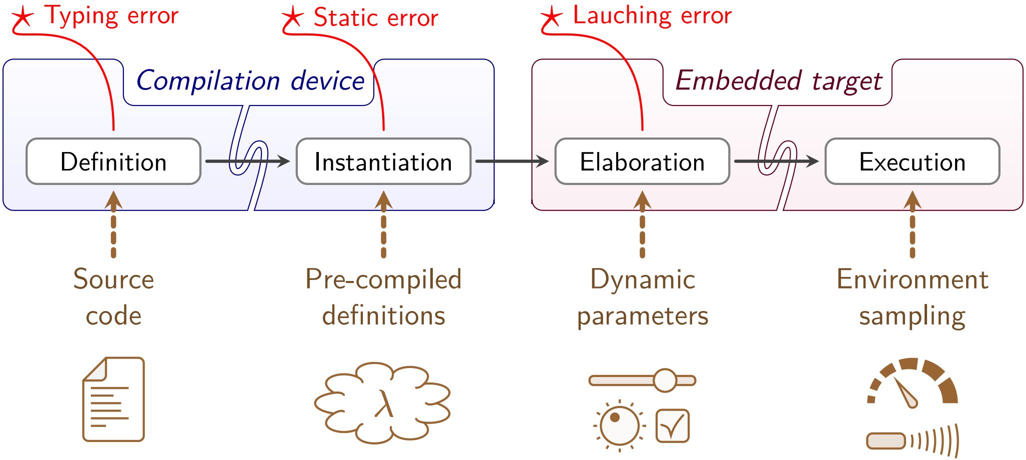
Embedded Synchronous Program Timeline



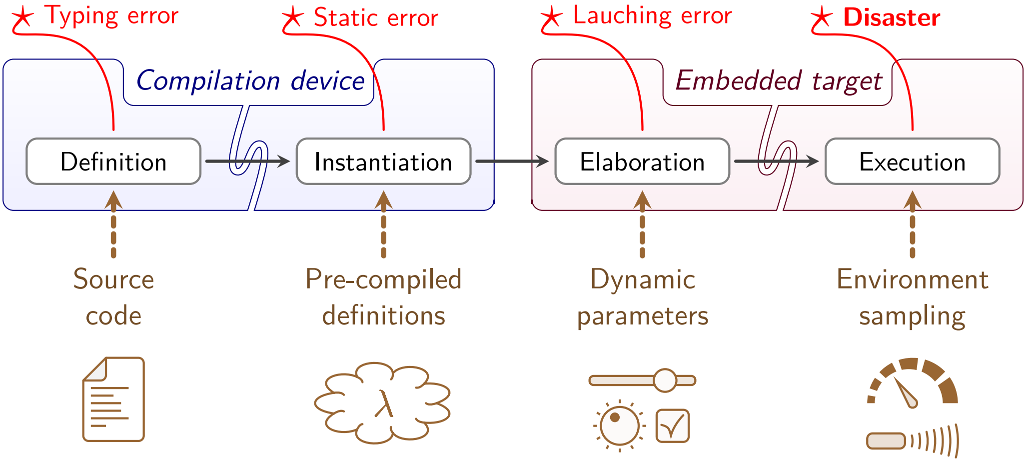
Embedded Synchronous Program Timeline



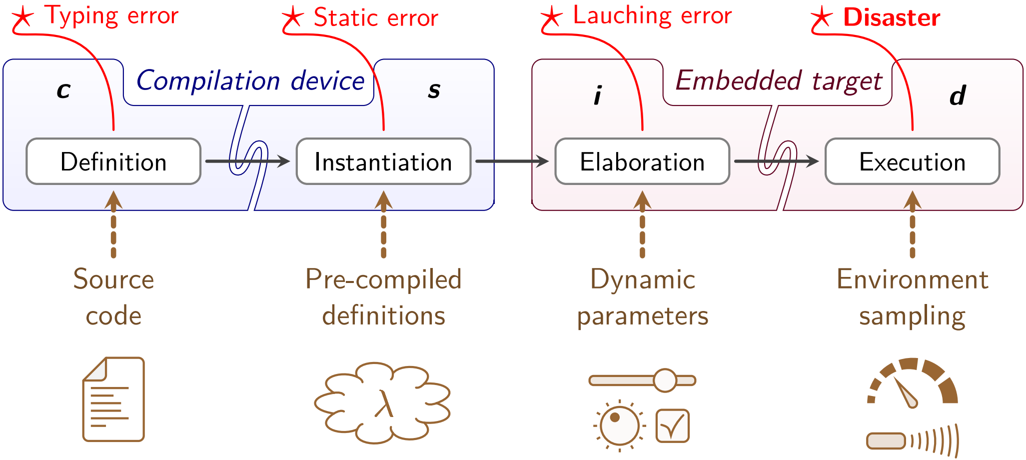
Embedded Synchronous Program Timeline



Embedded Synchronous Program Timeline



Embedded Synchronous Program Timeline



Binding-time analysis [NN88]

Conclusion

Polynomial size polymorphism

- Expressiveness / formal handling trade-off
- Decidable type and size checking
- Size reconstruction heuristics

Conclusion

Polynomial size polymorphism

- Expressiveness / formal handling trade-off
- Decidable type and size checking
- Size reconstruction heuristics

Coercions

- Marking of remaining checks
- Multiple analyses / code generation perspectives

Conclusion

Polynomial size polymorphism

- Expressiveness / formal handling trade-off
- Decidable type and size checking
- Size reconstruction heuristics

Coercions

- Marking of remaining checks
- Multiple analyses / code generation perspectives

Thank you!

References I

[Fla06] Cormac Flanagan.
Hybrid type checking.
In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[HE21] Troels Henriksen and Martin Elsman.
Towards size-dependent types for array programming.
In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 1–14, 2021.

[Mee83] Lambert Meertens.
Incremental polymorphic type checking in B.
In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 265–275, 1983.

References II

[Myc84] Alan Mycroft.
Polymorphic type schemes and recursive definitions.
In *International Symposium on Programming*, pages 217–228. Springer, 1984.

[NN88] Hanne R Nielson and Flemming Nielson.
Automatic binding time analysis for a typed λ-calculus.
Science of computer programming, 10(2):139–176, 1988.

[SSSV17] Artjoms Sinkarovs, Sven-Bodo Scholz, Robert Stewart, and Hans-Nikolai Vießmann.
Recursive array comprehensions in a call-by-value language.
In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages, IFL 2017, Bristol, UK, August 30 - September 01, 2017*, pages 5:1–5:12, 2017.

References III

[XP98] Hongwei Xi and Frank Pfenning.
Eliminating array bound checking through dependent types.
In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, 1998.

Complete syntax

η	$::=$		*Sizes*
	$\mid$	ι	variable
	$\mid$	n	constant
	$\mid$	$\eta \texttt{ + } \eta$	sum
	$\mid$	$\eta \texttt{ * } \eta$	product
τ	$::=$		*Types*
	$\mid$	α	variable
	$\mid$	$\texttt{<}\eta\texttt{>}$	singleton
	$\mid$	$\texttt{[}\eta\texttt{]}$	interval
	$\mid$	$\texttt{int}$	integer
	$\mid$	$\texttt{bool}$	boolean
	$\mid$	$\tau \to \tau$	function
σ	$::=$		*Type scheme*
	$\mid$	τ	simple type
	$\mid$	$\forall\iota.\ \sigma$	size quantif.
	$\mid$	$\forall\alpha.\ \sigma$	type quantif.

e	$::=$		*Expressions*
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$\lambda x{:}\tau.\ e$	abstraction
	$\mid$	$\texttt{true} \mid \texttt{false}$	boolean
	$\mid$	n	integer
	$\mid$	o	operateur
	$\mid$	$e\ {}_{\eta}$	size application
	$\mid$	$e\ {}_{\tau}$	type application
	$\mid$	$\Lambda\iota.\ e$	size abstraction
	$\mid$	$\Lambda\alpha.\ e$	type abstraction
	$\mid$	$\texttt{fix}\ x{:}\sigma = e$	fix-point
	$\mid$	$\texttt{let}\ x{:}\sigma = e\ \texttt{in}\ e$	local definition
	$\mid$	$\texttt{let size}\ \iota = e\ \texttt{in}\ e$	size definition
	$\mid$	$\texttt{<}\eta\texttt{>}$	size
	$\mid$	$e \triangleright \tau$	coercion
	$\mid$	$\texttt{case}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	by case def.
	$\mid$	$.$	dead branch