Flexible Type Analysis

by

Karl Crary & Stephanie Weirich
CMU Cornell

@ ICFP'99

presented by Kwang
11/11/00
4/6/00

Language LX

I. language to define / compute types

\+

language to define/compute values

type expr
τ, c

value expr
e

II. Thm 1 Well-formed type terms is strongly normalizing, confluent, preserves kinds, and is respected by equality.

"can run at compile-time"

Thm 2 Type-checking programs is decidable.

Thm 3 Type-checking is safe.

Why We Need Such A Language?

as a powerful/expressive intermediate language of our compiler.

* ML pgm (typed) ⟶ IL pgm (typed)

Morrisett et al '98

based on System F

- good for compiler debugging
- useful for tag-free g.c.
- useful for code-safety check. etc.

* need more expressive lang. than Morrisett's.

Consider our LET project:

ML pgm ⟶ IL pgm

- need a way to translate properties of ML pgms into those of IL pgms, or v.v.
- how wonderful if such translator can be defined & manipulated.

LX Kinds & Type Exprs

{Constructs for computing types}

kinds $k \rightarrow \text{Type}$ } primitive kinds

$\mid 1$

$\mid k \rightarrow k$

$\mid k \times k$

$\mid k + k$

$\mid j$ kind variable

$\mid \mu j . k$ inductive kind

type exprs $c, \tau \rightarrow \text{int} \mid \text{unit} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau$

$\mid \forall \alpha : k . \tau \mid \exists \alpha : k . \tau \mid \text{rec}_k\, c_1\, c_2$

$\mid \alpha \mid \lambda \alpha : k . c \mid c_1\, c_2$

$\mid \langle c_1, c_2 \rangle \mid c \downarrow 1 \mid c \downarrow 2$

$\mid \text{in}_1^{k_1 + k_2}\, c \mid \text{in}_2^{k_1 + k_2}\, c \mid \text{case}\ c\ \ \alpha_1 . c_1\ \ \alpha_2 . c_2$

$\mid \text{fold}_{\mu j . k}\, c \mid \text{pr}(j, \alpha : k, \beta : j \rightarrow k', c)$

$\mid *$

Noticeable language constructs for type terms are

- $rec_k\ c_1\ c_2$ — recursive type 표현하기

  e.g. type 'a t = A | B of ('a → 'a t) 의

  $'a\ t \equiv rec_{Type}\ (\lambda t.\lambda \alpha.\ c+(\alpha \to t\,\alpha))$

  $int\ t \equiv rec_{Type}\ (\lambda t.\lambda \alpha.\ c+(\alpha \to t\,\alpha))\ int$

- $fold_{\mu j.k}\ c$ — type을 구성하기 / type construction

  inductive kind의 타입 구성하기

  e.g. kind j = A of * | B of * × j 에서

  $A\ int \equiv fold_{\mu j.\_}\ (inj_1\ int)$

- $pr(j, \alpha : k, \beta : j \to k', c)$ — type을 들여보면서 일하기 / type deconstruction

  inductive kind의 타입 들여보기

  e.g. see following slides.

{Recursive type 표현하기}

$rec_k \; c_1 \; c_2$ "an instance of a recursive type"

type 'a t = ι + ('a → 'a t)

→ in ML,

type 'a t = A
| B of ('a → 'a t)

$t = \lambda\alpha{:}k.\, \iota + (\alpha \to t\alpha)$

Such t is

$fix\ \lambda t.\, \lambda\alpha{:}k.\, \iota + (\alpha \to t\alpha)$

written as

$rec_k\ (\lambda t.\, \lambda\alpha{:}k.\, \iota + (\alpha \to t\alpha))$

Hence

int t is written as

$rec_{Type}\ (\lambda t.\, \lambda\alpha{:}Type.\, \iota + (\alpha \to t\alpha))\ int$

(inductive kind 의 타입 구성하기)

$\text{fold}_{\mu j.k}\ C$

type을 가지고 inductive 한 구조 (data structure)
를 만드는 방법

<u>analogy</u> value를 가지고 inductive 한 구조를
만드는 방법.

kind j = A of Type 인데
| B of j × Type

A int B (A int, string) 는 type 으로 구성된

kind j 의 구조물들.

$$\text{"A int"} \equiv \text{fold}_{\mu j.\text{Type}+j\times\text{Type}}\left(\text{inj}_1^{\text{Type}+j\times\text{Type}}\ \text{int}\right)$$

$$\text{"B(A int, string)"} \equiv \text{fold}_{\mu j.\_}\ \overline{\text{inj}_2}\left(\left(\text{fold}_{\mu j.\_}\ \overline{\text{inj}_1}\ \text{int}\right) \times \text{string}\right)$$

(inductive kind의 타입 풀어보면서 타입을 계산하는 primitive rec. fn 정의하기)

$pr(j, \alpha : k, \beta : j \to k'. c)$

is, in high-level form,

$$\text{tfun } \beta(\text{fold}_{\mu j.k}(\alpha : k)) = c \ : k'$$

↑ recursive call to β must be inductive sub-component of $\mu j.k$.

β becomes primitive recursive, hence β always terminates. ⇐

e.g. $\text{tfun alzaal}(\text{fold}_{\mu j.\, 1+\text{Type}\times j}(\alpha : 1 + \text{Type} \times j))$

$$= \text{case } \alpha \text{ of } 1 \Rightarrow \alpha \times \alpha \mid t \times j \Rightarrow (t \times t) \times (\text{alzaal } j)$$

"'a list to 'a×'a list"

LX Value Exprs.

{Constructs for computing values}

$$e \rightarrow n \mid () \mid x \mid \lambda x{:}\tau.\, e \mid e_1 e_2$$

$$\mid \langle e_1, e_2 \rangle \mid e{\downarrow}1 \mid e{\downarrow}2$$

$$\mid in_1^{\tau_1+\tau_2} e \mid in_2^{\tau_1+\tau_2} e \mid \text{case } e \;\; x.e_1 \;\; x.e_2$$

$$\mid \Lambda\alpha{:}k.\, v \mid e[c] \mid \text{fix } f{:}\tau.\, e$$

$$\mid \text{pack } e \text{ as } \exists\alpha{:}k.\tau \text{ hiding } c$$

$$\mid \text{unpack } \langle \alpha, x \rangle = e_1 \text{ in } e_2$$

$$\mid \text{fold}_{rec_k\, c\, c'}\; e \mid \text{unfold } e$$

$$\mid \text{let}_\tau \langle \beta, \gamma \rangle = c \text{ in } e$$

$$\mid \text{let}_\tau (\text{fold } \beta) = c \text{ in } e$$

$$\mid \text{ccase}_\tau\; c \;\; \alpha.e_1 \;\; \alpha.e_2$$

$$v \rightarrow n \mid () \mid x \mid \lambda x{:}\tau.e \mid \text{fix } f{:}\tau.e$$

$$\mid \langle v_1, v_2 \rangle \mid v{\downarrow}1 \mid v{\downarrow}2 \mid in_1^{\tau_1+\tau_2} v \mid in_2^{\tau_1+\tau_2} v$$

$$\mid \Lambda\alpha{:}k.v \mid \text{fold}_{rec_k\, c\, c'}\; v \mid \text{pack } v \text{ as } \exists\alpha{:}k.\tau \text{ hiding } c$$

Introduction		Elimination
$\lambda x : \tau .\ e$	$\tau_1 \rightarrow \tau_2$	$e_1\ e_2$
$\langle e_1, e_2 \rangle$	$\tau_1 \times \tau_2$	$e \downarrow 1 \quad e \downarrow 2$
$\mathrm{in}_1^{\tau_1+\tau_2} e \quad \mathrm{in}_2^{\tau_1+\tau_2} e$	$\tau_1 + \tau_2$	$\mathrm{case}\ e\ \ x_1.e_1\ \ x_2.e_2$
$\Lambda \alpha : k .\ v$	$\forall \alpha : k . \tau$	$e[c]$
$\mathrm{pack}\ e$ $\mathrm{as}\ \exists \alpha : k . \tau$ $\mathrm{hiding}\ c$	$\exists \alpha : k . \tau$	$\mathrm{unpack}\ \langle \alpha, x \rangle = e_1$ $\mathrm{in}\ e_2$
$\mathrm{fold}_{\mathrm{rec}_k\, c\, c'}\ e$	$\mathrm{rec}_k\ c\ c'$	$\mathrm{unfold}\ e$

LX Static Semantics

$$\Delta \rightarrow \emptyset \mid \Delta, j \mid \Delta, \alpha : k$$

$$\Gamma \rightarrow \emptyset \mid \Gamma, x : \tau$$

Judgments

$\Delta \vdash k$	k is a well-formed kind
$\Delta \vdash c : k$	type expr. c has kind k
$\Delta \vdash c_1 = c_2 : k$	type exprs. c_1 & c_2 have the same kind k
$\Delta ; \Gamma \vdash e : \tau$	value expr. e has type τ

Typecase expr.

"$\mathrm{ccase}_\tau\ c\ \alpha.e_1\ \alpha.e_2$"

$$\frac{\Delta \vdash c = \mathrm{in}_1^{k_1+k_2}\, c' : k_1+k_2 \qquad \Delta ; \Gamma \vdash e_1[c'/\alpha] : \tau}{\Delta ; \Gamma \vdash \mathrm{ccase}_\tau\ c\ \alpha.e_1\ \alpha.e_2 : \tau}$$

$$\frac{\begin{array}{c}\Delta, \beta : k_1, \Delta' ; \Gamma[\mathrm{in}_1\, \beta/\alpha] \\ \vdash e_1[\mathrm{in}_1\, \beta/\alpha] : \tau[\mathrm{in}_1\, \beta/\alpha] \\ \Delta, \beta : k_2, \Delta' ; \Gamma[\mathrm{in}_2\, \beta/\alpha] \\ \vdash e_2[\mathrm{in}_2\, \beta/\alpha] : \tau[\mathrm{in}_2\, \beta/\alpha] \\ \Delta, \alpha : k_1+k_2, \Delta' \vdash c = \alpha : k_1+k_2\end{array}}{\Delta, \alpha : k_1+k_2, \Delta' ; \Gamma \vdash \mathrm{ccase}_\tau\ c\ \beta.e_1\ \beta.e_2 : \tau}$$

Refinement (Decomposition) Exprs

$$\text{"}\mathsf{let}_\tau\ \langle \beta, \gamma \rangle = c\ \mathsf{in}\ e\text{"}$$

$$\text{"}\mathsf{let}_\tau\ (\mathsf{fold}_{\mu j.k}\ \beta) = c\ \mathsf{in}\ e\text{"}$$

$$\frac{\Delta \vdash c = \langle c_1, c_2 \rangle : k_1 \times k_2 \qquad \Delta ; \Gamma \vdash e[c_1/\beta][c_2/\gamma] : \tau}{\Delta ; \Gamma \vdash \mathsf{let}_\tau\ \langle \beta, \gamma \rangle = c\ \mathsf{in}\ e : \tau}$$

$$\frac{\begin{array}{c}\Delta, \beta : k_1, \gamma : k_2, \Delta' ; \Gamma[\langle \beta, \gamma \rangle / \alpha] \vdash e[\langle \beta, \gamma \rangle / \alpha] : \tau[\langle \beta, \gamma \rangle / \alpha] \\ \Delta, \alpha : k_1 \times k_2, \Delta' \vdash c = \alpha : k_1 \times k_2\end{array}}{\Delta, \alpha : k_1 + k_2, \Delta' ; \Gamma \vdash \mathsf{let}_\tau\ \langle \beta, \gamma \rangle = c\ \mathsf{in}\ e : \tau}$$