ATTAPL 輪講
2.1 - 2.5
(2006/6/14)

米澤研 M2　佐藤秀明

今回の内容

- Dependent Types
  - 動機づけ
  - λLF：基本的な型システム
  - λLF の性質
  - algorithmic λLF
  - sigma types の導入

動機づけ

Dependent Types とは

- 項から型への関数
  - Vector: Nat 型の項をもらって型を返す関数

    (長さ 5 のベクタ) : Vector 5

- cf. 型から型への関数 (universal types)
  - Pair: 型を 2 つもらって型を返す関数

    (1, false) : Pair Nat Bool

Dependent Product Type (Pi Type)

- 型を項について抽象化したい
  - 例：Vector の初期化関数

    init : Πn:Nat. data → Vector n
    init k : data → Vector k
    init k t : Vector k

- Πx:S.T ＝「項 s:S を受け取って型 [x→s]T を返す関数」
  - ∀X.T ＝「型 A を受け取って型 [X→A]T を返す関数」

Pi Type の使用例

- Vector の操作関数
  - コンストラクタ

    empty : Vector 0

    cons : Πn:Nat. data → Vector n → Vector (n+1)

    ↓

    hd : data ∧ tl : Vector 5 ⇒ cons 5 hd tl : Vector 6
  - 要素取得

    first : Πn:Nat. Vector(n+1) → data

empty な vector を型システムで reject できる
↓
実行時の size check が不要

sprintf の表現例

- データ列の長さは format string の内容に依存

  sprintf : Πf:Format. Data(f) → String

- Data(f) は再帰的に構築可能

  Data("%d" :: cs) = Nat * Data(cs)
  Data("%s" :: cs) = String * Data(cs)
  Data(c :: cs) = Data(cs)
  Data([]) = Unit

  - これを現実の言語上でどう処理するかは challenging

Pi Type と Arrow Type の関係

- arrow type は pi type の特別なケース

> x が T の中に自由に出現しないとき
> $\Pi x{:}S.T \quad = \quad S \rightarrow T$

  - 型 T が項 x に依存 (depend) しない場合は arrow と同じ

- 例：

$\Pi x{:}Nat.Nat \quad = \quad Nat \rightarrow Nat$

$\Pi x{:}Nat.Vector\ x \quad \neq \quad Nat \rightarrow Vector\ x$

- 可読性のため、以降も arrow type を併用する

復習：Existential Types

- t : ∃X.T
  - ある型 S から型 [X→S]T への関数
- (*X, t) : (∃X, T) と表記することもある
  - ある型 S と型 [X→S]T の pair の集合
  - module 機構を説明しやすい
    - 型 X が実際には型 S であることをプログラマから隠蔽

Dependent Sum Type (Sigma Type)

- 項に depend する Existential types を導入
  - ある項 s から型 [x→s]T への関数

$$s:S \text{ かつ } t:T(s) \text{ のとき } (s, t) : \Sigma x:S.T(x)$$

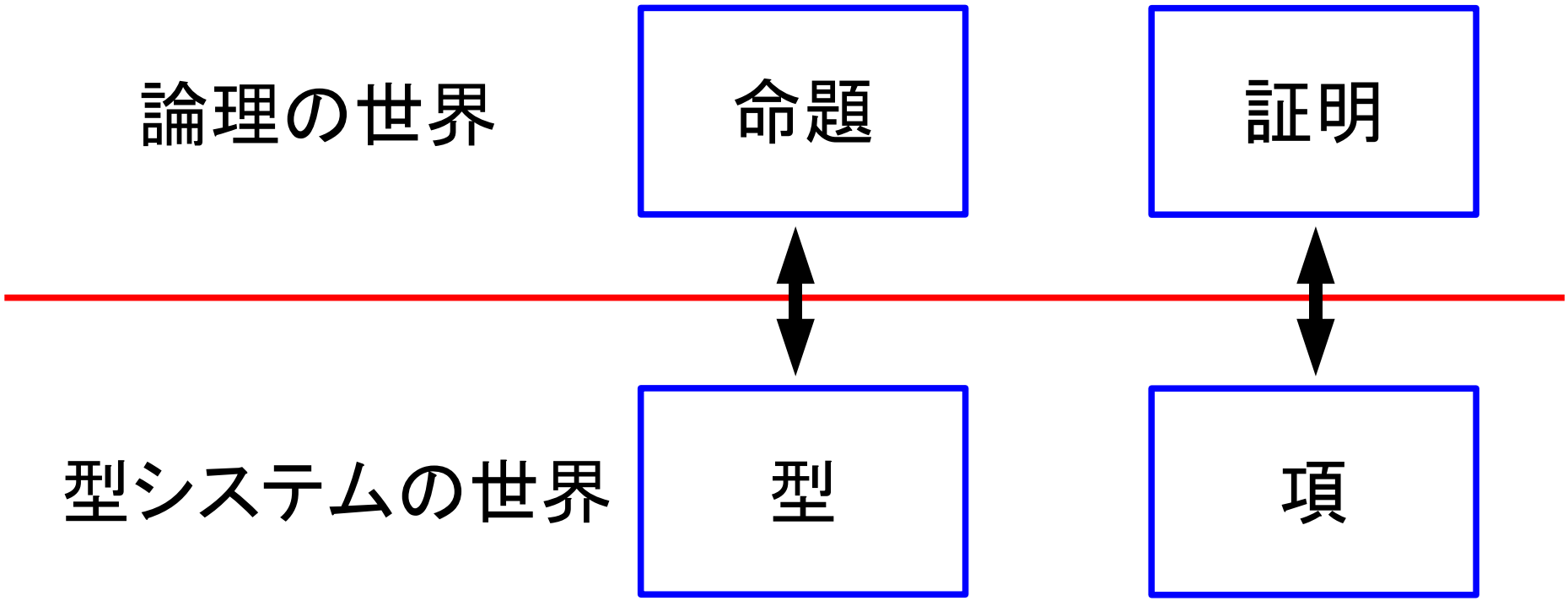
  - 依存関係がない場合はただの pair

x が T の中に自由に出現しないとき

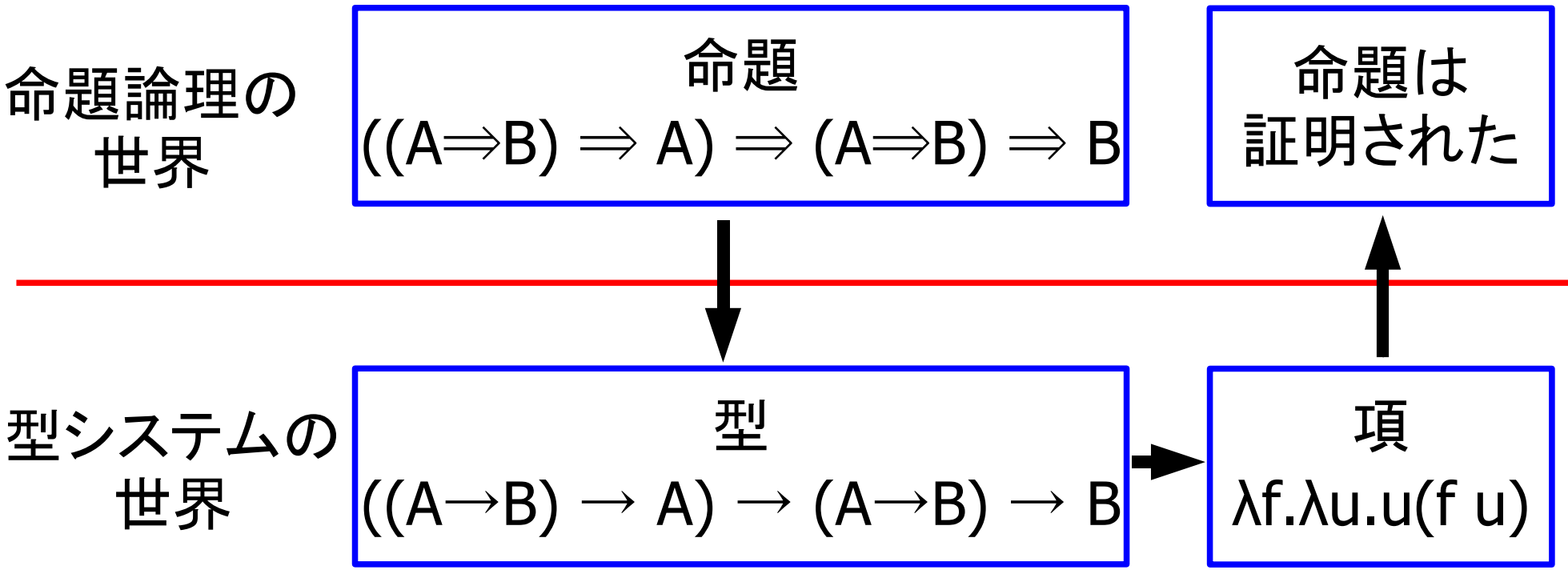
$$\Sigma x:S.T \quad = \quad S \times T$$

Curry-Howard 対応

- 論理と型システムとの美しい対応
  - 命題 A が証明 B をもつ ⇔ 型 A の項 B が存在する



Implication との対応



- A⇒B の証明 ＝ a:A を b:B に変換する関数

一階述語論理との対応

- $\forall x{:}S.T(x) \longleftrightarrow \Pi x{:}S.T(x)$
  - 例：vector の m 番目の要素を取得

    $\Pi m{:}Nat.\Pi n{:}Nat.$

    $\quad LessThan(m,n) \rightarrow Vector(n) \rightarrow Data$
  - 命題と型を自由に混ぜて書いている
- $\exists x{:}S.T(x) \longleftrightarrow \Sigma x{:}S.T(x)$
  - 例：ある二項演算が結合的であることの証明

    $\Sigma m{:}T \rightarrow T \rightarrow T.\ \Pi x{:}T.\Pi y{:}T.\Pi z{:}T.$

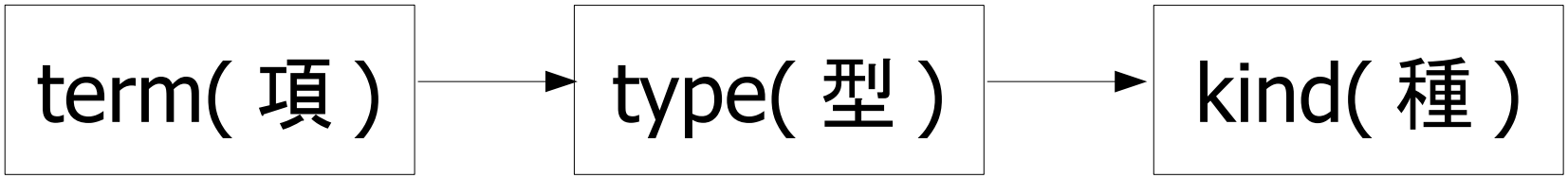
    $\quad Equal\ (m(x,m(y,z)))\ (m(m(x,y),z))$
  - この型をもつ項は
    (二項演算 m, m についての証明) の pair

Kind の導入

- kind: 型の型 (see TAPL, chapter 29)

  Vector :: Nat → *
  Vector 5 :: *
  (長さ 5 のベクタ) : Vector 5

  - 「 :: 」は kinding
    - cf. 「 : 」は typing
  - 「 * 」は proper な (項への依存を解消した) 型の kind



Logical Frameworks(1)

- syntax と証明システムを論理体系として構築
  - 対象言語 O をメタ言語 M で表現
  - 例 : 単純型付ラムダ計算 O に対する M 上の型チェッカ

Ty :: *　　(O での型を表現する M の項)

Tm :: Ty → *　　(M の型表現を O の型へ変換)

base : Ty　　(M における O の base type)

arrow : Ty → Ty → Ty　　(M で arrow 型の型表現を構成)

lam : ΠA:Ty.ΠB:Ty.(Tm A → Tm B) → Tm(arrow A B)

(O でのラムダ抽象を M で表現)

app : ΠA:Ty.ΠB:Ty.Tm(arrow A B) → Tm A → Tm B

(O での関数適用を M で表現)

Logical Frameworks(2)

- 型 A についての恒等関数

idA = lam A A (λx:Tm A.x) ⟷ idA = λx:A.x

- 型 A についてのチャーチ数の 2

```
two = λA:Ty.
  lam A (arrow (arrow A A) A) (λx:Tm A.
    lam _ _ (λf:Tm(arrow A A).
      app _ _ f
        (app _ _ f x)))
```

⟷

```
two =
  λx:A.
    λf:A→A.
      f
        (f x)
```

  - 一部引数は省略

- 型を項のレベルで自由に定義

λLF：基本的な型システム

First-Order Dependent Types (λLF)

- 単純型付を拡張
  - arrow type を pi type に置き換え
  - kind を導入
  - definitional equivalence を定義 (後述)
- syntax と rules は Figure 2-1(p. 51) に
  - 3 種類の rules が互いに依存
    →各種性質の証明も互いに依存した構造帰納法になる
    - well-formed kinds
    - kinding
    - typing

Syntax

- 型変数は初期 context でのみ宣言可能
  - プログラムの途中で新たに宣言できない
- 型は arrow type なし
  - 全て pi type で表現可能
- kind も arrow kind なし
  - Π を用いて表記

T ::=
- X
- Πx:T.T
- T t

K ::=
- *
- Πx:T.K

Typing Rules

- pi 抽象 / 適用のルール

$$\frac{\Gamma \vdash S :: * \qquad \Gamma, x{:}S \vdash t : T}{\Gamma \vdash \lambda x{:}S.t : \Pi x{:}S.T} \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : \Pi x{:}S.T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : [x \rightarrow t_2]T} \text{(T-APP)}$$

Kinding Rules

- 項に依存する型は初期 context でのみ導入可能
  - プログラム中で導入できるのは arrow type のみ

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x{:}T_1 \vdash T_2 :: *}{\Gamma \vdash \Pi x{:}T_1.T_2 :: *} \text{(K-PI)}$$

$\Pi x{:}T_1.T_2$ = $T1 \rightarrow T2$; $*$ = proper type

Well-Formed Kinds

- higher-order type operator を禁止
  - pi 抽象が引数にとれるのは proper type な項のみ
  - (WF-STAR) と (WF-PI) で制限

kind は　$\Pi x_1{:}T_1. \ldots .\Pi x_n{:}T_n.*$　の形をとり、

かつ $T_1 :: *, \ldots, T_n :: *$

Equivalence Rules

- 2 つの型が等しいことをどう判定するか？
  - 型の中に項が出現
- つまり：2 つの項が等しいことをどう判定するか？
  - 項以外の部分は構造等価でじゅうぶん
- 判定器を強力にしすぎるとundecidable になる
  - 任意の計算 / 証明が必要

等価性判定の例

- 簡単：β 等価

  $(\lambda x{:}S.x)\ z \equiv z$

- 簡単：算術演算

  Vector (3 + 4) ≡ Vector 7

- 困難：非現実的な証明作業

  f:Nat→Nat 、内部仕様は unknown
  x:Nat 、値は unknown
  任意の x について f x = 7 が示せれば
  Vector (f x) ≡ Vector 7 が得られる

λLF の等価性判定ポリシー

- 判定の本質は項どうしの β/η 等価性
  - 等価性判定ルールは Figure 2-2(p. 52) に

$$\frac{\Gamma, x{:}S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x{:}S.t)\ s \equiv [x \to s]t : [x \to s]T} \text{ (Q-BETA)}$$

$$\frac{\Gamma \vdash t : \Pi x{:}S.T \qquad x \text{ not in } FV(t)}{\Gamma \vdash \lambda x{:}S.(t\ x) \equiv t : \Pi x{:}S.T} \text{ (Q-ETA)}$$

- kinding 、typing の中で等価性ルールを適宜利用
  - Figure 2-1 の (K-CONV) と (T-CONV)

λLF の性質

Basic Properties(1)

- 特殊な notation を導入
  - 任意の judgements を Γ |- J と表記
  - Γ |- K かつ Γ |- K' を Γ |- K, K' と表記

- Lemma[Permutation and Weakening]

  Γ⊆Δ のとき、Γ |- J ⇒ Δ |- J

- Lemma[Substitution]

  Γ, x:S, Δ |- J　かつ　Γ |- s : S　ならば

  Γ, [x→s]Δ |- [x→s]J

Basic Properties(2)

- Lemma[Agreement]

  1. Γ |- T :: K ならば Γ |- K
  2. Γ |- t : T ならば Γ |- T :: *
  3. Γ |- K≡K' ならば Γ |- K, K'
  4. Γ |- T≡T' :: K ならば Γ |- T, T' :: K
  5. Γ |- t≡t' :: T ならば Γ |- t, t' : T

Evaluation Rules

- λLF についての β-reduction を定義

$$\frac{t \rightarrow_\beta t'}{\lambda x{:}T.t \rightarrow_\beta \lambda x{:}T.t'} \quad \text{(BETA-ABS)}$$

$$\frac{t_1 \rightarrow_\beta t_1'}{t_1\ t_2 \rightarrow_\beta t_1'\ t_2} \quad \text{(BETA-APP1)}$$

$$\frac{t_2 \rightarrow_\beta t_2'}{t_1\ t_2 \rightarrow_\beta t_1\ t_2'} \quad \text{(BETA-APP2)}$$

$$(\lambda x{:}T_1.t_1)\ t_2 \rightarrow_\beta [x \rightarrow t_2]\ t_1 \quad \text{(BETA-APPABS)}$$

Strong Normalization

- Theorem[Strong Normalization]

  $\Gamma \mid\!- t_1 : T$ ならば、

  $t_i \rightarrow_\beta t_{i+1}$ となる無限列 $(t_i)_{i \geqq 1}$ はない

- 証明

  1.λLF の型表現のみを単純型付きのものへ単純化 (N)

  - $N(X) = X$
  - $N(\Pi x{:}S.T) = N(S) \rightarrow N(T)$
  - $N(T\ t) = N(T)$

  2.$\Gamma \mid\!- t : T \Rightarrow N(\Gamma) \mid\!- N(t) : N(T)$ を示す

  3.単純型付きの Strong Normalization から題意を得る

Confluence

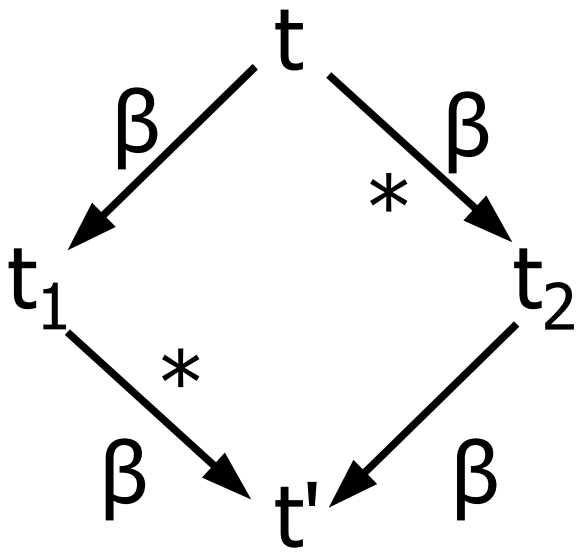
- Theorem[Confluence]

  関係 $\rightarrow_{\beta}$ は合流性をもつ

- 証明

  1.local confluence(下図) を証明

  2. 1. と strong normalization から題意を導く



Algorithmic λLF

Algorithmic λLF

- 適用するタイミングが不定の rule が存在
  - 実装が非決定的になる
    - (K-CONV) 、(T-CONV) 、...
- 実装に即した定義に書き換える必要性
  - syntax の構造に沿って決定的に動くアルゴリズム

- algorithmic λLF の定義は Figure 2-3, 2-4(p. 58) に

Kind Formation/Kinding/Typing

- 適用する rule が syntax の形から 1 つに決まる
- 等価性判定の実行箇所を限定
  - (KA-APP) と (TA-APP)

t ::=

x	(TA-VAR)
λx:T.t	(TA-ABS)
t t	(TA-APP)

T ::=

X	(KA-VAR)
Πx:T.T	(KA-PI)
T t	(KA-APP)

K ::=

*	(WFA-STAR)
Πx:T.K	(WFA-PI)

Weak Head Reduction

- Evaluation Rules を制限
  - head の位置でのみ β-reduction が起こる

$$\frac{t_1 \rightarrow_{wh} t_1'}{t_1\ t_2 \rightarrow_{wh} t_1'\ t_2} \qquad \text{(WH-APP1)}$$

$$(\lambda x{:}T_1.t_1)\ t_2 \rightarrow_{wh} [x \rightarrow t_2]\ t_1 \quad \text{(WH-APPABS)}$$

- Theorem[Weak Head Normal Forms]

  Γ |- t : T ならば、$t \rightarrow_{wh}{}^* t' \not\rightarrow_{wh}$ となる t' が唯一存在。

  - 証明 :[Strong Normalization] と
    $\rightarrow_{wh}$ の決定性から明らか。

Equivalence Rules

- 項の weak head normal form を利用
  - weak head normal form を求める関数 whnf を用いる
- well-typed な項 / 型が渡されることを仮定
  - equivalence rules 自体は well-typedness を確認しない

Well-Formed Context

- algorithmic λLF は変数の適切な型付を検査しない
  - (K-VAR) vs. (KA-VAR) と (T-VAR) vs. (TA-VAR)
- 本来は well-formed context の rule が必要
  - 初期 context 内のおかしな型付をはじく

$$\frac{}{\vdash \Phi} \qquad \text{(WFA-EMPTY)}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash T :: *}{\vdash \Gamma, x : T} \qquad \text{(WFA-TM)}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash K}{\vdash \Gamma, X :: K} \qquad \text{(WFA-TY)}$$

Soundness

- Lemma[Soundness of Algorithmic λLF]

  1.Γ |→ K ならば Γ |- K

  2.Γ |→ T::K ならば Γ |- T::K

  3.Γ |→ t:T ならば Γ |- t:T

  4.Γ |→ K,K' かつ Γ |→ K≡K' ならば Γ |- K≡K'

  5.Γ |→ T,T'::K かつ Γ |→ T≡T' ならば Γ |- T≡T'::K

  6.Γ |→ t,t':T かつ Γ |→ t≡t' ならば Γ |- t≡t':T

  ただし各々の場合において |→ Γ を仮定する。

- 証明
  - algorithmic λLF の導出に関する帰納法。

Completeness(1)

- Lemma[Completeness of Algorithmic λLF]

  1.Γ |- K ならば Γ |→ K

  2.Γ |- T::K ならば、ある K' が存在して
  Γ |→ T::K' かつ Γ |→ K≡K' かつ Γ |→ K'

  3.Γ |- t:T ならば、ある T' が存在して
  Γ |→ t:T' かつ Γ |→ T≡T' かつ Γ |→ T'::*

  4.Γ |- t≡t':T ならば Γ |→ t≡t'

  5.Γ |- T≡T'::K ならば Γ |→ T≡T'

Completeness(2)

- 証明

  1.λLF の各 rule を algorithmic λLF に写せることを示す。

  2.λLF の導出に関する帰納法。

- λLF の rule を algorithmic λLF へ写す例
  - (Q-TRANS)

    $\Gamma \vdash t_1,t_2,t_3:T$ かつ $\Gamma \vdash\!\rightarrow t_1 \equiv t_2$ かつ $\Gamma \vdash\!\rightarrow t_2 \equiv t_3$

    ならば $\Gamma \vdash\!\rightarrow t_1 \equiv t_3$
  - (Q-APP)

    $\Gamma \vdash t_1 t_2, s_1 s_2:T$ かつ $\Gamma \vdash\!\rightarrow t_1 \equiv s_1$ かつ $\Gamma \vdash\!\rightarrow t_2 \equiv s_2$

    ならば $\Gamma \vdash\!\rightarrow t_1 t_2 \equiv s_1 s_2$

Termination(1)

- ill-typed な項の weak head reduction は止まらない
  - 例 : $\lambda x{:}A.(x\ x)\ \lambda x{:}A.(x\ x)$
  - algorithmic λLF の停止性を脅かす？
- 実際には ill-typed な項を心配する必要はない
  - well-typed な項のみ等価性検査するよう制限されている

Termination(2)

- Theorem[Termination of Typechecking]
  algorithmic λLF の型検査アルゴリズムは停止する。
- 証明：
  - well-typed な 2 つの項 t_1 、t_2 について、等価性検査
    $\Gamma \mid\rightarrow t_1 \equiv t_2$ の導出過程が停止することを示す。
- アイデア：導出を辿るごとに単調減少する値の存在
  - 項のサイズ
  - 現在の項から normal form までの β-reduction の回数

Preservation

- Theorem[Preservation]

  Γ |- t : T かつ $t \rightarrow_{\beta} t'$ ならば Γ |- t' : T

- 詳細な証明は省略
  - 重要なケースは (BETA-APPABS) のみ
    - [Substitution][Agreement][Soundness][Completeness] を利用

Sigma Types の導入

Sigma Type (再掲)

- 項に depend する existential type を導入
  - ある項 s から型 [x→s]T への関数

$$s:S \text{ かつ } t:T(s) \text{ のとき } (s, t) : \Sigma x:S.T(x)$$

  - 依存関係がない場合はただの pair

x が T の中に自由に出現しないとき

$$\Sigma x:S.T \quad = \quad S \times T$$

- pi type ＝ 項に depend する universal type
  - 依存関係がない場合はただの arrow type

λLF with Sigma Types

- pair の構築 / 各要素取得を表現する要素を追加
  - ただの pair type も含めて sigma type で表現

t ::= ...
　　(t, t:Σx:T.T)
　　t.1
　　t.2

T ::= ...
　　Σx:T.T

- 新たに追加された要素は Figure 2-5(p. 62) に

明示的な型指定

- pair の 2 番目の要素は明示的な型指定が必要

  t ::= ...

  (t, t:Σx:T.T)

  - どの項を module の中に隠蔽すべきかを人間が指示

    S :: T→* かつ x : T かつ y : S x のとき、
    (x, y) の型は Σz:T.S z にも Σz:T.S x にもできる

Pair の Weak Head Reduction

- evaluation rules を制限

$$(t_1, t_2{:}T).i \rightarrow_\beta t_i \qquad \text{(BETA-PROJPAIR)}$$

$$\frac{t \rightarrow_\beta t'}{t.i \rightarrow_\beta t'.i} \qquad \text{(BETA-PROJ)}$$

weak head reduction

$$\frac{t_1 \rightarrow_\beta t_1'}{(t_1, t_2{:}T) \rightarrow_\beta (t_1', t_2{:}T)} \qquad \text{(BETA-PAIR1)}$$

$$\frac{t_2 \rightarrow_\beta t_2'}{(t_1, t_2{:}T) \rightarrow_\beta (t_1, t_2'{:}T)} \qquad \text{(BETA-PAIR2)}$$

algorithmic なλLF with Sigma Types

- 定義は Figure 2-6(p. 63) に
  - weak head reduction を term equivalence に利用
- 重要な性質は sigma type を入れても成り立つ
  - [Soundness]
  - [Completeness]
  - [Termination]