λ-Calculus

General Recursion and Polymorphism

陳亮廷 Chen, Liang-Ting

Formosan Summer School on Logic, Language, and Computation (FLOLAC) 2022

Institute of Information Science, Academia Sinica

PCF— System of Recursive Functions

PCF: $\lambda_\rightarrow$ with naturals and general recursion

Definition 1 (Terms)

Additional term formation rules are added to $\lambda_\rightarrow$ as follows.

$$\frac{}{\texttt{zero} : \texttt{Term}_{\mathsf{PCF}}} \qquad \frac{M : \texttt{Term}_{\mathsf{PCF}}}{\texttt{suc}\ M : \texttt{Term}_{\mathsf{PCF}}}$$

$$\frac{L : \texttt{Term}_{\mathsf{PCF}} \qquad M : \texttt{Term}_{\mathsf{PCF}} \qquad N : \texttt{Term}_{\mathsf{PCF}} \qquad x \in V}{\texttt{ifz}(M; x.\, N)\ L}$$

$$\frac{M : \texttt{Term}_{\mathsf{PCF}} \qquad x \in V}{\texttt{fix}\, x.\, M : \texttt{Term}_{\mathsf{PCF}}}$$

PCF: Typing rules

Definition 2

Additional term typing rules are added to $\lambda_{\rightarrow}$ as follows.

$$\frac{}{\Gamma \vdash \mathtt{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \mathtt{suc}\ M : \mathbb{N}}$$

$$\frac{\Gamma \vdash L : \mathbb{N} \qquad \Gamma \vdash M : \tau \qquad \Gamma, x : \mathbb{N} \vdash N : \tau}{\Gamma \vdash \mathtt{ifz}(M; x.\,N)\ L : \tau}$$

$$\frac{\Gamma, x : \tau \vdash M : \tau}{\Gamma \vdash \mathtt{fix}\, x.\, M : \tau}$$

- Substitution for **PCF** is defined similarly.
- Substitution respects typing judgements, i.e. $\Gamma \vdash N : \tau$ and $\Gamma, x : \tau \vdash M : \sigma$, then $\Gamma \vdash M[N/x] : \sigma$.

PCF: Dynamics

β-conversion for **PCF** is extended with three rules

$$
\begin{aligned}
\mathtt{fix}\, x.\, M &\longrightarrow_\beta M[\mathtt{fix}\, x.\, M/x] \\
\mathtt{ifz}(M; x.\, N)\ \mathtt{zero} &\longrightarrow_\beta M \\
\mathtt{ifz}(M; x.\, N)\ (\mathtt{suc} M) &\longrightarrow_\beta N[M/x]
\end{aligned}
$$

Similarly, a β-reduction $\longrightarrow_{\beta 1}$ extends $\longrightarrow_\beta$ to all parts of a term and $\longrightarrow_{\beta *}$ indicates finitely many β-reductions.

Theorem 3

PCF enjoys type safety.

Example

A term which never terminates can be defined easily.

$$
\begin{aligned}
& \texttt{fix}\, x.\, x && \longrightarrow_{\beta 1} x[\texttt{fix}\, x.\, x/x] \\
\equiv\ & \texttt{fix}\, x.\, x && \longrightarrow_{\beta 1} x[\texttt{fix}\, x.\, x/x] \\
\equiv\ & \texttt{fix}\, x.\, x && \longrightarrow_{\beta 1} x[\texttt{fix}\, x.\, x/x] \\
\equiv\ & \ldots
\end{aligned}
$$

Example: Predecessor and negation

$$\begin{aligned} \texttt{pred} &:= \lambda n : \mathbb{N}.\, \texttt{ifz}(\texttt{zero}; x.\, x)\ n && : \mathbb{N} \to \mathbb{N} \\ \texttt{not} &:= \lambda n : \mathbb{N}.\, \texttt{ifz}(\texttt{suc zero}; x.\, \texttt{zero})\ n && : \mathbb{N} \to \mathbb{N} \end{aligned}$$

Exercise

Evaluate the following terms to their normal forms.

1. `pred zero`
2. `pred (suc suc suc zero)`
3. `not (suc suc zero)`

F — Polymorphic Typed λ-Calculus

Polymorphic types

Given type variables $\mathbb{V}$, $\tau : \mathsf{Type}$ is defined by defined by

$$\frac{t \in \mathbb{V}}{t : \mathsf{Type}}\ (\text{tvar})$$

$$\frac{\sigma : \mathsf{Type} \qquad \tau : \mathsf{Type}}{\sigma \to \tau : \mathsf{Type}}\ (\text{fun})$$

$$\frac{\sigma : \mathsf{Type} \qquad t \in \mathbb{V}}{\forall t.\, \sigma : \mathsf{Type}}\ (\text{poly})$$

where t may or may not appear in σ.

The polymorphic type $\forall t.\, \sigma$ provides a generic type for every instance $\sigma[\tau/t]$ whenever t is instantiated by an actual type τ.

Examples

- $\texttt{id} : \forall t.\, t \to t$
- $\texttt{proj}_1 : \forall t.\, \forall u.\, t \to u \to t$
- $\texttt{proj}_2 : \forall t.\, \forall u.\, t \to u \to u$
- $\texttt{length} : \forall t.\, \texttt{list}\ t \to \texttt{nat}$
- $\texttt{singleton} : \forall t. t \to \texttt{list}(t)$

Free and bound variables, again

Definition 4

The *free variable* $\mathsf{FV}(\tau)$ of τ is defined inductively by

$$\begin{aligned} \mathsf{FV}(t) &= t \\ \mathsf{FV}(\sigma \to \tau) &= \mathsf{FV}(\sigma) \cup \mathsf{FV}(\tau) \\ \mathsf{FV}(\forall t.\, \sigma) &= \mathsf{FV}(\sigma) - \{t\} \end{aligned}$$

For convenience, the function extends to contexts:

$$\mathsf{FV}(\Gamma) = \{\, t \in \mathbb{V} \mid \exists (x : \sigma) \in \Gamma \wedge t \in \mathsf{FV}(\sigma) \,\}.$$

1. $\mathsf{FV}(t_1) = \{t_1\}$.
2. $\mathsf{FV}(\forall t.\, (t \to t) \to t \to t) = \emptyset$.
3. $\mathsf{FV}(x : t_1, y : t_2, z : \forall t.\, t) = \{t_1, t_2\}$.

Capture-avoiding substitution for type

Definition 5

The *(capture-avoidance) substitution* of a type ρ for the free occurrence of a type variable t is defined by

$$
\begin{aligned}
t[\rho/t] &= \rho \\
u[\rho/t] &= u && \text{if } u \neq t \\
(\sigma \to \tau)[\rho/t] &= \sigma[\rho/t] \to \tau[\rho/t] \\
(\forall t.\sigma)[\rho/t] &= \forall t.\sigma \\
(\forall u.\sigma)[\rho/t] &= \forall u.\sigma[\rho/t] && \text{if } u \neq t, u \notin \mathsf{FV}(\rho)
\end{aligned}
$$

Recall that $u \notin \mathsf{FV}(\rho)$ means that u is *fresh* for ρ.

Typed terms

Definition 6

On top of $\lambda_{\to}$, F has additional term formation rules as follows.

$$\frac{M : \mathtt{Term}_F \qquad t : \mathbb{V}}{\Lambda t.\, M : \mathtt{Term}_F}\ (\text{gen})$$

$$\frac{M : \mathtt{Term}_F \qquad \tau : \mathtt{Type}}{M\ \tau : \mathtt{Term}_F}\ (\text{inst})$$

1. $\Lambda t.\, M$ for type abstraction, or *generalisation*.
2. $M\ \tau$ for type application, or *instantiation*.

Example

Suppose $\texttt{length} : \forall t.\, \texttt{list}\ t \to \texttt{nat}$.

Then,

1. $\texttt{length nat}$
2. $\texttt{length bool}$
3. $\texttt{length}\ (\texttt{nat} \to \texttt{nat})$

are instances of `length` with types

1. $\texttt{list nat} \to \texttt{nat}$
2. $\texttt{list bool} \to \texttt{nat}$
3. $\texttt{list}\ (\texttt{nat} \to \texttt{nat}) \to \texttt{nat}$

System F: Typing judgement

A *type context* is a sequence of type variable

$$t_1, t_2, \ldots, t_n$$

F has two kinds of typing judgements.

- $\Delta \vdash \tau$ for τ for a valid type under the type context Δ
- $\Delta; \Gamma \vdash M : \tau$ for a well-typed term under the context Γ and the type context Δ.

For example,

$$t \vdash t \to t$$

is a judgement that $t \to$ is a valid type under the type context, t.

System F: Type formation

The justification of $\Delta \vdash \tau$ is constructed inductively by following rules.

$$\frac{t \text{ occurs in } \Delta}{\Delta \vdash t} \qquad \frac{\Delta, t \vdash \tau}{\Delta \vdash \forall t.\, \tau}$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}$$

Exercise

Derive the judgement

$$t : \tau \vdash t \to t$$

System F: Typing rules

The justification of $\Delta; \Gamma \vdash M : \sigma$ is defined inductively by following rules.

$$\frac{x : \sigma \in \Gamma}{\Delta; \Gamma \vdash x : \sigma}$$

$$\frac{\Delta, t; \Gamma \vdash M : \sigma}{\Delta; \Gamma \vdash \Lambda t.\, M : \forall t.\, \sigma} \ (\forall\text{-intro})$$

$$\frac{\Delta; \Gamma \vdash M : \sigma \to \tau \qquad \Delta; \Gamma \vdash N : \sigma}{\Delta; \Gamma \vdash M\ N : \tau}$$

$$\frac{\Delta \vdash \sigma \qquad \Delta; \Gamma, x : \sigma \vdash M : \tau}{\Delta; \Gamma \vdash \lambda x : \sigma.\, M : \sigma \to \tau}$$

$$\frac{\Delta; \Gamma \vdash M : \forall t.\, \sigma \qquad \Delta \vdash \tau}{\Delta; \Gamma \vdash M\ \tau : \sigma[\tau/t]} \ (\forall\text{-elim})$$

For convenience, $\vdash M : \tau$ stands for $\cdot; \cdot \vdash M : \tau$.

Typing derivation

The typing judgement $\vdash \Lambda t.\, \Lambda u.\, \lambda(x : t)(y : u).\, x : \forall t.\ t \to u \to t$ is derivable from the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{t, u \vdash t} \qquad \dfrac{\dfrac{}{t, u \vdash u} \qquad \dfrac{}{t, u; x : t, y : u \vdash x : t}}{t, u; x : t \vdash \lambda(y : u).\, x : u \to t}}{t, u; \cdot \vdash \lambda(x : t)(y : u).\, x : t \to u \to t}}{t; \cdot \vdash \Lambda u.\, \lambda(x : t)(y : u).\, x : \forall u.\, t \to u \to t}}{\vdash \Lambda t.\, \Lambda u.\, \lambda(x : t)(y : u).\, x : \forall t.\, \forall u.\, t \to u \to t}$$

Exercise

Derive the following judgements:

1. $\vdash \Lambda t.\, \lambda(x : t).\, x : \forall t.\, t \to t$
2. $\sigma; a : \sigma \vdash (\Lambda t.\, \lambda(x : t)(y : t).\, x)\ \sigma\ a : \sigma \to \sigma$
3. $\vdash \Lambda t.\, \lambda(f : t \to t)(x : t).\, f\ (f\ x) : \forall t.\, (t \to t) \to t \to t$

Hint. F is syntax-directed, so the type inversion holds.

System F: β-reduction

The β-conversion has two rules

$$(\lambda(x:\tau).\,M)\,N \longrightarrow_{\beta} M[x/N] \quad \text{and} \quad (\Lambda t.\,M)\,\tau \longrightarrow_{\beta} M[\tau/t]$$

For example,

$$(\Lambda t.\lambda x:t.\,x)\;\tau\;a \longrightarrow_{\beta} (\lambda x:t.\,x)[\tau/t]\;a \equiv (\lambda x:\tau.\,x)\;a \longrightarrow_{\beta} x[a/x] \equiv a$$

Similarly, β-conversion extends to subterms of a given term, introducing symbols $\longrightarrow_{\beta 1}$ and $\longrightarrow_{\beta *}$ in the same way.

Self application

Self-application is not typable in simply typed λ-calculus.

$$\lambda(x : t).\, x\, x$$

However, self-application is possible in System F.

$$\lambda(x : \forall t.t \to t).\, x\ (\forall t.t \to t)\ x$$

Exercise

Instantiate the first t with the type $\forall t.\, t \to t$.

Sum type

Definition 7

The *sum type* is defined by

$$\sigma + \tau := \forall t.(\sigma \to t) \to (\tau \to t) \to t$$

It has two injection functions: the first injection is defined by

$$\texttt{left}_{\sigma+\tau} := \lambda(x : \sigma).\, \Lambda t.\, \lambda(f : \sigma \to t)(g : \tau \to t).\, f\, x$$
$$\texttt{right}_{\sigma+\tau} := \lambda(y : \tau).\, \Lambda t.\, \lambda(f : \sigma \to t)(g : \tau \to t).\, g\, y$$

Exercise

Define

$$\texttt{either} : \forall u.\, (\sigma \to u) \to (\tau \to u) \to \sigma + \tau \to u$$

Product type

Definition 8 (Product Type)

The product type is defined by

$$\sigma \times \tau := \forall t.(\sigma \to \tau \to t) \to t$$

The pairing function is defined by

$$\langle \_, \_ \rangle := \lambda(x : \sigma)(y : \tau).\, \Lambda t.\, \lambda(f : \sigma \to \tau \to t).\, f\, x\, y$$

Exercise

Define projections

$$\texttt{proj}_1 : \sigma \times \tau \to \sigma \quad \text{and} \quad \texttt{proj}_2 : \sigma \times \tau \to \tau$$

Natural numbers i

The type of Church numerals is defined by

$$\texttt{nat} := \forall t.\,(t \to t) \to t \to t$$

Church numerals

$$\mathtt{c}_n : \texttt{nat}$$
$$\mathtt{c}_n := \Lambda t.\,\lambda(f : t \to t)\,(x : t).\,f^n\,x$$

Successor

$$\texttt{suc} : \texttt{nat} \to \texttt{nat}$$
$$\texttt{suc} := \lambda(n : \texttt{nat}).\,\Lambda t.\,\lambda(f : t \to t)\,(x : t)\,.\,f\,(n\;t\;f\;x)$$

Addition

$$\begin{aligned}&\mathtt{add} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}\\ &\mathtt{add} := \lambda(n : \mathtt{nat})\,(m : \mathtt{nat}) \quad \Lambda t.\, \lambda(f : t \to t)\,(x : t).\\ &\qquad\qquad\qquad\qquad\qquad\qquad (m\ t\ f)\ (n\ t\ f\ x)\end{aligned}$$

Multiplication

$$\begin{aligned}&\mathtt{mul} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}\\ &\mathtt{mul} := ?\end{aligned}$$

Conditional

$$\begin{aligned}&\mathtt{ifz} : \forall t.\, \mathtt{nat} \to t \to t \to t\\ &\mathtt{ifz} := ?\end{aligned}$$

System F allows us to define *iterator* like `fold` in Haskell.

$$\mathtt{fold}_{\mathtt{nat}} : \forall t.\, (t \to t) \to t \to \mathtt{nat} \to t$$

$$\mathtt{fold}_{\mathtt{nat}} := \Lambda t.\, \lambda(f : t \to t)(e_0 : t)(n : \mathtt{nat}).n\ t\ f\ e_0$$

Exercise

Define `add` and `mul` using $\mathtt{fold}_{\mathtt{nat}}$ and justify your answer.

1. $\mathtt{add}' := ? : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$
2. $\mathtt{mul}' := ? : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$

Lists

Definition 9

For any type σ, the type of lists over σ is

$$\texttt{list}\,\sigma := \forall t.\, t \to (\sigma \to t \to t) \to t$$

with “list constructors”:

$$\texttt{nil}_\sigma := \Lambda t.\lambda(h : t)(f : \sigma \to t \to t).\, h$$

and

$$\texttt{cons}_\sigma := \lambda(x : \sigma)(xs : \texttt{list}\,\sigma).\, \Lambda t.\lambda(h : t)(f : \sigma \to t \to t).f\, x\, (xs\ t\ h\ f)$$

of type $\sigma \to \texttt{list}\,\sigma \to \texttt{list}\,\sigma$.

Type erasure

Definition 10

The *erasing map* is a function defined by

$$\begin{aligned}
|x| &= x \\
|\lambda(x : \tau).\, M| &= \lambda x.\, |M| \\
|M\ N| &= (|M|\ |N|) \\
|\Lambda t.\, M| &= |M| \\
|M\ \tau| &= |M|
\end{aligned}$$

Proposition 11

Within System F, if $\vdash M : \sigma$ *and* $|M| \longrightarrow_{\beta 1} N'$*, then there exists a well-typed term* N *with* $\vdash N : \sigma$ *and* $|N| = N'$*.*

Type safety and normalisation

Theorem 12 (Type safety)

Suppose $\vdash M : \sigma$. *Then,*

1. $M \longrightarrow_{\beta 1} N$ *implies* $\vdash N : \sigma$;
2. M *is in normal form or there exists* N *such that* $M \longrightarrow_{\beta 1} N$

Type safety is proved by induction on the derivation of $\vdash M : \sigma$.

Theorem 13 (Normalisation properties)

F *enjoys the weak and strong normalisation properties.*

Proved by Girard's *reducibility candidates*.

Parametricity

What functions can you write for the following type?

$$\forall t.\, t \to t$$

Since t is arbitrary, we cannot inspect the content of t. What we can do with t is simply return it.

Theorem 14

Every term M of type $\forall t.\, t \to t$ is observationally equivalent[1] *to $\Lambda t.\, \lambda x : t.\, x$.*

[1] The notion of observational equivalence is beyond the scope of this lecture.

Parametricity: Theorems for free[2]

Assume F extended with the list type $\texttt{list}\ \tau$ for τ and the type $\mathbb{N}$ of naturals, denoted $\mathsf{F}_{\texttt{list},\mathbb{N}}$.

Then $\texttt{head} \circ \texttt{map}\ f = f \circ \texttt{head}$ for any $f : \tau \to \sigma$ where $\texttt{head} : \forall t.\ \texttt{list}\ t \to t$ can be proved by just reading the type of $\texttt{head}$ and $\texttt{tail}$!

Theorem 15

For any type σ in F (with lists) and $\cdot \vdash M : \sigma$, then

$$M \sim M : \mathcal{R}_{\sigma,\sigma}$$

[2] Philip Wadler. 1989. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89)*. ACM, New York, NY, USA, 347–359.

Undecidability of type inference

Theorem 16 (Wells, 1999)

It is undecidable whether, given a closed term M of the untyped lambda-calculus, there is a well-typed term M' in System F such that $|M'| = M$.

Two ways to retain decidable type inference:

1. Limit the expressiveness so that type inference remains decidable. For example, *Hindley-Milner type system* adapted by Haskell 98, Standard ML, etc. supports only a limited form of polymorphism but type inference is decidable.
2. Adopt *partial* type inference so that type annotations are needed for, e.g. top-level definitions and local definitions.

Check out *bidirectional type inference*.

Nameless Representation

Capture-avoiding but ill-defined substitution

The definition of capture-avoiding substitution is not well-defined. Recall that

$$
\begin{aligned}
x[L/x] &= L \\
y[L/x] &= y && \text{if } x \neq y \\
(M\,N)[L/x] &= M[L/x]\;N[L/x] \\
(\lambda x.\,M)[L/x] &= \lambda x.\,M \\
(\lambda y.\,M)[L/x] &= \lambda y.\,M[L/x] && \text{if } x \neq y \text{ and } y \notin \mathsf{FV}(L)
\end{aligned}
$$

The function $\_[L/x] \colon \texttt{Term}_V \to \texttt{Term}_V$ is not total, so it is not an instance of *structural recursion* (i.e. `fold`). In what sense, is the above well-defined?

1. Use *nominal technique* and the notion of α-structure recursion/induction. It requires some elements of group theory.
2. Use *nameless* representation.

Well-Scoped de Bruijn index representation i

An index i starting from 0 is used as a variable to represent the i-th enclosing λ (binder) 'from the inside out'. For example, a term with named variables

$$\lambda a.\, \lambda b.\, (\lambda c.\, c)\ (\lambda c.\, a\ b)$$

becomes

$$\lambda\ \lambda\ (\lambda\, 0)\ (\lambda\, 2\ 1)$$

Hint. It may be easier to think of a term in its tree representation.

Well-Scoped de Bruijn index representation ii

Definition 17 (de Bruijn representation with a local scope)

The term formation $\boxed{t \quad \mathtt{Term}_n}$ is defined inductively for $n \in \mathbb{N}$ by

$$\frac{0 \leq i < n}{i \quad \mathtt{Term}_n} \qquad\qquad \frac{t \quad \mathtt{Term}_{n+1}}{\lambda t \quad \mathtt{Term}_n}$$

$$\frac{t \quad \mathtt{Term}_n \qquad u \quad \mathtt{Term}_n}{t\,u \quad \mathtt{Term}_n}$$

$\boxed{t \quad \mathtt{Term}_n}$ means t has at most n many free variables.

Exercise

Translate the following terms to its de Bruijn index representation.

1. $\lambda x.\, x$
2. $\lambda s.\, \lambda z.\, s\ z$
3. $\lambda a.\, \lambda b.\, a\ (\lambda c.\, a\ b)$
4. $(\lambda x.\, x)\ (\lambda y.\, y)$
5. $\lambda x.\, y$
6. $x\ y\ z$

Substitution, revisited

How to reformulate β-reduction for terms in de Bruijn representation? Consider

$$(\lambda\,\lambda\,(\lambda\,0)\;(\lambda\,2\;1))\;\;t \longrightarrow_\beta (\lambda\;\;(\lambda\,0)\;(\lambda\,2\;1))\,[t/0]$$

The de Bruijn index increments under a binder so $[t/i]$ should be $[t'/i+1]$ where t' is the result of incrementing every index in t, e.g.,

$$\begin{aligned}(\lambda\,(\lambda\,0)\;(\lambda\,2\;1))\,[t/0] &= \lambda\,(\lambda\,0)[t'/1] \quad (\lambda\,2\;1)[t'/1] \\ &= \lambda\,(\lambda\,0[t''/2]) \quad (\lambda\;(2\;1)\,[t''/2]) \\ &= \lambda\,(\lambda\,0) \quad (\lambda\,2[t''/2]\;1[t''/2]) \\ &= \lambda\,(\lambda\,0) \quad (\lambda\,t''\;1)\end{aligned}$$

Simultaneous variable renaming

Definition 18

A *(variable) renaming* is a function ρ between $\mathbb{Z}_n$ and $\mathbb{Z}_m$.

Every renaming $\rho \colon \mathbb{Z}_n \to \mathbb{Z}_m$ extends to an action on terms:

$$\langle\rho\rangle\, i = \rho(i)$$

$$\langle\rho\rangle\,(t\ u) = \langle\rho\rangle\, t\ \ \langle\rho\rangle\, u$$

$$\langle\rho\rangle\, \lambda\, t = \lambda\, \langle\rho'\rangle\, t$$

where $\rho' : \mathbb{Z}_{n+1} \to \mathbb{Z}_{m+1}$ is defined as

$$\begin{aligned} \rho'(0) &= 0 \\ \rho'(i) &= 1 + \rho(i) && \text{if } i \neq 0 \end{aligned}$$

to avoid changing bound variables.

In particular, $wk \colon \mathtt{Term}_n \to \mathtt{Term}_{n+1}$ derived by $i \mapsto i + 1 \in \mathbb{Z}_{n+1}$ increments every index of a free variable by 1.

Simultaneous substitution

Definition 19

A *(simultaneous) substitution* is a function σ from $\mathbb{Z}_n$ to $\texttt{Term}_m$.

Every substitution extends to an action terms:

$$\langle\sigma\rangle\, i = \sigma(i)$$
$$\langle\sigma\rangle\, (t\ u) = \langle\sigma\rangle\, t\ \ \langle\sigma\rangle\, u$$
$$\langle\sigma\rangle\, \lambda\, t = \lambda\, \langle\sigma'\rangle\, t$$

where $\sigma' : \mathbb{Z}_{n+1} \to \texttt{Term}_{m+1}$ is defined as

$$\begin{aligned} \sigma'(0) &= 0 \\ \sigma'(i) &= wk\,(\sigma(i)) && \text{if } i \neq 0 \end{aligned}$$

Single substitution

Definition 20

A *single substitution* for t is a simultaneous substitution given by

$$\sigma(i) = \begin{cases} t & i = 0 \\ i & \text{otherwise.} \end{cases}$$

Exercise

1. Adopt α-equivalence to the de Bruijn representation.
2. Adopt β-equivalence to the de Bruijn representation.
3. Apply the new definition of substitution to compute `not True`.
4. Adopt the definitions of renaming and substitution to the de Bruijn level representation. N.B. we may also count the *i*-th enclosing binder 'from the outside in' using the same definition, called *the de Bruijn level*.

Homework

1. (2.5%) Extend PCF with the type $\mathbb{B}$ of boolean values with $\texttt{ifz}(M; N)\ \texttt{true} =_\beta M$ and $\texttt{ifz}(M; N)\ \texttt{false} =_\beta N$ including term formation rules, typing rules, and dynamics for $\mathbb{B}$.
   and $\texttt{pred}\ (\texttt{suc}\ n) = n$.
2. (2.5%) Define $\texttt{length}_\sigma : \texttt{list}\ \sigma \to \texttt{nat}$ calculating the length of a list in System F.