A Fuelled Self-Reducer For System T

Using simple types to encode complex ones

Greg Brown
greg.brown01@ed.ac.uk

University of Edinburgh

PEPM '25

Unreadable System T

```
(* sum : (nat -> nat -> nat -> nat -> nat) -> nat *)
let sum tree =
  let depth = tree 0 0 0 0 in
  let root  = tree 1 0 0 0 in
  let heap  = tree 2 in
  let go : nat -> nat = primrec depth with
    Z      -> fun i -> 0
  | S(acc) -> fun i ->
      let (tag, data) = (heap i 0 0, heap i 1) in
      if tag == 0 then
        data 0
      else
        acc (data 0) + acc (data 1)
  in
  go root
```

Outline

1. Type Encodings
2. Self Reducer

Unreadable System T

```
(* sum : (nat -> nat -> nat -> nat -> nat) -> nat *)
let sum tree =
  let depth = tree 0 0 0 0 in
  let root  = tree 1 0 0 0 in
  let heap  = tree 2 in
  let go : nat -> nat = primrec depth with
    Z      -> fun i -> 0
  | S(acc) -> fun i ->
      let (tag, data) = (heap i 0 0, heap i 1) in
      if tag == 0 then
        data 0
      else
        acc (data 0) + acc (data 1)
  in
  go root
```

Union Types[1]

- $A \sqcup B$ defined inductively
- $\texttt{inl} : A \to A \sqcup B$
- $\texttt{inr} : B \to A \sqcup B$
- $\texttt{prl} : A \sqcup B \to A$
- $\texttt{prr} : A \sqcup B \to B$

[1] Kiselyov, *Simply-typed encodings: PCF considered as unexpectedly expressive programming language*

System T$_{\sqcup}$

```
(* sum : (n -> (n |+| (n -> n -> (n |+| n |+| (n -> n))))) -> n *)
let sum t =
  let depth = prl (tree 0) in
  let root  = prl (tree 1) in
  let heap  = prr (tree 2) in
  let go : nat -> nat = primrec depth with
    Z      -> fun i -> 0
  | S(acc) -> fun i ->
      let (tag, data) = (prl (heap i 0), prr (heap i 1)) in
      if tag == 0 then
        prl data
      else
        acc ((prr data) 0) + acc ((prr data) 1)
  in
  go root
```

Product Types[1]

- $A \times B := \mathbb{N} \to (A \sqcup B)$
- `fst p = prl (p 0)`
- `snd p = prr (p 1)`
- `(x, y) = fun i ->`
  `if i == 0 then inl x else inr y`

System $T_{\sqcup\times}$

```
(* sum : (n * n * (n -> (n * (n |+| (n * n))))) -> n  *)
let sum tree =
  let (depth, root, heap) = tree in
  let go : nat -> nat = primrec depth with
    Z       -> fun i -> 0
  | S(acc) -> fun i ->
      let (tag, data) = heap i in
      if tag == 0 then
        prl data
      else
        let (left, right) = prr data in
        acc left + acc right
  in
  go root
```

Sum Types[1]

- $A + B := \mathbb{N} \times (A \sqcup B)$
- `left x = (0, inl x)`
- `right x = (1, inr x)`
- `either f g x =`

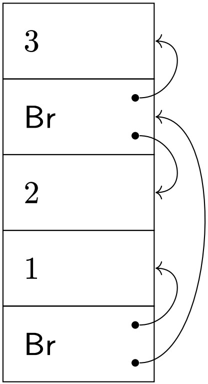
```
let (i, v) = x in
if i == 0 then f (prl v) else g (prr v)
```

System $T_{\times+}$

```
(* sum : (nat * nat * (nat -> nat + (nat * nat)) -> nat *)
let sum tree =
  let (depth, root, heap) = tree in
  let go : nat -> nat = primrec depth with
    Z      -> fun i -> 0
  | S(acc) -> fun i ->
      match heap i with
        Left(data)           -> data
      | Right(left, right) -> acc left + acc right
  in
  go root
```

Inductive Types[2]

- Heap with pointers
- Need depth for recursion
- Store root for efficiency



[2] Longley and Normann, *Higher-Order Computability*

```
(* sum : nat btree -> nat *)
let sum tree = fold tree with
  Lf x -> x
| Br(left, right) -> left + right
```

Examples of Regular Types

Examples

Lists	$\mu X.1 + A \times X$
Binary trees	$\mu X.A + X^2$
Finite trees	$\mu X.A \times \mathsf{List}\ X$

Non-Examples

- $\mu X.1 + ((X \to \mathbb{N}) \to \mathbb{N})$
- $\mu X.A + (\mathbb{N} \to X)$

Representing Terms

```
type term =
    Var of nat
  | Zero
  | Suc of term
  | Rec of term * term * term
  | Abs of term
  | App of term * term
```

Parallel Reduction Step

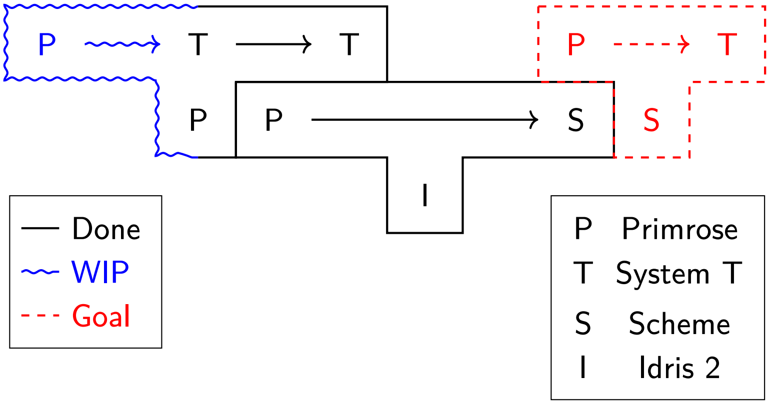
```
(* step : term -> term *)
step t = fold t with
  Rec(z, s, Zero)  -> z
| Rec(z, s, Suc t) -> App(s, Rec(z, s, t))
| App(Abs t, u)    -> subst t u

| Var n            -> Var n
| Zero             -> Zero
| Suc t            -> Suc t
| Rec(z, s, t)     -> Rec(z, s, t)
| Abs t            -> Abs t
| App(t, u)        -> App(t, u)
```

Computing Normal Forms

```
(* reduce : nat * term -> term *)
reduce (n, t) =
  fold n with
    Zero -> t
  | Suc t -> step t
```

Future Work


P
T
T
P
T
P
P
S
S
I
Done
WIP
Goal
P Primrose
T System T
S Scheme
I Idris 2

Summary

- Encode regular types in System T
- Use for fuelled self-reducer
- Ongoing work on implementation

Self-Reducer

Compiler