Type Inference for linear lambda calculus for hardware description

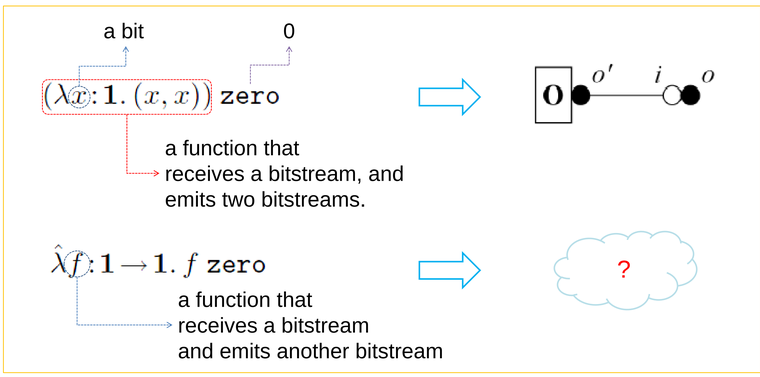
Jeongpyo Lim
Pohang University of Science and Technology

Motivation

Q. Is it possible to use λ-calculus to describe hardware circuits?
A. It is Possible but it demands some Modifications!

- function ≈ hardware
  application ≈ link of two components
- Because of higher-order functions in λ-calculus

< λ-calculus and hardware circuits>

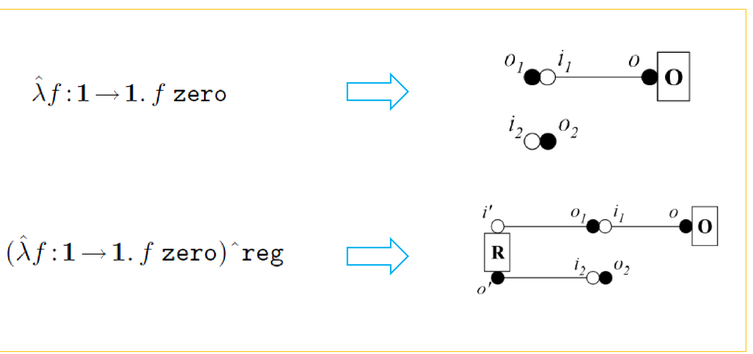


Linear λ-calculus(*l*λ)

Q. What is the meaning of 'linear'?
A. 'Something is linear' means that it can be used only once and it is not sharable!

☺. Bitstreams are sharable(not linear), but hardware circuits are not sharable(linear)

< Higher-order functions in *l*λ-calculus >



Problem : Current λ is impractical!

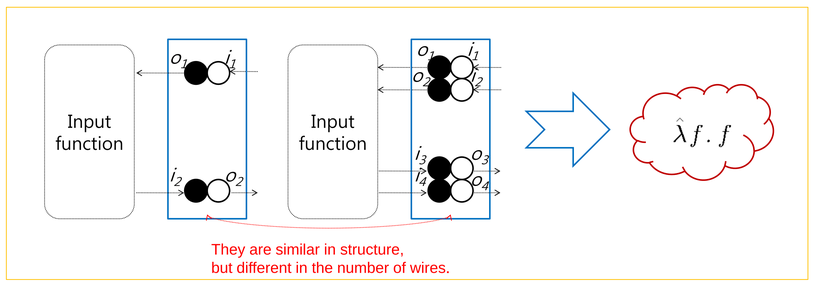
1. *l*λ requires programmers to write all type annotations!

<Example : a piece of FFT code>

```
let twoC : tforall 'a.(('a->'a) -->^2 ('a^2->'a^2)) =
           forall 'a => lfn f:'a->'a ==>2 fn (x,y):('a*'a) => ((f x),(f y));
let prodC : tforall 'a.(('a->'a^2) -->^2 ('a^4->'a^2)) =
           forall 'a => lfn f:'a^2->'a ==>2 fn ((x,y),(z,w)):(('a*'a)*('a*'a)) => ((f (x,z)),(f (y,w)));
let riffleC : tforall 'a.(('a^2->'a^2) -->^2 ('a^2->'a^2)) =
           forall 'a => lfn f:'a^2->'a^2 ==>2 fn ((x,y),(z,w)):(('a*'a)*('a*'a)) => ((f (x,z)), (f(y,w)));
let unriffleC : tforall 'a.(('a^2->'a^2) -->^2 ('a^2->'a^2)) =
           forall 'a => lfn f:'a^2->'a^2 ==>2 fn (p,q):('a^2*'a^2) =>
           proj (f p) of (x,z):('a*'a) => proj (f q) of (y,w):('a*'a) => ((x,y),(z,w));
let spreadC : tforall 'a.('a^4->'a^4) = forall 'a => fn ((x,y),(z,w)):(('a*'a)*('a*'a)) => ((x,z),(y,w));
```

2. Incomplete polymorphism

<polymorphism is…>



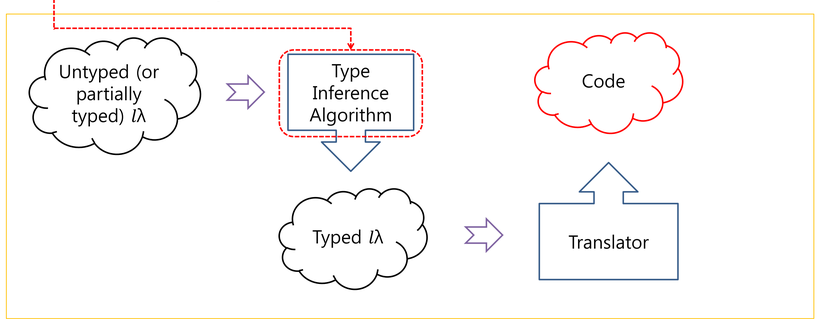
- Type variables in *l*λ range over not all monomorphic types but only sharable types.
- Incomplete polymorphism acts as an obstacle to the design of a type inference algorithm.

Goal : Complete polymorphism & Type inference

1. Extending *l*λ with general polymorphism

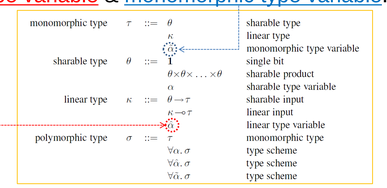
2. Type inference algorithm

<Translation process>



Solution

1. Complete polymorphism
   - Linear type variable & monomorphic type variable.

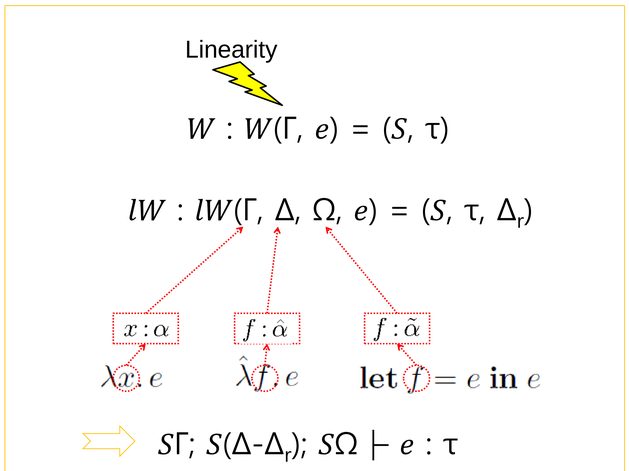


   - Conventional let-expression
     : enables programmers to define polymorphic expressions.

   - Polymorphic typing context

sharable typing context	Γ	::=	$\cdot \mid \Gamma, x:\theta$
linear typing context	Δ	::=	$\cdot \mid \Delta, f:\kappa$
polymorphic typing context	Ω	::=	$\cdot \mid \Omega, f:\sigma$

2. Type Inference algorithm(*lW*)
   : ≈ Revised version of the Hindley/Milner algorithm(*W*)



Conclusions

- Conventional *W* algorithm can be transformed to the type inference algorithm for linear λ-calculus(*lW*).
- The type inference algorithm can infer types of untyped expressions, and this allows programmers not to write all type annotations.