

CMPSCI 630: Programming Languages  
**Toward More Realistic Languages –  
 Functions, Natural Numbers and Recursion**

Spring 2009  
 (with thanks to Robert Harper)

**Toward More Realistic Languages – Functions, Natural  
 Numbers and Recursion**

We began by considering  $\mathcal{L}\{\text{num str}\}$

- Numbers, strings, some arbitrary primitive operations.
- Static and dynamic semantics; type safety.

We then considered adding functions to  $\mathcal{L}\{\text{num str}\}$

- First order, via function definitions
- Higher order, via function types

1

**Toward More Realistic Languages – Recursive Functions**

A more powerful approach to realism is to combine recursive functions and natural numbers, yielding general computational capability.

Harper describes two possibilities:

- One based on **primitive recursion**:  $\mathcal{L}\{\text{nat} \rightarrow\}$  or Gödel's T.
- Another on **general recursion**:  $\mathcal{L}\{\text{nat} \rightarrow\}$  or Plotkin's PCF.

Primitive recursion permits only **total** functions; termination is guaranteed, but some computations cannot be coded this way (not Turing complete), so not practical for a "realistic" language.

2

**Syntax for  $\mathcal{L}\{\text{nat} \rightarrow\}$**

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{nat}$	$\text{nat}$
		$  \text{arr}(\tau_1; \tau_2)$	$\tau_1 \rightarrow \tau_2$
Expr	$e$	$::= x$	$x$
		$  z$	$z$
		$  s(e)$	$s(e)$
		$  \text{rec}(e; e_0; x.y.e_1)$	$\text{rec } e \{z \Rightarrow e_0 \mid s(x) \text{ with } y \Rightarrow e_1\}$
		$  \text{lam}[\tau](x.e)$	$\lambda(x:\tau.e)$
		$  \text{ap}(e_1; e_2)$	$e_1(e_2)$

We write  $\bar{n}$  for the expression  $s(\dots s(z))$ , in which successor is applied  $n \geq 0$  times to zero.

3

**Syntax for  $\mathcal{L}\{\text{nat} \rightarrow\}$**

The expression  $\text{rec}(e; e_0; x.y.e_1)$  is called **primitive recursion**. It represents the  $e$ -fold iteration of the transformation  $x.y.e_1$  starting from  $e_0$ , where  $x$  is bound to the predecessor and  $y$  is bound to the result of the  $x$ -fold iteration.

Sometime **iteration**, written  $\text{iter}(e; e_0; y.e_1)$ , is considered an alternative to primitive recursion. Similar meaning, except only result of recursive call is bound to  $y$  in  $e_1$  and there is no binding for the predecessor. Clearly iteration is definable from recursion, by ignoring the predecessor binding. Conversely, can define primitive recursion from iteration and product by pairing simultaneous computation of predecessor with iteration of specified computation.

4

**Static Semantics for  $\mathcal{L}\{\text{nat} \rightarrow\}$**

$$\frac{}{\Gamma, x : \text{nat} \vdash x : \text{nat}}$$

$$\frac{}{\Gamma \vdash z : \text{nat}}$$

$$\frac{e : \text{nat}}{\Gamma \vdash s(e) : \text{nat}}$$

$$\frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \text{rec}(e; e_0; x.y.e_1) : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \quad x \# \Gamma}{\Gamma \vdash \text{lam}[\sigma](x.e) : \text{arr}(\sigma; \tau)}$$

$$\frac{\Gamma \vdash e_1 : \text{arr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{ap}(e_1; e_2) : \tau}$$

5

### Properties of Typing

#### Lemma 1 (Substitution)

If  $\Gamma, x : \tau \vdash e' : \tau'$  and  $\Gamma \vdash e : \tau$ , then  $\Gamma \vdash [e/x]e' : \tau'$

The proof is by induction on typing.

6

### Dynamic Semantics for $\mathcal{L}\{\text{nat} \rightarrow\}$

We will adopt an **eager** semantics for successor, so that values of type **nat** are numerals, and a **call-by-name** semantics for function applications. Variables range over computations, which are not necessarily values. These choices are not required, but are natural and convenient since every closed expression in  $\mathcal{L}\{\text{nat} \rightarrow\}$  has a value.

$$\frac{\overline{z \text{ val}}}{\frac{\frac{e \text{ val}}{s(e) \text{ val}}}{\text{lam}[\tau](x.e) \text{ val}}}$$

7

### Dynamic Semantics for $\mathcal{L}\{\text{nat} \rightarrow\}$

$$\frac{\frac{\frac{e \mapsto e'}{s(e) \mapsto s(e')}}{\text{ap}(e_1; e_2) \mapsto \text{ap}(e'_1; e_2)}}{\text{ap}(\text{lam}[\tau](x.e); e_2) \mapsto [e_2/x]e}}{\frac{\frac{e \mapsto e'}{\text{rec}(e; e_0; x.y.e_1) \mapsto \text{rec}(e'; e_0; x.y.e_1)}}{\text{rec}(e; e_0; x.y.e_1) \mapsto \text{rec}(e'; e_0; x.y.e_1)}}$$

8

### Dynamic Semantics for $\mathcal{L}\{\text{nat} \rightarrow\}$

$$\frac{\overline{\text{rec}(z; e_0; x.y.e_1) \mapsto e_0}}{\frac{\frac{s(e) \text{ val}}{\text{rec}(s(e); e_0; x.y.e_1) \mapsto [e, \text{rec}(e; e_0; x.y.e_1)/x, y]e_1}}{\text{rec}(s(e); e_0; x.y.e_1) \mapsto [e, \text{rec}(e; e_0; x.y.e_1)/x, y]e_1}}$$

Note that lazy binding to  $y$  means recursive call on  $e$  will not be executed unless  $y$  is required for evaluation of  $e_1$ .

9

### Safety

#### Lemma 2 (Canonical Forms)

Suppose that  $e : \tau$  and  $e \text{ val}$ .

1. If  $\tau = \text{nat}$ , then  $e = s(\dots s(z))$  for some number  $n \geq 0$  occurrences of the successor starting with zero.
2. If  $\tau = \tau_1 \rightarrow \tau_2$ , then  $e = \lambda(x:\tau_1).e_2$  for some  $e_2$ .

#### Theorem 3 (Safety)

1. If  $e : \tau$  and  $e \mapsto e'$ , then  $e' : \tau$ .
2. If  $e : \tau$ , then either  $e \text{ val}$  or there exists  $e'$  such that  $e \mapsto e'$ .

10

### Definability in $\mathcal{L}\{\text{nat} \rightarrow\}$

A function  $f : \mathcal{N} \rightarrow \mathcal{N}$  is **definable** in  $\mathcal{L}\{\text{nat} \rightarrow\}$  iff there exists an expression  $e_f : \text{nat} \rightarrow \text{nat}$  such that for every  $n \in \mathcal{N}$ :

$$e_f(\bar{n}) \equiv \overline{f(n)} : \text{nat}$$

where  $\equiv$  means **definitionally equivalent**.

For example, the doubling function  $d(n) = 2 \times n$  is definable by the expression

$$e_d = \lambda(x:\text{nat}.\text{rec } x\{z \Rightarrow z \mid s(u) \text{ with } v \Rightarrow s(s(v))\})$$

11

### Definability in $\mathcal{L}\{\text{nat} \rightarrow\}$

To see this, observe that  $e_f(\bar{0}) \equiv \bar{0} : \text{nat}$  and that, assuming  $e_d(\bar{n}) \equiv \bar{d}(n) : \text{nat}$ ,

$$\begin{aligned} e_d(\overline{n+1}) &\equiv \mathbf{s}(e_d(\bar{n})) \\ &\equiv \mathbf{s}(\mathbf{s}(2 \times n)) \\ &= 2 \times (n+1) \\ &= \overline{d(n+1)} \end{aligned}$$

Even the Ackermann function is definable in  $\mathcal{L}\{\text{nat} \rightarrow\}$ !

12

### Non-Definability in $\mathcal{L}\{\text{nat} \rightarrow\}$

It is impossible to define an infinite loop in  $\mathcal{L}\{\text{nat} \rightarrow\}$ :

#### Theorem 4

If  $e : \tau$ , then there exists  $v$  val such that  $e \mapsto^* v$ .

But a diagonalization argument can be used to show that there exist functions on the natural numbers that are not definable in  $\mathcal{L}\{\text{nat} \rightarrow\}$ . (See Harper for details.)

13

### Toward More Realistic Languages – $\mathcal{L}\{\text{nat} \rightarrow\}$

General recursion does not guarantee termination – functions are **partial** – but supports Turing complete computation.

Thus general recursion is a more practical starting point for a “realistic” language –  $\mathcal{L}\{\text{nat} \rightarrow\}$  or Plotkin’s PCF.

14

### Toward More Realistic Languages – $\mathcal{L}\{\text{nat} \rightarrow\}$

General recursion characterizes recursive definitions in terms of **fixed points**.

- The solution to a recursive definition is allowed to be a **partial function** that is an **approximation** to the solution.
- The type system no longer ensures termination – it is left to the programmer to ensure that recursive functions are total, i.e., that all loops terminate.

15

### Syntax for $\mathcal{L}\{\text{nat} \rightarrow\}$

Category	Item	Abstract	Concrete
Type	$\tau$	$::= \text{nat}$	$\text{nat}$
		$  \text{parr}(\tau_1; \tau_2)$	$\tau_1 \rightarrow \tau_2$
Expr	$e$	$::= x$	$x$
		$  z$	$z$
		$  \mathbf{s}(e)$	$\mathbf{s}(e)$
		$  \text{ifz}(e; e_0; x.e_1)$	$\text{ifz } e \{ z \Rightarrow e_0 \mid \mathbf{s}(x) \Rightarrow e_1 \}$
		$  \text{lam}[\tau](x.e)$	$\lambda(x:\tau.e)$
		$  \text{ap}(e_1; e_2)$	$e_1(e_2)$
	$  \text{fix}[\tau](x.e)$	$\text{fix } x:\tau \text{ is } e$	

16

### Syntax for $\mathcal{L}\{\text{nat} \rightarrow\}$

The expression  $\text{fix}[\tau](x.e)$  is called **general recursion**.

The expression  $\text{ifz}(e; e_0; x.e_1)$  branches according to whether  $e$  evaluates to  $z$ , binding the predecessor to  $x$  in the case that it does not.

17

### Static Semantics for $\mathcal{L}\{\text{nat} \rightarrow\}$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash z : \text{nat}}$$

$$\frac{e : \text{nat}}{\Gamma \vdash s(e) : \text{nat}}$$

$$\frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{nat} \vdash e_1 : \tau}{\Gamma \vdash \text{ifz}(e; e_0; x.e_1) : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{lam}[\tau_1](x.e) : \text{parr}(\tau_1; \tau_2)}$$

18

### Static Semantics for $\mathcal{L}\{\text{nat} \rightarrow\}$

$$\frac{\Gamma \vdash e_1 : \text{parr}(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{ap}(e_1; e_2) : \tau}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{fix}[\tau](x.e) : \tau}$$

Deceptively simple, but note that we **assume** the typing we are trying to establish!

19

### Properties of Typing

The structural rules are admissible for the static semantics. In particular:

#### Lemma 5 (Substitution)

If  $\Gamma, x : \tau \vdash e' : \tau'$  and  $\Gamma \vdash e : \tau$ , then  $\Gamma \vdash [e/x]e' : \tau'$

The proof is by induction on typing.

20

### Dynamic Semantics for $\mathcal{L}\{\text{nat} \rightarrow\}$

We consider a call-by-name semantics for function application in  $\mathcal{L}\{\text{nat} \rightarrow\}$  and require that successor evaluate its argument.

$$\frac{}{z \text{ val}}$$

$$\frac{e \text{ val}}{s(e) \text{ val}}$$

$$\frac{}{\text{lam}[\tau](x.e) \text{ val}}$$

21

### Dynamic Semantics for $\mathcal{L}\{\text{nat} \rightarrow\}$

$$\frac{e \mapsto e'}{s(e) \mapsto s(e')}$$

$$\frac{e \mapsto e'}{\text{ifz}(e; e_0; x.e_1) \mapsto \text{ifz}(e'; e_0; x.e_1)}$$

$$\frac{}{\text{ifz}(z; e_0; x.e_1) \mapsto e_0}$$

$$\frac{s(e) \text{ val}}{\text{ifz}(s(e); e_0; x.e_1) \mapsto [e/x]e_1}$$

22

### Dynamic Semantics for $\mathcal{L}\{\text{nat} \rightarrow\}$

$$\frac{e_1 \mapsto e'_1}{\text{ap}(e_1; e_2) \mapsto \text{ap}(e'_1; e_2)}$$

$$\frac{}{\text{ap}(\text{lam}[\tau](x.e); e_2) \mapsto [e_2/x]e}$$

$$\frac{}{\text{fix}[\tau](x.e) \mapsto [\text{fix}[\tau](x.e)/x]e}$$

Final rule **unwinds** the recursion.

23

### Safety

#### Theorem 6

1. If  $e : \tau$  and  $e \mapsto e'$ , then  $e' : \tau$ .
2. If  $e : \tau$ , then either  $e$  is a value, or there exists  $e'$  such that  $e \mapsto e'$ .

24

### Definability for $\mathcal{L}\{\text{nat} \rightarrow\}$

We define **general recursive functions** with syntax:

$$\text{fun } x(y:\tau_1):\tau_2 \text{ is } e$$

where  $x$  stands for the function itself and  $y$  is its argument, and both are bound in  $e$ .

25

### Definability for $\mathcal{L}\{\text{nat} \rightarrow\}$

Dynamic semantics for general recursive functions:

$$\frac{}{\text{fun } x(y:\tau_1):\tau_2 \text{ is } e(e_1) \mapsto [\text{fun } x(y:\tau_1):\tau_2 \text{ is } e, e_1/x, y]e}$$

Recursively substituting the function itself for  $x$  in its body.

26

### Definability for $\mathcal{L}\{\text{nat} \rightarrow\}$

General recursive functions are definable using ordinary (non-recursive) functions and general recursion.

Specifically, define  $\text{fun } x(y:\tau_1):\tau_2 \text{ is } e$  to stand for

$$\text{fix } x:\tau_1 \rightarrow \tau_2 \text{ is } \lambda(y:\tau_1.e)$$

It is easy to check that the static and dynamic semantics of recursive functions are derivable from this definition.

27

### Definability for $\mathcal{L}\{\text{nat} \rightarrow\}$

Primitive recursion can also be defined in terms of general recursion.

Specifically, define  $\text{rec } e\{z \Rightarrow e_0 \mid s(x) \text{ with } y \Rightarrow e_1\}$  to be the expression  $e'(e)$  where  $e'$  is the general recursive function

$$\text{fun } f(u:\text{nat}):\tau \text{ is if } z \Rightarrow e_0 \mid s(x) \Rightarrow [f(x)/y]e_1$$

It is easy to check that the static and dynamic semantics of primitive recursion functions are derivable in  $\mathcal{L}\{\text{nat} \rightarrow\}$  using this expansion.

28

### Definability for $\mathcal{L}\{\text{nat} \rightarrow\}$

Because  $\mathcal{L}\{\text{nat} \rightarrow\}$  admits partial functions, definability is more difficult to characterize here than for  $\mathcal{L}\{\text{nat} \rightarrow\}$ .

It turns out that definability in  $\mathcal{L}\{\text{nat} \rightarrow\}$  corresponds to **partial recursive functions** and that Church's Law says partial recursive functions are exactly what can be computed with any conceivable programming language.

Therefore,  $\mathcal{L}\{\text{nat} \rightarrow\}$  is as powerful as any other programming language – if perhaps not as user friendly!

29