

CMPSCI 630: Programming Languages
**Toward More Realistic Languages –
 Functions: Syntax and Semantics**

Spring 2009
 (with thanks to Robert Harper)

Introduction

Formal definitions of programming languages.

1. **Abstract syntax:** first-order (ast) and higher-order (abt).
2. **Static semantics:** typing rules.
3. **Dynamic semantics:** execution rules.
4. Formal definition of **type safety**.
5. **Proving** a language safe.

1

Toward Realistic Programming Languages

We started by studying $\mathcal{L}\{\text{num str}\}$, a simple language of expressions:

- Numbers and strings.
- Variables, simple operations, binding.

Then we added some basic data types and related operations:

- Product types and sum types
- Pattern matching

2

Toward Realistic Programming Languages

Next we will study a fundamental programming language construct – the **function** – also known as:

- Method (in Java, etc.)
- Procedure (in Algol60, Ada, etc.)

We will begin by considering **first-order** or **second-class** functions, added to $\mathcal{L}\{\text{num str}\}$ via **function definitions**.

We will then consider **first-class** functions, add to $\mathcal{L}\{\text{num str}\}$ by introducing **function types**, yielding a **higher-order** language.

3

First-Order Functions

First order functions:

- are created by **functions definitions**, which give them names
- have **domain** and **range** of **base** types (e.g., `num` or `str` in $\mathcal{L}\{\text{num str}\}$)
- do not introduce new types for themselves
- are called **second-class** functions, since they are not values in the same sense as are instances of the language's base types (e.g., `num` or `str` in $\mathcal{L}\{\text{num str}\}$)

4

First-Order Functions

The language $\mathcal{L}\{\text{num str fun}\}$ extends $\mathcal{L}\{\text{num str}\}$ with function definitions and function applications, as defined by the following grammar:

<i>Category</i>	<i>Item</i>	<i>Abstract</i>	<i>Concrete</i>
Expr	e	$::= \text{fun}[\tau_1; \tau_2](x_1.e_2; f.e)$	$\text{fun } f(x_1:\tau_1):\tau_2 = e_2 \text{ in } e$
		$ \text{call}[f](e)$	$f(e)$

Variable f ranges over distinguished class of variables called **function names**

5

First-Order Functions

Expression $\text{fun}[\tau_1; \tau_2](x_1.e_2; f.e)$ binds f within e to pattern $x_1.e_2$, an abstractor which has parameter x_1 and expression e_2 .

Domain is type τ_1 and **range** is type τ_2

Expression $\text{call}[f](e)$ instantiates abstractor bound to f with argument e .

6

Static Semantics

Static semantics for $\mathcal{L}\{\text{num str fun}\}$ are defined, as usual, by hypothetical inductive judgements, $\Gamma \vdash e : \tau$, but hypotheses can be of two forms:

- $x : \tau$, declaring type of variable x to be τ
- $f(\tau_1) : \tau_2$, declaring that f is a function name with domain τ_1 and range τ_2 .

Second form sometimes called **function header** based on resemblance to concrete syntax of first part of a function definition

7

Static Semantics

Static semantics for $\mathcal{L}\{\text{num str fun}\}$ defined by extending those for $\mathcal{L}\{\text{num str}\}$ with the following judgements:

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2 \quad \Gamma, f(\tau_1) : \tau_2 \vdash e : \tau}{\Gamma \vdash \text{fun}[\tau_1; \tau_2](x_1.e_2; f.e) : \tau}$$

$$\frac{\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau_1}{\Gamma, f(\tau_1) : \tau_2 \vdash \text{call}[f](e) : \tau_2}$$

8

Function Substitution

Structural property of substitution takes a form matching the form of hypotheses governing function names.

The **function substitution** operation, $[[x.e/f]]e'$, is inductively defined similarly to ordinary substitution, but with restriction that function name f can only occur within e' as part of a function call. For such occurrences, substitution rule is:

$$\overline{[[x.e/f]]\text{call}[f](e') = \text{let}(e'; x.e)}$$

Thus, at call sites to f , x is bound to e' within e to instantiate the pattern substituted for f .

9

Properties of Typing

Lemma 1 (Substitution)

If $\Gamma, f(\tau_1) : \tau_2 \vdash e : \tau$ and $\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2$, then $\Gamma \vdash [[x_1.e_2/f]]e : \tau$

The proof is by induction on derivation of the structure of e' .

10

Dynamic Semantics

Dynamic semantics for $\mathcal{L}\{\text{num str fun}\}$ defined using function substitution, by extending those for $\mathcal{L}\{\text{num str}\}$ with the following judgement:

$$\overline{\text{fun}[\tau_1; \tau_2](x_1.e_2; f.e) \mapsto \overline{[[x_1.e_2/f]]e}}$$

Function substitution eliminates all applications of f in e , so no rule for evaluating function applications is needed.

Semantics is either **call-by-name** or **call-by-value** according to whether the `let` binding is lazy or eager.

Safety for $\mathcal{L}\{\text{num str fun}\}$ could be proven, but is a corollary of safety for $\mathcal{L}\{\text{num str } \rightarrow\}$, which we consider next.

11

Higher-Order Functions

$\mathcal{L}\{\text{num str fun}\}$ segregates (second-class) functions from (first-class) expressions.

To consolidate function definitions with expression definitions, we:

- **reify** the abstractor into a form of expression, called **lambda abstraction**, written $\text{lam}[\tau](x.e)$
- **generalize** function execution to **application**, written $\text{ap}(e_1; e_2)$, where e_1 can be any expression, not just a function name.

12

Higher-Order Functions

Lambda abstraction and application are, respectively, the introduction and elimination forms for the **function type**, $\text{arr}(\tau_1; \tau_2)$, whose elements are functions with domain τ_1 and range τ_2 .

Functions of function type are **first-class**, since their domains and ranges can be arbitrary types, including function types.

A language with function types is called **higher-order** since functions can be passed as arguments to and returned as results from other functions.

Higher-order languages are powerful and consequently subtle and have led to some notorious programming language design errors.

13

Higher-Order Functions

The language $\mathcal{L}\{\text{num str } \rightarrow\}$ that enriches $\mathcal{L}\{\text{num str}\}$ with function types is defined by the following grammar:

Category	Item	Abstract	Concrete
Type	τ	$::= \text{arr}(\tau_1; \tau_2)$	$\tau_1 \rightarrow \tau_2$
Expr	e	$::= \text{lam}[\tau](x.e)$ $\text{ap}(e_1; e_2)$	$\lambda(x:\tau.e)$ $e_1(e_2)$

In $\text{arr}(\tau_1; \tau_2)$, τ_1 is the **domain type**, τ_2 is the **range type**. Note that since $\tau_1 \rightarrow \tau_2$ is a type, so are **higher order function types** such as $(\tau_1 \rightarrow \tau_2) \rightarrow (\tau_3 \rightarrow \tau_4)$, etc.

14

Static Semantics

Static semantics for $\mathcal{L}\{\text{num str } \rightarrow\}$ defined by extending those for $\mathcal{L}\{\text{num str}\}$ with the following parametric inductive judgements:

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

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

15

Properties of Typing

Lemma 2 (Inversion)

Suppose that $\Gamma \vdash e : \tau$

1. If $e = \text{lam}[\tau_1](x.e)$, then $\tau = \text{arr}(\tau_1; \tau_2)$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.
2. If $e = \text{ap}(e_1; e_2)$, then there exists τ_2 such that $\Gamma \vdash e_1 : \text{arr}(\tau_2; \tau)$ and $\Gamma \vdash e_2 : \tau_2$

Proof by **rule induction** on the typing rules.

16

Properties of Typing

Lemma 3 (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 derivation of the first judgement.

17

Dynamic Semantics – Call-by-Name

In **call-by-value** semantics, the argument is evaluated before the function is applied to the resulting value.

In **call-by-name** semantics, the argument is passed to the function in unevaluated form and its evaluation is deferred until it is actually needed.

18

Dynamic Semantics – Call-by-Name

Dynamic semantics for call-by-name $\mathcal{L}\{\text{num str} \rightarrow\}$ defined by extending those for $\mathcal{L}\{\text{num str}\}$ with the following inductive judgements:

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

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

$$\frac{}{\text{ap}(\overline{\text{lam}[\tau_2](x.e_1)}; e_2) \mapsto [e_2/x]e_1}$$

19

Safety

Theorem 4 (Preservation)

If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Proof: The proof proceeds by induction on evaluation. ■

20

Safety

Lemma 5 (Canonical Forms)

If $e \text{ val}$ and $e : \text{arr}(\tau_1; \tau_2)$ then $e = \overline{\text{lam}[\tau_1](x.e_2)}$ for some x and e_2 such that $x : \tau_1 \vdash e_2 : \tau_2$

The proof is by **induction on typing**.

Theorem 6 (Progress)

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

Proof: The proof is by **induction on typing**. ■

21

Evaluation Semantics for Functions

An inductive definition of evaluation semantics for $\mathcal{L}\{\text{num str} \rightarrow\}$ extends that for $\mathcal{L}\{\text{num str}\}$ with the following judgements:

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

$$\frac{e_1 \Downarrow \overline{\text{lam}[\tau](x.e)} \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{\text{ap}(e_1; e_2) \Downarrow v}$$

Theorem 7

1. If $e \Downarrow v$ then $e \mapsto^* v$.
2. If $e \mapsto^* v$, where $v \text{ val}$, then $e \Downarrow v$.

22

Static Scope

Dynamic semantics for $\mathcal{L}\{\text{num str} \rightarrow\}$ is defined for **closed** expressions – those with no free variables.

Hence variables will never be encountered during evaluation; closed expressions will have been substituted for them before they are needed in the evaluation.

This is known as **static scope** or **static binding**.

23

Dynamic Scope

An alternative evaluation strategy for $\mathcal{L}\{\text{num str } \rightarrow\}$ is **dynamic scope** or **dynamic binding**.

Two crucial differences:

- evaluation is defined for **open** terms – those having free variables – although it is an error to evaluate a variable
- binding of variables is specified by a special form of substitution that **incurs** rather than **avoids** capture of free variables

24

Replacement vs. Substitution

Distinguish the capture-incurring form of substitution by calling it **replacement**, written as $[x \leftarrow e_1]e_2$.

Let $e = \lambda(x:\sigma.y)$ (with free variable y) and let $e' = \lambda(y:\tau.f(y))$ (with variable f).

Substitution: $[e/f]e' = \lambda(y':\tau.\lambda(x:\sigma.y)(y'))$, with bound variable y renamed y' to avoid confusion with free variable y in e .

Replacement: $[f \leftarrow e]e' = \lambda(y:\tau.\lambda(x:\sigma.y)(y))$, which has no free variables – the free y in e is captured by the binding for y in e' .

25

Implications of Dynamic Scope

Dynamic scope does not preserve typing: If $\sigma \neq \tau$ then

$$[f \leftarrow e]e' = \lambda(y:\tau.\lambda(x:\sigma.y)(y))$$

is not well-typed even though both e and e' are.

Hence dynamic scope is usually restricted to **untyped**, or **untyped**, languages.

26

Implications of Dynamic Scope

The "virtue" of dynamic scope is that it enables parameterization of functions with variable that don't need to be passed as arguments:

- $\lambda(x:\sigma.e)$ with y free in e represents a family of functions, one for each choice of y .
- Replacement, instead of substitution, allows y to be set by the context in which the function is used, rather than the context where it was defined – hence "dynamic scope"

27

Implications of Dynamic Scope

But this "virtue" means that **names of bound variables matter**, which violates α -equivalence:

$e' = \lambda(y:\tau.f(y))$ and $e'' = \lambda(y':\tau.f(y'))$ are α -equivalent

But $[f \leftarrow e]e' = \lambda(y:\tau.\lambda(x:\sigma.y)(y))$ and $[f \leftarrow e]e'' = \lambda(y':\tau.\lambda(x:\sigma.y)(y'))$ are quite different.

Thus dynamic scope does violence to notions of modularity or separation of concerns!

28

Summary

Functions are a fundamental programming language construct.

Dynamic binding is a bad idea because it violates fundamental principles of modularity.

29