

04—Typing and Denotational Semantics of simPL

CS4215: Programming Language Implementation

Martin Henz

February 3, 2012

Generated on Thursday 2 February, 2012, 18:17

- 1 Typing of simPL
 - Type Environments
 - Typing Relation for simPL
 - Type Safety of simPL
- 2 Denotational Semantics of simPL

Example

Is $x + 3$ well-typed?

Type Environments

A *Type environment*, denoted by Γ , keeps track of the type of identifiers appearing in the expression.

$\Gamma(x)$ returns the type that is known by environment Γ for the identifier x .

Environment Extension

If $\Gamma[x \leftarrow t]\Gamma'$, then Γ' behaves like Γ , except that the type of x is t .

Example

Let $\Gamma = \emptyset$.

$\emptyset[\text{AboutPi} \leftarrow \text{int}]\Gamma'$

$\Gamma'(\text{AboutPi}) = \text{int}$

$\Gamma'[\text{Square} \leftarrow \text{int} \rightarrow \text{int}]\Gamma''$

$\text{dom}(\Gamma'') = \{\text{AboutPi}, \text{Square}\}$

Typing Relation

The set of well-typed expressions is defined by the ternary *typing relation*, written $\Gamma \vdash E : t$, where Γ is a type environment such that $E \bowtie X$ and $X \subseteq \text{dom}(\Gamma)$.

“The expression E has type t , under the assumption that its free identifiers have the types given by Γ .”

Examples

- $\Gamma' \vdash \text{AboutPi} * 2 : \text{int}$
- $\Gamma'' \vdash \text{fun}\{\text{int} \rightarrow \text{int}\} x \rightarrow \text{AboutPi} * (\text{Square } 2) \text{ end} : \text{int} \rightarrow \text{int}$

Examples

$\Gamma' \vdash \text{fun } \{\text{int} \rightarrow \text{int}\} \text{ x} \rightarrow \text{AboutPi} * (\text{Square } 2) \text{ end} : \text{int} \rightarrow \text{int}$

does not hold, because `Square` occurs free in the expression, but the type environment Γ' to the left of the \vdash symbol is not defined for `Square`.

Definition of Typing Relation

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \text{[VarT]}$$

If $\Gamma(x)$ is not defined, then this rule is not applicable. In this case, we say that there is no type for x derivable from the assumptions Γ .

Definition of Typing Relation (cont'd)

————— [NumT]
 $\Gamma \vdash n : \text{int}$

————— [TrueT]
 $\Gamma \vdash \text{true} : \text{bool}$

————— [FalseT]
 $\Gamma \vdash \text{false} : \text{bool}$

Definition of Typing Relation (cont'd)

For each primitive operation p that takes n arguments of types t_1, \dots, t_n and returns a value of type t , we have exactly one rule of the following form.

$$\frac{\Gamma \vdash E_1 : t_1 \quad \dots \quad \Gamma \vdash E_n : t_n}{\Gamma \vdash p[E_1, \dots, E_n] : t} \text{ [PrimT]}$$

Definition of Typing Relation (cont'd)

ρ	t_1	t_2	t
+	int	int	int
-	int	int	int
*	int	int	int
/	int	int	int
&	bool	bool	bool
	bool	bool	bool
\	bool		bool
=	int	int	bool
<	int	int	bool
>	int	int	bool

Definition of Typing Relation (cont'd)

$$\Gamma \vdash E : \text{bool} \quad \Gamma \vdash E_1 : t \quad \Gamma \vdash E_2 : t$$

$$\Gamma \vdash \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ end} : t$$

Definition of Typing Relation (cont'd)

$$\Gamma_1[x_1 \leftarrow t_1]\Gamma_2 \cdots \Gamma_n[x_n \leftarrow t_n]\Gamma_{n+1} \quad \Gamma_{n+1} \vdash E : t$$

$$\Gamma_1 \vdash \text{fun } \{t_1 * \cdots * t_n \rightarrow t\} x_1 \dots x_n \rightarrow E \text{ end} : t_1 * \cdots * t_n \rightarrow t$$

Definition of Typing Relation (cont'd)

$$\begin{array}{l} \Gamma[f \leftarrow t_1 * \dots * t_n \rightarrow t] \Gamma_1 \\ \Gamma_1[x_1 \leftarrow t_1] \Gamma_2 \dots \Gamma_n[x_n \leftarrow t_n] \Gamma_{n+1} \\ \Gamma_{n+1} \vdash E : t \end{array}$$

$$\Gamma \vdash \text{recfun } f \{t_1 * \dots * t_n \rightarrow t\} x_1 \dots x_n \rightarrow E \text{ end} : t_1 * \dots * t_n \rightarrow t$$

Definition of Typing Relation (cont'd)

$$\Gamma \vdash E : t_1 * \dots * t_n \rightarrow t \quad \Gamma \vdash E_1 : t_1 \quad \dots \quad \Gamma \vdash E_n : t_n$$

$$\Gamma \vdash (E E_1 \dots E_n) : t$$

Well-Typedness

An expression E is well-typed, if there is a type t such that $E : t$.

Example Proof

$$\frac{}{\emptyset \vdash 2 : \text{int}}$$
$$\frac{}{\emptyset \vdash 3 : \text{int}}$$
$$\frac{}{\emptyset \vdash 2 * 3 : \text{int}}$$
$$\frac{}{\emptyset \vdash 7 : \text{int}}$$
$$\frac{}{\emptyset \vdash 2 * 3 > 7 : \text{bool}}$$

Example Proof

$$\begin{array}{c}
 \frac{\frac{}{\Gamma \vdash x : \text{int}} \quad \frac{}{\Gamma \vdash 1 : \text{int}}}{\frac{}{\Gamma \vdash x+1 : \text{int}}} \\
 \frac{\emptyset[x \leftarrow \text{int}]\Gamma \quad \frac{}{\Gamma \vdash x+1 : \text{int}}}{\frac{}{\emptyset \vdash \text{fun } \{\text{int} \rightarrow \text{int}\} x \rightarrow x+1 \text{ end} : \text{int} \rightarrow \text{int}}} \\
 \frac{}{\emptyset \vdash \text{fun } \{\text{int} \rightarrow \text{int}\} x \rightarrow x+1 \text{ end} : \text{int} \rightarrow \text{int}} \quad \frac{}{\emptyset \vdash 2 : \text{int}} \\
 \frac{}{\emptyset \vdash (\text{fun } \{\text{int} \rightarrow \text{int}\} x \rightarrow x+1 \text{ end } 2) : \text{int}}
 \end{array}$$

Unique Type

Lemma

For every expression E and every type assignment Γ , there exists at most one type t such that $\Gamma \vdash E : t$.

More Properties of Typing Relation

Lemma

Typing is not affected by “junk” in the type assignment. If $\Gamma \vdash E : t$, and $\Gamma \subset \Gamma'$, then $\Gamma' \vdash E : t$.

Lemma

Substituting an identifier by an expression of the same type does not affect typing. If $\Gamma[x \leftarrow t']\Gamma'$, $\Gamma' \vdash E : t$, and $\Gamma \vdash E' : t'$, then $\Gamma \vdash E'' : t$, where $E[x \leftarrow E']E''$.

Type Safety

Type safety is a property of a given language with a given static and dynamic semantics. It says that if a program of the language is well-typed, certain problems are guaranteed not to occur at runtime.

What do we consider as “problems”?

Components of Type Safety

Progress. Well-typed expressions are values or can be further evaluated.

Preservation. Well-typed expressions do not change their type during evaluation.

Definition of Type Safety

A programming language with a given typing relation $\dots \vdash \dots : \dots$ and one-step evaluation \mapsto is called type-safe, if the following two conditions hold:

- ① **Preservation.** If E is a well-typed program with respect to $\dots \vdash \dots : \dots$ and $E \mapsto E'$, then E' is also a well-typed program with respect to \vdash .
- ② **Progress.** If E is a well-typed program, then either E is a value or there exists a program E' such that $E \mapsto E'$.

Preservation in simPL

If for a simPL expression E and some type t holds $E : t$ and if $E \mapsto_{\text{simPL}} E'$, then $E' : t$.

Progress in simPL

Let simPL' be simPL without division.

If for a simPL' expression E holds $E : t$ for some type t , then either E is a value, or there exists an expression E' such that $E \mapsto_{\text{simPL}'} E'$.

Is perfect typing possible?

The type safety of simPL' ensures that evaluation of a well-typed simPL' expression does not get stuck.

Can we say the reverse by claiming that any expression for which the dynamic semantics produces a value is well-typed?

Is perfect typing possible?

Unfortunately, some expressions produce a value although they are not well-typed. Example:

```
if true then 1 else false end
```

Interesting to note: Perfect typing is not possible for languages such as simPL'.

Stepping back

Summary so far

- Typing allows us to focus on well-typed programs
- Well-typed programs “behave well” (progress, preservation)

Outlook

We will focus on well-typed programs and develop a semantics that eliminates many of the efficiency and engineering issues encountered with dynamic semantics.

1 Typing of simPL

2 Denotational Semantics of simPL

- Denotational Semantics of simPL0
- Denotational Semantics of simPL1
- Denotational Semantics of simPL2
- Denotational Semantics of simPL3
- simPL4

A Critique of Current Approach

- Contraction relies substitution; mathematically rather complex.
- Primitive operations that are not total functions, such as division, make the evaluation process get stuck. We want a more explicit way of handling runtime errors.
- simPL contains many “unreasonable” programs, which complicates the definition of a dynamic semantics
- Dynamic semantics cannot be extended easily to other language paradigms such as imperative programming.

A Glimpse of Denotational Semantics

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{+[E_1, E_2] \mapsto v_1 + v_2}$$

- syntactic domains
- semantic domains
- semantic functions

Sublanguages of `simPL`

- `simPL0`; integer and boolean expressions
- `simPL1`; add `let` and `if`
- `simPL2`; add division
- `simPL3`; add functions
- `simPL4`; add recursive functions

Syntactic Domain of simPL0

$$\frac{}{n}$$

$$\frac{}{\text{true}}$$

$$\frac{}{\text{false}}$$

$$\frac{E_1 \quad E_2}{p[E_1, E_2]} \quad p \in \{ |, \&, +, -, *, =, >, < \}$$

$$\frac{E}{p[E]} \quad p \in \{ \backslash \}$$

Semantic Domains

Domain name	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\dots, -2, -1, 0, 1, 2, \dots\}$	ring of integers
EV	Bool + Int	expressible values

Expressible values **EV** are values that are the result of evaluating an expression.

Semantic Function

The semantic function

$$\cdot \mapsto \cdot : \mathbf{simPL0} \rightarrow \mathbf{EV}$$

expresses the meaning of elements of **simPL0**, by defining the value of each element.

Semantic Rules

$$\begin{array}{ccc}
 \frac{}{\text{true} \mapsto \text{true}} & \frac{}{\text{false} \mapsto \text{false}} & \frac{n \mapsto_{\mathbf{N}} i}{n \mapsto i}
 \end{array}$$

The function $\mapsto_{\mathbf{N}}$ transforms the simPL0 integer syntax into an element of **Int**.

Semantic Rules (cont'd)

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{E_1 + E_2 \mapsto v_1 + v_2}$$

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{E_1 - E_2 \mapsto v_1 - v_2}$$

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{E_1 * E_2 \mapsto v_1 \cdot v_2}$$

Semantic Rules (cont'd)

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{E_1 \& E_2 \mapsto v_1 \wedge v_2}$$

$$\frac{E \mapsto v}{\backslash E \mapsto \neg v}$$

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{E_1 | E_2 \mapsto v_1 \vee v_2}$$

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{E_1 = E_2 \mapsto v_1 \equiv v_2}$$

Semantic Rules (cont'd)

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{E_1 > E_2 \mapsto v_1 > v_2}$$

$$\frac{E_1 \mapsto v_1 \quad E_2 \mapsto v_2}{E_1 < E_2 \mapsto v_1 < v_2}$$

Example

$1 + 2 > 3 \mapsto \text{false}$ holds because $1 + 2 \mapsto 3$
and $3 > 3$ is *false*.

simPL1

Add the following to simPL0:

- conditionals,
- identifiers,
- let

Syntactic Domain of simPL1

$E \quad E_1 \quad E_2$

if E then E_1 else E_2 end

$E \quad E_1 \quad \dots \quad E_n$

x

let $x_1 = E_1 \dots x_n = E_n$ in E end

Motivation: Semantic Domains of simPL1

We need to introduce environments that allow us to keep track of the binding of identifiers. These environments map identifiers to *denotable values*.

Semantic Domains of simPL1

Semantic domain	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\dots, -2, -1, 0, 1, 2, \dots\}$	ring of integers
EV	Bool + Int	expressible values
DV	Bool + Int	denotable values
Id	alphanumeric string	identifiers
Env	Id \rightsquigarrow DV	environments

Operations on Environments

We introduce an operation $\Delta[x \leftarrow v]$,
which denotes an environment that works like Δ ,
except that $\Delta[x \leftarrow v](x) = v$

Semantic Functions for simPL1

Define $\cdot \mapsto \cdot$ using auxiliary semantic function $\cdot \Vdash \cdot \mapsto \cdot$ that gets an environment as additional argument.

$\cdot \mapsto \cdot : \mathbf{simPL1} \rightarrow \mathbf{EV}$

$$\emptyset \Vdash E \mapsto v$$

$$E \mapsto v$$

Auxiliary Semantic Function for `simPL1`

$\cdot \Vdash \cdot \rightsquigarrow \cdot : \mathbf{Env} * \text{simPL1} \rightarrow \mathbf{EV}$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash \text{true} \rightsquigarrow \text{true}$$
$$n \rightsquigarrow_{\mathbf{N}} i$$

$$\Delta \Vdash n \rightsquigarrow i$$

$$\Delta \Vdash \text{false} \rightsquigarrow \text{false}$$

$$\Delta \Vdash x \rightsquigarrow \Delta(x)$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash E_1 + E_2 \rightsquigarrow v_1 + v_2$$

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash E_1 - E_2 \rightsquigarrow v_1 - v_2$$

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash E_1 * E_2 \rightsquigarrow v_1 \cdot v_2$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash E_1 \& E_2 \rightsquigarrow v_1 \wedge v_2$$

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash E_1 \mid E_2 \rightsquigarrow v_1 \vee v_2$$

$$\Delta \Vdash E \rightsquigarrow v$$

$$\Delta \Vdash \backslash E \rightsquigarrow \neg v$$

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash E_1 = E_2 \rightsquigarrow v_1 \equiv v_2$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash E_1 > E_2 \rightsquigarrow v_1 > v_2$$

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash E_1 < E_2 \rightsquigarrow v_1 < v_2$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash E \rightsquigarrow true \quad \Delta \Vdash E_1 \rightsquigarrow v_1$$

$$\Delta \Vdash \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ end} \rightsquigarrow v_1$$

$$\Delta \Vdash E \rightsquigarrow false \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ end} \rightsquigarrow v_2$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n] \Vdash E \rightsquigarrow v \quad \Delta \Vdash E_1 \rightsquigarrow v_1 \quad \cdots \quad \Delta \Vdash E_n \rightsquigarrow v_n$$

$$\Delta \Vdash \text{let } x_1 = E_1 \cdots x_n = E_n \text{ in } E \text{ end} \rightsquigarrow v$$

Example

...

$$\emptyset[\text{AboutPi} \leftarrow 3] \Vdash \text{AboutPi} + 2 \rightsquigarrow 5$$

$$\emptyset \Vdash \text{let AboutPi} = 3 \text{ in AboutPi} + 2 \text{ end} \rightsquigarrow 5$$

$$\text{let AboutPi} = 3 \text{ in AboutPi} + 2 \text{ end} \rightsquigarrow 5$$

Syntactic Domain of simPL2

The language simPL2 adds division to simPL1.

$$\frac{E_1 \quad E_2}{E_1/E_2}$$

The difficulty lies in the fact that division on integers is a partial function, not being defined for 0 as second argument.

Semantic Domains of simPL2

Domain name	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\dots, -2, -1, 0, 1, 2, \dots\}$	ring of integers
EV	Bool + Int + $\{\perp\}$	expressible values
DV	Bool + Int	denotable values
Id	alphanumeric string	identifiers
Env	Id \rightsquigarrow DV	environments

Modify Auxiliary Semantic Function

The semantic function $\cdot \Vdash \cdot \mapsto \cdot$ is modified to take the occurrence of the error value \perp into account.

Auxiliary Semantic Function (cont'd)

$$\begin{array}{c}
 \Delta \Vdash E_1 \rightsquigarrow \perp \\
 \hline
 \Delta \Vdash E_1 + E_2 \rightsquigarrow \perp
 \end{array}
 \qquad
 \begin{array}{c}
 \Delta \Vdash E_2 \rightsquigarrow \perp \\
 \hline
 \Delta \Vdash E_1 + E_2 \rightsquigarrow \perp
 \end{array}$$

$$\begin{array}{c}
 \Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2 \\
 \hline
 \Delta \Vdash E_1 + E_2 \rightsquigarrow v_1 + v_2 \quad \text{if } v_1, v_2 \neq \perp
 \end{array}$$

Auxiliary Semantic Function (cont'd)

$$\Delta \Vdash E_1 \rightsquigarrow \perp$$

$$\Delta \Vdash E_1/E_2 \rightsquigarrow \perp$$

$$\Delta \Vdash E_2 \rightsquigarrow \perp$$

$$\Delta \Vdash E_1/E_2 \rightsquigarrow \perp$$

$$\Delta \Vdash E_2 \rightsquigarrow 0$$

$$\Delta \Vdash E_1/E_2 \rightsquigarrow \perp$$

$$\Delta \Vdash E_1 \rightsquigarrow v_1 \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

if $v_1, v_2 \neq \perp$ and $v_2 \neq 0$

$$\Delta \Vdash E_1/E_2 \rightsquigarrow v_1/v_2$$

Auxiliary Semantic Function (cont'd)

$$\Delta \Vdash E \rightsquigarrow \perp$$

$$\Delta \Vdash \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ end} \rightsquigarrow \perp$$

$$\Delta \Vdash E \rightsquigarrow \text{true} \quad \Delta \Vdash E_1 \rightsquigarrow v_1$$

$$\Delta \Vdash \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ end} \rightsquigarrow v_1$$

$$\Delta \Vdash E \rightsquigarrow \text{false} \quad \Delta \Vdash E_2 \rightsquigarrow v_2$$

$$\Delta \Vdash \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ end} \rightsquigarrow v_2$$

Auxiliary Semantic Function (cont'd)

$$\Delta \Vdash E_i \rightsquigarrow \perp$$

for $i, 1 \leq i \leq n$

$$\Delta \Vdash \text{let } x_1 = E_1 \cdots x_n = E_n \text{ in } E \text{ end} \rightsquigarrow \perp$$

$$\Delta[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n] \Vdash E \rightsquigarrow v \quad \Delta \Vdash E_1 \rightsquigarrow v_1 \cdots \Delta \Vdash E_n \rightsquigarrow v_n$$

$$\Delta \Vdash \text{let } x_1 = E_1 \cdots x_n = E_n \text{ in } E \text{ end} \rightsquigarrow v$$

otw'

Example

For any environment Δ ,
 $\Delta \Vdash 5+(3/0) \rightsquigarrow \perp$, since
 $\Delta \Vdash 3/0 \rightsquigarrow \perp$.

Syntactic Domain of simPL3

Add (non-recursive) function definition and application.

$$\frac{E}{\text{fun } \{ t \} x_1 \dots x_n \rightarrow E \text{ end}}$$

$$\frac{E \ E_1 \ \dots \ E_n}{(E \ E_1 \ \dots \ E_n)}$$

Semantic Domains of simPL3

Domain name	Definition	Explanation
Bool	$\{true, false\}$	ring of booleans
Int	$\{\dots, -2, -1, 0, 1, 2, \dots\}$	ring of integers
EV	Bool + Int + $\{\perp\}$ + Fun	expressible values
DV	Bool + Int + Fun	denotable values
Id	alphanumeric string	identifiers
Env	Id \rightsquigarrow DV	environments
Fun	DV $*$ \dots $*$ DV \rightsquigarrow EV	function values

Auxiliary Semantic Function (cont'd)

$\Delta \Vdash \text{fun } \{ t \} x_1 \dots x_n \rightarrow E \text{ end} \rightsquigarrow f$

f is function such that

$f(y_1, \dots, y_n) = v$, where

$\Delta[x_1 \leftarrow y_1] \cdots [x_n \leftarrow y_n] \Vdash E \rightsquigarrow v$

Evaluation of Function Application

$$\Delta \Vdash E \rightsquigarrow f \quad \Delta \Vdash E_1 \rightsquigarrow v_1 \quad \dots \quad \Delta \Vdash E_n \rightsquigarrow v_n$$

$$\Delta \Vdash (E \ E_1 \ \dots \ E_n) \rightsquigarrow f(v_1, \dots, v_n)$$

Syntactic Domain of simPL4

E

`recfun f { t } $x_1 \dots x_n \rightarrow E$ end`

Discussion simPL4

We would like to add the following rule to our definition of \mapsto .

$$\Delta \Vdash \text{recfun } g \{ t \} x_1 \dots x_n \rightarrow E \text{ end } \mapsto v$$

f is function such that

$$f(y_1, \dots, y_n) = v, \text{ where}$$

$$\Delta[x_1 \leftarrow y_1] \cdots [x_n \leftarrow y_n]$$

$$[g \leftarrow f] \Vdash E \mapsto v$$

Problem

The symbol f occurs on the right hand side of the definition of f . How do we know that such a function f exists? Some expressions have unique solutions for f , others have multiple solutions.

Example:

```
(recfun f {int -> int} x -> (f x) end 0)
```

Theory of fixpoints ...

Overview of Next Lecture

Friday 10/2:

- A Virtual Machine for `simPL`