Computability and Self-interpretation

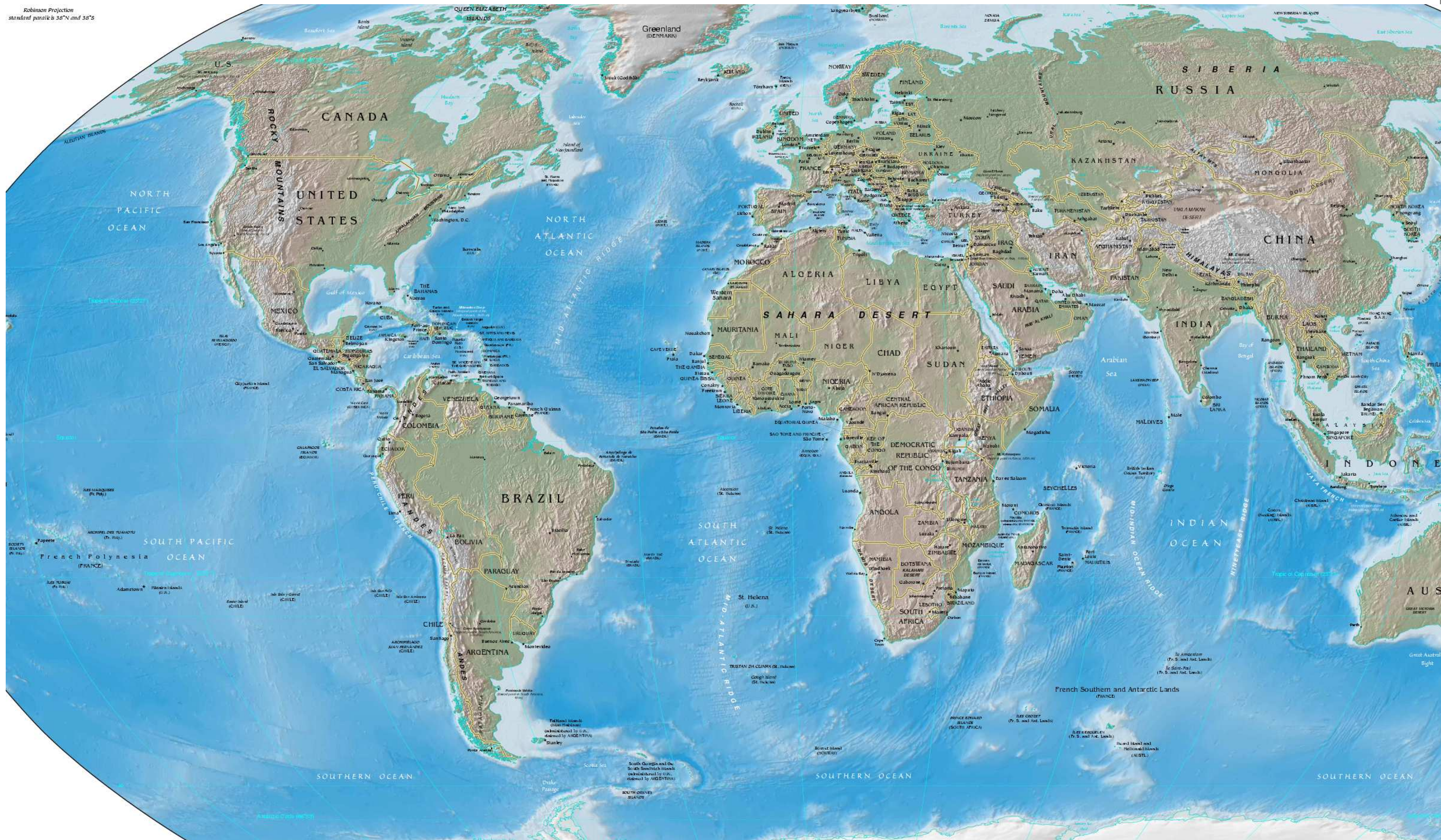
Bengt Nordström

bengt@cs.chalmers.se

ChungAng University, Seoul, Korea

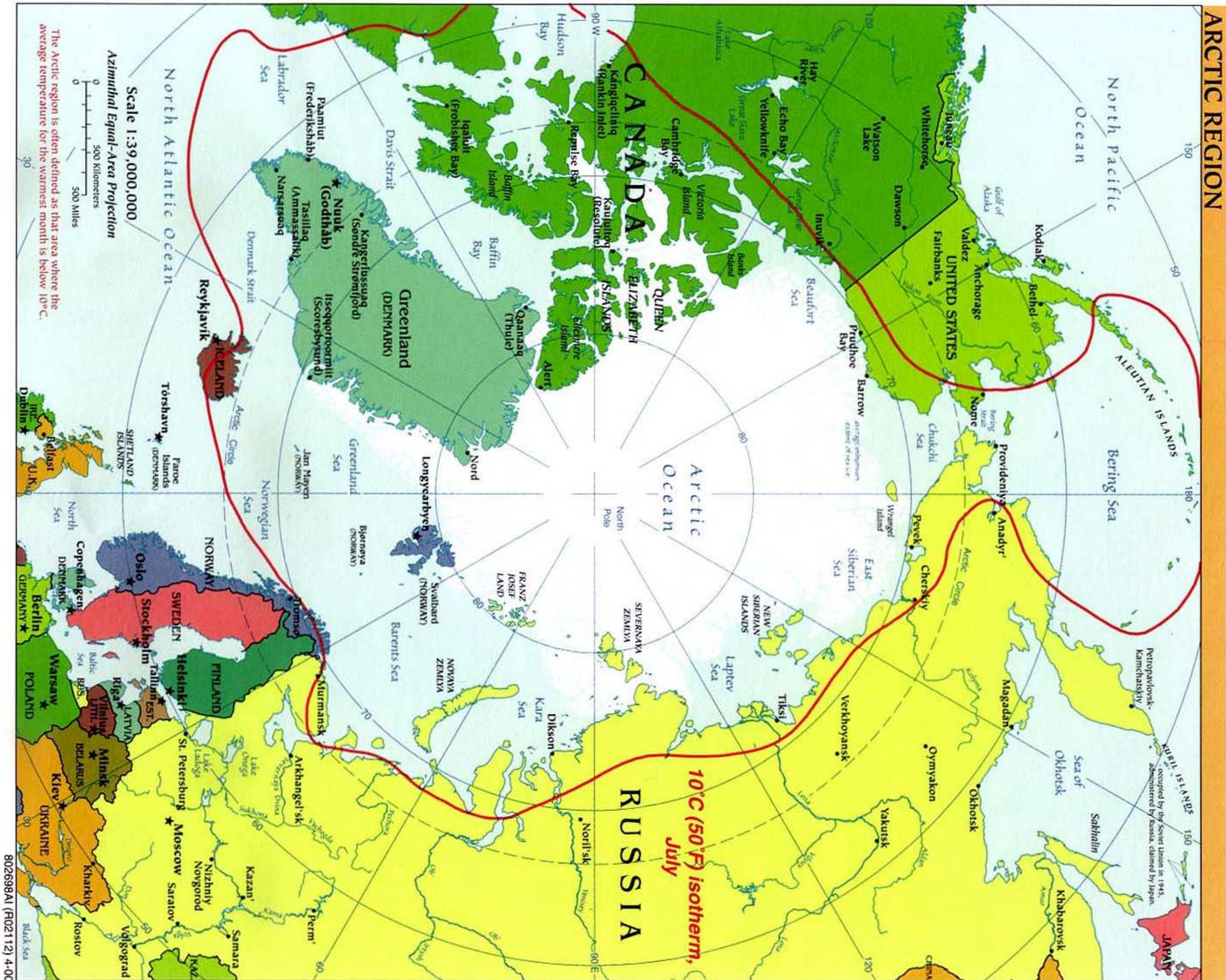
on leave from Chalmers University, Göteborg, Sweden

The world



Sweden and Korea

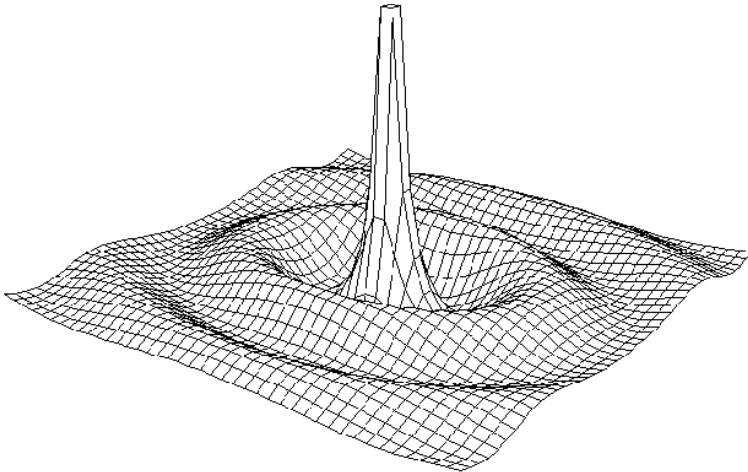
The arctic region



What is a function?

- Functions in mathematics: The function

$$f(x,y) = \frac{4 * cos\sqrt{2 * x^2 + y^2}}{\sqrt{x^2 + y^2}}$$



- and in a programming language:

```
f(x,y) = 4*cos(sqrt(2*x^2+y^2))/sqrt(x^2+y^2
```

Deterministic functions

- Functions in programming languages are in general *not* deterministic (not the same output for the same input)
- A function in mathematics is always deterministic (since we want $f(n) = f(n)$)

Is this the only difference?

Definition

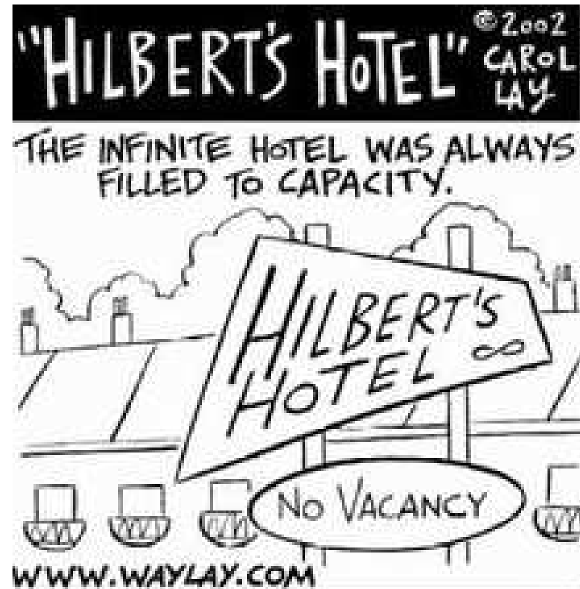
- We say that f is a (*mathematical*) *function* if it is a set of pairs
$$\{(n_1, m_1), \ldots, (n_i, m_i), \ldots\}$$
such that the first component uniquely decides the second component, i.e. if both (n, m) and (n, k) are in f then $m = k$.
- A *program* (*function in a programming language*) is a method which when given an object a either terminates with an object b or does not terminate.

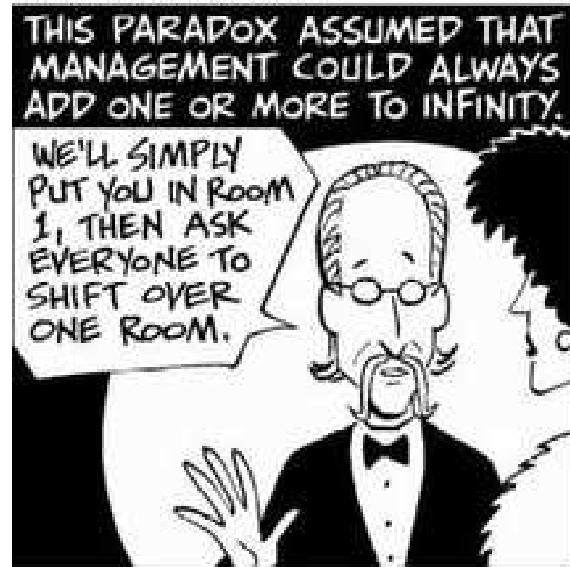
Are these function concepts the same?

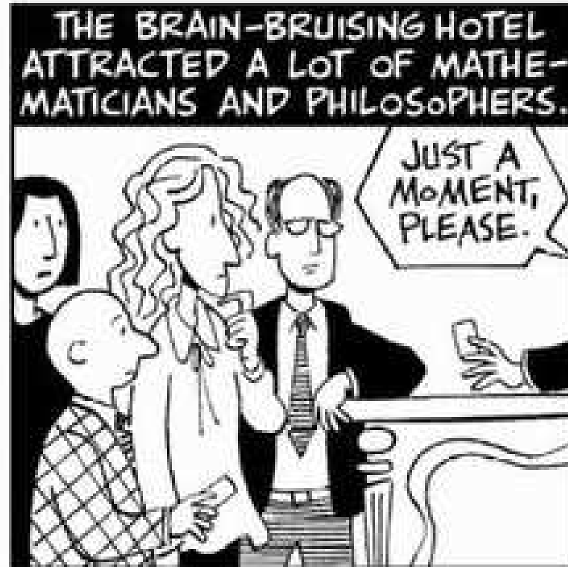
There are more functions than programs

There are more functions taking natural numbers as input and output then program taking natural numbers as input and output.
What does this mean?

Hilbert's Hotel







Not all functions can be implemented

- The set of programs fit in Hilbert's hotel.
- The set of functions from N to N does not.

Why?

Examples of noncomputable functions:

It is an open mathematical problem whether Goldbach's conjecture is true.

Conjecture 1 (Goldbach) *Every even number greater than 2 can be written as a sum of two prime numbers*

We can then ask ourselves if the constant function $g \in \mathbf{N} \rightarrow \mathbf{N}$ defined by

$$g\ n = \begin{cases} 1 & \text{if Goldbach's conjecture is true,} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

is computable?

Another example

Your friend claims that he has written a program $\mathbf{halts}$ such that

$$\mathbf{halts}\ x = \begin{cases} \mathbf{true} & \text{if the computation of } x \text{ terminates,} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

Do you believe him? Let's write down this program:

$$T\ x = \ \text{ if } \mathbf{halts}\ x \text{ then } \mathbf{loop} \text{ else } \mathbf{true}$$

The program $\mathbf{loop}$ is a program which never terminates. We now see that the program T reverses the termination behaviour of its argument, i.e.

$T\ a$ terminates if and only if a does not terminate.

Another example, contd.

So, if your friend succeeded to write the program $\mathbf{halts}$, then you have been able to construct a program T such that

$T\ a$ terminates if and only if a does not terminate.

And this holds for all boolean values a! In particular for the boolean value s defined recursively by

$$s = T\ s$$

Does s terminate? From the definition of s it terminates if $T\ s$ terminates. But this terminates if the argument s does not terminate. So s terminates if s does not terminate. We have a contradiction, so your friend must be lying when he said that he had written the $\mathbf{halts}$ program.

Conclusion:

It is impossible to write a program $\mathbf{halts}$ such that

$$\mathbf{halts}\ x = \begin{cases} \mathbf{true} & \text{if the computation of } x \text{ terminates,} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

This is a typical result in the scientific field *Computability*, a field which started around 70 years ago by people like Church, Turing, von Neuman, Markov, Curry and Kleene. They were interested in what mathematical problems cannot be solved by computers. It is interesting to note that this was before the computer age. In fact, their results stimulated the building of the first computers.

To represent a program as data

We can represent a program as data. For instance as a text string (source code) or as a tree (abstract syntax tree).
A self-interpreter for a language L is a program in L which can interpret an arbitrary L-program. So, it takes a representation of a program p as input and outputs a representation of the result of executing the program p.

Can we make a simple self-interpreter?

This is not simple: the language must be simple so that the interpreter is simple. But it must also be powerful so that a simple program can express its interpreter.
Think about making a self-interpreter for Java!

Description of a Programming Language

A precise description of a programming language consists of the following parts:

- Concrete syntax (for instance using BNF)
- Abstract syntax
- Semantics (informal and formal)
- Syntactic conventions

Concrete syntax of χ

$\text{“}\textit{any character string}\text{“}$	string
$e_1\ e_2$	application
$\lambda x \rightarrow e$	abstraction
$\mathsf{C}\ e$	constructor application
$\mathbf{case}\ e\ \mathbf{of}\ \{C_1 : e_1,\ \ldots\}$	case-expression
$\{l_1 = e_1;\ \ldots\}$	structure
$e.\mathsf{l}$	projection
$\mathbf{rec}\ x = e\ \mathbf{end}$	recursion
x	variable

Constructors are identifiers starting with a capital letter. Variables and labels starts with small letters. The variable e stands for an expression in χ, $\mathbf{x}$ ranges over variables, $\mathbf{C}$ over constructors and $\mathbf{l}$ over labels.

Abstract syntax

We assume that $e \in \chi, i \in \mathbf{Id}, t \in \mathbf{T}(\mathbf{Id}, \chi)$. The set χ is defined by the following inductive definition:

$$
\begin{array}{l}
\mathbf{string}(i) \in \mathbf{Exp} \quad \text{if} \quad i \in \mathbf{Id} \\
\mathbf{apply}(e_1,\ e_2) \in \chi \\
\mathbf{lambda}(i, e) \in \chi \\
\mathbf{constr}(i, e) \in \chi \\
\mathbf{case}(e, t) \in \chi \\
\mathbf{struct}(t) \in \chi \\
\mathbf{proj}((, e), i) \in \chi \\
\mathbf{rec}(i, e) \in \chi \\
\mathbf{var}(i) \in \chi
\end{array}
$$

Informal semantics

- The simplest program is a string “$abc\cdots$“, it computes to itself.
- The program $\lambda x \rightarrow e$ is a value, it cannot be further computed.
- The program $e\ e'$ is computed by fi rst computing the value of e. If this is $\lambda x \rightarrow e$ then we continue by computing the value of the expression obtained by substituting e' for all free occurrences of x in e.
- The program $(\mathbf{c}\ e)$ is a value, it cannot be further computed.
- The program $\mathbf{case}\ e\ \mathbf{of}\ \{\mathbf{C_1} : e_1,\ \ldots\}$ is computed by fi rst computing the value of e. If this is on the form $\mathbf{C_i}\ e$ then we continue by computing the value of $(e_i\ e)$.
- The program $\{\mathbf{l_1} = e_1;\ \ldots\}$ is a value.
- The program e.l is computed by fi rst computing the value of e. If this is on the form $\{\mathbf{l_1} = e_1;\ \ldots\}$ then we continue by computing the value of e_i.
- Finally, the program $\mathbf{rec}\ i = e\ \mathbf{end}$ is computed by computing the value of the expression obtained by substituting $\mathbf{rec}\ i = e\ \mathbf{end}$ for all free occurrences of the identifi er $\mathbf{i}$ in the expression e.

Operational semantics

$$\frac{e_1 \longrightarrow \mathbf{lambda}(i, e_3) \qquad e_3[i \leftarrow e_2] \longrightarrow d}{\mathbf{apply}(e_1, e_2) \longrightarrow d}$$

$$\frac{e \longrightarrow \mathbf{constr}(i, e') \qquad \mathbf{lookup}(t, i, e'') \qquad \mathbf{apply}(e'', e') \longrightarrow d}{\mathbf{case}(e, t) \longrightarrow d}$$

$$\frac{e \longrightarrow \mathbf{struct}(t) \qquad \mathbf{lookup}(t, i, e') \qquad e' \longrightarrow d}{\mathbf{proj}((, e), i) \longrightarrow d}$$

$$\frac{e[i \leftarrow \mathbf{rec}(i, e)] \longrightarrow d}{\mathbf{rec}(i, e) \longrightarrow d}$$

$$\mathbf{lambda}(i, e) \longrightarrow \mathbf{lambda}(i, e)$$

$$\mathbf{constr}(i, e) \longrightarrow \mathbf{constr}(i, e)$$

$$\mathbf{struct}(t) \longrightarrow \mathbf{struct}(t)$$

The substitution operation

$$\begin{aligned}
\mathbf{apply}(e_1,\ e_2)[i \leftarrow e] &= \mathbf{apply}(e_1[i \leftarrow e],\ e_2[i \leftarrow e]) \\
\mathbf{lambda}(i, e')[i \leftarrow e] &= \mathbf{lambda}(i, e') \\
\mathbf{lambda}(j, e')[i \leftarrow e] &= \mathbf{lambda}(j, e'[i \leftarrow e]) \quad \text{if } i \neq j \\
\mathbf{constr}(j, e')[i \leftarrow e] &= \mathbf{constr}(j, e'[i \leftarrow e]) \quad \text{constructors are not substituted} \\
\mathbf{case}(e', t)[i \leftarrow e] &= \mathbf{case}(e'[i \leftarrow e], t[i \leftarrow e]_T) \\
\mathbf{struct}(t)[i \leftarrow e] &= \mathbf{struct}(t[i \leftarrow e]_T) \\
\mathbf{proj}((,e)', i)[i \leftarrow e] &= \mathbf{proj}((,)[e' \leftarrow i]e, i) \quad \text{labels are not substituted} \\
\mathbf{rec}(i, e')[i \leftarrow e] &= \mathbf{rec}(i, e') \\
\mathbf{rec}(j, e')[i \leftarrow e] &= \mathbf{rec}(j, e'[i \leftarrow e]) \quad \text{if } i \neq j \\
\mathbf{var}(i)[i \leftarrow e] &= e \\
\mathbf{var}(j)[i \leftarrow e] &= \mathbf{var}(j) \quad \text{if } i \neq j
\end{aligned}$$

We assume that $e, e' \in \chi$, e is closed and $i \in \mathbf{Id}$. The expression $e'[i \leftarrow e]$ stands for the expression obtained by substituting e for all free occurences of the variable

Substitution on tables

$$\phi[i \leftarrow e]_T = \phi$$
$$(t, j : e')[i \leftarrow e]_T = (t[i \leftarrow e]_T, j : e'[i \leftarrow e])$$

An ordinary substitution on each entry in the table.

Examples of syntactic conventions

Local definitions

The expression $\mathbf{let}\ x = e_1\ \mathbf{in}\ e$ stands for the expression $(\lambda x{\rightarrow}e\ e_1)$. The expression $\mathbf{let}\ x_1 = e_1, x_2 = e_2\ \mathbf{in}\ e$ stands for $\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ (\mathbf{let}\ x_2 = e_2\ \mathbf{in}\ e)$ and so on.

Tuples

The expression $\langle e_1, \ldots, e_n \rangle$, where $n \geq 0$ stands for the expression $\{\text{arg1} = e_1; \ldots; \text{argn} = e_n\}$, so $\langle\rangle$ is another way of writing $\{\}$ and $\langle a, b \rangle$ stands for $\{\text{arg1} = a; \text{arg2} = b\}$, etc.

Examples of syntactic conventions, cont'd

Pattern-matching on tuples

The expression $\lambda\langle x_1, \ldots, x_n\rangle \rightarrow e$ stands for the expression $\lambda x \rightarrow \mathbf{let}\ x_1 = x.\text{arg1}, \ldots, x_n = x.\text{argn}\ \mathbf{in}\ e$

Pattern-matching

Inside a case-expression $\mathbf{case}\ e\ \mathbf{of}\ \{\mathbf{c_1} : e_1,\ \ldots\}$ the expression $\mathbf{c_1} : e_1$ is called a case-branch.

There will be certain convenient ways of writing case-branches:

A case-branch of the form $\mathbf{C}\ p : e$ will stand for $\mathbf{C} : \lambda p \rightarrow e$, where the pattern p is either a variable or a pattern of the form $\langle x_1, \ldots, x_n\rangle$.

Examples of syntactic conventions, cont'd

These abbreviations are made so that we can use a more traditional style of writing case-expressions, for instance the program

$$\textbf{case C } a \textbf{ of } \{\textbf{C } x : e\}$$

computes to the value of $e[x \leftarrow a]$, the expression obtained by substituting a for all free occurrences of the variable x in e. etc.

The self-evaluator for χ

Application:

$$\frac{e_1 \longrightarrow \mathbf{lambda}(i, e_3) \qquad e_3[i \leftarrow e_2] \longrightarrow d}{\mathbf{apply}(e_1, e_2) \longrightarrow d}$$

$$\begin{aligned}\textbf{eval}\ \textsf{apply}\langle e_1, e_2\rangle =&\textbf{case eval}\ e_1\ \textbf{of}\ \{\\ &\textsf{lambda}\langle i, e_3\rangle : \textbf{eval}\ (\textbf{subst}\ e_3\ i\ e_2)\}\end{aligned}$$

self-evaluator, cont'd

Pattern-matching:

$$\frac{e \longrightarrow \mathbf{constr}(i, e') \qquad \mathbf{lookup}(t, i, e'') \qquad \mathbf{apply}(e'', e') \longrightarrow d}{\mathbf{case}(e, t) \longrightarrow d}$$

$$\begin{aligned}\textbf{eval}\ \textsf{case}\langle e, t\rangle = &\textbf{case eval}\ e\ \textbf{of}\ \{\\ &\textsf{constr}\langle i, e'\rangle : \textbf{let}\ e'' = \textbf{lookup}\ t\ i\ \textbf{in eval}\ \textsf{apply}\langle e'', e'\rangle\}\end{aligned}$$

self-evaluator, cont'd

Projection:

$$\frac{e \longrightarrow \mathbf{struct}(t) \qquad \mathbf{lookup}(t,i,e') \qquad e' \longrightarrow d}{\mathbf{proj}((,e),i) \longrightarrow d}$$

$$\begin{aligned}\mathbf{eval}\ \mathsf{proj}\langle e,i\rangle =& \mathbf{case\ eval}\ e\ \mathbf{of}\ \{\\ & \mathsf{struct}\langle t\rangle : \mathbf{let}\ e' = \mathbf{lookup}\ t\ i\ \mathbf{in\ eval}\ e'\}\end{aligned}$$

Recursion

$$\frac{e[i \leftarrow \mathbf{rec}(i,e)] \longrightarrow d}{\mathbf{rec}(i,e) \longrightarrow d}$$

$$\mathbf{eval}\ \mathsf{rec}\langle i,e\rangle = \mathbf{eval}\ (\mathbf{subst}\ e\ i\ (\mathsf{rec}\langle i,e\rangle))$$

self-evaluator, cont'd

String:

$$\mathbf{string}(i) \longrightarrow \mathbf{string}(i)$$

$$\mathbf{eval}\ \mathsf{string}\langle i\rangle = \mathsf{string}\langle i\rangle$$

Abstraction:

$$\mathbf{lambda}(i, e) \longrightarrow \mathbf{lambda}(i, e)$$

$$\mathbf{eval}\ \mathsf{lambda}\langle i, e\rangle = \mathsf{lambda}\langle i, e\rangle$$

Constructor application:

$$\mathbf{constr}(i, e) \longrightarrow \mathbf{constr}(i, e)$$

$$\mathbf{eval}\ \mathsf{constr}\langle i, e\rangle = \mathsf{constr}\langle i, e\rangle$$

Structure:

$$\mathbf{struct}(t) \longrightarrow \mathbf{struct}(t)$$

Putting the equations together:

$$
\begin{array}{l}
\textbf{rec eval} = \lambda\mathbf{p}\rightarrow\textbf{case } p \textbf{ of } \{ \\
\quad \mathsf{apply}\langle e_1, e_2\rangle : \quad \textbf{case eval } e_1 \textbf{ of } \{ \\
\qquad\qquad \mathsf{lambda}\langle i, e_3\rangle : \textbf{eval } (\textbf{subst } e_3\ i\ e_2)\} \\
\quad \mathsf{case}\langle e, t\rangle : \quad \textbf{case eval } e \textbf{ of } \{ \\
\qquad\qquad \mathsf{constr}\langle i, e'\rangle : \textbf{let } e'' = \textbf{lookup } t\ i \textbf{ in eval } \mathsf{apply}\langle e'', e'\rangle\} \\
\quad \mathsf{proj}\langle e, i\rangle : \quad \textbf{case eval } e \textbf{ of } \{\mathsf{struct}\langle t\rangle : \textbf{let } e' = \textbf{lookup } t\ i \textbf{ in eval } e'\} \\
\quad \mathsf{rec}\langle i, e\rangle : \quad \textbf{eval } (\textbf{subst } e\ i\ (\mathsf{rec}\langle i, e\rangle)) \\
\quad \mathsf{string}\langle i\rangle : \mathsf{string}\langle i\rangle \\
\quad \mathsf{lambda}\langle i, e\rangle : \mathsf{lambda}\langle i, e\rangle \\
\quad \mathsf{constr}\langle i, e\rangle : \mathsf{constr}\langle i, e\rangle \\
\quad \mathsf{struct}\langle t\rangle : \mathsf{struct}\langle t\rangle\} \\
\textbf{end}
\end{array}
$$