Bengt Nordström,[1]
Department of Computing Science,
Chalmers and University of Göteborg,
Göteborg, Sweden

2016-11-14

1 The language χ

The main purpose of the language χ is to illustrate some basic results in computability. The standard computational models (Turing machines, Church's λ-calculus, Post's deduction systems, Herbrand's and Gödel's general recursive functions etc) were designed in the 1930's long before the development of modern programming languages. For somebody used to modern functional programming languages these models seem rather awkard, in particular the complicated codings of programs and other inductively defined objects using (a coding of) natural numbers. The strategy used is to first find a representation of natural numbers and then using some version of prime factorization to represent other inductively defined objects. It is simpler to use a computation model with a direct representation of inductively defined objects.

Compared to the untyped λ-calculus, the language χ has constructors and case-expressions. There is also an explicit operator for general recursion.

Some of the expressions in the language are called values (canonical expressions), they are the expressions which are the result of a computation.

Definition 1 (value) *An expression is called a value if it computes to itself.*

A value is either a lambda-expression or an application of a primitive constant (constructor) to a list of values.

Here is a list of all ways of forming programs and their informal semantics:

- The program $\lambda\, x.\ e$ is a function which takes one argument (denoted by the variable x) to an expression e (which may depend on x). This program is a value, it cannot be computed further.

[1]The document has been modified by Nils Anders Danielsson. The main changes:

- N-ary application, n-ary lambdas and nullary constants have been replaced by binary application, unary lambdas and n-ary constructor applications.
- Constructors and variables are now encoded by (representations of) natural numbers, not single constructors.
- Call-by-name has been replaced by call-by-value. A section about equality between programs has been removed.
- The old definition of χ-computable allowed the program to give ill-formed output when the function was undefined. Now the program must fail to terminate.

- A constructor application $\mathsf{c}(e_1, \ldots, e_n)$ is a value if all of the arguments are values. Note that the number of arguments, n, can be zero. The intuition is that c is like a constructor in (strict) functional programming languages, it is used to represent elements in inductively defined types.

- The program $(e_1\ e_2)$ is the application of the function e_1 to the argument e_2. It is computed by first computing the value of e_1. If the resulting value is on the form $\lambda\, x.\ e$ then the value of e_2 is computed. If this expression has the value d, then the value is the value of $e[x \leftarrow d]$, the expression obtained by substituting the expression d for all free occurrences of the variable x in e.

- The program **case** d **of** $\{\mathsf{c}_1(x_1, \ldots, x_{n_1}) \to e_1; \ldots\}$ expresses pattern matching. The value of the expression is obtained by first computing the value of d. If this is on the form $\mathsf{c}_i(d_1, \ldots, d_{n_i})$ then the value is the value of $e_i[x_{n_i} \leftarrow d_{n_i}] \ldots [x_1 \leftarrow d_1]$.

- Finally, the program **rec** $x = e$ is a recursive program. It is computed by computing the value of the substitution $e[x \leftarrow$ **rec** $x = e\,]$.

The remainder of this section contains a more precise description of the language.

1.1 Concrete syntax of the core language

We assume that we have two disjoint sets of identifiers: variables and constructors. The variable e ranges over expressions in χ, x ranges over variables and c over constructors.

The concrete syntax of the language χ is shown in Figure 1. In the text superfluous parentheses around expressions will sometimes be used. Furthermore parentheses around applications will sometimes be omitted. Such applications should be read in a left-associative way. For instance, the expression $e_1\ e_2\ e_3$ stands for $((e_1\ e_2)\ e_3)$.

1.2 Abstract syntax

In order to be precise about the operational semantics, we first give the abstract syntax of the language as an inductively defined set Exp. The relation between the concrete and abstract syntax should be obvious. It is the task of a parser to translate from the concrete to the abstract syntax.

The set Exp is defined by the inductive definition in Figure 2. The definition assumes that we have a set Var of variables and a set Const of constants (both

$(e_1\ e_2)$	application
$\lambda x.\ e$	abstraction
$\textbf{case}\ e\ \textbf{of}\ \{\mathsf{c}_1(x_1, \ldots, x_{n_1}) \rightarrow e_1;\ \ldots\}$	case-expression
$\textbf{rec}\ x = e$	recursion
x	variable
$\mathsf{c}(e_1, \ldots, e_n)$	constructor application

Figure 1: Concrete syntax

$$\begin{array}{rll}
\text{apply}(e_1, e_2) \in \text{Exp} & \text{if} & e_1, e_2 \in \text{Exp} \\
\text{lambda}(x, e) \in \text{Exp} & \text{if} & x \in \text{Var}, e \in \text{Exp} \\
\text{case}(e, bs) \in \text{Exp} & \text{if} & e \in \text{Exp}, bs \in \text{List}(\text{Br}) \\
\text{rec}(x, e) \in \text{Exp} & \text{if} & x \in \text{Var}, e \in \text{Exp} \\
\text{var}(x) \in \text{Exp} & \text{if} & x \in \text{Var} \\
\text{const}(c, es) \in \text{Exp} & \text{if} & c \in \text{Const}, es \in \text{List}(\text{Exp}) \\
\text{branch}(c, xs, e) \in \text{Br} & \text{if} & c \in \text{Const}, xs \in \text{List}(\text{Var}), e \in \text{Exp}
\end{array}$$

Figure 2: Inductive definition of the abstract syntax

in bijective correspondence with the natural numbers), as well as sets $\text{List}(A)$ of lists of elements from sets A. The set $\text{List}(A)$ has elements of the form nil or $\text{cons}(a, as)$, where $a \in A$ and $as \in \text{List}(A)$.

1.3 Operational semantics based on substitutions

The operational semantics will be an inductive definition of the computation relation. The relation

$$e \longrightarrow d,$$

defined in Figure 3, expresses that the computation of the expression e has the value d. It is a relation between closed expressions.

The relation is defined using two substitution operations. If $e, e' \in \text{Exp}$, e is closed, and $x \in \text{Var}$, then $e'[x \leftarrow e]$ stands for the expression obtained by substituting e for all free occurrences of the variable $\text{var}(x)$ in e'. This operation is defined in detail in the appendix. The operation is used to give a semantics for applications of lambda abstractions. It is somewhat more tricky to give a semantics

$$\frac{e_1 \longrightarrow \mathrm{lambda}(x, e) \qquad e_2 \longrightarrow d_2 \qquad e[x \leftarrow d_2] \longrightarrow d}{\mathrm{apply}(e_1, e_2) \longrightarrow d}$$

$$\frac{e \longrightarrow \mathrm{const}(c, es) \qquad \mathrm{lookup}(c, bs, xs, e') \qquad e'[xs \leftarrow es] \mapsto e'' \qquad e'' \longrightarrow d}{\mathrm{case}(e, bs) \longrightarrow d}$$

$$\frac{e[x \leftarrow \mathrm{rec}(x, e)] \longrightarrow d}{\mathrm{rec}(x, e) \longrightarrow d}$$

$$\frac{}{\mathrm{lambda}(x, e) \longrightarrow \mathrm{lambda}(x, e)}$$

$$\frac{es \longrightarrow_L ds}{\mathrm{const}(c, es) \longrightarrow \mathrm{const}(c, ds)}$$

$$\frac{}{\mathrm{nil} \longrightarrow_L \mathrm{nil}}$$

$$\frac{e \longrightarrow d \quad es \longrightarrow_L ds}{\mathrm{cons}(e, es) \longrightarrow_L \mathrm{cons}(d, ds)}$$

Figure 3: Operational semantics based on substitution

for case expressions, because the list of expressions in a constructor application may not have the same length as the list of variables in a matching branch. The *relation* $e[xs \leftarrow es] \mapsto e'$ is inhabited (for $e, e' \in \mathrm{Exp}$, $es \in \mathrm{List(Exp)}$ with all expressions closed, and $xs \in \mathrm{List(Var)}$) if xs and es have the same length n, and in that case e' is $e[xs_n \leftarrow es_n] \ldots [xs_1 \leftarrow es_1]$ (where xs_n is the n-th element in xs, and similarly for the other indexed expressions). Note that the order of the individual substitution operations matters if several variables in xs are equal. The relation is defined in the following way:

$$\frac{}{e[\mathrm{nil} \leftarrow \mathrm{nil}] \mapsto e}$$

$$\frac{e[xs \leftarrow ds] \mapsto e'}{e[\mathrm{cons}(x, xs) \leftarrow \mathrm{cons}(d, ds)] \mapsto e'[x \leftarrow d]}$$

The operational semantics also makes use of the lookup relation. Intuitively, $\mathrm{lookup}(c, bs, xs, e)$ (for $c \in \mathrm{Const}$, $bs \in \mathrm{List(Br)}$, $xs \in \mathrm{List(Var)}$ and $e \in \mathrm{Exp}$) is true if the first branch associated to the constant c in the list bs is $\mathrm{branch}(c, xs, e)$.

It is inductively defined by the following clauses:

$$\frac{}{\mathrm{lookup}(c, \mathrm{cons}(\mathrm{branch}(c, xs, e), bs), xs, e)}$$

$$\frac{\mathrm{lookup}(c, bs, xs, e) \qquad c \neq c'}{\mathrm{lookup}(c, \mathrm{cons}(\mathrm{branch}(c', xs', e'), bs), xs, e)}$$

This is an *inductive* definition, so there is no other way of giving a direct proof that the lookup relation is true. In particular, this means that $\mathrm{lookup}(c, \mathrm{nil}, xs, e)$ is always false.

The operational semantics also makes use of the $\longrightarrow_L$-relation. This relation describes how a list is computed: we just compute each element of it.

We can see from the definition of the operational semantics that the $\longrightarrow$-relation is deterministic, i.e. a partial function. Notice also that the value of a program is on one of the following forms:

- $\mathrm{lambda}(x, e)$, where $x \in \mathrm{Var}$, $e \in \mathrm{Exp}$, or
- $\mathrm{const}(c, es)$, where $c \in \mathrm{Const}$, $es \in \mathrm{List}(\mathrm{Exp})$, and every element in es is a value.

There is no need for the subprogram e to be evaluated. We are not evaluating inside lambdas, because we want to use a naive definition of substitution.

2 The extensional halting problem

It is not possible to construct a closed program halt_E in χ with the property that, for any closed expression p,

$$\mathrm{halt}_E\ (\lambda x.\ p) \longrightarrow \begin{cases} \mathsf{True}(), & \text{if } p \text{ terminates,} \\ \mathsf{False}(), & \text{otherwise,} \end{cases} \tag{1}$$

where "p terminates" means that there is some v such that $p \longrightarrow v$. We will prove this by assuming that there is such a program halt_E and deriving a contradiction.

Note the use of a lambda in $\mathrm{halt}_E\ (\lambda x.\ p)$. If p does not terminate, then $\mathrm{halt}_E\ p$ always fails to terminate. It may not be so surprising that it is impossible to implement halt_E: What could this program do with its argument $\lambda x.\ p$? Running it, by applying it to, say, $\mathsf{True}()$, does not work, because p might not terminate.

Let us now prove that halt_E cannot be defined by constructing a function on closed expressions that inverts the termination behaviour of its argument. We could define a function $\textbf{terminv}_E$ in the following way:

$$\begin{aligned}&\textbf{terminv}_E \in \text{Exp} \to \text{Exp}\\&\textbf{terminv}_E(x) =_{\mathsf{def}} \textbf{case}\ \text{halt}_E\ x\ \textbf{of}\ \{\mathsf{True}() \to \textbf{loop}; \mathsf{False}() \to \mathsf{Zero}()\}\end{aligned}$$

Note that the concrete syntax in the right-hand side of the definition of $\textbf{terminv}_E$ should be interpreted as abstract syntax, and that x is not a χ variable, but a meta-variable that stands for an element in Exp. The term **loop** is also not a χ variable, but an abbreviation that stands for a non-terminating program:

$$\textbf{loop} =_{\mathsf{def}} \textbf{rec}\ x = x$$

We notice that the program $\textbf{terminv}_E(x)$ terminates if and only if the closed program x does not terminate. This holds for all closed programs x, in particular for the program

$$\textbf{strange} =_{\mathsf{def}} \textbf{rec}\ x = \textbf{terminv}_E(x)$$

We can now see that **strange** stands for a program which terminates if and only if the program $\textbf{terminv}_E(\textbf{strange})$ terminates. Thus **strange** terminates if and only if it does not terminate, contradicting the assumption that there is a program halt_E.

The version of the halting problem which we have discussed here could be called the extensional halting problem, because we give a program as input to the function which is to decide if the program terminates. This means that the function which is to decide termination cannot inspect the syntax of the program. Maybe it is possible to solve the halting problem if we work with a more syntactical representation of the programs? We will later show that this is not the case. In order to state this problem in a precise way we need to introduce some more concepts.

3 Representing mathematical objects in χ

There is a simple way to represent an element in an arbitrary inductively defined set as a program in χ. If the element has the form

$$c(e_1, \ldots, e_n)$$

and we represent e_i by $\ulcorner e_i \urcorner$, then we can represent the element by inventing a new constructor c and define the representation by

$$\ulcorner c(e_1, \ldots, e_n) \urcorner =_{\mathsf{def}} \mathsf{c}(\ulcorner e_1 \urcorner, \ldots, \ulcorner e_n \urcorner).$$

We call this the *standard representation* of an inductively defined set.

3.1 Representing N in χ

As an example, the elements in the set of natural numbers, N, are represented as the following expressions in our programming language:

$$\begin{array}{l}\mathsf{Zero}(),\\ \mathsf{Succ}(\mathsf{Zero}()),\end{array}$$

and so on.

Notice that these programs are written in the concrete syntax of the programming language χ. If we want to see them as mathematical objects in the set Exp then they are written (**c**$_{\text{Const}}$ stands for the element in Const which corresponds to the identifier **c**)

$$\begin{array}{l}\mathrm{const}(\mathsf{Zero}_{\mathrm{Const}}, \mathrm{nil}),\\ \mathrm{const}(\mathsf{Succ}_{\mathrm{Const}}, \mathrm{cons}(\mathrm{const}(\mathsf{Zero}_{\mathrm{Const}}, \mathrm{nil}), \mathrm{nil})),\end{array}$$

and so on.

Thus the mathematical object 1 (as an element in the set N) is very different from its representation $\ulcorner \mathsf{1} \urcorner$ as an element in the mathematical set Exp. However, the concrete syntax of 1 in χ is chosen so that there is almost no distinction.

3.2 Representing χ-programs in χ

The standard representation gives us a possibility to represent elements in an inductively defined set as programs in χ. But the set Exp representing the abstract syntax of programs is itself an inductively defined set! So, we can use this technique to represent χ-programs in χ by using the standard representation of its abstract syntax:

$$\begin{aligned}\ulcorner \mathrm{apply}(e_1, e_2) \urcorner &= \mathsf{Apply}(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner)\\ \ulcorner \mathrm{lambda}(x, e) \urcorner &= \mathsf{Lambda}(\ulcorner x \urcorner, \ulcorner e \urcorner)\\ \ulcorner \mathrm{case}(e, bs) \urcorner &= \mathsf{Case}(\ulcorner e \urcorner, \ulcorner bs \urcorner)\\ \ulcorner \mathrm{rec}(x, e) \urcorner &= \mathsf{Rec}(\ulcorner x \urcorner, \ulcorner e \urcorner)\\ \ulcorner \mathrm{var}(x) \urcorner &= \mathsf{Var}(\ulcorner x \urcorner)\\ \ulcorner \mathrm{const}(c, es) \urcorner &= \mathsf{Const}(\ulcorner c \urcorner, \ulcorner es \urcorner)\\ \ulcorner \mathrm{branch}(c, xs, e) \urcorner &= \mathsf{Branch}(\ulcorner c \urcorner, \ulcorner xs \urcorner, \ulcorner e \urcorner)\end{aligned}$$

Elements in List(A) are also encoded in the standard way:

$$\begin{aligned}\ulcorner \mathrm{nil} \urcorner &= \mathsf{Nil}()\\ \ulcorner \mathrm{cons}(a, as) \urcorner &= \mathsf{Cons}(\ulcorner a \urcorner, \ulcorner as \urcorner)\end{aligned}$$

We assumed above that Var and Const are in bijective correspondence with the natural numbers, so we can use natural numbers, encoded in the standard way, to represent variables and constants.

3.2.1 A comment on the representation

As an example, the program which applies the identity function to zero is in the concrete syntax written as

$$(\lambda x.\, x)\ \mathsf{Zero}().$$

This program corresponds to the following mathematical object (element in Exp):

$$\text{apply}(\text{lambda}(x_{\text{Var}}, \text{var}(x_{\text{Var}})), \text{const}(\mathsf{Zero}_{\text{Const}}, \text{nil}))$$

This object is in turn represented by the following expression:

$$\mathsf{Apply}(\mathsf{Lambda}(\ulcorner x_{\text{Var}} \urcorner, \mathsf{Var}(\ulcorner x_{\text{Var}} \urcorner)), \mathsf{Const}(\ulcorner \mathsf{Zero}_{\text{Const}} \urcorner, \mathsf{Nil}()))$$

If we assume that the variable x_{Var} and the constructor $\mathsf{zero}_{\text{Const}}$ are both represented by the natural number 0, then we get the following, final expression:

$$\mathsf{Apply}(\mathsf{Lambda}(\mathsf{Zero}(), \mathsf{Var}(\mathsf{Zero}())), \mathsf{Const}(\mathsf{Zero}(), \mathsf{Nil}()))$$

This is how we are able to represent a program as a value inside the language itself.

4 Representing a program as a mathematical object

We just showed how it is possible to represent an object in an inductively defined set as a program in χ. A natural question to ask is how one can go in the other direction. How can we see a program as a mathematical object? There are two ways.

4.1 The intensional representation

We have just seen how we can see a program p as an element $p \in$ Exp. In this exposition we identify a program with its abstract syntax as represented in the set Exp. This is the intensional view of a program. The mathematical object which represents the program carries all “relevant” syntactic information. If we were explicit about it, we would distinguish a program “p“ from its abstract syntax p (which is a mathematical object in the set Exp). We can see the step from p to “p“ as printing the program p. The inverse step is parsing.

4.2 The extensional representation

It is also possible to think of the mathematical object which a program stands for as its value (which is an element in Exp). This is the view of denotational semantics. We can define a partial function $[\![\cdot]\!] \in \mathrm{Exp} \overset{\sim}{\rightarrow} \mathrm{Exp}$ by

$$[\![p]\!] = v \quad \text{iff} \quad p \longrightarrow v.$$

So, when we are using our extensional glasses $[\![\cdot]\!]$, we see no differences between programs which have equal values.

Note that this representation is still somewhat intensional. For instance, the α-equivalent programs $\lambda\, x.\ x$ and $\lambda\, y.\ y$ are seen as different, as are the programs $\lambda\, x.\ x$ and $\lambda\, x.\ (\lambda\, y.\ y)x$.

5 Programming in χ

Let us now look at a simple χ program that adds two natural numbers. To construct the addition function we proceed as follows. We want to construct a program `add` which satisfies the following equations:[2]

```
add Zero()  m = m                                 (1)
add Succ(n) m = Succ(add n m)
```

It is clear that we can replace these equations by one:

```
add l m = case l of                               (2)
            { Zero()  -> m
            ; Succ(n) -> Succ(add n m)
            }
```

Note that if (2) holds for all l and m, then

```
add Zero() m
 =
case Zero() of { Zero()  -> m
               ; Succ(n) -> Succ(add n m)
               }
 = {by the computation rule for case}
m
```

and

[2]It is left as an exercise for the reader to come up with a suitable definition of what the equality sign means here.

```
add Succ(n) m
 =
case Succ(n) of { Zero()  -> m
                ; Succ(n) -> Succ(add n m)
                }
 = {by the computation rule for case}
Succ(add n m)
```

We can move the variables of the left hand side in (2) to the right hand side:

```
add = \l. \m. case l of                              (3)
                { Zero()  -> m
                ; Succ(n) -> Succ(add n m)
                }
```

So, we have to find the fixpoint `add` solving the equation

```
add = F add
```

where `F` stands for the following program:

```
\a. \l. \m. case l of
              { Zero()  -> m
              ; Succ(n) -> Succ(a n m)
              }
```

These kinds of fixpoints are found by the recursion operator:

```
rec add = F add
```

If we call this program `adds`, we notice that

```
adds
 = {external definition}
rec add = F add
 = {computation rule of rec}
F (rec add = F add)
 = {external definition}
F adds
```

So, we can finally define addition by the following program:

```
rec add = \l. \m.
       case l of
         { Zero()  -> m
         ; Succ(n) -> Succ(add n m)
         }
```

6 An interpreter for χ in χ

Definition 2 *A self-evaluator for the language χ is a program* eval *in χ such that*

- *if* $p \longrightarrow v$ *then* $\mathrm{eval}\ \ulcorner p \urcorner \longrightarrow \ulcorner v \urcorner$*, and*
- *if* $\mathrm{eval}\ \ulcorner p \urcorner \longrightarrow v'$ *then* $p \longrightarrow v$ *for some v satisfying* $\ulcorner v \urcorner = v'$.

Or, expressed differently: $[\![\mathrm{eval}\ \ulcorner p \urcorner]\!] = \ulcorner [\![p]\!] \urcorner$. Here, $a = b$ means that either both a and b are defined and equal, or both a and b are undefined.

We have noticed that it is easy to represent the abstract syntax of a program in χ as an expression in χ. We will now continue and define an interpreter for the language in itself. We start with the weak evaluation relation, $\longrightarrow$, and treat one clause of the inductive definition of the operational semantics at a time:

Application:

$$\frac{e_1 \longrightarrow \mathrm{lambda}(x, e) \qquad e_2 \longrightarrow d_2 \qquad e[x \leftarrow d_2] \longrightarrow d}{\mathrm{apply}(e_1, e_2) \longrightarrow d}$$

Here we assume that we have access to an implementation of substitution, `subst`:

```
eval Apply(e1, e2) =
  case eval e1 of
    { Lambda(x, e) -> eval (subst x (eval e2) e)
    }
```

Case:

$$\frac{e \longrightarrow \mathrm{const}(c, es) \qquad \mathrm{lookup}(c, bs, xs, e') \qquad e'[xs \leftarrow es] \mapsto e'' \qquad e'' \longrightarrow d}{\mathrm{case}(e, bs) \longrightarrow d}$$

Here we assume that we have access to a `lookup` operation, as well as substitution for lists, `substs`:

```
eval Case(e, bs) =
  case eval e of
    { Const(c, es) -> case lookup c bs of
      { Branch(_, xs, e) -> eval (substs xs es e)
      }
    }
```

Recursion

$$\frac{e[x \leftarrow \mathrm{rec}(x, e)] \longrightarrow d}{\mathrm{rec}(x, e) \longrightarrow d}$$

```
eval Rec(x, e) = eval (subst x Rec(x, e) e)
```

Abstraction:

$$\frac{}{\mathsf{lambda}(x, e) \longrightarrow \mathsf{lambda}(x, e)}$$

```
eval Lambda(x, e) = Lambda(x, e)
```

Constructor

$$\frac{es \longrightarrow_L ds}{\mathsf{const}(c, es) \longrightarrow \mathsf{const}(c, ds)}$$

Here we assume that we have access to a `map` function:

```
eval Const(c, es) = Const(c, map eval es)
```

To summarize, our aim is to construct a program `eval` which solves the following system of five equations:

```
eval Apply(e1, e2) =
  case eval e1 of
    { Lambda(x, e) -> eval (subst x (eval e2) e)
    }
eval Case(e, bs) =
  case eval e of
    { Const(c, es) -> case lookup c bs of
      { Branch(_, xs, e) -> eval (substs xs es e)
      }
    }
eval Rec(x, e) = eval (subst x Rec(x, e) e)
eval Lambda(x, e) = Lambda(x, e)
eval Const(c, es) = Const(c, map eval es)
```

We can solve these equations using the recursion operator and a case-expression:

```
rec eval = \p. case p of
  { Apply(e1, e2) ->
      case eval e1 of
        { Lambda(x, e) -> eval (subst x (eval e2) e)
        }
  ; Case(e, bs) ->
      case eval e of
        { Const(c, es) -> case lookup c bs of
          { Branch(_, xs, e) -> eval (substs xs es e)
          }
        }
  ; Rec(x, e)    -> eval (subst x Rec(x, e) e)
  ; Lambda(x, e) -> Lambda(x, e)
  ; Const(c, es) -> Const(c, map eval es)
  }
```

Filling in the missing pieces above is left as an exercise for the reader.

7 Computability

Suppose that A and B are inductively defined sets which can be represented in χ. If $a \in A$ then $\ulcorner a \urcorner$ is the representation of a in χ.

Definition 3 (A program computes a function) *If $\phi \in A \rightharpoonup B$ then we say that the program p in χ computes the partial function ϕ if, for all $a \in A$,*

- *if $\phi(a)$ is defined, then $p \ulcorner a \urcorner \longrightarrow \ulcorner \phi(a) \urcorner$, and*
- *if $p \ulcorner a \urcorner \longrightarrow b$ for some b, then $\phi(a)$ is defined and $b = \ulcorner \phi(a) \urcorner$.*

(Or, alternatively, for all $a \in A$, $[\![p \ulcorner a \urcorner]\!] = \ulcorner \phi(a) \urcorner$.)

Definition 4 (χ-computable) *We say that a partial function is* χ-computable *if there is a program in χ which computes it.*

As an example, if $\phi \in (N \times N) \rightharpoonup N$ then $p \in \chi$ computes ϕ if, for all n, $m \in N$, $[\![p\ \mathsf{Pair}(\ulcorner n \urcorner, \ulcorner m \urcorner)]\!] = \ulcorner \phi(n, m) \urcorner$. Notice that the self-evaluator is a proof that the $\longrightarrow$-relation is computable.

8 Examples of noncomputable functions

8.1 The intensional halting problem of self-application is not computable

Consider the partial function $\Theta \in \text{Exp} \rightharpoonup \text{Bool}$ defined by

$$\Theta(p) = \begin{cases} \text{true}, & \text{if } p \ulcorner p \urcorner \text{ terminates,} \\ \text{false}, & \text{otherwise.} \end{cases} \tag{2}$$

The program p is an element in Exp and $\ulcorner p \urcorner$ is its representation in χ. Notice that both p and $\ulcorner p \urcorner$ are elements in the set Exp!

The function Θ is not computable. If it were, then there would exist a program **selfhalts** in χ which computes Θ, i.e.

$$[\![\textbf{selfhalts}\ \ulcorner p \urcorner]\!] = \ulcorner \Theta(p) \urcorner.$$

Such a program **selfhalts** would have the property that

$$\textbf{selfhalts}\ \ulcorner p \urcorner \longrightarrow \begin{cases} \mathsf{True}(), & \text{if } p \ulcorner p \urcorner \text{ terminates,} \\ \mathsf{False}(), & \text{otherwise.} \end{cases} \tag{3}$$

In a similar way as when we proved the extensional halting problem we can define the program

$$\mathbf{terminv}_I =_{\text{def}} \lambda x.\ \mathbf{case\ selfhalts}\ x\ \mathbf{of}\ \{\mathsf{True}() \to \mathbf{loop}; \mathsf{False}() \to \mathsf{Zero}()\}.$$

We notice that the program $\mathbf{terminv}_I \ulcorner p \urcorner$ terminates if and only if the program $p \ulcorner p \urcorner$ does not terminate. This fact holds for all programs p, in particular for $\mathbf{terminv}_I$ itself. In this case we can conclude that $\mathbf{terminv}_I \ulcorner \mathbf{terminv}_I \urcorner$ terminates if and only if the program $\mathbf{terminv}_I \ulcorner \mathbf{terminv}_I \urcorner$ does not terminate. This is clearly a contradiction and we have to conclude that Θ is not computable.

8.2 The intensional halting problem is not computable

Consider the partial function halts $\in$ Exp $\times$ Exp$\tilde{\to}$Bool defined by

$$\text{halts}(p, n) = \begin{cases} \text{true}, & \text{if } p \ulcorner n \urcorner \text{ terminates,} \\ \text{false}, & \text{otherwise.} \end{cases} \tag{4}$$

This function is not computable. If it were, then there would exist a program **phalts** in χ such that

$$\mathbf{phalts}\ \mathsf{Pair}(\ulcorner p \urcorner, \ulcorner n \urcorner) \longrightarrow \begin{cases} \mathsf{True}(), & \text{if } p \ulcorner n \urcorner \text{ terminates,} \\ \mathsf{False}(), & \text{otherwise.} \end{cases}$$

Suppose that x is a code of a program p. We can come to a contradiction if we are able to construct a program which computes to $\mathsf{True}()$ if $p \ulcorner p \urcorner$ terminates, and computes to $\mathsf{False}()$ otherwise (this would be a construction of the non-existing program **selfhalts** above).

But **phalts** $\mathsf{Pair}(x, x)$ is such a program.

Hence, the program

$$\lambda x.\ \mathbf{phalts}\ \mathsf{Pair}(x, x)$$

is a program, which computes the halting problem of self-application. We have reached a contradiction and our assumption that we can compute the extensional halting problem must be wrong.

8.3 Half of the halting problem is computable

The partial function halfhalts $\in$ Exp$\tilde{\to}$Bool defined by

$$\text{halfhalts}(p) = \begin{cases} \text{true}, & \text{if } p \text{ terminates,} \\ \text{undefined}, & \text{otherwise} \end{cases} \tag{5}$$

is computable.

We want to find a program **phalfhalts** in χ such that

$$\textbf{phalfhalts}\ \ulcorner p \urcorner \longrightarrow \mathsf{True}() \quad \text{if } p \text{ terminates.}$$

Note that Definition 2 implies that p terminates if and only if the program eval $\ulcorner p \urcorner$ terminates. Thus eval is almost a function that computes **phalfhalts**, the only difference is that if eval p terminates, then it computes to the representation of the value of p, while a function which computes halfhalts must terminate with $\mathsf{True}()$. We can thus implement **phalfhalts** by converting the output of eval, which in the relevant cases must be the representation of some value in Exp, to $\mathsf{True}()$:

```
\x. case eval x of
      { Lambda(x, e) -> True()
      ; Const(c, es) -> True()
      }
```

When we apply this program to a code $\ulcorner p \urcorner$, we see that it terminates with $\mathsf{True}()$ if p terminates, and is undefined otherwise.

A The substitution operation

If $e, e' \in$ Exp, e is closed and $x \in$ Var, then the expression $e'[x \leftarrow e]$ stands for the expression obtained by substituting e for all free occurrences of the variable var(x) in e'. It is defined in Figure 4 together with corresponding operations for branches and lists of expressions or branches.

$$
\begin{aligned}
\mathrm{apply}(e_1, e_2)[x \leftarrow e] &= \mathrm{apply}(e_1[x \leftarrow e], e_2[x \leftarrow e]) \\
\mathrm{lambda}(x, e')[x \leftarrow e] &= \mathrm{lambda}(x, e') \\
\mathrm{lambda}(y, e')[x \leftarrow e] &= \mathrm{lambda}(y, e'[x \leftarrow e]) \quad \text{if } x \neq y \\
\mathrm{case}(e', bs)[x \leftarrow e] &= \mathrm{case}(e'[x \leftarrow e], bs[x \leftarrow e]_{\mathrm{BL}}) \\
\mathrm{rec}(x, e')[x \leftarrow e] &= \mathrm{rec}(x, e') \\
\mathrm{rec}(y, e')[x \leftarrow e] &= \mathrm{rec}(y, e'[x \leftarrow e]) \quad \text{if } x \neq y \\
\mathrm{var}(x)[x \leftarrow e] &= e \\
\mathrm{var}(y)[x \leftarrow e] &= \mathrm{var}(y) \quad \text{if } x \neq y \\
\mathrm{const}(c, es)[x \leftarrow e] &= \mathrm{const}(c, es[x \leftarrow e]_{\mathrm{EL}})
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{branch}(c, xs, e')[x \leftarrow e]_{\mathrm{B}} &= \mathrm{branch}(c, xs, e') \quad \text{if } x \in xs \\
\mathrm{branch}(c, xs, e')[x \leftarrow e]_{\mathrm{B}} &= \mathrm{branch}(c, xs, e'[x \leftarrow e]) \quad \text{if } x \notin xs
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{nil}[x \leftarrow e]_{\mathrm{EL}} &= \mathrm{nil} \\
\mathrm{cons}(d, ds)[x \leftarrow e]_{\mathrm{EL}} &= \mathrm{cons}(d[x \leftarrow e], ds[x \leftarrow e]_{\mathrm{EL}})
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{nil}[x \leftarrow e]_{\mathrm{BL}} &= \mathrm{nil} \\
\mathrm{cons}(b, bs)[x \leftarrow e]_{\mathrm{BL}} &= \mathrm{cons}(b[x \leftarrow e]_{\mathrm{B}}, bs[x \leftarrow e]_{\mathrm{BL}})
\end{aligned}
$$

Figure 4: Definition of substitution