WORKING WITH T^+ IN MINLOG

KENJI MIYAMOTO

This note describes how to do programming in the term calculus T^+ of Minlog.

1. Building terms and types via parser

In this note we input types and terms by means of the procedures `py` and `pt` standing for *Parse tYpes* and for *Parse Terms*, respectively. These procedure take as an argument a string written in a special syntax representing a type or a term. Here strings are double quoted expressions (e.g. `"abcde"`) as common in programming practices.

1.1. **Types.**

Recall that types are defined by the following syntax in the theory of Minlog.

$$\tau, \sigma ::= \alpha \mid \iota_{\vec{\tau}} \mid \tau \to \sigma$$

where we assume α is a type variable, $\iota_{\vec{\alpha}}$ is an algebra with parameters $\vec{\alpha}$, and $\to$ is a constructor of the arrow type. When Minlog is loaded, some types including a type variable α are already available. Feeding a string `alpha` to the procedure `py`, we get a type variable α. Here lines staring with "`>`" are supposed to be entered by a user, and other lines are responses of Minlog.

```
> (py "alpha")
(tvar -1 "alpha")
```

The returned value `(tvar -1 "alpha")` is an internal representation of the type variable α which is not very helpful for us, although such a representation is useful for Minlog. We get a rather user friendly output by means of the procedure `pp` standing for *Pretty Printing* in the following way.

```
> (pp (py "alpha"))
alpha
```

Type variables can be indexed by natural numbers.

```
> (pp (py "alpha3"))
alpha3
```

When the number is omitted, it is internally indexed by -1 but through `pp` the index is not displayed. In order to declare a new type variable name, use the procedure `add-tvar-name`.

```
> (add-tvar-name "sigma" "tau")
ok, type variable sigma added
ok, type variable tau added
```

Arrow types are formed by `=>`. For example to make an arrow type from α to α you can apply `py` to a string `"alpha=>alpha"`.

Date: 10. Juni 2016.

```
> (pp (py "alpha=>alpha"))
alpha=>alpha
```

Algebras are usable as base types.

```
> (pp (py "unit=>boole"))
unit=>boole
```

If an algebra has parameters (for example the sum type $\alpha_0 + \alpha_1$), we need to feed the type parameters. The sum type and the product type in Minlog are defined as algebras `ysum` and `yprod`, respectively, and are available as follows.

```
> (pp (py "alpha0 ysum alpha1"))
alpha0 ysum alpha1
> (pp (py "alpha0 yprod alpha1"))
alpha0 yprod alpha1
```

There is also a product type `@@` which is distinct from `yprod`. It is not defined as an algebra, but given as a primitive type in the same sense as the arrow type.

```
> (pp (py "alpha0@@alpha1"))
alpha0@@alpha1
```

1.2. **Terms.** Recall that the syntax of terms is defined as follows in the theory.

$$t, s ::= x \mid \lambda_x t \mid ts \mid C \mid D$$

where x is a variable, $\lambda_x t$ is an abstraction, ts is an application, C is a constructor and D is defined constant. By means of `pt` variable names are parsed as terms of variables. Strings representing types to `py` work as variable names to `pt`. Parentheses can be omitted when there is no confusion.

```
> (pt "alpha0 ysum alpha1")
(term-in-var-form
  (alg "ysum" (tvar 0 "alpha") (tvar 1 "alpha"))
  (var (alg "ysum" (tvar 0 "alpha") (tvar 1 "alpha"))
       -1
       1
       ""))
> (pp (pt "alpha0 ysum alpha1"))
(alpha0 ysum alpha1)
> (pp (pt "(alpha0 ysum alpha1)"))
(alpha0 ysum alpha1)
```

It is possible to add new variable names by the procedure `add-var-name`. In the following lines we use `x`, `b`, and `f` as variable names of the types α, boolean and $\alpha \to \alpha$, respectively.

```
> (add-var-name "x" (py "alpha"))
ok, variable x: alpha added
> (add-var-name "b" (py "boole"))
ok, variable b: boole added
> (add-var-name "f" (py "alpha=>alpha"))
ok, variable f: alpha=>alpha added
> (add-var-name "y" (py "alpha1"))
```

```
ok, variable y: alpha1 added
> (add-var-name "z" (py "alpha2"))
ok, variable z: alpha2 added
```

Indices are used in the following way. Note that the underscore is used to avoid confusion when a type name is used as a variable name.

```
> (pp (pt "f0"))
f0
> (pp (pt "f_0"))
f0
> (pp (pt "alpha0 ysum alpha1_0"))
(alpha0 ysum alpha1)_0
> (pp (pt "alpha0 ysum alpha1_0"))
(alpha0 ysum alpha1)_0
> (pp (pt "alpha0 ysum alpha_0"))
(alpha0 ysum alpha)_0
```

For example, `(pp (pt "(alpha0 ysum alpha1)0"))` does not yield an expected result. In order to reset the declared variable name, the variable name has to be once removed.

```
> (remove-variable-name "f")
ok, variable f is removed
```

Now it is possible to add a variable name `f` again.

There is a special syntax for abstractions. We use brackets instead of lambda, where commas ("`,`") can be used for listing variables. The following is examples for $\lambda_x x$ and $\lambda_{x_0,x_1} x_0$.

```
> (pp (pt "[x]x"))
[x]x
> (pp (pt "[x0,x1]x0"))
[x0,x1]x0
```

Applications ts are written as `t s`. For example $\lambda_{g^{\alpha\to\alpha_1\to\alpha_2},f^{\alpha\to\alpha_1},x^\alpha}(gx(fx))$ is dealt with as follows.

```
> (add-var-name "g" (py "alpha=>alpha1=>alpha2"))
ok, variable g: alpha=>alpha1=>alpha2 added
> (pp (pt "[g,f,x]g x(f x)"))
[g,f,x]g x(f x)
```

The type of term can be seen by using the procedure `term-to-type`.

```
> (pp (term-to-type (pt "[g,f,x]g x(f x)")))
(alpha=>alpha1=>alpha2)=>(alpha=>alpha1)=>alpha=>alpha2
```

The term is normalized by means of the procedure `nt` standing for *Normalize Term*.

```
> (pp (nt (pt "([x]x)x1")))
x1
> (pp (nt (pt "([g,f,x]g x(f x))g1 f1 x1")))
g1 x1(f1 x1)
```

Constructors are used in a similar way. The constructors `tt` and `ff` of the boolean algebra has in Minlog the names `True` and `False`. By means of the procedure `display-alg` the list of constructors and their types are displayed.

```
> (display-alg "boole")
boole
	True: boole
	False: boole
> (display-alg "ysum")
ysum
	InL: alpha1=>alpha1 ysum alpha2
	InR: alpha2=>alpha1 ysum alpha2
> (display-alg "yprod")
yprod
	PairConstr: alpha1=>alpha2=>alpha1 yprod alpha2
```

If the algebra has type parameters, constructors require types to be fed.

```
> (pp (pt "(InL boole alpha)b"))
(InL boole alpha)b
> (pp (term-to-type (pt "(InL boole alpha)b")))
boole ysum alpha
> (pp (pt "(InR alpha boole)x"))
(InR alpha boole)x
> (pp (term-to-type (pt "(InR alpha boole)x")))
boole ysum alpha
> (pp (pt "(PairConstr alpha boole)x b"))
x pair b
> (pp (term-to-type (pt "(PairConstr alpha boole)x b")))
alpha yprod boole
```

Note that the type parameter looks opposite for the case of `InR`. In the case of the product algebra, you can use `pair` instead of `PairConstr` with parameters.

```
> (pp (pt "x pair b"))
x pair b
```

It does not require the type parameters because it can be inferred. The type inference is not possible for `InL` and `InR` in the same way (Why?). Corresponding to the primitive product type `@@` there is the pairing `x@y`.

From now, we say $\boldsymbol{B}$, $\boldsymbol{N}$ and $\boldsymbol{L}_\alpha$ for the algebras boolean, naturals, and lists of α. Naturals and lists are available by loading libraries as follows.

```
(set! COMMENT-FLAG #f)
(libload "nat.scm")
(libload "list.scm")
(set! COMMENT-FLAG #t)
```

The result of `display-alg` is as follows.

```
> (display-alg "nat" "list")
nat
Zero: nat
Succ: nat=>nat
list
Nil: list alpha
Cons: alpha=>list alpha=>list alpha
```

Instead of writing `"(Cons alpha)x xs"` where `x` and `xs` are of type `alpha` and `list alpha`, respectively, one can write `"x::xs"`. Also instead of writing `(Cons alpha)x((Cons alpha)x1((Cons alpha)x2(Nil alpha)))` one can write `"x::x1::x2:"`.

The first and the last lines are there just to suppress the messages from Minlog during the loading. After loading the `nat.scm` library, `n` and `m` are available as variable names of natural numbers. The case distinction $\mathcal{C}_\iota^\tau$ is in Minlog the `if` construct. Recall that the type of $\mathcal{C}_{\boldsymbol{B}}^\alpha$ is $\boldsymbol{B} \to \alpha \to \alpha \to \alpha$. The term $\mathcal{C}_{\boldsymbol{B}}^\alpha \mathrm{tt} x_0 x_1$ is given as follows.

```
> (pp (pt "[if True x0 x1]"))
[if True x0 x1]
```

As the type of $\mathcal{C}_{\boldsymbol{N}}^\alpha$ is $\boldsymbol{N} \to \alpha \to (\boldsymbol{N} \to \alpha) \to \alpha$, $\mathcal{C}_{\boldsymbol{N}}^\alpha n x_0 \lambda_n x_1$ is given in Minlog as follows.

```
> (pp (pt "[if n x_0 ([n]x_1)]"))
[if n x_0 ([n]x_1)]
```

Recursion and corecursion operators are defined for given algebra ι and type parameter τ. Assuming `iota` is an algebra and `tau` is a type, the recursion and corecursion operators are specified by the strings `"(Rec iota=>tau)"` and `"(CoRec tau=>iota)"`. Recall the following types of recursion and corecursion operators.

$$
\begin{aligned}
\mathcal{R}_{\boldsymbol{N}}^\tau &: \boldsymbol{N} \to \tau \to (\boldsymbol{N} \to \tau \to \tau) \to \tau \\
\mathcal{R}_{\boldsymbol{L}_\sigma}^\tau &: \boldsymbol{L}_\sigma \to \tau \to (\sigma \to \boldsymbol{L}_\sigma \to \tau \to \tau) \to \tau \\
{}^{\mathrm{co}}\mathcal{R}_{\boldsymbol{N}}^\tau &: \tau \to (\tau \to \boldsymbol{U} + (\boldsymbol{N} + \tau)) \to \boldsymbol{N} \\
{}^{\mathrm{co}}\mathcal{R}_{\boldsymbol{L}_\sigma}^\tau &: \tau \to (\tau \to \boldsymbol{U} + \sigma \times (\boldsymbol{L}_\sigma + \tau)) \to \boldsymbol{L}_\sigma
\end{aligned}
$$

The above operators are given in Minlog as follows.

```
> (pp (pt "(Rec nat=>tau)"))
(Rec nat=>tau)
> (pp (term-to-type (pt "(Rec nat=>tau)")))
nat=>tau=>(nat=>tau=>tau)=>tau
> (pp (pt "(Rec list sigma=>tau)"))
(Rec list sigma=>tau)
> (pp (term-to-type (pt "(Rec list sigma=>tau)")))
list sigma=>tau=>(sigma=>list sigma=>tau=>tau)=>tau
> (pp (pt "(CoRec tau=>nat)"))
(CoRec tau=>nat)
> (pp (term-to-type (pt "(CoRec tau=>nat)")))
tau=>(tau=>uysum(nat ysum tau))=>nat
> (pp (pt "(CoRec tau=>list sigma)"))
(CoRec tau=>list sigma)
> (pp (term-to-type (pt "(CoRec tau=>list sigma)")))
tau=>(tau=>uysum(sigma@@(list sigma ysum tau)))=>list sigma
```

Here we find further algebras and types in the case of corecursion. The algebra `uysum alpha` is defined to be $\mu_\xi(\xi, \alpha \to \xi)$.

```
> (display-alg "uysum")
```

```
uysum
        DummyL: uysum alpha1
        InrUysum: alpha1=>uysum alpha1
```

It is used to substitute `ysum` when the left parameter type is empty.

General recursion operator $\mathcal{F}^{\tau}_{\sigma} : (\sigma \to \boldsymbol{N}) \to \sigma \to (\sigma \to (\sigma \to \tau) \to \tau) \to \tau$ is in Minlog given as follows.

```
> (pp (pt "(GRec tau sigma)"))
(GRec tau sigma)
> (pp (term-to-type (pt "(GRec tau sigma)")))
(tau=>nat)=>tau=>(tau=>(tau=>sigma)=>sigma)=>sigma
```

It is possible to introduce constants with computation rules. Constants are defined by means of the procedure `add-program-constant` which takes two arguments the name and the type.

```
> (add-program-constant "Y" (py "(tau=>tau)=>tau"))
ok, program constant Y: (tau=>tau)=>tau
of t-degree 0 and arity 1 added
```

To use such constants with type parameters in `pt`, the parameters should be given in the same way as above.

```
> (pp (pt "(Y tau)"))
(Y tau)
```

By means of the procedure `add-computation-rule` computation rules are given.

```
> (add-computation-rule "(Y tau)(tau=>tau)"
                        "(tau=>tau)((Y tau)(tau=>tau))")
ok, computation rule (Y tau)(tau=>tau) ->
                     (tau=>tau)((Y tau)(tau=>tau)) added
```

There is also the procedure `add-computation-rules` which takes more than one pair of computation rules.

```
> (add-program-constant "Predecessor" (py "nat=>nat"))
ok, program constant Predecessor: nat=>nat
of t-degree 0 and arity 1 added
> (add-computation-rules "Predecessor Zero" "Zero"
                         "Predecessor(Succ n)" "n")
ok, computation rule Predecessor Zero -> Zero added
ok, computation rule Predecessor(Succ n) -> n added
```

2. Examples

We give examples.

2.1. **Even or not.** The function `NEven` is of type $\boldsymbol{N} \to \boldsymbol{B}$, such that $\texttt{NEven}(2n) = \texttt{tt}$ and $\texttt{NEven}(2n+1) = \textsf{ff}$. By using recursion operator and case distinction, it is defined to be $\lambda_n(\mathcal{R}^{\boldsymbol{B}}_{\boldsymbol{N}}\ n\ \texttt{tt}\ \lambda_{n,b}(\mathcal{C}^{\boldsymbol{B}}_{\boldsymbol{B}}\ b\ \textsf{ff}\ \texttt{tt}))$. In Minlog,

```
"[n](Rec nat=>boole)n True ([m,b][if b False True])"
```

2.2. **Half of a natural number.** The function `NHalf` of type $\boldsymbol{N} \to \boldsymbol{N}$, such that $\texttt{NHalf}(2n) = n$ and $\texttt{NHalf}(2n+1) = n$. By using general recursion operator, $\lambda_n(\mathcal{F}^{\boldsymbol{N}}_{\boldsymbol{N}}\ \lambda_n n\ n\ \lambda_{n,f^{\boldsymbol{N}\to\boldsymbol{N}}}(\mathcal{C}^{\boldsymbol{N}}_{\boldsymbol{N}}\ n\ \mathsf{0}\ \lambda_n(\mathcal{C}^{\boldsymbol{N}}_{\boldsymbol{N}}\ n\ \mathsf{0}\ \lambda_n(\mathsf{Succ}(fn))))))$ defines `NHalf`. In Minlog it is given as follows, assuming `f` is a variable name declared to be of type `nat=>nat`.

```
"[n](GRec nat nat)([n]n)
                  n
                  ([n,f][if n Zero
                              ([n][if n Zero
                                        ([n]Succ(f n))])])"
```

2.3. **Filtering a list.** The function `Filter` is of type $\boldsymbol{L}_\tau \to (\tau \to \boldsymbol{B}) \to \boldsymbol{L}_\tau$ such that from the given list elements satisfying the given boolean valued function are chosen to be output.

```
"[xs,h](Rec list tau=>list tau)xs
                               (Nil tau)
                               ([x,xs,xs1][if (h x)
                                              (x::xs1)
                                              xs1])"
```

2.4. **Cototal ideal of $\boldsymbol{N}$.** Cototal ideals can be defined by means of corecursion operator. An infinite number can be defined as an example of cototal ideals. This is given by ${}^{\mathrm{co}}\mathcal{R}^{\boldsymbol{U}}_{\boldsymbol{N}}$ Dummy $\lambda_u(\mathsf{InR}_{\boldsymbol{U},\boldsymbol{N}+\boldsymbol{U}}(\mathsf{InR}_{\boldsymbol{U},\boldsymbol{N}}u))$. In Minlog the following string represents it.

```
"(CoRec unit=>nat)Dummy([u]((InrUysum nat ysum unit)
                            ((InR unit nat)u)))"
```

Since corecursion operator is in general not terminating, normalization of corecursion operator is delayed under the use of `nt`. In order to undelay it, there is the procedure `undelay-delayed-corec` which takes a term and a number (a positive integer in Scheme). As an example, we unfold the above defined term for three times. For readability we abbreviate the definition of the infinite natural number as `Inf`.

```
> (pp (nt (undelay-delayed-corec (pt "Inf") 3)))
Succ(Succ(Succ(Inf)))
```

3. Reference

Minlog Reference Manual and Minlog Tutorial, which are in the official Minlog package.