A Note On Type Checking Linear Functional Languages

Apostolos Syropoulos
366, 28th October Str.
GR-671 00 Xanthi, GREECE
apostolo@platon.ee.duth.gr

Abstract

A linear functional language is based on a linear λ-calculus. Any linear functional program must observe linearity and it has to be type correct. Type correctness can be ensured by means of a type-checking system. A type checker for a linear functional language is essentially a type-checker for an ordinary functional language, modified to properly handle the modality *ofcourse*.

1 Introduction

In ordinary logic it is valid to use an assumption more than once or even to ignore it. This is justified by the silent hypothesis that an assumption which is true, it will remain true. However, this is not the case in real life. Consider, for example, the assumption: *Today is a sunny day.* This assumption may be valid on a summer day in Athens[1], but not on a winter day in Stockholm. The inventor of linear logic designed a logical system that takes under consideration the course of time and its consequences [Gir87]. So, it is valid to use an assumption only once, because it holds only once. At the same time we cannot use an assumption more than once for the same reason. Moreover, we cannot ignore an assumption, simple because it exists and we cannot pretend that it does not! In other words we must use each assumption only once.

Ordinary functional programming languages are based on λ-calculus. Accordingly a linear functional language should be based on a linear λ-calculus (e.g., as it is presented in [Laf88]). Once a linear λ-calculus is available we can use it as a simple, thou impractical, programming notation. A valid program in this notation has to be linear correct and type correct. Since our main focus is on type correctness, the reader interested in linear correctness should consult either [HS92] or [Laf88] for more information on the subject. Type checking a linear functional program means to be able to handle successfully the modality *ofcourse*. Here we present a type-checker that can handle successfully this modality. The type-checker has been developed as part of a project to implement a linear functional language (the language is described in [HS92]).

2 The Modality *ofcourse*

The modality of *ofcourse* (denoted by an exclamation mark) is a *indirect* introduction of *weakening* (1) and *contraction* (2):

$$\frac{\Gamma \vdash B}{\Gamma, !A \vdash B} \tag{1}$$

$$\frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \tag{2}$$

The modality is defined by the equation

$$!A = A \& \mathbf{1} \& (!A \otimes !A)$$

The categorical combinators `read` $: !A \to A$, `kill` $: !A \to \mathbf{1}$, and `dupl` $: !A \to !A \otimes !A$ are used to extract the elements of an *ofcoursed* object, which in turn has been created with the combinator `make` $: A \to !A$.

[1]The smoke does not count as a cloud :-)

3 The Typing Rules

The type rules for a linear functional language have no context, but this is only a direct consequence of the a fact that every correct linear functional program must obey linearity, among other things. A type checker as those described in [Jon87, FH88] can handle most constructs. However, it has to be modified in order to be able to handle situations that involve the combinator make. Every expression has a *storage*.

Definition 3.1 *Let e be an expression with free variables $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$ (t_n is either a generic or a concrete type). Then the set $[t_1, t_2, \ldots, t_n]$ is called the storage of e.*

The storage of an expression is called *valid* either if it is empty or if it consists of type objects of the form $!\alpha$. In case a storage consists only of type objects of the form α_i (i.e., some *generic* type), then we can transform it by replacing each α_i with a $!\alpha_i$. If an expression has a valid storage it is possible to perform the type checking in the usual way. As an example consider the expression $\lambda x.\mathtt{make}x$. The body of the λ abstraction is an expression involving make. The subexpression x has storage $[\beta]$, which is transformed to $[!\beta]$, according to the above rules. The whole λ abstraction has type $!\beta \rightarrow !!\beta$.

4 Implementing the Type Rules

In [HS92] we describe a simple linear functional language called *D*emocritos. An experimental implementation of this language exists, written in the functional language LML, and it is available from:

`http://platon.ee.duth.gr/~apostolo`

as `linear.tar.gz`. The type checker of this implementation is based on the one by Peter Hancock, which is described in chapter nine of [Jon87]. Here we describe the part of it directly related to the modality.

The first step in the design of a type-checker is to decide about the representation of type expressions.

```
  type Tvname == (List Int) /* type variable name is */
and                         /* a list of integers    */
  type TEXP = TVAR Tvname +  /* a type expr is either a type variable */
              TCONS String (List TEXP) /* or a type construct */
```

The next step involves the definition of certain *type forming operators*, such as int for integers etc.

```
and
  imp texp1 texp2 = TCONS "imp" [texp1;texp2]
  /* denotes that a function has type texp1 --> texp2 */
and
  ofcourse texp = TCONS "!" [texp]
```

The modality *ofcourse* has its own type forming operator for obvious reasons. The core of a type checker is the unification procedure. This procedure tries to find the most general type of an expression by solving type equations. In order to solve type equations we need certain *substitution* operators (or functions, if you prefer this term), which transform type expression. Our type checker uses the standard substitution operators (the *identity* substitution and the δ substitution), plus a new one the η substitution:

```
and
  id_subst t = TVAR t
and
  delta t texp t' = if t=t' then texp
                    else TVAR t'
and
  eta tvns tvn = if mem tvn tvns
                 then ofcourse (TVAR tvn)
                 else TVAR tvn
```

The η substitution is used in the transformation of a generic type to an *ofcoursed* one, only if this generic type is a member of a specific storage; otherwise it behaves exactly like the identity substitution. We now turn our attention to the type checker itself. Here we consider only the make case.

```
1   ||tc gamma ns   (EMake e) =
2     let rec
3         fv0 = freeVarsOf [] e
4         and
5         fv = map mkname fv0
6         and
7         tvns = gettvnEnv gamma fv
8         and
9         gamma' = updateEnv gamma fv
10        and
11        tfv = map (gettype gamma') fv
12        and
13        areofcoursed = isofcoursed tfv
14        and
15        ns0 = deplete ns
16        and
17        tvn = next_name ns0
18        and
19        ns1 = deplete ns0
20        and
21        tvn1 = next_name ns1
22    in if null fv
23       then tcmake tvn tvn1 (id_subst) (tc gamma ns1 e)
24        else if areofcoursed
25             then  tcmake tvn tvn1 (eta tvns)(tc gamma' ns1 e)
26             else Failure("Improper Storage ", (EMake e))
```

The type checker is a function that gets three arguments

1. gammma is a type environment that associates type schemes with each of the free variables of the expression to be type-checked.
2. ns is a supply of type variable names, and
3. the last argument is the expression to be type-checked

and it returns a tuple consisting of

1. a substitution ϕ defined on the unknown type variables of gamma
2. a type t derived for the expression that is being type-checked in a the possible altered type environment.

Line 1, in the above code fragment, identifies the case we are dealing with. In line 3 we get the free variables (if any) of the (sub-)expression e. In lines 11–13 we get the storage of the expression and the result of the *validity* check of this storage. The validity check is performed by the following function:

```
  isofcoursed [] = true
||isofcoursed ((TVAR _).tt) = isofcoursed tt
||isofcoursed ((TCONS constr _).tt) = (constr = "!") & isofcoursed tt
```

In lines 14–21 we create some new type names which will be used later. Next, in line 22, we check to see if the storage is empty or not. If it is empty we perform ordinary type checking, otherwise we must check if it is valid. In case it is valid we perform type checking in the transformed environment. In case it is not valid, the function stops and reports a `Improper Storage` error. The function `tcmake` is the one which calls the unification procedure, which deduces the most general type of the expression, if that is possible.

```
and
  tcmake tvn tvn1 zeta (Ok(phi, t, e)) =
              tcmake´ tvn zeta (unify phi
              (imp  (TVAR tvn1) (ofcourse (TVAR tvn1)),
              imp t (TVAR tvn))) e
||tcmake tvn tvn1 zeta (Failure s) = Failure s
and
  tcmake´ tvn zeta (Failure s) e = Failure(s, (EMake e))
||tcmake´ tvn zeta (Ok(phi)) e = Ok(scomp phi zeta, phi tvn, (EMake e))
```

The function `tcmake´` is actually used to get the result of the unification. If the unification procedure has failed in its job, then we must report that, otherwise we continue by returning the deduced type.

References

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley Publishing Company, Inc., Wokinghan, England, 1988.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[HS92] Pétur Hilmarsson and Apostolos Syropoulos. Democritos: A linear functional language. Electronic document available at http://platon.ee.duth.gr/~apostolo/demo.ps.gz, May 1992.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.