# Incorporating Static Type-Checking into Functions Defined on Context-Free Languages *

Haiming Chen and Yunmei Dong

Computer Science Laboratory, Institute of Software, Chinese Academy of Sciences
Beijing 100080, P R China
{chm,dym}@ios.ac.cn

**Abstract.** This paper presents a type-checking method for the functional language LFC. A distinct feature of LFC is that it uses context-free (CF) languages as data types to represent compound data structures. This makes LFC a dynamically typed language, since the inclusion relation of two CF languages is undecidable. To improve runtime efficiency, we incorporated static type-checking into LFC, utilizing a decidable subset of the inclusion relation, i.e. the sentential form relation. The type-checking also converts the expressions with implicit structures to structured representation.

## 1 Introduction

LFC [7, 2] is a functional language that has been used for software specification [7, 4], rapid prototyping [4], programming language processing [3], and so on. The language is based on the theory of a new kind of recursive functions, which are defined on context-free (CF) languages [5, 6]. A distinct feature of LFC is that it uses CF languages as data types (called CF type) to represent compound data structures. The merits and disadvantages of using CF languages as data types has been discussed in detail in [2]. Below is a trivial example for the demonstration of LFC, which defines the evaluation of arithmetic expressions.

*Example 1.* A CF grammar of arithmetic expressions:

```
<E>-><T>|<E>+<T>|<E>-<T> <T>-><F>|<T>*<F>|<T>/<F> <F>-><Int>|(<E>)
```

Functions:

```
dec eval: E -> Int;
var e : E;  t : T;
def eval(t)= evalterm(t);
    eval(e[]"+"[]t)= add(eval(e),evalterm(t));
    eval(e[]"-"[]t)= sub(eval(e),evalterm(t));
dec evalterm : T -> Int;
var f : F;  t : T;
```

* Research supported by NSFC under Grant No. 60103008.

```
def evalterm(f)= evalfact(f);
    evalterm(t[]"*"[]f)= mul(evalterm(t), evalfact(f));
    evalterm(t[]"/"[]f)= div(evalterm(t), evalfact(f));
dec evalfact : F -> Int;
var i : Int;  e : E;
def evalfact(i)=i;
    evalfact("("[]e[]")")=eval(e);
```

In the above, three types (`E`, `T`, `F`) are defined in the CF grammar. `Int` is a built-in type representing integers. Functions are defined by structural induction on CF types. The function `eval`, for example, is defined by induction on type `E`. Three mutually recursive functions (`eval`, `evalterm`, `evalfact`) are defined. Built-in functions defined on `Int` (i.e. `add`, `sub`, `mul`, `div`) are used.

Since the inclusion relation of two CF languages is undecidable, LFC in its nature is a dynamically typed language (see Section 3), which means that type-checking is performed at run time (dynamic type-checking). This is the same as the way of Scheme implementation, which is expensive in time.

Type checking [14] is an important issue of programming languages. Dynamically typed languages offer more programming flexibility at the cost of efficiency. Statically typed languages, on the other hand, which do type-checking at compile time (static type-checking), will improve the quality and time efficiency of programs at the cost of loss of flexibility. A major goal of type system research is to combine the flexibility with the security.

In order to improve the time efficiency of LFC we incorporated static type-checking into LFC in the desire to maximally reduce the need for dynamic-type checking. Although the inclusion relation of CF languages is not decidable, a special subset of the relation is decidable, i.e. the sentential form relation, which can be statically checked. Moreover, most of the expressions in actual LFC programs appear to satisfy this relation according to the statistic data of our experiments. So, despite the static type-checking is not complete, it undertakes most (or even all) of the type-checking task in practice. Consequently the run-time efficiency is effectively improved by removing most (or all) dynamic type-checking. Expressions of CF types have implicit structures (see Section 4), which are disadvantageous for efficient implementation. They will be converted into explicitly structured representation in the type-checking. Structure reconstruction technique is presented. Part of the very earlier and preliminary results of the work of this paper has been reported in [1], this paper contains a more complete and correct presentation of this work.

CF languages have been used in various areas of software. One programming language that uses CF language as data type is the SNOBOL string manipulation language, which is known as a powerful and flexible programming language for naive programmers. Recently, along with the popularity of XML, type systems of XML become an important and active research topic, which is closely related to type systems of CF types, since a DTD of XML documents is precisely a CF grammar for the documents. Though the technique introduced in this paper is presented in the context of LFC, it is general for type systems of CF types.

The rest of this paper is organized as follows. Section 2 introduces types and expressions. Section 3 discusses issues of type systems with CF types and presents the type-checking method. Section 4 describes structure reconstruction of expressions. Section 5 sketches implementation. Section 6 concludes.

## 2 Types and Expressions

Let $V_N$ denote the set of nonterminal symbols, $V_T$ the set of terminal symbols ($V_N \cap V_T = \emptyset$), $P$ the set of productions $P = \{X \to \alpha \mid X \in V_N, \alpha \in (V_N \cup V_T)^*\}$. For any production $Y \to \alpha$, $\alpha$ is called a *term* of $Y$. Denote $Term(Y) = \{\alpha \mid Y \to \alpha \in P\}$.

We can define a CF grammar $G_X = (V_N, V_T, X, P)$ for each nonterminal symbol $X \in V_N$, where nonterminal symbol $X$ is called the start symbol. $G_X$ produces a CF language written $L(G_X)$. Terms of $X$ are also called terms of $L(G_X)$. We also write $L(X)$ instead of $L(G_X)$ when no ambiguity arises. A BNF-like notation is used to represent CF grammar. Denote $Term(L(X))$ the same as $Term(X)$.

A CF type is just a CF language, i. e., a set that contains all of the valid sentences of the language. A CF grammar $G$ uniquely defines a CF type $L(G)$. For simplicity, we consider the name of a CF type the same as the start symbol of its grammar. Hence in Example 1 type `E` actually means type $L(\texttt{E})$, which is the language of all arithmetic expressions.

The basic operations for constructing a CF type are union ($\cup$) and concatenation (., usually omitted).

Concatenation: A production is formed by a concatenation of nonterminal and terminal symbols. We call a string of nonterminal symbols and terminal symbols a *grammatical string*. A known simple property of a such string is:

*Property 1.* If $\alpha$ is a grammatical string, $\alpha$ defines a CF type denoted by $L(\alpha)$.

Union: For a group of productions $X \to \alpha_1 \mid \ldots \mid \alpha_n$, where $\alpha_i \in (V_N \cup V_T)^*$, $i = 1, \ldots n$, we have $L(X) = L(\alpha_1) \cup \ldots \cup L(\alpha_n)$. $L(X)$ is the union of $L(\alpha_i)$.

The definition of a function consists of declarations and equations, as we have seen in Example 1. The basic way to define a function is structural induction on grammar(s) [5, 6]. The right-hand side of each equation is an expression. Syntax of a core of expressions is represented as follows.

| | |
|---|---|
| $c$ | Constant |
| $v$ | Variable |
| $f$ | Function name |
| $e ::= c \mid v \mid e_1 e_2 \mid f(e_1, \ldots, e_n)$ | Expression |

Note $e_1 e_2$ is the concatenation of expressions, not the function application.

## 3 Type Checking Method

There could be an inclusion or intersection relation between two CF languages. A type system containing CF types hence excludes the popular Hindley-Milner systems [9, 11], implemented in a number of programming languages like ML[12], which are based around equality constraints, and is close to a subtyping system [13]. In such a system the subtype relation between two CF types is simply inclusion between the two CF languages that they denote. In a subtyping system, a subtyping relation satisfies that if $\omega$ is a subtype of $\omega'$, then there exists an implicit conversion or coercion from values of type $\omega$ to values of type $\omega'$. Thus expressions of type $\omega$ can appear in any context allowing expressions of type $\omega'$.

However, neither the inclusion nor the intersection relation of two CF languages is decidable. This leads to the fact that the subtype relation of CF types is not decidable. As a result, a type system with CF types in general must involve dynamic type-checking. To incorporate static type-checking into such a type system, there are some issues to be solved. (1) How to assign a type to an expression from type declarations (i. e., typing of expressions). (2) How to check (a subclass) subtype relation (which is in general undecidable).

### 3.1 Typing of Expressions

To deal with CF types, we first annotate every expression of CF type with a "virtual type" — a grammatical string — as the "minimal type" of the expression, in next subsection we will see its use. According to Property 1 a grammatical string defines a CF type, but this CF type may not reflect any real CF types used, so it is a virtual type. We call such a grammatical string gives the *least type* of the expression.

Let $Exp$ denote the set of expressions, $Decl$ denote all sets of variable and function declarations, $Gstr$ denote the set of grammar strings. $decl \in Decl$, $\sigma, \sigma_i, \tau, \tau_i \in Gstr$. The following are the inference rules for the least type.

$$\frac{}{\vdash c : c} \quad \text{[CON]}$$

$$\frac{}{decl \vdash x : \sigma \;\; x : \sigma \in decl} \quad \text{[VAR]}$$

$$\frac{decl \vdash e_1 : \sigma_1, \quad decl \vdash e_2 : \sigma_2}{decl \vdash e_1 e_2 : \sigma_1 \sigma_2} \quad \text{[CONCAT]}$$

$$\frac{decl \vdash f : \tau_1 \times \ldots \times \tau_n \longrightarrow \tau, \quad decl \vdash e_i : \tau_i', \quad \tau_i' \subseteq \tau_i}{decl \vdash f(e_1, \ldots, e_n) : \tau} \quad \text{[APP]}$$

The least type of an expression $e$ is denoted by $L(e,\ decl)$, and is abbreviated as $L(e)$. Sometimes we need to distinguish between the grammatical string itself and the least type it gives (i.e. $L(e)$). So we also denote the grammatical string inferred from $e$ by $\gamma(e, decl)$.

It is easy to know that for any CF type $l$, $e$ has type $l$ if and only if $L(e) \subseteq l$. This is why $L(e)$ is called the least type of $e$.

*Example 2.* For the definition of `eval` in Example 1, we have *decl*={`e`: `E`, `t`: `T`, `eval`: `E` → `Int`}, then
$\gamma$(`e[]"+"[]t`, *decl*)=`E+T`, $L$(`e[]"+"[]t`)=$L$(`E+T`)=$L$(`E`){`+`}L(`T`)

For any given CF type $l$, we generally cannot determine if $L(e) \subseteq l$ holds (unless in some special situations), consequently in general we actually cannot assign any useful type to $e$. So actually we cannot infer any real type for an expression. The type checking is, given all typing information, to check if an expression is compatible with the declared type.

Since the inclusion relation is undecidable, we generally cannot assign least types to function applications by the APP rule. As we will see in the next subsection, however, static type-checking is considered for a decidable subset of inclusion relation, thus the consequent replacement of general inclusion by the decidable subset in the APP rule also enables the least type assignment.

### 3.2 The Method

There would be two ways to solve the type problem of CF types. One way is to impose constraints on expression so that we can statically check the compatibility of an expression with the declared type. This will lose some flexibility of the language.

Another way is to use a dynamic type system, but perform static type-checking when possible. This would decrease the run-time efficiency if compared with the previous way, but would be more efficient than the totally dynamic type-checking. We chose this way to retain a maximal flexibility. Although the inclusion relation of CF languages is not decidable, a subset of the relation is decidable, i.e. the sentential form relation, which can be statically checked. Given a CF grammar $G_X = (V_T, V_N, X, P),\ \alpha \in (V_T \cup V_N)^*$, if $\alpha$ is derivable from $X$, i. e., $X \stackrel{*}{\Rightarrow} \alpha$, then $\alpha$ is called a *sentential form* of $L(G_X)$, denoted by $L(G_X) \mapsto \alpha$.

Clearly if $\alpha$ is a sentential form of $L(G_X)$, then $L(\alpha) \subseteq L(G_X)$, thus $L(\alpha)$ is a subtype of $L(G_X)$. It is known that the sentential form recognition problem is decidable.

*Example 3.* For the CF grammar of arithmetic expressions in Example 1, `T`, `E+T`, `T+T`, `T+T*F` are all sentential forms of `E` (or more accurately, $L$(`E`)), and hence are subtypes of `E`.

Bellow we give a brief description of the type-checking method.

The following two definitions define the concepts of subtyping and type coercion.

**Definition 1.** *$l_1$ is called a subtype of $l_2$ if $l_1 \subseteq l_2$.*

**Definition 2.** *We say $e$ has type $l$ if $L(e)$ is a subtype of $l$.*

The following property follows immediately.

*Property 2.* Whether type of $e$ can be coerced to $l$ is not decidable.

The notion of a checkable type is defined as follows.

**Definition 3.** *$l$ is called a checkable type of expression $e$ if $l \mapsto \gamma(e, decl)$.*

*Example 4.* Following Example 1 and Example 3, we know `t`, `e[]"+"[]t`, `t[]"+"[]t`, `t[]"+"[]t[]"*"[]f` all have a checkable type `E`.

Checkable types have the following properties.

*Property 3.* Checkable type is not unique.

*Proof.* This is obvious for CF languages.

*Property 4.* The type of an expression can be coerced to any of its checkable types.

*Proof.* This can be inferred from the fact that if $l \mapsto \gamma(e, decl)$ then $L(e) \subseteq l$ hence $L(e)$ is a subtype of $l$.

*Property 5.* Whether an expression $e$ has a checkable type $l$ is decidable.

*Proof.* This is derived from the decidability of sentential form relation.

Therefore, type coercion is decidable for checkable types.

*Property 6.* There exists $l$, such that $l$ is not a checkable type of $e$, but the type of $e$ can be coerced to $l$.

Consider a simple example. Given CF grammar $\langle l \rangle \rightarrow \langle n \rangle \mid \langle l \rangle \langle n \rangle$, $decl = \{x \in n, y \in l\}$, and expression $e = xy$, then $\gamma(e,\ decl) = nl$ is not a sentential form of $l$, so $l$ is not a checkable type of $e$. On the other hand $L(e)$ equals $l$, thus $L(e)$ is subtype of $l$, then the type of $e$ can be coerced to $l$.

This property shows there are other coercible types than checkable types.

From the above we can obtain the type-checking method. First we statically check if the declared type $l$ of expression $e$ is a checkable type of $e$, using sentential form parsing technique (see Section 5 for an effective optimization by using term checking). If so, then the type is correct. Otherwise we check if $e$ can be dynamically coerced to type $l$, i.e. determine if the value of $e$ is a sentence of $l$ at run-time. This can be done by sentence parsing technique, after the expression is bound.

A *run-time environment* is a set of mappings from variable names to values. If $E$ is a run-time environment, $e$ is an expression, we denote $E \mid e$ the value of $e$ in $E$, whose definition is a trivial induction on the structure of expressions.

**Definition 4.** *We say the type of an expression $e$ can be coerced to $l$ in a run-time environment $E$ if $E \mid e \in l$.*

*Property 7.* Whether type of expression $e$ can be coerced to $l$ in a run-time environment $E$ is decidable.

*Proof.* This is exactly the sentence recognition problem of CF languages, which is decidable.

*Property 8.* If the type of an expression $e$ can be coerced to $l$ in a run-time environment $E$, then $L(e) \cap l \neq \emptyset$.

*Proof.* According to Definition 4, if the type of expression $e$ can be coerced to $l$ in a run-time environment $E$, then $E \mid e \in l$. But $E \mid e \in L(e)$, thus $L(e) \cap l \neq \emptyset$.

**Corollary 1.** *If $L(e) \cap l = \emptyset$, then the type of expression $e$ cannot be coerced to $l$ in any run-time environment.*

*Property 9.* If $L(e) \subseteq l$, then the type of expression $e$ can be coerced to $l$ in any run-time environment.

*Proof.* $E \mid e \in L(e)$ for any run-time environment $E$, on the other hand $L(e) \subseteq l$, therefore $E \mid e \in l$.

From the above it can be seen that, if $L(e)$ is a subtype of $l$, then the type of $e$ can be dynamically converted to $l$; if $L(e)$ does not intersect with $l$, then the type of $e$ cannot be dynamically converted to $l$. But the dynamic conversion permits the cases of $L(e) \not\subseteq l$ and $L(e) \cap l \neq \emptyset$. Consequently, if an expression needs dynamic type-checking, the correctness of the type of this expression may depend on run-time environment, so type correctness can not be guaranteed. To remedy this defect, a type-warning message will be given to user when an expression cannot be statically type checked, so that the user has a chance to decide if the type is correct.

If an expression $e$ can be statically checked to have a type $l$, from the above we have $l \mapsto \gamma(e, decl)$ and hence $L(e) \subseteq l$. Then by Property 9 type soundness can be ensured, since $E|e$ must have the type $l$ for any run-time environment $E$.

## 4 Structure Reconstruction

Recall the definition of expressions in Section 2, we can see that an expression of CF type is naturally formed by concatenation. On the other hand, the expression cannot reflect the phrase structure it should have for its CF type, in other word, the expression has an implicit structure.

This kind of implicit structure representation is disadvantageous for the implementation. In order to achieve an efficient implementation of LFC, an intermediate representation is necessary that can reflect the phrase structures of expressions. We have designed such an intermediate representation called Vtree, the core is a concise and efficient representation of parse trees.

Suppose in a grammar each production has a unique *label*. A production $X \to \alpha$ with label $p$ is denoted by $p : X \to \alpha$.

**Definition 5.** *A Vtree is defined inductively as follows.*

*1. A variable $v$ of type $X$ is a Vtree of type $X$.*

*2. For a production $p : X \to a$, $a \in V_T^*$, $p$ is a Vtree of type $X$.*

*3. For a function $f : X_1 \times \ldots \times X_n \longrightarrow X$, if $t_i$ is a Vtree of type $X_i$, $i = 1, \ldots, n$, then $f(t_1, \ldots, t_n)$ is a Vtree of type $X$.*

*4. For a production $p : X \to a_1 X_1 \ldots a_n X_n a_{n+1}$ ($a_i \in V_T^*$, $X_i \in V_N$, $n \geq 1$), if $t_i$ is a Vtree of type $X_i$, $i = 1, \ldots, n$, then $p(t_1, \ldots, t_n)$ is a Vtree of type $X$.*

*5. There is not any other Vtree besides the above.*

Vtree can naturally reflect phrase structures, so it has explicit structure.

*Example 5.* Rewrite the CF grammar for expressions in Example 1 as follows.

```
p1:<E>-><T>      p2:<E>-><E>+<T>  p3:<E>-><E>-<T>  p4:<T>-><F>
p5:<T>-><T>*<F> p6:<T>-><T>/<F>  p7:<F>-><Int>    p8:<F>->(<E>)
```

Given the types of functions and variables in Example 1, the following are some valid Vtrees with their types (`Vtree:type` means the Vtree `vtree` has type `type`).

```
e: E    eval(p2(e,t)): Int
p2(eval(p2(e,t),p5(evalterm(p7(10)),p4(p7(20)))): E
```

## 5 Implementation and Experiment

A type checking algorithm for LFC has been designed and adopted in the implementation of LFC, which performs type checking and converts functions into an internal representation based on Vtree. For the limited size of the paper, it is not described here. In the following we indicate some important implementation issues.

When checking if an expression $e$ has a checkable type $l$, in order to be more efficient, before using sentential form parser we first check if $e$ corresponds to a term of $l$, if not then call the parser. According to our statistic data (see below), over 80% of the expressions that are sentential forms correspond to terms. Since term checking is straightforward, while sentential form checking is more expensive, type-checking efficiency can be considerably improved by this optimization.

To build Vtree, if an expression corresponds to a term of its expected type, the construction of a Vtree from it is straightforward and very efficient. Otherwise a parser for CF languages is needed. The CF types in LFC do not have any constraint, so a general CF language parser is required. The sentence parser is based on Earley's algorithm [8] which generates the rightmost derivation of a sentence. For a sentence of length $n$, the worst parse time is $O(n^3)$. The sentential form parser is derived from Earley's algorithm. In practice, since most expressions correspond to terms of their declared types, the need for parsing is not often.

Construction of Vtree from the derivation of a sentential form is an extension of parse tree construction from the derivation of a sentence. The key is to represent the rightmost derivation of sentential form by introducing the notion

of an empty production *nil* to keep empty positions in a derivation. An empty production may have any nonterminal symbol on the left-hand side, and produces nothing. For example, suppose we have a grammar $p1 : \langle A\rangle \rightarrow a\langle B\rangle b\langle B\rangle$ $p2 : \langle B\rangle \rightarrow c$, and sentential forms (1) $acb\langle B\rangle$, (2) $a\langle B\rangle bc$. The rightmost derivations of the two sentential forms are: (1) $[p1, nil, p2]$, (2) $[p1, p2]$. Here we use a list notation to represent the derivations.

Experiments with LFC have been made on many trivial and nontrivial examples. Table 1 gives the statistic data of some of the examples. In the table, the column titled "Sentential form (term)" denotes expressions that correspond to terms of their declared types and hence satisfy sentential forms relation. Next column denotes the rest of expressions that are sentential forms of their declared types. It shows that most of the expressions (96% in the experiment) satisfy the sentential form relation, among these expressions most (over 80% in the experiment) correspond to terms of their declared types. Most of the type-checking work is then done statically, runtime efficiency is therefore improved. From our experiments the execution times are reduced by about two to twenty times, if compared with the implementation without static type-checking. How much the efficiency is gained mainly depends on how many expressions are not sentential forms of the declared types. Since most of the static type-checking can be reduced to the term checking, the compilation time for type-checking is also reduced.

**Table 1.** Statistic data of sentential forms in expressions

| | Funcs | Exprs | Sentential form (term) | Sentential form (not term) | Others |
|---|---|---|---|---|---|
| Number | 135 | 2720 | 2207 | 399 | 114 |
| Ratio | | | 0.81 | 0.15 | 0.04 |

Data in Table 1 also suggest that if expressions are constrained to satisfy sentential form relation, expressiveness will not be lost much, therefore this constraint can be used to obtain static type system of CF types in practice. Actually we also have a statically typed version of LFC.

## 6 Conclusion

We present a type-checking method for LFC which supports CF languages as data types. In this method both static and dynamic type-checking are involved. Structures of expressions are also reconstructed to represent phrase structures. The method has been used in the interpreter and compiler of LFC. Experiments show that the runtime efficiency can be effectively improved. We also have a statically typed version of LFC, in which expressions are restricted to sentential forms.

This paper presents a general framework and related implementation techniques for dealing with CF types. When applied to a particular domain, for example the XML processing, it is possible to further tune and improve the

method according to properties of the domain. In many applications CF types can be restricted to some subclasses of CF languages, so more efficient techniques for the subclasses can then be used instead of the general techniques.

A recent research on XML is type systems of XML. There have been several work on it from different approaches. For example, XDuce [10] uses regular expression types, which are based on tree automata theory, and supports subtyping. The work presented in this paper obviously can be applied to XML. Compared with XDuce, for type-checking efficiency the time complexity is exponential in worst case for XDuce [10], and is polynomial ($O(n^3)$) in worst case for LFC. And as is discussed in the paper, the time for type-checking of LFC can be reduced by term checking. Moreover, it is possible to make further improvement to the type system particularly for processing XML.

## References


1. H. Chen and Y. Dong. Practical type checking of functions defined on context-free languages. Proc. the Sixth Int. Conf. for Young Computer Scientists (ICYCS'2001), Hangzhuo, China, Beijing: International Academic Publishers, World Publishing Corporation. Oct. 2001, 1261-1263.
2. H. Chen and Y. Dong, A formal specification language facilitating specification acquisition, Chinese Journal of Computers, 25(5), 2002, 459–466. (in Chinese)
3. H. Chen and Y. Dong. Yet another meta-language for programming language processing. ACM SIGPLAN Notices, 37(6), 2002, 28–37.
4. H. Chen and Y. Dong. Modeling and prototyping of software systems. Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002), November 2002, Cambridge, USA, 617–622.
5. Y. Dong, Recursive functions of context free languages (I) — The definitions of CFPRF and CFRF, Science in China, Series F, 45(1), 2002, 25–39.
6. Y. Dong, Recursive functions of context free languages (II) — Validity of CFPRF and CFRF definitions, Science in China, Series F, 45(2), 2002, 1–21.
7. Y. Dong, K. Li, H. Chen, et al., Design and implementation of the formal specification acquisition system SAQ, Proc. Conf. on Software: Theory and Practice, IFIP 16th World Computer Congress 2000, Beijing, China, 2000, 201–211.
8. J. Earley. An efficient context-free parsing algorithm. Comm. ACM. 13, 1970, 94–102.
9. R. Hindley. The principal type-scheme of an object in combinatory logic. Trans. Amer. Math. Soc. 146, 1969, 29–60.
10. H. Hosoya, B. Pierce. XDuce: a statically typed XML processing language, ACM Transactions on Internet Technology, 3(2), 2003, 117–148.
11. R. Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17(3), 1978, 348–375.
12. R. Milner, M. Tofte. and R. Harper, The definition of Standard ML. The MIT Press, 1990.
13. J. C. Mitchell. Type inference with simple subtype. Journal of Functional Programming, 1(3), 1991, 245–285.
14. B.C. Pierce. Types and Programming Languages. MIT Press, Cambridge, Mass. 2002.