An Abstract Model for Programming Languages

Hongjun Zheng *Naixiao Zhang*
Department of Computer Science & Technology
Peking University, Beijing 100871, P.R. China
Email: znx@sxx0.math.pku.edu.cn
naixiao@pku.edu.cn

Abstract This paper presents an abstract model for programming languages with algebraic approach, and proposes a concept of abstraction for programming languages. Then the paper describes the semantics of inheritance, extension and shielding of programming languages and the semantics of language family to reveal connections between programming languages in language family. [+]

Keywords programming language, language abstraction, abstract model, language family

1. Introduction

With the development of computer science, programming languages are constantly emerging. The differences between languages are the results of different selections of the criteria for language design and different relative priorities of the criteria, some of them mutually conflicting. Most of these languages differ markedly one from another in style, substance and appearance. The unnecessary diversity and differences of languages bring about many difficulties not only for using languages, but also for studying on connections between languages. Programming language designers are attempting to alleviate the resulting problems by combining the merits of several paradigms into a single more comprehensive language; but the signs of success in such a combibation are difficult to recognise. A solution to these problems may emerge from a wider understanding and agreement about the nature of programming and the choices available to the programming language designer and user[6].

This paper presents a concept of language abstraction and an abstract model for programming languages through studying into the essence of programming languages based on the research of Polya language[++] and model-oriented transformational software development methodology[13]. Then the paper describes the semantics of language inheritance, extension, and shielding and also the semantics of language family[13]. We hope that the abstract model of programming languages the paper proposed may contribute to building up a theoretical foundation for studying on the connections between languages. Furthermore, the abstract model mentioned above may be a theoretical model, which can be implemented for language design and development. The

[+] The research reported herein has been funded by the National Natural Science Fund.

[++] Polya language was proposed by Prof. David Gries of Cornell University.

implementation of this theoretical model can make it possible to separate language using from language defining and so do language development from program development. As a result of it, it is possible to improve the succinctness of language itself by the separation, and decrease the complexity of software development. The framework discussed in this paper may allow us to envision feasible solution to the problems raised by the limitations of current software technology.

2. Language Abstraction

Describing programming languages with algebraic approach can enable programming languages clear and arrangible[2], and also it is a beneficial attempt to study connections between languages (providing an abstract model for languages). A *signature* $\Sigma=(S,F)$[4], which consists of a set of sorts S (basic grammar elements) and a set of operations F (combinatorial relations among grammar elements), can be used to describe the *abstract grammar* of a language[7]. A set of semantic equations assign the semantics to Σ, i.e. to the language which the signature Σ corresponds to. Different semantic equations assign different semantics to Σ. The signature and its semantic equations form an *abstract model* of a language. The abstract model of a language is independent of the concrete notation of the language. Many authors have the similar idea that describe abstract grammar of a language with signature[1][9][11].

The general principle of describing the abstract grammar for a language with a signature $\Sigma=(S,F)$ is that each sort in S corresponds to a basic grammar element of the language— nonterminal in its BNF; and an operation in F corresponds to the constructive relation among basic grammar elements of the language. Namely, a BNF production P: $s_0::=t_1s_1t_2s_2\ \ldots\ t_ns_nt_{n+1}$, where t_i is either a terminal or an empty ε and s_i is a nonterminal, corresponds to the operation P in the signature that is as P: $s_1\times s_2\times\ \ldots\ \times s_n\rightarrow s_0$ $(s_i\in S,\ 0\le i\le n)$. For example, we consider a tiny Pascal-like language as follows, which we call it a kernel language K:

```
prog ::=  BEGIN stmt END
stmt ::=  stmt ‘;’ stmt
       |  IF expr THEN stmt ELSE stmt FI
       |  WHILE expr DO stmt OD
       |  vid:=expr  |  skip
expr ::=  int  |  vid
       |  expr ‘+’ expr  |  expr ‘*’ expr  |  NOT expr
```

Let’s see about part of semantics of K informally. In K, there is only integer type expression. Boolean type can be implemented by integer type like the way in C language. Denote by $eval$: $EXPR\rightarrow INT$ the evaluation function for expressions in K. Then we can evaluate integer expression as boolean value as follows:

$$eval(\underline{\textbf{NOT}}\ \text{expr}) = \underline{\textbf{LET}}\ v=eval(\text{expr})\ \underline{\textbf{IN}}\ v=0\rightarrow 1, 0$$

and the semantics of **IF** expr **THEN** stmt$_1$ **ELSE** stmt$_2$ **FI** can be specified informally as:

$$eval(\text{expr})=0\rightarrow \text{stmt}_2\ , \text{stmt}_1\ .$$

We can describe the abstract grammar of K by a signature $\Sigma_K=(S_K,F_K)$, where

$$
\begin{array}{llll}
S_K = \{ \text{int, id, stmt, expr, prog} \} & ; & & \\
F_K = \{ \text{program} : \text{stmt}\rightarrow\text{prog} & ; & \text{comp} : \text{stmt}\times\text{stmt}\rightarrow\text{stmt} & ; \\
\quad \text{if} : \text{expr}\times\text{stmt}\times\text{stmt}\rightarrow\text{stmt} & ; & \text{while} : \text{expr}\times\text{stmt}\rightarrow\text{stmt} & ; \\
\quad \text{assign} : \text{id}\times\text{expr}\rightarrow\text{stmt} & ; & \text{skip} : \rightarrow\text{stmt} & ; \\
\quad \text{int\_expr} : \text{int}\rightarrow\text{expr} & ; & \text{id\_expr} : \text{id}\rightarrow\text{expr} & ; \\
\quad \text{add} : \text{expr}\times\text{expr}\rightarrow\text{expr} & ; & \text{mul} : \text{expr}\times\text{expr}\rightarrow\text{expr} & ; \\
\quad \text{not} : \text{expr}\rightarrow\text{expr} & ; & & \}
\end{array}
$$

The terminals may be "*forgotten*" here, as they are not essential and can be determined uniquely by the operation's name P[2]. We can see, from the "forgotten" principle, that not only the set of sorts in Σ is abstract, but also the operation representing is abstract. Signature Σ_L, which is constructed by the principle above, is called the abstract grammar of a language L. The process of constructing the signature Σ_L is that of the grammar abstraction. This process is much like that of [1][10]. Furthermore, the sort in Σ_L reflects the analyticity of the abstract grammar, and the operation in Σ_L reflects the syntheticality of the abstract grammar.

To assign semantics to signature Σ_L, it is necessary to introduce a set of Σ_L equations, which are used to describe the semantics of Σ_L. Σ_L and the set of semantic equations, say E_L, form a *language* (Σ_L, E_L). Different E_L assigns different semantics to Σ_L.

We adopt the means of *transformational semantics*[2][7][12] to assign semantics to Σ_L, and regard Σ_L equations as transformation rules. Transformational semantics is to assume that the semantics of a language L_1 is known, and constucts in language L_1 are used to describe every construct in another language L by the means of transformation rule, then we can reach the semantics of the language L[7]. We call language L *abstract language*, and language L_1 *concrete language* or implementing language[12]. If the process above is continued, we can gain a sequence of languages like $L_n, L_{n-1}, \ldots, L_1, L$. Abstract language and concrete language are relative to the different place in the sequence. Abstract language can be concrete language, and vice versa. For abstract language L in the sequence, language $L_n, L_{n-1}, \ldots, L_1$ are all concrete languages of L, and for abstract language L_1, languages $L_n, L_{n-1}, \ldots, L_2$ are all concrete languages of L_1, and so forth. The most concrete language L_n in the sequence is called kernel language, and other languages in the sequence are intermedia languages. If L_n is a machine language, the sequence above represents a process of language compiling, i.e. the process of language translation from high-level language to machine language[10][11].

Definition 1 A Σ_L **equation** is a pair $<lhs, rhs>$, usually is of the form $lhs = rhs$, where lhs is a term of ground term algebra $T(\Sigma_L)$ [7][4], and rhs is a term of ground term algebra $T(\Sigma_{L''})$. L is an abstract language, and L'' is the concrete language relative to L.

For instance, we assume that the semantics of K is fixed by E_K, then we can specify a language $M = (\Sigma_M, E_M)$ with $K = (\Sigma_K, E_K)$ as follows:

$$
\begin{array}{ll}
S_M = S_K; & \\
F_M = F_K + \{ & \text{and\_expr} : \text{expr}\times\text{expr}\rightarrow\text{expr} ; \\
 & \text{or\_expr} : \text{expr}\times\text{expr}\rightarrow\text{expr} ; \\
 & \text{repeat} : \text{stmt}\times\text{expr}\rightarrow\text{stmt} \}
\end{array}
$$

$E_M = E_K + \{$ and_expr(expr, expr) = mul(expr, expr) ;
or_expr(expr, expr) = add(expr, expr) ;
repeat(stmt, expr) = comp(stmt , while(not(expr) , stmt)) }

The added constructs *and_expr*, *or_expr* and *repeat* in the language M are called "syntactic sugar" in [5]. The three contructs above are expressible in K as showed in E_M. By Felleisen[5], M is a definitional extension of K. This kind of extension is a basis of Section 3 in this paper. We can also specify another language $N = (\sum_N, E_N)$ that is on the contrary of extension like in M, where

$S_N = S_K$;
$F_N = F_K -$ { add:expr $\times$ expr$\rightarrow$expr } ;
$E_N = E_K -$ { $eval$(expr '+' expr) } ;

Since $\sum_L$ and $\sum_{L''}$ are the abstract grammars of language L and L'', terms in the ground term algebra $T(\Sigma_L)$ and $T(\Sigma_{L''})$ are well-formed terms, i.e. the terms that are built up from the operator symbols of $\sum_L$ and $\sum_{L''}$. Equations in E_L are the same as transformation rules in transformational semantic description of the language L, they are closed under semantic equivalence inference by reflexitivity, transitivity and symmetry of equality and by substitution. The equation $lhs = rhs$ means that the sentence lhs of language L can be transformed into the sentence rhs of language L'' preserving semantic consistence.

Definition 2 A **language** $L = (\sum_L, E_L)$ consists of a signature $\sum_L$ and a set of $\sum_L$ equations E_L. We define $(\sum_L, E_L)$ as an **abstract model** of programming language L.

From the abstract model of language given above, the grammar of a language L can be abstracted as a set of sorts and operations on the sorts, i.e. as a signature $\sum_L$, whose semantics can be abstracted as a set of $\sum_L$ equations E_L. So a language L can be abstracted as the *abstract model* $(\sum_L, E_L)$ including grammar abstraction and semantic abstraction. The abstract model of a language defined in Def. 2 provides a theoretical framework for studying connections between languages in Section 3.

In Def. 2, we assume that $S_L \cap F_L \cap E_L = \{\perp\}$, where $\perp$ represents undefined element in S_L, F_L and E_L respectively.

3. Language Inheritance, Extension and Shielding — Language Family

In this section, we will discuss the semantics of *inheritance, extension* and *shielding* from L to a new language L'. The relation between L and L' is that between concrete and abstract language.

Definition 3 $\sum_L = (S_L, F_L)$ and $\sum_{L'} = (S_{L'}, F_{L'})$ are signatures of language L and L' respectively. Define the **signature mapping** s from $\sum_L$ to $\sum_{L'}: \sum_L \rightarrow \sum_{L'}$ as a pair (h, g), where $h: S_L \rightarrow S_{L'}$ is the mapping from S_L into $S_{L'}$ and $h(\perp) = \perp$; $g: F_L \rightarrow F_{L'}$ is a partial homomorphism from F_L into $F_{L'}$, where $g(\perp) = \perp$. For $s_i \in S_L (0 \le i \le n)$, and an operation P: $s_1 \times s_2 \times \ldots \times s_n \rightarrow s_0 \in F_L$, we define

$$g(\ P\colon s_1 \times s_2 \times \ \dots\ \times s_n \to s_0\ ) = \begin{cases} \perp & \exists s_i\ h(s_i) = \perp \ or\ P \notin F_{L'} \\ P\colon\ h(s_1) \times h(s_2) \times\ .\ .\ .\ \times h(s_n) \to h(s_0) \end{cases}$$

and the operation $P\colon\ h(s_1) \times h(s_2) \times\ \dots\ \times h(s_n) \to h(s_0) \in F_{L'}$.

If both h and g are identity mappings, then the signature mapping s is an identity mapping; if both h and g are injections, then the signature mapping s is an injection; if both h and g are surjections, then the signature mapping s is a surjection.

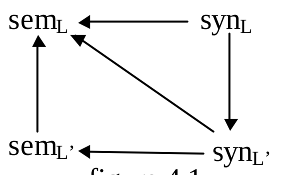
Definition 4 Let $L=(\sum_L, E_L)$ and $L'=(\sum_{L'}, E_{L'})$ be two languages, s be a signature mapping: $\sum_L \to \sum_{L'}$.If equation (s $(lhs)=rhs) \in E_{L'}$ for any equation $lhs=rhs$ in E_L, we also call s a **language mapping** : $L \to L'$.

For languages K and M in Section 2, there exists a signature mapping from $\sum_K$ to $\sum_M$, say, $s_1=(h_1, g_1)$ according to Def. 3, where both h_1 and g_1 are identity mappings, thus s_1 is an identity mapping. Notice, h_1 is a surjection and injection because of $S_M = S_K$, but g_1 is not a surjection because of $F_K \subset F_M$. So, s_1 is not a surjection. Then, in terms of Def. 4, $s_1$is a language mapping: $K \to M$ too. Furthermore, for languages K and N, there also exists a signature mappping from $\sum_K$ to $\sum_N$, say, $s_2=(h_2, g_2)$ where h_2 is an identity mapping: $S_K \to S_K$, and g_2 is a surjection because the operation $add \notin F_N$, i.e. $F_K \supset F_N$, and

g_2(add: expr $\times$ expr$\to$expr)=$\perp$; $g_2(\perp)=\perp$;

Thus g_2 is a surjection but not injection, and so does s_2, because h_2 is an identity mapping. Also in terms of Def. 4, s_2 is a language mapping:$K \to N$ too.

In fact, the signature mapping and the language mapping in Def. 3 and Def. 4 respectively are a special case of commutable communication diagram of Rus in[10]. We can simplify that diagram by the two definitions as figure 4.1, where sem_L and $\text{sem}_{L'}$ are the transformational semantics of L and L' respectively, and syn_L and $\text{syn}_{L'}$ are the syntax of L and L' respectively. For more details, refer to [10][11].


$$\begin{tikzcd} \text{sem}_L & \text{syn}_L \arrow[l] \arrow[d] \arrow[dl, phantom] \\ \text{sem}_{L'} \arrow[u] & \text{syn}_{L'} \arrow[l] \arrow[ul] \end{tikzcd}$$
figure 4.1

In Def. 4, lhs is a term of ground term algebra $T(\Sigma_L)$, and rhs is a term of ground term algebra $T(\Sigma_{L''})$ where L'' is a concrete language of L and also of L'. The relationship between L, L' and L'' expressed by a language sequence like in Section 2 is of L'', L, L'. As ground term consists of sorts and operations in a signature, s (lhs) means to map sorts and operations of $\sum_L$ in lhs into corresponding sorts and operations of $\sum_{L'}$. Assume that s $(lhs)=\perp$ if s $(\ t\ )=\perp$, where t is a sort or an operation on some sorts in lhs, and equation (s $(lhs)=rhs) \in \{\perp\}$ if s $(lhs)=\perp$. We define that s $(\ t\ )=\perp$ if and only if $h(\ t\ )=\perp$ or $g(\ t\ )=\perp$.

Without loss of generality, if a signature mapping s :$\sum_L \to \sum_{L'}$ is an injection and also a surjection, then its language mapping s :$L \to L'$ means semantic inheritance between language L and L', i.e. language L' inheriting language L completely just as an identical language mapping s_0: $K \to K$. If a signature mapping s :$\sum_L \to \sum_{L'}$ is an injection but not a surjection, then its language mapping s :$L \to L'$ means extension relation under the semantic inheritance between L and L'. Language L' is an extended language and also a definitional extension of language L under semantic inheritance. As showed in s_1, M is a definitional extension of K. If a signature mapping s :$\sum_L \to \sum_{L'}$ is a surjection but not an injection, then its language mapping s : $L \to L'$

means shielding relation between language L and L'. Language L' shields some construct structures of language L in semantics, like that language N shields the *add* operation of language K in s_2.

Theorem The **category**[4] AL has languages as objects and mappings between languages as arrows.

Proof：(1) identical language mapping exists.

Let language L be $L=(\Sigma_L, E_L)$, apparently, identity mapping $s:\Sigma_L \rightarrow \Sigma_L$ on signature Σ_L exists, e.g. s_0. Then for any equation $(lhs=rhs) \in E_L$, $s(lhs)=lhs$, so equation $(s(lhs)=rhs) \in E_L$.

Based on Def. 4, signature mapping $s:\Sigma_L \rightarrow \Sigma_L$ is a language mapping:$L \rightarrow L$, and also be an identity mapping.

(2) Composition of language mappings is still a language mapping.

Let languages be $L_1=(\Sigma_{L1}, E_{L1})$, $L_2=(\Sigma_{L2}, E_{L2})$, $L_3=(\Sigma_{L3}, E_{L3})$ and language mappings be $s_1:L_1 \rightarrow L_2$, $s_2:L_2 \rightarrow L_3$. Then from Def. 4, there exist signature mappings $s_1:\Sigma_{L1} \rightarrow \Sigma_{L2}$, $s_2:\Sigma_{L2} \rightarrow \Sigma_{L3}$ corresponding to those language mappings respectively. We can construct the composition of signature mappings based on Def. 3, i.e. $s_3=s_1 \cdot s_2: \Sigma_{L1} \rightarrow \Sigma_{L3}$ as the form of $<h_1 \cdot h_2, g_1 \cdot g_2>$, where $<h_1, g_1>$ and $<h_2, g_2>$ are the representations of s_1 and s_2 respectively, and " $\cdot$ " is composition operation of mappings.

Also from Def. 4, for any equation $(lhs=rhs) \in E_{L1}$, equation $(s_1(lhs)=rhs) \in E_{L2}$. If equation $(s_1(lhs)=rhs) \notin \{\perp\}$, then equation $(s_2(s_1(lhs))=rhs) \in E_{L3}$, namely, equation $(s_3(lhs)=rhs) \in E_{L3}$. If equation $(s_1(lhs)=rhs) \in \{\perp\}$, then equation $(s_2(s_1(lhs))=rhs) \in \{\perp\}$, namely, equation $(s_3(lhs)=rhs) \in \{\perp\} \subseteq E_{L3}$.

Above all, for any equation $(lhs=rhs) \in E_{L1}$, equation $(s_3(lhs)=rhs) \in E_{L3}$. According to Def. 4, s_3 is a language mapping $L_1 \rightarrow L_3$, i.e. composition of language mappings is still a language mapping.

From (1), (2) above and the definition of category[4], the **category** AL has languages as objects and mappings between languages as arrows. □

In category AL, a **based language** consists of a language L and all arrows $\{L_i \rightarrow L\}$, which point to L, represented as $<L, \{L_i \rightarrow L\}>$, where $L_i \rightarrow L$ represents language mapping between L_i and L, L_i are **base language** of the language L.

A *based language* may have many base languages, some base languages may also be based languages, namely, some based language may take other based languages as its base languages, just illustrated as figure 4.2.

In this figure, L_1, L_2 and L_4 are base languages of L. From the figure, we can also see that the based

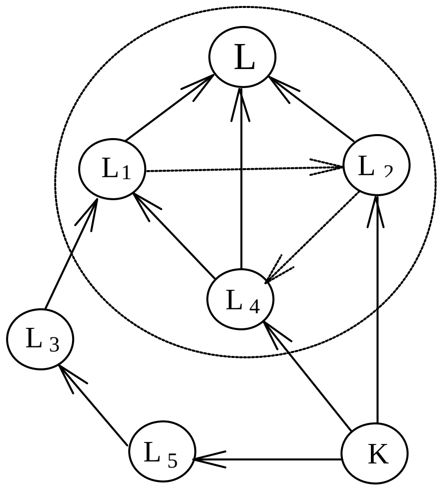


figure 4.2

language L has three base languages L_1, L_2 and L_4, however, the base languages L_1, L_2 and L_4 themselves are also based languages. All languages in the figure construct a *language family model* under inheritance, extension and shielding relations. It is possible that there are not the three relations above between real languages, so we represent this situation by dotted arrows, e.g. between L_1 and L_2, L_2 and L_4. Fortunately, we can construct one of the three relations between L_1 and L_2, L_2 and L_4 based on Def. 3 and Def. 4 exploiting the undefined element $\perp$ and the translator T_1 in [11]. In the language family, which is constructed by language inheritance, extension and shielding, languages are in different levels. Each language in different language levels in the language family has its own *expressive abstract degree*. The higher a *language level* is, the higher the language expressive abstract degree is, and the stronger the language descriptive power is. The lowest base language (e.g. K in the figure 4.2) is called *kernel language*[13], just as the most concrete language. All languages in the language family are achieved from the kernel language by language inheritance, extension and shielding undder the assumption that the semantics of the kernel language is known. From the point of *category theory*, a based language (like L in figure 4.2) consists of base languages (like L_1, L_2 and L_4 in figure 4.2) and mappings between them, and they altogether construct a *cocone*[4]. That is, languages L, L_1, L_2 and L_4 in figure 4.2, and mappings between them construct a cocone, becuase they satisfy that

$$(L_2 \to L)\cdot(L_1 \to L_2) = L_1 \to L$$

$$(L_4 \to L)\cdot(L_2 \to L_4) = L_2 \to L$$

$$(L_1 \to L)\cdot(L_4 \to L_1) = L_4 \to L$$

i.e. L, L_1 and L_2 are commutable, so do L, L_2, L_4 and L, L_1, L_4, where "•" is composition operation of mappings. The meaning of a cocone in language family implies that the methods and ways to development of a language are diverse. We still take figure 4.2 as an example for discussing that the development of language L has many ways. We can develop L from L_4 directly by inheritance, extension and shielding, and also can develop L_1 from L_4, then L from L_1 by the three ways above. The significance of the diversity of methods and ways for developing a language exists in that we can design directly, based on a language L_i (L_1, L_2 or L_4), another language L which is different from L_i in descriptive power and descriptive way, and also can design several different languages whose descriptive powers and descriptive ways range between L_i and L, which are all developed based on the kernel language (like K in figure 4.2). Then language users can also have several selections, they can select those languages which can very fit in their special needed descriptive ways to solve the problems they concern in their own special domains. As a result of it, the problem, which is mentioned in Section 1, of diversity of programming languages and incompatibility between languages may be resolved.

4. Language Abstraction is a Distillation of Data Abstraction

Abstract data type[3] is a new stage of *data abstraction*. The grammar of an abstract data type A can also be described by a signature. We call this signature $\sum_A$. The sorts in the signature $\sum_A$ of an abstract data type A represent *data types*, and the operations represent *operation* *relations* between the data types. However, the sorts in the signature $\sum_L$ of a language L represent *grammar elements* of the language L, the operations represent *constructive relations* between the grammar elements. So $\sum_L$ is the distillation of $\sum_A$ in the eyes of the contents that $\sum_L$ and $\sum_A$ expressed and the level of the objects they described, i.e. $\sum_L$ describes the grammar of the language L, and $\sum_A$ describes the grammar of the abstract data type *A*.

Also being the same as $\sum_L$, abstract data type A can be assigned semantics by a set of equations E_A (a sequence of axioms). An equation $lhs = rhs$ in E_A represents *logic relation* between operations and data types in the abstract data type *A*, where lhs and rhs both are terms of term algebra[7], i.e. finite operations on data types. Nevertheless, equations in E_L, which the language L corresponds to, represent the *transformational relations* between languages under semantic equivalence. Easy to see, E_L is different from E_A in the level of objects they used and in that of properties of the relations they described.

On the other hand, programming languages can be considered as hierarchical types based on abstract data types theory. From this point, the context-free syntax of a language corresponds to the signatures of the types; the context conditions of a language construct are expressed by total, boolean-valued terms; the semantics of the languages is specified by a set of conditional equations[2]. We can gain a complete specification of a programming language with this approach. However, it is difficult, by way of hierarchical types, to show clearly relations (e.g. inheritance, extension and shielding relations) between languages because of the hierarchichal types. From our points in Section 3, and based on the language abstraction and the abstract model from it, we can see clearly the inheritance, extension and shielding relations between languages. Under these relations, languages construct a language family and category AL as showed in Section 3. A programming language family, in fact, is organised as hierarchical languages which are distillation of hierarchical types.

Above all, comparing language abstraction $(\sum_L, E_L)$ with data abstraction $(\sum_A, E_A)$, language abstraction is a distillation of data abstraction.

5. Related Work and Conclusion

Several authors have discussed about the model of programming languages and related topics. Milner[8] proposed a fully abstract model of typed lambda-calculi and showed that under certain conditions, there exists, for an extended typed lambda-calculus, a fully abstract model including algebraic model. We can also envision that there exists an abstract model we have proposed for programming languages. Pair[9] and Broy *et al.*[2] studied on the algebraic definition of programming languages. Both two papers emphasized the algebraic semantics of programming languages based on abstract data types. Under the proposal that "programming languages should be studied in terms of algebraic theories"[Goguen], Broy et al. described, analyzed and constrasted the transformational semantics and weakly terminal semantics. We combine transformational semantics to abstract data types theory to specify programming languages. From the point of language implementation, Rus[10][11] gave an algebraic model for programming languages. BNF plays central role in his framework. However, in his papers, Rus did not discuss how to describe semantics of programming languages in his model. Moreover, under some conditions, e.g., the undefined condition in this paper, the consistency relation in his papers between languages may not exist. Felleisen[5] provided a formal framework for comparing the expressive power of programming languages. He took Scheme as an example language, and discussed about the coservative and definitional extension of Scheme. One of the purposes of our work is to apply and explain his research work in the abstract model of this paper. Under the transformational semantics of programming languages, the mapping $s_1 : K \to M$ in Section 3 means a conservative and definitional extension of language K to M in our abstract model. In the framework of Felleisen, it is difficult to explain the meaning of the mapping like $s_2 : K \to N$, because he discussed only about language extension. However, there are some differences between language extension and shielding to some extend after all.

Language abstraction discussed in this paper is the distillation of data abstraction. This concept builds up a theoretical model for researching connections between languages. Language family developed from kernel language is a powerful language development tool and supporting environment of the model-oriented transformational software development methodology. While we develop the Polya-BD[12][13], which is a double environment for language development and program development, we also have advanced a mechanism, based on the language abstract model we have proposed, for language abstraction and encapsulation called *Garment*, whose details will be given in another prepared paper[13]. Garment is an implementation of the abstract model of language. By the mechanism, we can implement language inheritance, extension and shielding, and can construct a language family consisted of languages which are different domain-oriented. Language users can work on any language level they need in language family in order to develop software easily. Further work is to study on the sufficiency of kernel language, the correctness and semantics of language transformation, the sufficient completeness and consistence of language extension.

References

[1] Bjorner, D. Toward the Meaning of VDM 'M'. *LNCS* 352; TAPSOFT'89: 1—35; 1989.

[2] Broy, M. *et al.* On the Algebraic Definition of Programming Languages. *ACM Transaction on Programming Languages and Systems* 9(1): 54—99; 1987.

[3] Cardelli, L. and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys* 17(4): 471—522; 1985.

[4] Chen, Yiyun. *Category Theory in Computer Science.* Science & Technology University Press, China; 1994.

[5] Felleisen, M. On the Expressive Power of Programming Languages. *LNCS* 432; ESOP'90: 134—151; 1990.

[6] Hoare, C.A.R. The Varieties of Programming Languages. *LNCS* 351; TAPSOFT'89: 1—18; 1989.

[7] Lu, Ruqian. *Formal Semantics of Computer Languages*, Science Press, China; 1994.

[8] Milner, M. Fully Abstract Models of Typed Lambda-Calculi. *Theoretical Computer Science* 4(1): 1—22; 1977.

[9] Pair, C. Abstract Data Types and Algebraic Semantics of Programming Languages. *Theoretical Computer Science* 18: 1—31; 1982.

[10] Rus, T. An Algebraic Model for Programming Languages. *Computer Languages* 12(3/4): 173—195; 1987.

[11] Rus, T. An Algebraic Tool for Language Processing. *Computer Languages* 20(4): 213—238; 1994.

[12] Zhang, Naixiao Notation of Program Transformation in Programming Languages —on Transformational Language. *Chinese Journal of Software* 4(5): 17—23; 1993.

[13] Zhang, Naixiao *et al.* On Model-Oriented Transformational Software Development Methodology. *Theoretical Computer Science*(China) 2: 54—64; 1994.

[14] Zhang, Naixiao and Zheng, Hongjun Garment — A New Mechanism of Abstraction and Encapsulation. *Technical Report*; 1996.

Summary

(*An Abstract Model for Programming Languages*)

This paper presents a concept of language abstraction and an abstract model for programming languages through studying into the essence of programming languages based on the research of Polya language and model-oriented transformational software development methodology. Then the paper describes the semantics of language inheritance, extension, and shielding and also the semantics of language family.

In Section 2, we describe the *abstract grammar* of a language with a *signature* $\Sigma=(S,F)$, which consists of a set of sorts S (basic grammar elements) and a set of operations F (combinatorial relations among grammar elements), and adopt the means of *transformational semantics* to assign semantics to Σ **,** in which we regard Σ equations E_L as transformation rules. So a language L can be abstracted as the *abstract model* (Σ_L, E_L) including grammar abstraction and semantic abstraction. The abstract model of a language provides a theoretical framework for studying connections between languages based on which we discuss in Section 3 the semantics of *inheritance, extension* and *shielding* from L to a new language L'.

According to the definitions of signature mapping and language mapping in the paper, we conclude, without loss of generality, that if the signature mapping $s : \Sigma_L \to \Sigma_{L'}$ is an injection and also a surjection, then the language mapping $s : L \to L'$ means semantic inheritance between language L and L'; if signature mapping $s : \Sigma_L \to \Sigma_{L'}$ is an injection but not a surjection, then the language mapping $s : L \to L'$ means extension relation under the semantic inheritance between L and L'; and if signature mapping $s : \Sigma_L \to \Sigma_{L'}$ is a surjection but not an injection, then the language mapping $s : L \to L'$ means shielding relation between language L and L'. We also give several examples to illustrate the conclusion above. Furthermore, we proof the thoerem in this paper, that languages and mappings between them constitute a category AL. Then, discuss about the semantics of language family in the category AL.

In Section 4, we try to explain, by comparing language abstraction (Σ_L, E_L) with data abstraction (Σ_A, E_A), that language abstraction proposed in the paper is a distillation of data abstraction from three points: grammar, semantics and language specification method.

Finally, we discuss about some related work which have something to do with this paper, and give some concluding remarks in Section 5.

Two referees suggested by the authors:

Professor David Gries

Address:	Computer Science Department Cornell University Ithaca, New York 14853 - 7501 U.S.A

Professor Dines Bjorner

Address:	International Institute for Software Technology The United Nations University (UNU/IIST) P.O. Box 3085 18/F Ed. Banco Luso Intl. 1 - 3, Rua Dr. Pedro Jose Lobo Macau
Email:	db@iist.unu.edu