A Type System and Type Soundness for the Calculus of Aspect-Oriented Programming Languages

Dinesh Gopalani, M. C. Govil, and K. C. Jain

***Abstract*—The formal study of class of functional and procedure-oriented programming languages is well-defined and uses λ-calculus as the main tool. With the advent of object calculi, the formal study of object-oriented programming languages is also well developed and understood. Since the paradigm of aspect-oriented programming is new, formal theory for the same is under development. The proposed untyped aspect calculus provides direct support for aspects and other construct of aspect-oriented programming languages however without considering typing information. We propose here the simple-typed aspect calculus - a type system for the aspect calculus. The proposed simple-typed aspect calculus includes typing rules for all the terms of untyped calculus and the same are discussed in the paper. The theorem of type soundness along with its proof for the aspect calculus is also derived and discussed here. The proof is based on method of induction and states that reduction rules defined in the operational semantics of untyped aspect calculus is consistent with the type system of simple-typed aspect calculus. The proposed theory is very useful in studying and understanding various properties related to existing aspect-oriented languages.**

***Index Terms*—aspect-oriented programming, formal study, object calculi, reduction rules, simple-typed aspect calculus, type soundness, type system, typing rules.**

I. Introduction

The object calculi [1] proposed by Abadi and Cardelli, treat objects rather than functions as their main primitive constructs and define operations on these objects directly. This approach used by the object calculi overcomes the problem of complex encoding of objects as functions which usually occurs when λ-calculus [2], [3] is used to model features of object-oriented programming languages. With the object calculi, the formal study of object-oriented programming languages is well developed and understood however the calculi do not provide direct support for aspects and other related constructs of aspect-oriented programming languages [4], [5]. Since the paradigm of aspect-oriented programming is new, formal theory for the same is under development. The scope of object calculi is primarily confined to the family of conventional object-oriented languages but can also be used for aspect-oriented languages, however this may result into some lengthy and complex encodings of aspects in terms of conventional objects. To circumvent this problem, we propose aspect calculi which provide direct support for aspects and other related constructs of aspect-oriented programming paradigm.

The existing approaches to deal with the formal theory of aspect-oriented languages mainly include Parameterized Aspect Calculus [6], A Calculus of Untyped Aspect-oriented Programs [7], and A Theory of Aspects [8]. However our approach is quite different in the sense that we propose a formal theory for aspect-oriented programming languages by providing aspect calculi which are extensions to object calculi and based on imperative execution model. The Untyped Aspect Calculus [9] deals with aspects as basic primitives and define operations on these primitives, however without considering any typing information. We propose here a type system for the calculus which includes object and aspect types and typing rules. The proposed typed calculus is named as "Simple-typed Aspect Calculus". Type safety or type soundness is an important property of any type system and according to which a term must preserve its type in the process of reduction or evaluation. The theorem of type soundness along with its proof for the aspect calculi is also devised. The proof is based on method of induction and states that reduction rules defined in the operational semantics of untyped aspect calculus is consistent with the type system of simple-typed aspect calculus. The proposed theory is very useful in studying and understanding the existing aspect-oriented languages as well as designing and implementing the new ones.

Manuscript received November 16, 2011; revised February 3, 2012.

Dinesh Gopalani is with the Department of Computer Engineering, Malaviya National Institute of Technology, Jaipur, India. e-mail:dgopalani@rediffmail.com, dg@mnit.ac.in

M. C. Govil is currently Principal of Government Mahila Engineering College, Ajmer, India and on deputation from the Department of Computer Engineering, Malaviya National Institute of Technology Jaipur, India. e-mail:govilmc@yahoo.com

K. C. Jain is with the Department of Mathematics, Malaviya National Institute of Technology, Jaipur, India. e-mail:jainkc_2003@yahoo.com

II. Simple-Typed Aspect Calculus

The type system for aspect calculi is devised and discussed in this section. Here typing constructs for aspects and other terms are given in a very simplified form. The calculus does not include higher constructs like polymorphism and self types.

A. Syntax

The syntax of the simple-typed aspect calculus includes two types - object type and aspect type. The object type

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
SYNTAX OF SIMPLE-TYPED ASPECT CALCULUS

$S, T ::=$	types
$A, B ::=$	object types
$[l_i : B_i^{\,i\in 1..n}]$	object type (l_i distinct)
$C, D ::=$	aspect types
$[l_i : D_i^{\,i\in 1..p}, d_i : D_i^{\,i\in p+1..q}]$	aspect type (l_i, d_i distinct)
$a, b ::=$	terms
x	variable
$[l_i = \varsigma(x_i : B_i) b_i^{\,i\in 1..n}]$	object (l_i distinct)
$[l_i = \varsigma(x_i : D_i) b_i^{\,i\in 1..p}, d_i : p = \varsigma(x_i : D_i) b_i^{d\ i\in p+1..q}]$	aspect (l_i, d_i distinct)
$a.l$	method invocation
$a.l \Leftarrow \varsigma(x : A) b$	method update
$clone(a)$	clone
$let\ x = a\ in\ b$	let
$b^{\,d} ::=$	advice body term
$let\ x = proceed(a)\ in\ b$	proceed
$return(a)$	return
$p, q ::=$	pointcut
$\neg p$	negation
$p \wedge q$	conjunction
$p \vee q$	disjunction
$call(l_i)$	method call
$get(l_i)$	field get
$set(l_i)$	field set
$target(a)$	receiver object

$[l_i : B_i^{\,i\in 1..n}]$ indicates that there are n methods in the object and method bodies have types $B_1,\ B_2,\ \ldots\ B_n$ respectively for each method labeled l_i, where $i \in 1..n$. An aspect type is extended form of an object type since an aspect can have advice in addition to conventional methods. All the conventional and advice methods of an aspect must have distinctive labels.

All terms defined for the untyped version of aspect calculus, i.e., variable, object and aspect terms, method invocation, method update, clone, let, proceed, return terms, and pointcut primitives are also included here along with type information wherever needed. An object term is defined as $[l_i = \varsigma(x_i : B_i) b_i^{\,i\in 1..n}]$ with type as $[l_i : B_i^{\,i\in 1..n}]$, and an aspect term is represented as $[l_i = \varsigma(x_i : D_i) b_i^{\,i\in 1..p}, d_i : p = \varsigma(x_i : D_i) b_i^{d\ i\in p+1..q}]$ with type $[l_i : D_i^{\,i\in 1..p}, d_i : D_i^{\,i\in p+1..q}]$. The method and advice are represented by the notation $\varsigma(x : T)b$, where the bound variable $x : T$ represents self of type T and body b that produces the result. A method invocation written as $a.l$, executes the body of method labeled l of the term a with self parameter bound to term a. A method update term is given as $a.l \Leftarrow \varsigma(x : A)b$, where A is the type of the self parameter associated with the new method, replaces method labeled l of term a with new method $\varsigma(x : A)b$. The cloning operation $clone(a)$ produces a new object/aspect with the same method labels as a, with each component sharing the methods of the corresponding component of a. The let term is very important and provides support for imperative execution.

TABLE II
TYPING RULES FOR ENVIRONMENT AND VARIABLES

$$\frac{}{\Phi \vdash \diamond}\ (Env\ \Phi) \qquad \frac{\Gamma \vdash T \qquad x \notin dom(\Gamma)}{\Gamma, x : T \vdash \diamond}\ (Env\ x) \qquad \frac{\Gamma', x : T, \Gamma'' \vdash \diamond}{\Gamma', x : T, \Gamma'' \vdash x : T}\ (Val\ x)$$

A let term $let\ x = a\ in\ b$, first evaluates the term a, binds the result to variable x, and then evaluates the second term b with binding of that variable x in scope.

An advice can take either a proceed or a return action hence two terms - $proceed$ and $return$ terms are introduced. A proceed term $let\ x = proceed(a)\ in\ b$ transfers the control to a subsequent aspect in the aspect precedence and a return term $return(a)$ is used for transferring the control back to the caller. The pointcut p used in the description of an advice includes negation, conjunction and disjunction, method call and method execution, field get and field set, and target object. Table I gives the syntax for the simple-typed aspect calculus.

B. Typing Rules

Typing rules for all the terms of the calculus are described here. The typing rules mainly use two kind of judgments, i.e., type judgment $\Gamma \vdash T$ and value typing judgment $\Gamma \vdash a : T$. Here Γ represents the typing context or typing environment which is a sequence of binding of variables with their types. A type judgment $\Gamma \vdash T$ states that T is a well-formed type in the environment Γ. And a value typing judgment $\Gamma \vdash a : T$ states that term a has type T in the environment Γ.

The typing rules given in Table II are meant for building environments and to get the types of variables from an environment. The type rule $(Env\ \Phi)$ is the most fundamental rule and does not require any premise judgments. It states that the empty environment is a well-formed environment. The rule $(Env\ x)$ is used to extend an environment Γ to a longer environment $(\Gamma, x : T)$, if and only if T is a valid type in Γ and x is not in domain of Γ. The rule $(Val\ x)$ is used to extract the type for variable x from the environment Γ. Here the notation $\Gamma', x : T, \Gamma''$ means that the typing information of variable x may be available somewhere in the environment.

Next we define the rules for object and aspect typing and the same are given in Table III. The rule $(Type\ Object)$ is related to object typing and according to this rule, an object type $[l_i : B_i^{\,i\in 1..n}]$ is well-formed in the environment Γ, provided that all B_i's are well-formed types in Γ. The rule $(Type\ Aspect)$ is related to aspect typing and very much similar to the previous rule $(Type\ Object)$. The only difference here is that the types are added for advice methods along with the conventional methods. As per the rule $(Val\ Object)$, an object of type $[l_i : B_i^{\,i\in 1..n}]$ can be formed from a collection of n methods whose self parameters are of the

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE III
TYPING RULES FOR OBJECT AND ASPECT TERMS

$$\frac{\Gamma \vdash B_i \quad \forall\, i \in 1..n}{\Gamma \vdash [l_i : B_i^{\,i\in 1..n}]}\ (Type\ Object)\ (l_i\ distinct) \qquad \frac{\Gamma \vdash D_i \quad \forall\, i \in 1..q}{\Gamma \vdash [l_i : D_i^{\,i\in 1..p}, d_i : D_i^{\,i\in p+1..q}]}\ (Type\ Aspect)\ (l_i, d_i\ distinct)$$

$$\frac{\Gamma, x_i : A \vdash b_i : B_i \quad \forall\, i \in 1..n}{\Gamma \vdash [l_i = \varsigma(x_i) b_i^{\,i\in 1..n}] : A}\ (Val\ Object)\ (where\ A \equiv [l_i : B_i^{\,i\in 1..n}])$$

$$\frac{\Gamma, x_i : C \vdash b_i : D_i \ \ \forall i \in 1..p \quad \Gamma, x_i : C \vdash b_i^d : D_i \ \ \forall i \in p+1..q}{\Gamma \vdash [l_i = \varsigma(x_i) b_i^{\,i\in 1..p}, d_i : p = \varsigma(x_i) b_i^{\,d\ \ i\in p+1..q}] : C}\ (Val\ Aspect)\ (where\ C \equiv [l_i : D_i^{\,i\in 1..p}, d_i : D_i^{\,i\in p+1..q}])$$

TABLE IV
TYPING RULES FOR OTHER TERMS

$$\frac{\Gamma \vdash a : [l_i : B_i^{\,i\in 1..n}] \quad j \in 1..n}{\Gamma \vdash a.l_j : B_j}\ (Val\ Inv)$$

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A \vdash b : B_j \quad j \in 1..n}{\Gamma \vdash a.l_j \Leftarrow \varsigma(x : A) b \ : \ A}\ (Val\ Update)\ (where\ A \equiv [l_i : B_i^{\,i\in 1..n}])$$

$$\frac{\Gamma \vdash a : T}{\Gamma \vdash clone(a) : T}\ (Val\ Clone) \qquad \frac{\Gamma \vdash a : S \quad \Gamma, x : S \vdash b : T}{\Gamma \vdash let\ x = a\ in\ b \ : \ T}\ (Val\ Let)$$

$$\frac{\Gamma \vdash a : S \quad \Gamma, x : S \vdash b : T}{\Gamma \vdash let\ x = proceed(a)\ in\ b : T}\ (Val\ Proceed) \qquad \frac{\Gamma \vdash a : T}{\Gamma \vdash return(a) \ : \ T}\ (Val\ Return)$$

same type $[l_i : B_i^{\,i\in 1..n}]$ and whose bodies are of types $B_1,\ B_2,\ ...,\ B_n$ respectively. Next the rule $(Val\ Aspect)$ defines typing of an aspect term. According to this rule, an aspect term $[l_i = \varsigma(x_i) b_i^{\,i\in 1..p}, d_i : p = \varsigma(x_i) b_i^{\,d\ \ i\in p+1..q}]$ is of type $C \equiv [l_i : D_i^{\,i\in 1..p}, d_i : D_i^{\,i\in p+1..q}]$ provided that the self parameters x_i's are must be of same type, i.e., C and method bodies are of types $D_1,\ D_2,\ ...,\ D_p$ and advice bodies are of types $D_{p+1},\ D_{p+2},\ ...,\ D_q$ respectively.

Table IV defines typing rules for all the remaining terms of the calculus. The rule $(Val\ Inv)$ states that the invocation of method l_j (where $j \in 1..n$) of an object of type $[l_i : B_i^{\,i\in 1..n}]$ produces the result of type B_j. And the rule $(Val\ Update)$ preserves the type of an object whose method is updated. A method l_j (where $j \in 1..n$) of an object of type $[l_i : B_i^{\,i\in 1..n}]$ can be updated with the new method $\varsigma(x)b$ provided that the body b of this new method is of type B_j under the assumption that self x has the same object type $[l_i : B_i^{\,i\in 1..n}]$. According to the rule $(Val\ Clone)$, the term $clone(a)$ has type T provided that the original term a is also of type T. The rule $(Val\ Let)$ states that the term $let\ x = a\ in\ b$ has type T provided that the term a is of type S and the term b is of type T with the assumption that the variable x is of type S in the environment. Next we have the typing rules for the proceed and return terms which may appear as part of advice body. The typing rule for proceed is very much similar to the typing rule for let term and is given as $(Val\ Proceed)$. Finally, the typing rule $(Val\ Return)$ states that the term $return(a)$ has type T provided that the corresponding term a is also of type T in the environment Γ.

III. TYPE SOUNDNESS

Now it is to be proved that the operational semantics of the untyped aspect calculus is consistent with the given type system of the simple-typed aspect calculus. This consistency property is known as the type soundness or type safety. The type soundness property states that reduction or evaluation must preserve types, i.e., if a well-typed term t has type T and when the term t reduced to result v, then result v must has the same type T. To prove the type soundness for the calculus we need to provide typing for results and the method store as given below. Using this typing information, the theorem of type soundness and its proof is devised and also discussed in this section.

A. Result and Method Store Typing

For the proof of type soundness property for our calculus it is necessary to provide types to results as well as for the method store. The type of method store σ_m is represented as Σ, which provides mappings from each store location to its corresponding method type. A method type $M ::= [l_i : T_i^{\,i\in 1..n}] \rightarrow T_j$ consists of two components, where the first component $[l_i : T_i^{\,i\in 1..n}]$ is the type of self object/aspect and second component T_j is the result type of the corresponding method/advice body. Since method type consists of two components, $\Sigma_s(\iota)$ represents self type associated with the store location ι and $\Sigma_b(\iota)$ represents result type of body associated with the location ι. Next we define the following judgments which are required here:

$\vDash M$	well-formed method type judgment
$\Sigma \vDash \diamond$	well-formed method store type judgment
$\Sigma \vDash v : T$	result typing judgment
$\Sigma \vDash \sigma_m$	method store typing judgment
$\Sigma \vDash S_e : \Gamma$	environment stack type judgment

The first judgment $\vDash M$ defines the well-formedness of the method type M. The well-formedness of the method

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

store type is given by the judgment $\Sigma \vDash \diamond$. Then we have the judgment $\Sigma \vDash v : T$ meant for the typing of results. According to this, the result v has type T with method store type as Σ. All the locations in v are assigned types in Σ. The judgment $\Sigma \vDash \sigma_m$ states that the method store σ_m is compatible with the method store type Σ. Here all method closures stored in locations of σ_m are compatible to the method types associated with those locations in method store type Σ. Finally the last judgment $\Sigma \vDash S_e : \Gamma$ defines that the environment stack S_e is compatible with the typing context Γ with respect to method store type Σ.

Typing rules for the method store and other related constructs are defined and given in Table V. According to the rule $(Method\ Store\ Type)$, the method store type consists of method types M_i's associated with the locations ι_i's for all $i \in 1..m$, is well-formed provided that M_i is well formed for all $i \in 1..m$. There are two rules for the result typing as there are two possible results - object and aspect. The rule $(Result\ Type - Object)$ states that the result of an object term $[l_i = \iota_i^{\ i\in 1..n}]$ has type $[l_i : \Sigma_b(\iota_i)^{\ i\in 1..n}]$ provided that the corresponding method store type Σ is well formed and the types associated with locations ι_i are such that $\Sigma_s(\iota_i) \equiv [l_i : \Sigma_b(\iota_i)^{\ i\in 1..n}]$, for all $i \in 1..n$. Similarly, for aspect result typing the rule $(Result\ Type - Aspect)$ is defined and according to which an aspect value $[l_i = \iota_i^{\ i\in 1..p},\ d_i : p = \iota_i^{\ i\in p+1..q}]$ has type $[l_i : \Sigma_b(\iota_i)^{\ i\in 1..p},\ d_i : \Sigma_b(\iota_i)^{\ i\in p+1..q}]$ provided that the store type has $\Sigma_s(\iota_i) \equiv [l_i : \Sigma_b(\iota_i)^{\ i\in 1..p}]$ and $\Sigma_s(\iota_i) \equiv [d_i : \Sigma_b(\iota_i)^{\ i\in p+1..q}]$, for all $i \in 1..p+q$. Next two rules $(Env\ Stack\ \phi\ Typing)$ and $(Env\ Stack\ S_e\ Typing)$ define the typing rules for the environment stack used in the underlying operational semantics. The rule $(Env\ Stack\ \phi\ Typing)$ is meant for empty environment stack and states that the empty stack is always compatible with the empty typing context provided that corresponding method store type is well-formed. The rule $(Env\ Stack\ S_e\ Typing)$ states that the extended environment stack $S_e, x \mapsto v$ is compatible with the typing context $\Gamma, x : T$ with respect to store type Σ, provided that the stack S_e is compatible with the context Γ and the result v has type T with respect to the same store type Σ, and the variable x must not be there in the domain of the context Γ. The last rule here provides the typing for the method store and named as $(Method\ Store\ Typing)$. According to this rule, the method store $\iota_i \mapsto \langle \varsigma(x_i) b_i, S_{e_i} \rangle^{\ i\in 1..n}$ is compatible with the method store type Σ provided that the corresponding stacks S_{e_i}'s are compatible with the typing contexts Γ_i's and the method bodies b_i's have types $\Sigma_b(\iota_i)$ in the typing context $\Gamma_i,\ x_i : \Sigma_s(\iota_i)$, for all $i \in 1..n$.

TABLE V
RULES FOR RESULT AND METHOD STORE TYPING

$$\frac{\vDash M_i \quad \forall i \in 1..m}{\iota_i \mapsto M_i^{\ i\in 1..m} \vDash \diamond}\ (Method\ Store\ Type)$$

$$\frac{\Sigma \vDash \diamond \quad \Sigma_s(\iota_i) \equiv [l_i : \Sigma_b(\iota_i)^{i\in 1..n}]}{\Sigma \vDash [l_i = \iota_i^{\ i\in 1..n}] : [l_i : \Sigma_b(\iota_i)^{\ i\in 1..n}]}\ (Result\ Type - Object)$$

$$\frac{\Sigma \vDash \diamond \quad \Sigma_s(\iota_i) \equiv [l_i : \Sigma_b(\iota_i)^{\ i\in 1..p}] \quad \Sigma_s(\iota_i) \equiv [d_i : \Sigma_b(\iota_i)^{\ i\in p+1..q}] \quad \forall i \in 1..p+q}{\Sigma \vDash [l_i = \iota_i^{\ i\in 1..p}, d_i : p = \iota_i^{\ i\in p+1..q}] : [l_i : \Sigma_b(\iota_i)^{i\in 1..p}, d_i : \Sigma_b(\iota_i)^{i\in p+1..q}]}\ (Result\ Type - Aspect)$$

$$\frac{\Sigma \vDash \diamond}{\Sigma \vDash \phi : \phi}\ (Env\ Stack\ \phi\ Typing)$$

$$\frac{\Sigma \vDash S_e : \Gamma \quad \Sigma \vDash v : T \quad x \notin dom(\Gamma)}{\Sigma \vDash S_e, x \mapsto v : \Gamma, x : T}\ (Env\ Stack\ S_e\ Typing)$$

$$\frac{\Sigma \vDash S_{e_i} : \Gamma_i \quad \Gamma_i,\ x_i : \Sigma_s(\iota_i) \vdash b_i : \Sigma_b(\iota_i) \quad \forall i \in 1..n}{\Sigma \vDash \iota_i \mapsto \langle \varsigma(x_i) b_i, S_{e_i} \rangle^{\ i\in 1..n}}\ (Method\ Store\ Typing)$$

B. Proof of Type Soundness

The result typing and store typing described in the previous sub-section are required for the proof of type soundness. Now we provide the theorem and proof for type soundness for the aspect calculi. Before giving the theorem and its proof we give a definition for the extension of method store type and a lemma related to this extension.

Definition 1: A store type Σ' is an extension of the store type Σ (written as $\Sigma' \supseteq \Sigma$) if $dom(\Sigma') \supseteq dom(\Sigma)$ and for all $\iota \in dom(\Sigma)$, $\Sigma'(\iota) = \Sigma(\iota)$.

Lemma 3.1: If $\Sigma \vDash S_e : \Gamma$ and $\Sigma' \vDash \diamond$ with $\Sigma' \supseteq \Sigma$, then $\Sigma' \vDash S_e : \Gamma$.

Theorem 3.1:
If $\Gamma \vdash a : T$
$(\sigma_m\ ,\ \sigma_a).(S_e, S_d) \vdash a \rightsquigarrow v.(\sigma_m'\ ,\ \sigma_a')$
$\Sigma \vDash \sigma_m$
$\Sigma \vDash S_e : \Gamma$
then for some $\Sigma' \supseteq \Sigma$,
$\Sigma' \vDash \sigma_m'$
$\Sigma' \vDash v : T$

The above theorem states that with the following assumptions:

- A term a has type T in the context Γ.
- With the method store σ_m, aspect sequence σ_a, environment stack S_e, and the advice stack S_d, the term a reduces to a result v; method store and aspect sequence get changed in the process and these updated structures are represented by σ_m' and σ_a', respectively.
- The method store σ_m is compatible with the method store type Σ.
- The environment stack S_e is compatible with the typing context Γ with respect to method store type Σ.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

then we have the following conclusion for some method store type $\Sigma' \supseteq \Sigma$:

- The updated method store σ'_m is compatible with the extended method store type Σ'.
- Also the result v has type T with respect to the extended method store type Σ'.

Proof: The proof for the above theorem is given using method of induction on the reduction of terms of the calculus as defined by the underlying operational semantics. For every term of the calculus we have derived the proof, however due to space constraint here we provide the proof for only some of the terms and is given below as cases for each of these terms.

Case Variable Term (x):
For the reduction of variable term the rule $(Red\ Var)$ is applicable and given in the operational semantics of the untyped aspect calculus [9]. The reduction rule for variable term is given as below:

$$\frac{(Red\ Var)\quad (\sigma_m, \sigma_a).(\ (S_e',\ x \mapsto v,\ S_e''),\ S_d) \vdash \diamond}{(\sigma_m, \sigma_a).(\ (S_e',\ x \mapsto v,\ S_e''),\ S_d) \vdash x \rightsquigarrow v.(\sigma_m, \sigma_a)}$$

By hypothesis

$\Gamma \vdash x : T$
$(\sigma_m, \sigma_a).(\ (S_e',\ x \mapsto v,\ S_e''),\ S_d) \vdash x \rightsquigarrow v.(\sigma_m, \sigma_a)$
$\Sigma \vDash \sigma_m$
$\Sigma \vDash S_e',\ x \mapsto v,\ S_e'' : \Gamma$
we need to prove for some $\Sigma' \supseteq \Sigma$,
$\Sigma' \vDash \sigma_m$
$\Sigma' \vDash v : T$

Since $\Gamma \vdash x : T$, we must have the typing context $\Gamma \equiv \Gamma', x : T, \Gamma''$. The judgment $\Sigma \vDash S_e',\ x \mapsto v,\ S_e'' : \Gamma$ in the hypothesis must have been derived using the $(Env\ Stack\ S_e\ Typing)$ rule with premise $\Sigma \vDash v : T$. Here for the reduction of variable term the method store remains unchanged, so we can take $\Sigma' \equiv \Sigma$. Now we conclude that $\Sigma' \vDash \sigma_m$ and $\Sigma' \vDash v : T$.

Case Aspect Term ($[l_i = \varsigma(x_i)b_i^{\ i\in 1..p},\ d_i : p = \varsigma(x_i)b_i^{d\ i\in p+1..q}]$):
Now we discuss the proof of the Theorem 3.1 for the case when the term is an aspect term. The reduction rule for aspect term $(Red\ Aspect)$ is as given below.

$$\frac{\begin{array}{l}(Red\ Aspect)\quad (l_i,\ d_i,\ \iota_i\ distinct)\\ a \equiv [l_i = \varsigma(x_i)b_i^{\ i\in 1..p},\ d_i : p = \varsigma(x_i)b_i^{d\ i\in p+1..q}]\\ v \equiv [l_i = \iota_i^{\ i\in 1..p},\ d_i : p = \iota_i^{\ i\in p+1..q}]\end{array}\quad (\sigma_m\ ,\ \sigma_a).(S_e, S_d) \vdash \diamond \quad \iota_i \notin dom(\sigma_m) \quad \forall i \in 1..q}{(\sigma_m, \sigma_a).(S_e, S_d) \vdash a \rightsquigarrow v.((\sigma_m, \iota_i \mapsto \langle \varsigma(x_i)b_i, S_e\rangle^{i\in 1..p},\ \iota_i \mapsto \langle \varsigma(x_i)b_i^d, S_e\rangle^{\ i\in p+1..q}),\ \sigma_a\ +\ v)}$$

By hypothesis

$\Gamma \vdash [l_i = \varsigma(x_i)b_i^{\ i\in 1..p},\ d_i : p = \varsigma(x_i)b_i^{d\ i\in p+1..q}] : C$
$(\sigma_m, \sigma_a).(S_e, S_d) \vdash a \rightsquigarrow v.((\sigma_m, \iota_i \mapsto \langle \varsigma(x_i)b_i, S_e\rangle^{i\in 1..p},\ \iota_i \mapsto \langle \varsigma(x_i)b_i^d, S_e\rangle^{\ i\in p+1..q}),\ \sigma_a\ +\ v)$,
where $a \equiv [l_i = \varsigma(x_i)b_i^{\ i\in 1..p},\ d_i : p = \varsigma(x_i)b_i^{d\ i\in p+1..q}]$,
$v \equiv [l_i = \iota_i^{\ i\in 1..p},\ d_i : p = \iota_i^{\ i\in p+1..q}]$
$\Sigma \vDash \sigma_m$
$\Sigma \vDash S_e : \Gamma$
we need to prove for some $\Sigma' \supseteq \Sigma$,
$\Sigma' \vDash \sigma_m,\ \iota_i \mapsto \langle \varsigma(x_i)b_i, S_e\rangle^{\ i\in 1..p},\ \iota_i \mapsto \langle \varsigma(x_i)b_i^d, S_e\rangle^{i\in p+1..q}$
$\Sigma' \vDash [l_i = \iota_i^{\ i\in 1..p},\ d_i : p = \iota_i^{\ i\in p+1..q}] : C$

Let $C \equiv [l_i : D_i^{\ i\in 1..p},\ d_i : D_i^{\ i\in p+1..q}]$ and we take $\Sigma' \equiv \Sigma, \iota_i \mapsto (C \rightarrow D_i)^{\ i\in 1..q}$, by using $(Method\ Store\ Type)$ and using $\iota_i \notin dom(\Sigma)$ since $\iota_i \notin dom(\sigma_m)$ for all $i \in 1..q$, we have $\Sigma' \vDash \diamond$ and $\Sigma' \vDash S_e : \Gamma$ by using Lemma 3.1.

$$\frac{\text{Proof of } \Sigma' \vDash \sigma_m,\ \iota_i \mapsto \langle \varsigma(x_i)b_i, S_e\rangle^{\ i\in 1..p},\ \iota_i \mapsto \langle \varsigma(x_i)b_i^d, S_e\rangle^{\ i\in p+1..q}}{}$$

Using the hypothesis $\Gamma \vdash [l_i = \varsigma(x_i)b_i^{\ i\in 1..p},\ d_i : p = \varsigma(x_i)b_i^{d\ i\in p+1..q}] : C$, we have $\Gamma, x_i : C \vdash b_i : D_i,\ \forall i \in 1..q$ by the rule $(Val\ Aspect)$. Further it can be written as

$$\Gamma, x_i : \Sigma'_s(\iota_i) \vdash b_i : \Sigma'_b(\iota_i),\ \forall i \in 1..q \qquad (1)$$

The method store σ_m before the reduction of the aspect term may be of the form $\sigma_m \equiv \tau_j \mapsto \langle \varsigma(x_j)b_j, S_{e_j}\rangle^{\ j\in 1..m}$. Now the hypothesis $\Sigma \vDash \sigma_m$ has come from the $(Method\ Store\ Typing)$ rule, with $\Sigma \vDash S_{e_j} : \Gamma_j$ and $\Gamma_j, x_j : \Sigma_s(\tau_j) \vdash b_j : \Sigma_b(\tau_j),\ \forall j \in 1..m$. By Lemma 3.1 we have $\Sigma' \vDash S_{e_j} : \Gamma_j$. Also $\Sigma'(\tau_j) = \Sigma(\tau_j)$ for all $j \in 1..m$ as Σ' is an extension of Σ, so we have

$$\Gamma_j, x_j : \Sigma'_s(\tau_j) \vdash b_j : \Sigma'_b(\tau_j),\ \forall j \in 1..m \qquad (2)$$

Now using equations (1) and (2) above, and using the rule $(Method\ Store\ Typing)$ we have the following conclusion for $\Sigma' \supseteq \Sigma$:

$$\Sigma' \vDash \sigma_m, \iota_i \mapsto \langle \varsigma(x_i)b_i, S_e\rangle^{i\in 1..p},\ \iota_i \mapsto \langle \varsigma(x_i)b_i^d, S_e\rangle^{i\in p+1..q}$$

$$\frac{\text{Proof of } \Sigma' \vDash [l_i = \iota_i^{\ i\in 1..p},\ d_i : p = \iota_i^{\ i\in p+1..q}] : C}{}$$

Since $\Sigma' \equiv \Sigma, \iota_i \mapsto (C \rightarrow D_i)^{\ i\in 1..q}$ and $\Sigma' \vDash \diamond$, and using the rule $(Result\ Type - Aspect)$, we have the following conclusion for $\Sigma' \supseteq \Sigma$:

$$\Sigma' \vDash [l_i = \iota_i^{\ i\in 1..p},\ d_i : p = \iota_i^{\ i\in p+1..q}] : C$$

Case Invocation Term ($a.l$):
There are two reduction rules for the invocation term of the calculus as there are two possibilities while invocation. Here we discuss the proof for one such rule as the other one will have very similar proof. The reduction rule $(Red\ Inv)$ when no advice present is given below.

$$\frac{(Red\ Inv) \qquad (No\ Advice\ present)}{\begin{array}{c}(\sigma_m,\sigma_a).(S_e,S_d) \vdash a \rightsquigarrow v.(\sigma'_m,\sigma'_a)\ v_1 < v_2 .. < v_n \in \sigma'_a \\ match(v, l_j, v_1, \phi, inv) = \phi \quad \sigma'_m(v.l_j) = \langle \varsigma(x_j) b_j, S'_e \rangle \\ (\sigma'_m\ ,\ \sigma'_a).(\ (S'_e, x_j \mapsto v),\ S_d) \vdash b_j \rightsquigarrow v\,'.(\sigma''_m\ ,\ \sigma''_a)\end{array}}{(\sigma_m\ ,\ \sigma_a).(S_e,S_d) \vdash a.l_j \rightsquigarrow v\,'.(\sigma''_m\ ,\ \sigma''_a)}$$

By hypothesis

$\Gamma \vdash a.l_j : B_j$
$(\sigma_m\ ,\ \sigma_a).(S_e,S_d) \vdash a.l_j \rightsquigarrow v\,'.(\sigma''_m\ ,\ \sigma''_a)$
$\Sigma \vDash \sigma_m$
$\Sigma \vDash S_e : \Gamma$

we need to prove for some $\Sigma'' \supseteq \Sigma$,

$\Sigma'' \vDash \sigma''_m$
$\Sigma'' \vDash v\,' : B_j$

Since $\Gamma \vdash a.l_j : B_j$ we must have $\Gamma \vdash a : [l_j : B_j, ...]$ by the typing rule $(Val\ Inv)$, and let $A \equiv [l_j : B_j, ...]$. Now by induction hypothesis for the reduction of term a:

$\Gamma \vdash a : A$
$(\sigma_m\ ,\ \sigma_a).(S_e,S_d) \vdash a \rightsquigarrow v.(\sigma'_m\ ,\ \sigma'_a)$
$\Sigma \vDash \sigma_m$
$\Sigma \vDash S_e : \Gamma$

there exists some store type $\Sigma' \supseteq \Sigma$,

$\Sigma' \vDash \sigma'_m$
$\Sigma' \vDash v : A$

Let $v = [l_i = \iota_i{}^{i \in 1..n}]$ and we have $\sigma'_m(v.l_j) = \langle \varsigma(x_j) b_j, S'_e \rangle$. Now here $\Sigma' \vDash \sigma'_m$ have come through rule $(Method\ Store\ Typing)$ using $\Sigma' \vDash S'_e : \Gamma_j$ and $\Gamma_j, x_j : \Sigma'_s(\iota_j) \vdash b_j : \Sigma'_b(\iota_j)$. And $\Sigma' \vDash [l_i = \iota_i{}^{i \in 1..n}] : A$ (as $v \equiv [l_i = \iota_i{}^{i \in 1..n}]$) have come through $(Result\ Type - Object)$, using $A \equiv \Sigma'_s(\iota_j) \equiv [l_i : \Sigma'_b(\iota_j)\ {}^{i \in 1..n}]$. Since $A \equiv [l_j : B_j, ...]$, so we have $\Sigma'_b(\iota_j) \equiv B_j$. Using this, now we have $\Gamma_j, x_j : A \vdash b_j : B_j$.

Also using $(Env\ Stack\ S_e\ Typing)$, we have $\Sigma' \vDash S'_e, x_j \mapsto [l_i = \iota_i{}^{i \in 1..n}] : \Gamma_j, x_j : A$.

Let $\Gamma' \equiv \Gamma_j, x_j : A$ and again using induction hypothesis but this time for the reduction of method body term b_j, for the following assumptions

$\Gamma' \vdash b_j : B_j$
$(\sigma'_m\ ,\ \sigma'_a).(\ (S'_e, x_j \mapsto v),\ S_d) \vdash b_j \rightsquigarrow v\,'.(\sigma''_m\ ,\ \sigma''_a)$
$\Sigma' \vDash \sigma'_m$
$\Sigma' \vDash S'_e : \Gamma'$

we have the following conclusion for some store type $\Sigma'' \supseteq \Sigma'$,

$\Sigma'' \vDash \sigma''_m$
$\Sigma'' \vDash v\,' : B_j$

Finally we have the conclusion $\Sigma'' \vDash \sigma''_m$ and $\Sigma'' \vDash v\,' : B_j$ for the store type $\Sigma'' \supseteq \Sigma$ (as $\Sigma' \supseteq \Sigma$ and $\Sigma'' \supseteq \Sigma'$).

■

IV. Conclusion

The paradigm of aspect-oriented programming languages is quite new but is of significant value to research in the field of programming languages. With the advent of aspect-oriented languages like AspectJ and AspectC++, the commercial adoption of this new paradigm is also expected. The formalization for this new class of programming languages is still under development. The type system for the aspect-oriented programming languages in the form of simple-typed aspect calculus is presented here. The proposed typed calculus is very useful in studying various properties of current aspect-oriented languages. The type safety or type soundness is an important property of any type system. The proof of type soundness for the aspect calculi is also devised using the method of induction and the same is discussed in the paper.

References

[1] M. Abadi and L. Cardelli, *A Theory of Objects*. New York: Springer-Verlag, 1996.
[2] A. Church, "An Unsolvable Problem of Elementary Number Theory," *American Journal of Methematics*, vol. 58, no. 2, pp. 345–363, April 1936.
[3] ——, "A Formulation of the Simple Theory of Types," *The Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, June 1940.
[4] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar, "Aspect-Oriented Programming," *Proceedings of the European Conference on Object-oriented programming (ECOOP), Springer-Verlag*, 1997.
[5] T. Elard, R. Filman, and A. Badar, "Aspect-Oriented Programming : Introduction," *Communication of the ACM*, October 2001.
[6] C. Clifton, G. T. Leavens, and M. Wand, "Formal Definition of the Parameterized Aspect Calculus," *Technical Report 03-12b, Iowa State University*, Nov. 2003.
[7] R. Jagadeesan, A. Jeffrey, and J. Riely, "A Calculus of Untyped Aspect-Oriented Programs," *In European Conference on Object-Oriented Programming, Darmstadt, Germany*, July 2003.
[8] D. Walker, S. Zdancewic, and J. Ligatti, "A Theory of Aspects," *In Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, Aug. 2003.
[9] D. Gopalani and M. C. Govil, "Untyped Aspect Calculus : Formal Theory of Aspect-Oriented Programming Languages," *IEEE 2nd International Advance Computing Conference*, pp. 195–200, February 2010.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)