Type Theory and Language Constructs for Objects with States

H. Xu and S. Yu

Department of Computer Science
The University of Western Ontario
London, Ontario, Canada

Abstract

In current class-based Object-Oriented Programming Languages (OOPLs), object types include only static features. How to add object dynamic behaviors modeled by Harel’s statecharts into object types is a challenging task. We propose adding states and state transitions, which are largely unstated in object type theory, into object type definitions and typing rules. We argue that dynamic behaviors of objects should be part of object type definitions. We propose our type theory, the $\tau$-calculus, which refines Abadi and Cardelli’s $\varsigma$-calculus, in modeling objects with their dynamic behaviors. In our proposed type theory, we also explain that a subtyping relation between object types should imply the inclusion of their dynamic behaviors. By adding states and state transitions into object types, we propose modifying programming language constructs for state tracking.

Keywords: object types, object dynamic behaviors, $\varsigma$-calculus, $\tau$-calculus, programming language constructs, state tracking.

1 Introduction

Type theories for OOPLs have been proposed by many authors. In A Theory of Objects [1], Abadi and Cardelli developed their object calculi ($\varsigma$-calculus) which were stated as a method loosely modeling object-based languages. The type abstraction of object in $\varsigma$-calculus is conceptually simple and basically reflects the objects in current OOPLs. However, it lacks necessary expressiveness in certain situations. For example, we may have a class Person in

---

1 Email: hxu@csd.uwo.ca
2 Email: syu@csd.uwo.ca

1571-0661 © 2006 Elsevier B.V. Open access under CC BY-NC-ND license.
which the method void getDivorced (void) is included. An instance (object) of Person may be in one of the following four states (status): Single, Married, Divorced, and Widowed, and the method getDivorced should be called legally only when the object is in the state Married. It would be wrong for getDivorced to be called when the object is in another state. However, neither ζ-calculus nor any other current type system can handle such basic problems. In current type systems, any method of an object can be legally called at any time as long as the object still exists. The state or status of an object is not a consideration of the type systems. Corresponding to the real world, the set of methods of an object that can be called at a certain time may not include all the methods of the object. It depends on what state the object is in, which again depends on the state transitions that define the dynamic behaviors of the object. We argue that these characteristics of object dynamics should be reflected in object type theory. The problems related to states and state transitions of an object may surely be solved in other ways, but clearly they are part of the type system and better to be solved in the type system.

Object dynamic behaviors are already a phase of object-oriented modeling. Statecharts were introduced by David Harel in 1987 [5] and then incorporated into object-oriented modeling methods and languages such as OMT [10] and UML [2] to describe the dynamic behaviors of objects. Finite state machines (FSMs), in a form directly mapped from statecharts, have become a standard model for representing object behaviors.

Quite a number of other models also concerned states and state machines in types or object types. However, they were in different context or for different purposes. In a relatively early paper [14], states (not object states) were introduced in programming languages for enhancing software reliability. In [4], the authors proposed the language feature of re-classification to change an object’s behavior dynamically. However, object dynamic behavior is irrelevant to Harel’s statecharts. So the concept of “dynamic behavior” defined in their paper is fundamentally different from ours. In [3], a programming model of typestates for objects was developed. However, their type definition contains all of an object’s fields and, thus, their object type is an implementation-dependent entity. In [8], the authors proposed that the behavior of the objects of a subtype should also satisfy the behavior of the objects of its supertype. But the properties described in the paper were not directly connected to states and state transitions. In [9], the author proposed to integrate state machines and OOPLs, in which a state of an object is represented as a set of virtual bindings rather than being a clearly defined entity. In [11], the authors proposed typing objects with states, but focused on formalizing non-uniform concurrent objects. Several papers presented type-based general
methods (not methods in a class) for resource usage analysis [7] or resource usage analysis via scoped methods [15]. There are also papers aiming to specify state machines in OOPLs [12] or to initialize some kind of state-oriented programming in implementing hierarchical state machines [13]. However, all these ideas are very different from introducing states and state transitions into object types.

We propose our \(\tau\)-calculus for the typed system which comprises formal system fragments. The most fundamental formal system fragments are the object typing and subtyping rules with formally defined states and state transitions. Our \(\tau\)-calculus is viewed as an improvement of the \(\varsigma\)-calculus. States and state transitions are being introduced as an essential part of a class. That is, each class has its own states and state transition functions. We also introduce programming language constructs for implementing states and state transitions. The syntax developed for class is easy to understand and suitable for most of OOPLs. The OOPL type checking system can then include state tracking algorithm and provide a higher degree of program correctness to report object dynamic behavior errors in a program.

The idea of introducing states and state transitions into object types was motivated by David Harel’s statecharts. However, they have become different entities. Statecharts have been used for modeling the behaviors of objects normally in the modeling stage and before the programming is done, but objects with states are defined in the programs which are in the implementation stage. More importantly, a statechart models the whole status of an object, but the states of an object defined by a programmer may reflect only a small part of the total behaviors of the object, when states are relevant.

2 Object Types and States

Each object is a dynamic entity. During the life time of an object, the object may change into different states and it may have different behaviors when it is in different states. As we have described in the introduction, the set of methods of an object that can be legally called may be different when the object is in a different state. After a method is called, the object may be transformed into another state. The state transitions of an object (a class) can usually be described by a state diagram.

The state diagram of an object or a system is essentially a Deterministic Finite Automaton (DFA) [6] which can be described as \(M = (Q, \Sigma, s_0, \delta)\), where \(Q\) is the set of states; \(\Sigma\) is the set of methods; \(s_0\) is the starting state - the state when an object is created; \(\delta\) is the set of transitions defined by the function \(\delta(p, a) = q\) for \(p, q \in Q\) and \(a \in \Sigma\).
Fig. 1. (a) State diagram of Dryer and (b) its abstract DFA model

Assume that the state diagram of Dryer and its abstract DFA model are provided in Figure 1, where \( Q = \{0, 1, 2, 3\} \) and 0,1,2,3 represent the named states of OffSlow, OffFast, SlowHeating, and FastHeating, respectively; \( \Sigma = \{l_1 = \text{chg2Fast}, l_2 = \text{chg2Slow}, l_3 = \text{turnOff}, l_4 = \text{turnOn}\} \) and the labels \( l_1, l_2, l_3, l_4 \) are the method names of the object Dryer in lexicographical order; the transition function set \( \delta \) is defined as: \( \delta(0,l_4) = 2, \delta(0,l_1) = 1, \delta(1,l_4) = 3, \delta(1,l_2) = 0, \delta(2,l_3) = 0, \delta(2,l_1) = 3, \delta(3,l_3) = 1, \) and \( \delta(3,l_2) = 2. \)

For an object Dryer in a state \( q \in \{0, 1, 2, 3\} \), if a message \( e \in \Sigma \) is received (a method \( e \) is called or invoked) and \( \delta(q,e) = p \) is well defined for some \( p \in Q \), we consider that the transition to \( p \) is a valid transition. Otherwise, \( \delta(q,e) \), e.g., \( \delta(2,l_4) \), is not defined - a Dryer can not be turned on while it is in state SlowHeating. Then the transition set denoted by \( S_i = \{(q,p) | p,q \in Q \land \delta(q,l_i) = p\} \) contains all the proper state changes for a Dryer in receiving a message labeled by \( l_i \) during its execution. The attempt of a method call for Dryer in state \( q \in Q \) may not be successful when a corresponding state transition is not defined. Then the state changes of a Dryer, resulted from a sequence of method invocations, should be in coherence with the state transitions defined in object dynamics. Otherwise, type errors should be recognized by the type system and reported.

We observe that an object type is irrelevant to the states and state transitions in the object according to the rules in \( \varsigma \)-calculus. As a result, an algorithm, which is based on the rules for weak reduction in \( \varsigma \)-calculus and constitutes an interpreter for \( \varsigma \)-terms, cannot exclude type errors of such object misbehavior. Assume that object \( a \) has the type \([l_i : B_i]_{i \in 1..n}\) and the algorithm used for method invocation check in \( \varsigma \)-calculus is defined recursively as:

\[
\text{Outcome}(a,l_j) \triangleq \\
\begin{align*}
& \text{let } o = \text{Outcome}(a) & \cdots \\
& \text{in } \text{if } o \text{ has form } [l_i = \varsigma(x_i)b_i\{x_i\}_{i \in 1..n}] \text{ and } j \in 1..n \\
& \text{then } \text{Outcome}(b_j\{o\}) \\
& \text{else } \text{wrong}
\end{align*}
\]
But in our proposed type theory, a method invocation $a.l_j$ reduces to the result of the substitution of the host object for the self (or this) parameter in the body of the method named $l_j$ only when a state transition for the host object in the current state exists. In another word, the reduction of $a.l_j \rightarrow b_j\{x_j \leftarrow o\}$ for $l_j \ (j \in 1..n)$ is subject to the condition that there exists a valid state change defined in state transition functions for the object in the current state. Afterwards, the object may transform into another one. On line(1), the statement let $a=\text{Outcome}(a)$ implies that there exists a valid state transition for object $a$ to be transformed into $o$ while the method $l_j$ is called. The structure abstraction and the primitive semantics of object are well established in $\zeta$-calculus. We then consider to modify the object type by adding essential object dynamics modeling into its fundamental framework. As the result, the state machines composed of states and state transitions become essential entities in object type construction. Note that the states used in this paper are clearly different from the concept of typestates [3,14]. Object types are different from object classes. Object types are abstract entities which can be constructed from class definitions but leave out the implementation details.

3 $\tau$-Calculus for Object Types and Class Types

We start with a type system which is composed of several formal system fragments for object types and subtypes. Some fundamental properties of a type system, such as the reduction theorem, are supported by both our type theory and Abadi and Cadelli’s type theory.

First, we list the syntax fragment which is in fact implicit in the rules of $\Delta_{Ob}$ and for the later fragments.

<table>
<thead>
<tr>
<th>Syntax Fragment for $\Delta_{Ob}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B ::= \text{types}$</td>
</tr>
<tr>
<td>$[Q, (l_i : B_i) :: S_i^i \in 1..n]$ object type ($l_i$ distinct, $Q$ represents state set, $S_i$ is the set of state transitions for method $l_i$)</td>
</tr>
<tr>
<td>$a, b ::= \text{terms}$</td>
</tr>
<tr>
<td>$[l_i = \tau(x_i : A_i)b_i^i \in 1..n]$ object ($l_i$ distinct)</td>
</tr>
<tr>
<td>$a.l$ method call</td>
</tr>
</tbody>
</table>

Similar to what in $\zeta$-calculus, fields are just a special kind of methods and, thus, represented uniformly as methods. The method update feature in OOPLs allows an object dynamically change its behavior in execution. This makes the type of an object hard to trace in a type system. We do not put this
feature into discussion in this paper because it has very limited usage in class-based OOPLs which form the mainstream of object-oriented programming.

**Definition 3.1** The states and state transitions that are associated with an object are formally described as a triple \( A = (Q, \Sigma, S) \):

(i) \( Q \) is the set of states. Each node in the state diagram is a state \( q \in Q \).
(ii) \( \Sigma \) is the set of methods, i.e., \( \Sigma = \{ l_i \mid i = 1 \ldots n \} \), where \( n \) is the total number of methods.
(iii) \( S = \bigcup_{i=1}^{n} S_i \), where \( S_i = \{(p, q) \mid p, q \in Q \land \delta(p, l_i) = q \} \) and \( \delta(p, l_i) = q \) is a state transition function from \( p \) by \( l_i \) to \( q \).

In contrast to what appeared in \( \zeta \)-calculus, the enforcement of association between a method and its transition set in object types of \( \tau \)-calculus is important, but sometimes in a hidden form and underestimated. In some cases, we can model an object with only one state \( p \) (\( Q = \{p\} \)) and all the transitions cause no state change (\( S_i = \{(p, p)\} \) for \( i=1\ldots n \)). In these cases, the states of the objects can be ignored.

Four kinds of judgements are used in fragment \( \Delta_{Ob} \) in \( \tau \)-calculus: (1) state set and state transition set judgment \( E \vdash Q, S \) for \( Q \sim S \), stating that \( Q \) is a well-formed state set and \( S \) is a state transition set satisfying \( S \subseteq \{(p, q) \mid p, q \in Q \} \), is well defined in the typing environment \( E \), (2) a type judgement \( E \vdash B \), stating that \( B \) is a well-formed type in \( E \), (3) a value type judgement \( E \vdash (b : B) :: S \), stating that \( b \) has type \( B \) which is bound by state transition set \( S \) in \( E \), and (4) state activation correctness judgement \( E \vdash a@q \in Q \) and \( \exists(q, p) \in S \), stating that an object \( a \) is in the state \( q \) and there exists a state transition \((q, p)\) for \( a \) is a well-formed environment in \( E \).

---

**\( \Delta_{Ob} \) (\( \tau \)-calculus):**

- **Type Object**
  \[
  \frac{E \vdash Q, B_i \land E \vdash Q \land S_i \forall i \in 1..n}{E \vdash [Q, (l_i:B_i)::S_i^i \in 1..n]} \]
  where \( Q \land S_i \) means \( S_i \subseteq \{(p, q) \mid p, q \in Q \} \)

- **Value Object**
  \[
  \frac{E \vdash a:A \land E \vdash (l_i:B_i)::S_i \forall i \in 1..n}{E \vdash [l_i = \tau(a:A) b_i^i \in 1..n] : A} \]
  where \( A = \{Q, (l_i : B_i) :: S_i^{i \in 1..n}\} \)

- **Value Select**
  \[
  \frac{E \vdash a: \{Q, (l_i:B_i) :: S_i^{i \in 1..n}\} \land E \vdash a@q \in Q \land \exists(q, p) \in S_j \land j \in 1..n}{E \vdash (a.l_j) \land (a@q) = a@p} \]
  where the notation \( a@q \uparrow = a@p \) means updating state \( q \) to state \( p \) for object \( a \).

Rule **Type Object** states that the object type \( \{Q, (l_i : B_i) :: S_i^{i \in 1..n}\} \) is well-formed in \( E \), provided that there exist a well-formed state set \( Q \) and a set of
state transition sets $S_i^{i \in 1..n}$ satisfying $Q \curvearrowright S_i$ ($S_i \subset \{(p, q) \mid p, q \in Q\}$) in $E$. We always assume that, when writing $[Q, (l_i : B_i) :: S_i^{i \in 1..n}]$, that the labels $l_i$ must be distinct. We identify object types $[Q, (l_i : B_i) :: S_i^{i \in 1..n}]$ by sorting components $(l_i : B_i) :: S_i$ in certain order, e.g., lexicographical order of $l_i$.

Rule Val Object states that an object type $[Q, (l_i : B_i) :: S_i^{i \in 1..n}]$ can be formed from a collection of $n$ methods whose self parameter (e.g. this pointer referring to host object in C++ and Java) has type $[Q, (l_i : B_i) :: S_i^{i \in 1..n}]$ and whose bodies have type $B_1 :: S_1, ..., B_n :: S_n$. In detail, $B_i$ is the result type produced by the method body $b_i$ which is also bound by the state transition set $S_i$ for $i \in 1..n$. Note that this pointer is embedded in every method body and the circularity is used by the self parameter.

Rule Val Select describes how to enforce type correctness to a method invocation $a.l_j$. If there is an well-formed object type $[Q, (l_i : B_i) :: S_i^{i \in 1..n}]$ and method $l_j$ indexed by $j$, $1 \leq j \leq n$, can be correctly invoked only when there exists a valid transition $(q, p) \in S_j$ for the object $a$ in current state $q$ $(a@q)$. The state in object $a$ is updated to state $p$ afterwards.

Correspondingly, we can represent class types for an object type based on our $\tau$-calculus. Let $A \equiv [Q, (l_i : B_i) :: S_i^{i \in 1..n}]$ be an object type, then $Class(A) \triangleq [\text{new} : A, \ l_i : A \rightarrow (B_i :: S_i)^{i \in 1..n}]$ is a class that can generate objects of type $A$. These classes have the form: $[\text{new} = \tau(z:Class(A)) \mid q = \tau(P:Q, qo: P)z@qo, l_i = \tau(x:A, qP, s_i;S_i)z. l_i(x):s_i(q)^{i \in 1..n}], l_i \vdash S_i = \lambda(x:A).\lambda(q:P)(s_i(q))(b_i)^{i \in 1..n}]$

An object type is an implementation independent entity in contrast to an object class which is an implementation dependent entity [16]. Therefore, $P : Q$ stands for that a particular choice of the state set $P$ in class implementation is an instance of the formally described state set $Q$ in the triple, e.g., the state set $P = \{\text{HasJob}, \text{NoJob}\}$ may be chosen to implement the class Person whose type contains the state set $Q = \{0_{Employed}, 1_{Unemployed}\}$. So there are the mappings of states from HasJob to 0Employed and from NoJob to 1Unemployed for Person. Similarly, $s_i : S_i$ stands for that the sets of state transition functions $s_i^{i = 1..n}$ associated with the state set $P$ are the instances of those $S_i^{i = 1..n}$ associated with the state set $Q$. Note that $q_0$ is the starting state when an object is created. The state transition functions, denoted by $\lambda(q : P)(s_i(q))^{i \in 1..n}$, are a part of methods defined in a class. As a result, a method invocation will first check the state correctness and then do the rest computation. Similar to what in $\varsigma$-calculus, an ad hoc inheritance relation on class types “Class($A'$) may inherit from Class($A$) iff $A' <: A$” is set to follow the principle of method reuse for objects with states.
4 Inheritance and Subtyping

Although an object class is not an object type, a class is often taken as a type-defining construct in OOPLs. An insidious problem with inheritance in OOPLs is how to distinguish subtyping relation between object types, which indicates the inclusion of behaviors, from other purposes such as code reuse for classes. Objects of the same class are of the same type, but object of the same type may not belong to the same class. Inheritance may indicate a subtyping relation or code reuse or both. If subtyping relation is sound between $A$ and $B$ ($A <: B$), an object $o_A$ of subclass $A$ can emulate the behaviors of any object $o_B$ of superclass $B$. Let $p \overset{l_i}{\rightarrow} q$ be a state transition from state $p$ to state $q$ by the method call of $l_i$. Assume an arbitrary valid computation in terms of a sequence of state changes for $o_B$ is represented by $(q_0) \overset{l_i\ldots l_k}{\rightarrow} (q_m) \equiv q_0 \overset{l_1}{\rightarrow} q_1 \overset{l_2}{\rightarrow} \ldots \overset{l_k}{\rightarrow} q_m$. This property $\varphi(B)$ of $o_B$ must be properly inherited by $o_A$ in a form of behavioral inclusion polymorphism. As a result, there should be no type errors for $o_A$ to simulate the computation: $(q_0) \overset{l_i\ldots l_k}{\rightarrow} (q_m)$. To enforce that the objects of a subtype are able to simulate those of its supertype for the same sequence of method invocations, the subtyping relation between two objects indicates a relation between two state machines, denoted by $M_A \preceq M_B$ where $M_A = (Q_A, \Sigma_A, S_A)$ and $M_B = (Q_B, \Sigma_B, S_B)$ satisfying $Q_A \supseteq Q_B, \Sigma_A \supseteq \Sigma_B$, and for each method $l_i \in \Sigma_B, S_{A_i} \supseteq S_{B_i} (i = 1 \ldots |\Sigma_B|)$.

- **Sub Object 1 ($\tau$) ($l_i$ distinct):**

  \[
  E \vdash (Q \subseteq \hat{Q}) \quad E \vdash B_i \land (\hat{Q} \cap \hat{S}_i) \quad E \vdash (Q \cap \Sigma_j) \land (S_j \subseteq \hat{S}_j) \quad \forall i \in 1..n+m \ \forall j \in 1..n
  \]

  \[
  E \vdash [Q, (l_i:B_i)::S_i^{i\in 1..n+m}] <: [Q, (l_j:B_j)::S_j^{j\in 1..n}]
  \]

Rule **Sub Object 1 ($\tau$)** states a general subtyping relation between object types supporting the behavioral inclusion polymorphism with single inheritance. Let $[\hat{Q}, (l_i:B_i)::\hat{S}_i^{i\in 1..n+m}]$ and $[Q, (l_j:B_j)::S_j^{j\in 1..n}]$ be object types for $\hat{o}$ and $o$ respectively. If there exist $Q \subseteq \hat{Q}$, and $\hat{Q} \cap \hat{S}_i$ for each method $l_i$ indexed by $i$ ($i = 1..n+m$), and $(Q \cap \Sigma_j) \land (S_j \subseteq \hat{S}_j)$ for each method $l_j$ indexed by $j$ ($j = 1..n$), then the object type of $\hat{o}$ is a subtype of type of $o$.

5 Program Language Constructs and State Tracking

We first provide an example of our new programming language constructs for **states** and **state changes** in a **class** definition for class **Dryer** (see Figure 2(a)). We also use the current class structure (see Figure 2(b)) to implement states and state transitions for the purpose of comparison. At the end of a constructor heading in the new construct, the initial state is specified. On line 1 of Figure 2, it is an explicit declaration of state set for the class. The syntax
used in our class \textit{Dryer} definition is C++ syntax except the added syntax for states and state transitions.

```cpp
Class Dryer {
    /* (a) Our proposed class construct */
1. state: {OffSlow, OffFast, SlowHeating, FastHeating};
2. public:
3.  Dryer(int voltage) : ~Dryer() {} {−OffSlow } \_voltage(voltage) {}  
4.  void turnOn() : {OffSlow->SlowHeating, OffFast->FastHeating} {...}
5.  void turnOff() : {SlowHeating->OffSlow, FastHeating->OffFast} {...}
6.  void chg2Fast() : {OffSlow->OffFast, SlowHeating->FastHeating} {...}
7.  void chg2Slow() : {OffFast->OffSlow, FastHeating->SlowHeating} {...}
8.  .../* Other methods if necessary */
9. private:
10. int \_voltage;
11.} /* End of new class construct */
Class Dryer {
    /* (b) Traditional C++ class construct implementing states */
12. enum state {OffSlow, OffFast, SlowHeating, FastHeating};
13. public:
14.  Dryer(int voltage) : \_voltage(voltage) {}  
15.  ~Dryer() {}  
16.  void turnOn() { switch (\_mystate) { case OffSlow: \_mystate=SlowHeating;  
17.  break; case OffFast: \_mystate=FastHeating; break; default: } ...}  
18.  void turnOff() { switch (\_mystate) { case SlowHeating: \_mystate=OffSlow;  
19.  break; case FastHeating: \_mystate=OffFast; break; default: } ...}  
20.  void chg2Fast() { switch (\_mystate) { case OffSlow: \_mystate=OffFast;  
21.  break; case SlowHeating: \_mystate=FastHeating; break; default: } ...}  
22.  void chg2Slow() { switch (\_mystate) { case OffFast: \_mystate=SlowFast;  
23.  break; case FastHeating: \_mystate=SlowHeating; break; default: } ...}  
24. private:
25.  state \_mystate; int \_voltage; } /* End of C++ class construct */
```

Fig. 2. (a) Our proposed class construct for \textit{Dryer} and (b) the traditional C++ class construct for \textit{Dryer} with states

One may argue that the above class definitions can be implemented as what is shown in Figure 2(b), so programmers must implement the states and state transitions as part inside method code. The advantage of using our proposed class construct is obvious. First, it is in a much simpler form. The states and state transitions associated with each method reflect a more intuitive mapping from statechart. Second, inheritance (subtyping) clearly indicates the dynamic behavioral inclusion, e.g., if “\textit{class SmartDryer : public Dryer}” is declared, then the state set and the methods together with their associated state transitions in class \textit{Dryer} are inherited to class \textit{SmartDryer}. Moreover, we can add new state transitions for the same state set in the subclass. Third, the new constructs for states and state transitions are easy to implement at the compilation stage.

Based on our proposed object type theory and programming language constructs, a preprocessor can be implemented to report type errors of object misbehavior. If we do not consider parallelism at this stage, the methods of objects are called in sequence. We can repeat the following steps:
(i) For each class, build a deterministic finite automaton (DFA) for the state transition function, e.g., in the form of a table.

(ii) For every object created, initialize its state to the initial state. After a method is called, check if the state transition is defined. If so, change its state; otherwise, report the state transition error.

The type checking algorithm actually has two phases. In the first phase, it checks inconsistency of state transitions at the compilation time. In the second phase, it detects inconsistency in state transitions (guarded transitions) with efficiency during the run time.

6 Conclusion

We propose that object dynamic behavior, in respect to the general existence of states and state changes in objects, should be as an essential component in object type theory and subtyping modeling. This is considered as an important improvement to the object typing framework that appeared in $\lambda$-calculus by Abadi and Cardelli. The object type with states is more precise, and can ensure a higher level correctness of type systems built for objects. As a result, type errors caused by method calls at improper state can be reported. We believe that this approach in building our object type system is generally applicable in OOPLs. More work, such as the implementation of a preprocessor, should be continued in the near future.

References


[16] Yu, S., Class-is-type is inadequate for object reuse, ACM Sigplan Notice, 36, No. 6(2001), 50-59.