SCHOOL: a Small Chorded Object-Oriented Language

S. Drossopoulou, A. Petrounias, A. Buckley, S. Eisenbach

{ *s.drossopoulou, a.petrounias, a.buckley, s.eisenbach* } @ *imperial.ac.uk*
Department of Computing, Imperial College London, United Kingdom

Abstract

Chords are a declarative synchronisation construct based on the Join-calculus, available in the programming language C-omega. To our knowledge, chords have no formal model in an object-oriented setting.In this paper we suggest SCHOOL, a formal model for an imperative, object-oriented language with chords. We give an operational semantics and type system, and can prove soundness of the type system.

1 Introduction

A *chorded program* [1] consists of class definitions, each class defining one or more chords. A chord has a signature and a body. A chord's signature is an aggregate that comprises at most one *synchronous* method and zero or more *asynchronous* methods.

A chord body is executed when an object has received at least one message for *each* of the chord's synchronous and asynchronous method signatures. Potentially multiple method calls are needed to invoke a chord's body. This reflects the notion of *join* from the *Join*-Calculus [3], where the *join-pattern* consists of the methods comprising the chord signature.

For instance, the following chord, an unbounded buffer:

int get() & async put(int x) { *return x;* }

will execute the body and return x only when there is a *simultaneous* presence of invocations to both of the methods in its signature.

The method *get* is synchronous, and hence will block its caller until there is a message present for method *put* and hence it can join. The latter method is asynchronous, a subtype of *void*, and returns immediately to its caller; thus messages sent to it must be queued by the receiving object until consumed by the joining of the chord.

Those chords whose signatures contain a *synchronous* method are called *synchronous* chords. Chords with only *asynchronous* methods in their signature are called *asynchronous* chords.

2 Semantics

We present SCHOOL (see overview in figure 1) in the form of structural operational semantics (found in figure 2) and an accompanying type system (in figure 3). An extended version of this paper with additional material, a more thorrough coverage of chords in general, and hand-written proofs of soundness can be found from the following website:`slurp.doc.ac.uk/school`.

Expressions and Programs

The syntax of SCHOOL expressions is: method call, sequence of expressions, the receiver ($this$), a parameter (x), and the values $null$ (for the null pointer) and $voidVal$ (for the result of an execution that returns $void$ or for the result of a call to an *asynchronous* method).

We also define SCHOOL programs, which are tuples of mappings. We do not give a syntax for programs, and therefore can omit rather mechanical definitions of derived functions which lookup methods and superclasses.

A program consists of 1) a mapping from a class and method name to the method's signature in that class, 2) a mapping from a class and method name to all chords in which the method name is the *synchronous* part, 3) a mapping from a class name to the set of the class's *asynchronous* chords, and 4) a mapping from a class name to the name of its superclass.

A method signature contains a return type, a method name and a parameter type. The name of the formal parameter is derived from the name of the method: for a method called `mth`, the parameter will be called `mth_x`. These restrictions are, of course, inconvenient for programming but are not essential to our study of chords and types, and they allow a considerably more succinct presentation.

We represent a chord as a set of asynchronous method names along with the expression representing the chord's body. Thus, the distinction between a *synchronous chord* and an *asynchronous chord* is whether the chord appears in the image of the second or the third component of a SCHOOL program. A method name can appear in any number of chords.

For ease of notation we also define the following four lookup functions: the function $\mathcal{M}\ (\ P,\ c,\ m\ )$ is the projection of the first component of P, and returns m's signature in class c; the function $\mathcal{SC}hs\ (\ P,\ c,\ m\ )$ is the projection of the second component of P, and finds the *synchronous* chords to which m belongs, returning their *asynchronous* method names plus their bodies; the function $\mathcal{AC}hs\ (\ P,\ c\ )$ is the projection of the third component of P, and

returns the set of *asynchronous* chords for class c; finally, $\mathcal{M}^a\ (\ P,\ c\ )$ gives all *asynchronous* method names present in class c's chord definitions.

Objects, Messages and the Heap

One can view chord invocation as message-passing between objects. A caller object sends a message comprising of a name and an argument to a receiver object. A call to an asynchronous method returns immediately, but the corresponding chord body may not yet be ready to run. Therefore, messages that target asynchronous methods are queued within the receiver object.

Consequently an object comprises 1) the name of its defining class and 2) one queue for each *asynchronous* method signature in its class. Thus, the state of an object is represented by its queues.

Queues are modelled as mappings from method identifiers to *multisets of values* representing the actual parameter passed when the asynchronous method was called. The use of multisets allows a natural presentation of the non-deterministic nature of handling asynchronous methods call, whereby asynchronous calls are not guaranteed to be handled in the order they were made, even if they were made consecutively from the same thread [1]. We need to have *multisets* rather than sets in order to model the situation where an asynchronous method was called twice with the same parameter.

An interesting observation is that any object that can access another object can write to its queues by calling asynchronous methods. However, only the chord body associated with an asynchronous method signature can read from the method's queue. Reading from a queue consumes one of its elements.

The heap maps addresses (in $\mathbb{N}$) to objects. Once an object is allocated at an address, there is no way to remove it. Thus, in terms of address-to-value mappings, the heap grows monotonically. However, the queues within each object grow and shrink as messages are sent to and consumed from queues, as described earlier.

Operational Semantics

SCHOOL operational semantics, found in figure 2, are in the same style as [2]. The aim of the rules is to abstract as much away from scheduling as possible. Hence, we welcome non-determinism whenever there is choice, thus maximizing the possible behaviours of programs. There are three rules of particular interest: Async, Join, and Strung. These three rules capture the essence of chord invocation in SCHOOL.

Async describes invocation of an *asynchronous* method: the value representing void is immediately returned, and the actual argument is placed in the appropriate queue of the receiving object.

Join describes invocation of a *synchronous* method. The caller will block until all the *asynchronous* methods present in the chord containing the in-

voked method have at least one message each in their respective queues at the receiving object. Notice that the choice of chord is non-deterministic, as is the choice of participating queue elements. Once the chord joins, the messages are consumed from the queues and the current expression becomes the body of the chord.

STRUNG describes execution of *asynchronous* chords. As there is no caller waiting for the chord to join, it is the responsibility of the (abstract) scheduler to decide which chord to choose. Essentially, this rule exhibits non-deterministic choice at three levels: the selection of object in the heap, the selection of *asynchronous* chord, and the selection of elements from the participating queues. The body of the chord will execute in a new thread (we call this *spawning*).

Type Judgements

The judgment $\mathrm{P},\ \Gamma \vdash e : t$ describes the static type of a source level expression e, while the judgment $\mathrm{P},\ h \vdash e : t$ describes the dynamic type of a runtime expression e. The complete SCHOOL type system can be found in figure 3.

Well-Formed Programs

A well-formed SCHOOL source program (WF-PRGM) is comprised of well-formed class declarations (WF-CLASS). A class declaration is well-formed if its superclass is a class, *i.e.,* `Object` or a class defined in the program, any method overridden from the superclass has the same signature up to `async` or `void`, all *synchronous* chords are well-formed, and all *asynchronous* chords are well-formed.

A *synchronous* chord is well-formed when the return type of the chord's *synchronous* method signature coincides with the type of the chord body. The chord body is typed in a context where formal parameters take the types mentioned in *synchronous* and *asynchronous* method signatures, and *this* takes the type of the current class. Any other method signatures in the chord's signature must have a return type of `async`.

An *asynchronous* chord is well-formed when the chord body has type `void`, when typed in a context where the formal parameters take the types mentioned in the *asynchronous* method signatures.

With regard to method overriding, our system allows a method that returns `void` to be overridden in a subclass by a method that returns `async`. It also allows a method that returns `async` to be overridden in a subclass by a method that returns `void`. C_ω only allows a method that returns `void` to be overridden by a method that returns `async`. While overriding such as we allow may not be good programming practice, it does not affect the soundness of the type system, and so is allowed.

Furthermore, C_ω imposes restrictions on the overriding of methods when involved in chords, in order to avoid the inheritance anomaly [4]. The inheritance anomaly, however, is concerned with preservation of synchronisation properties and is unrelated to type soundness. Therefore, our system does not impose similar restrictions.

Finally, we do not require the class hierarchy to be acyclic. Although this property is useful for a compiler, it is not essential for type soundness.

Well-Formed Heaps

A well-formed heap (WF-HEAP) requires that every value in an object's queues must have a type according to the parameter type in the corresponding asynchronous method signature.

Soundness

The evaluation rules for SCHOOL preserve types throughout execution. We prove this property through a subject-reduction theorem [5]. The proof technique is standard.

We first define appropriate substitutions, σ, which map identifiers onto addresses in a type preserving way.

Definition 1 (Appropriate Substitution)
For a substitution $\sigma = Id \cup \{ \textit{this} \} \longrightarrow Addr$, a heap h, and an environemtn Γ, we have:

$$\left.\begin{array}{l} dom(\Gamma) \;=\; dom(\sigma) \\ \Gamma(id) \;=\; c \implies __,\ h \vdash \sigma(id) \;:\; c \end{array}\right\} \implies \Gamma,\ h \vdash \sigma$$

We can easily prove that an appropriate substitution, σ, when applied to an expression e turns it into a runtime expression, of the same type as the original expression.

Lemma 1 (Substitution)

$$\left.\begin{array}{l} P,\ \Gamma \vdash e \;:\; t \\ P,\ h \vdash \sigma \end{array}\right\} \implies P,\ h \vdash [e]_\sigma \;:\; t$$

Proof 1 *By induction on expression e.*

Furthermore, if a runtime expression has a certain type in a heap h, then it preserves its type in any heap h' where the objects have the same classes as the corresponding objects in h.

Lemma 2 (Preservation)
If: $\forall\ \iota\ dom(h) \;:\; h(\iota) \;=\; [\![\ c \parallel __\ ]\!] \implies h'(\iota) \;=\; [\![\ c \parallel __\ ]\!]$
then: $P,\ h \vdash e \;:\; t \implies P,\ h' \vdash e \;:\; t$

Proof 2 *By structural induction on expression e.*

We can prove subject reduction for the sequential case:

Lemma 3 (Subject Reduction - Sequential)

$$\left.\begin{array}{l} P \vdash h \\ \vdash P \\ P,\ h \vdash e : t \\ e,\ h \leadsto e',\ h' \end{array}\right\} \Longrightarrow \begin{array}{l} P \vdash h' \\ P,\ h' \vdash e' : t \end{array}$$

Proof 3 *By structural induction on the derivation* $\leadsto$ *.*

Finally, we can prove subject reduction for the multithreaded case:

Theorem 1 (Subject Reduction - Threads)
For SCHOOL heaps h and h', program P*, expressions e_1, ... e_n, e'_1, ... e'_m, types t_1, ... t_n, if:*

- $\vdash \mathrm{P}$ *and* $\mathrm{P} \vdash h$,
- $e_1,\ \ldots,\ e_n,\ h \leadsto e'_1,\ \ldots,\ e'_m,\ h'$
- $\mathrm{P},\ h \vdash e_i : t_i \qquad \forall i \in 1..n$

then, there exist types $t'_1,\ \ldots,\ t'_m$ so that:

- $\mathrm{P} \vdash h'$
- $\mathrm{P},\ h' \vdash e'_i : t'_i \qquad \forall i \in 1..m$
- $\{\ t_1,\ \ldots,\ t_n\ \} \cup \{\ void\ \} = \{\ t'_1,\ \ldots,\ t'_m\ \} \cup \{\ void\ \}$

Proof 4 *By case analysis on* $\leadsto$ *and application of lemma 3.*

3 Conclusions and Future Work

We designed SCHOOL with the aim of studying the features essential to an understanding of chords in an imperative, object-oriented setting. We made various design decisions to keep our language simple and the description minimal. Consequently, only classes and chords were necessary; the operational semantics requires only half a page, and ten rules!

We have also incorporated subclasses in SCHOOL, and were thus able to formally confirm that although inheritance and synchronisation do not generally mix well [4], the issues are unrelated to type soundness. Thus, in SCHOOL, we allow a method returning `void` to be overridden by a method returning `async`, and vice-versa. We also allow a method defined in one chord to be part of another chord in a subclass. The restrictions on overriding and method declaration in C_ω are thus unrelated to typing issues; rather, they attempt to preserve how a method is synchronised in subclasses.

In further work, we would like to extend SCHOOL to study interesting interactions with other language features. Although features like generics, packages, inner classes, overloading, and various control structures are probably orthogonal to chords, we expect that the introduction of delegates and exceptions may throw some interesting questions.

More interesting will be the study of the combination of SCHOOL and

explicit synchronisation mechanisms as in Java and $C^\sharp$, like locks and monitors. Furthermore, we would like to design extensions of chords to incorporate more advanced features, such as preemptions, priorities and transactions. We will use SCHOOL to express our designs.

It would also be interesting to consider issues around the scheduling for chords. The semantics of SCHOOL is non-deterministic, and thus abstract away one important property of chords as in Polyphonic $C^\sharp$; namely that any chord which *can* run (*i.e.,* whose queues are not empty), will *eventually* run. One could try to characterise such *fair* execution strategies through a further refinement of the operational semantics. More interesting would be a formal understanding of particular scheduling mechanisms, and proof of their properties.

Finally, another challenging direction is the use of chords in program understanding and verification. In [1], asynchronous methods correspond to states, and some synchronous method calls correspond to state change. Although this analogy cannot be expected to always hold, it would be interesting and useful to study how state transition diagrams can be mapped into chorded programs, and vice-versa. Such an approach would then allow the application of model-checkers.

References

[1] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for* C$^\sharp$, ACM Trans. Program. Lang. Syst. **26** (2004), pp. 769–804.

[2] Drossopoulou, S., *An abstract model of java dynamic linking and loading*, in: R. Harper, editor, *Third International Workshop, Types in Compilation (TIC 2000)*, LNCS **2071** (2000), pp. 53–84.

[3] Fournet, C. and G. Gonthier, *The reflexive CHAM and the join-calculus*, in: *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages* (1996), pp. 372–385.

[4] Matsuoka, S. and A. Yonezawa, *Analysis of inheritance anomaly in object-oriented concurrent programming languages*, in: *Research directions in concurrent object-oriented programming* (1993), pp. 107–150.

[5] Wright, A. K. and M. Felleisen, *A syntactic approach to type soundness*, Inf. Comput. **115** (1994), pp. 38–94.

Abstract Syntax

$$
\begin{array}{rcl}
e^s \in Expr^S & ::= & \texttt{null} \mid \text{voidVal} \mid \texttt{this} \mid x \\
& \mid & \texttt{new}\ \text{c} \mid e^s.m\ (\ e^s\ ) \mid e^s\ \ ;\ \ e^s \\
MethSig & ::= & t\ m\ (\ c\ ) \\
t \in Type & ::= & \texttt{void} \mid \texttt{async} \mid c \\
x,\ c,\ m \in Id & &
\end{array}
$$

Lookup Functions

$$
\begin{array}{rcl}
\mathcal{M}\ (\ \text{P},\ c,\ m\ ) & = & P{\downarrow_1}\ (\ c,\ m\ ) \\
\mathcal{SC}hs\ (\ \text{P},\ c,\ m\ ) & = & P{\downarrow_2}\ (\ c,\ m\ ) \\
\mathcal{AC}hs\ (\ \text{P},\ c\ ) & = & P{\downarrow_3}\ (\ c\ ) \\
\mathcal{M}^a & : & Program \times Id^c \rightarrow \mathcal{P}\ (\ Id^m\ ) \\
\mathcal{M}^a & = & \{\ m \mid \mathcal{M}\ (\ P,\ c,\ m\ )\ =\ \texttt{async}\ m\ (\ __\ )\ \}
\end{array}
$$

Program Representation

$$
\begin{array}{rcl}
Program & = & Id^c \times Id^m \rightarrow MethSig \\
& & \times \\
& & Id^c \times Id^m \rightarrow \mathcal{P}\ (\ Chord\ ) \\
& & \times \\
& & Id^c \rightarrow \mathcal{P}\ (\ Chord\ ) \\
& & \times \\
& & Id^c \rightarrow Id^c \\
Chord & = & \mathcal{P}\ (\ Id^m\ ) \times Expr
\end{array}
$$

Runtime Entities

$$
\begin{array}{rcl}
Heap & = & \mathbb{N} \rightarrow Object \\
Object & = & Id^c \times Queues \\
Queues & = & Id^m \rightarrow multiset(Val) \\
e \in Expr & ::= & voidVal \mid nullPtrEx \mid v \\
& \mid & \texttt{new}\ \text{c} \mid e.m\ (\ e\ ) \mid e\ \ ;\ \ e \\
v \in Val & ::= & null \mid \iota \\
\iota \in \mathbb{N} & &
\end{array}
$$

Well-Formedness

$$
\frac{\text{P} \vdash c \diamond_{cl} \implies \text{P} \vdash c}{\vdash \text{P}}\ \text{WF-PRGM}
$$

$$
\frac{\begin{array}{l}
\text{P} \vdash \text{P}{\downarrow_4}\ (\ c\ ) \diamond_{cl} \\
\mathcal{M}\ (\ \text{P},\ \text{P}{\downarrow_4}\ (\ c\ ),\ m\ )\ =\ t\ m\ (\ t'\ ) \implies \mathcal{M}\ (\ \text{P},\ c,\ m\ )\ =\ t''\ m\ (\ t'\ ) \\
\quad \text{where}\ \ t''\ =\ t\ \text{ or }\ t,\ t'' \in \{\ void,\ async\ \} \\
\mathcal{SC}hs\ (\ \text{P},\ c,\ m\ ) \ni (\ \{\ m_1,\ \ldots,\ m_n\ \},\ e\ ) \implies \\
\quad \forall i \in 1..n\ :\ \exists\, t_i : \mathcal{M}\ (\ \text{P},\ c,\ m_i\ )\ =\ \texttt{async}\ m_i\ (\ t_i\ ) \\
\quad\quad \exists\, t,\ t' : \mathcal{M}\ (\ \text{P},\ c,\ m\ )\ =\ t\ m\ (\ t'\ ) \\
\quad\quad \text{P},\ (\ m_1_x \mapsto t_1,\ \ldots,\ m_n_x \mapsto t_n,\ m_x \mapsto t',\ this \mapsto c\ ) \vdash e : t \\
\mathcal{AC}hs\ (\ \text{P},\ c\ ) \ni (\ \{\ m_1,\ \ldots,\ m_n\ \},\ e\ ) \implies \\
\quad \forall i \in 1..n\ :\ \exists\, t_i : \mathcal{M}\ (\ \text{P},\ c,\ m_i\ )\ =\ \texttt{async}\ m_i\ (\ t_i\ ) \\
\quad\quad \text{P},\ (\ m_1_x \mapsto t_1,\ \ldots,\ m_n_x \mapsto t_n,\ this \mapsto c\ ) \vdash e : void
\end{array}}{\text{P} \vdash c}\ \text{WF-CLASS}
$$

$$
\frac{h\ (\ \iota\ )\ =\ [\![\ c \mid\mid qs\ ]\!],\ \ \mathcal{M}\ (\ \text{P},\ c,\ m\ )\ =\ \texttt{async}\ m\ (\ t\ ),\ \ v \in qs\ (\ m\ ) \implies \text{P},\ h \vdash v : t}{\text{P} \vdash h}\ \text{WF-HEAP}
$$

Fig. 1. SCHOOL Overview

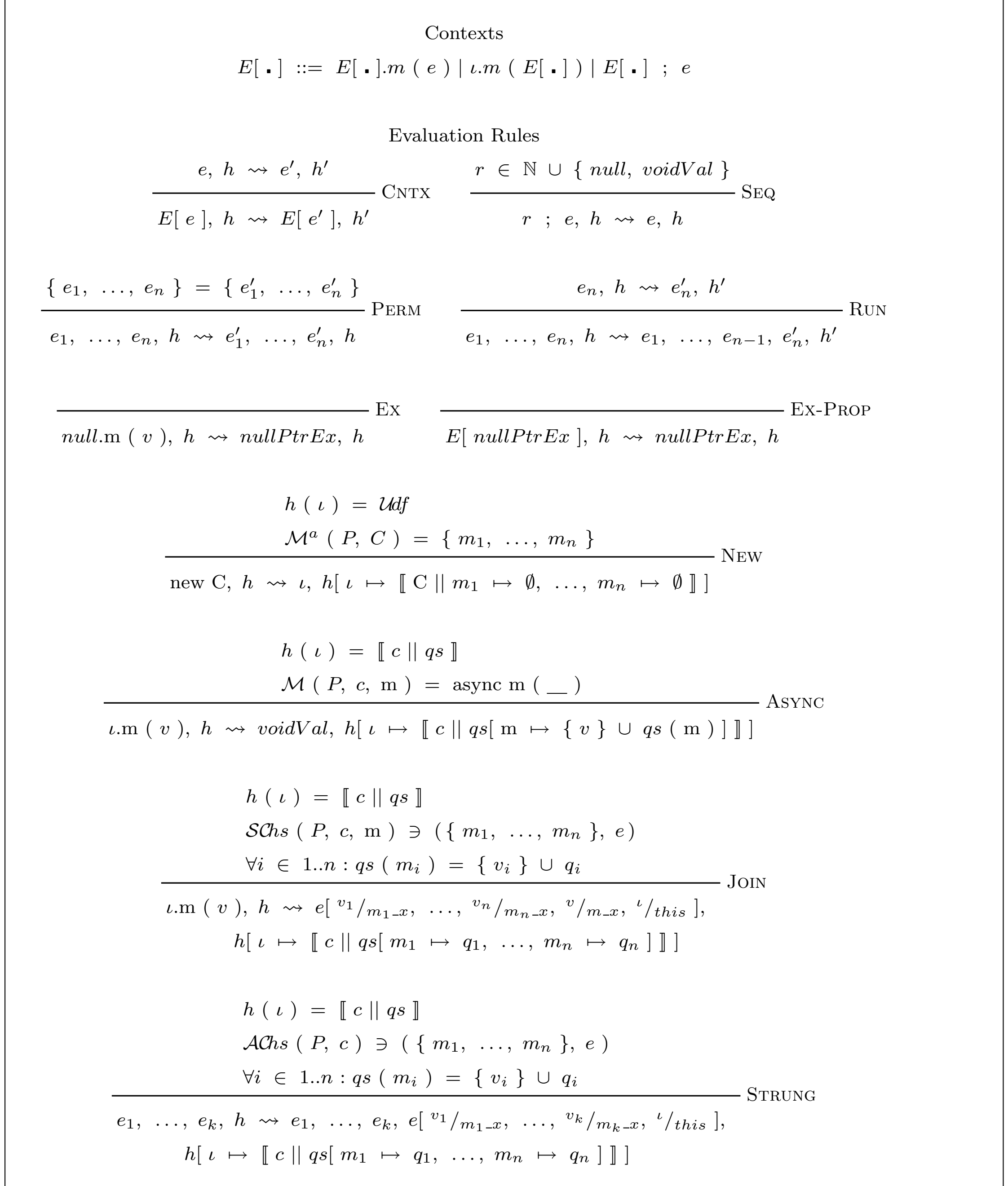

Fig. 2. SCHOOL Operational Semantics

Class and Type Declarations

$$\frac{P{\downarrow_4}\ (\ c\ )\ \neq\ \mathcal{U}\!\mathit{df}}{P\ \vdash\ c \diamond_{cl}}\ \textsc{Def-Class-1} \qquad \frac{}{P\ \vdash\ \mathit{Object} \diamond_{cl}}\ \textsc{Def-Class-2}$$

$$\frac{t\ \in\ \{\ \mathit{void},\ \mathit{async}\ \}}{P\ \vdash\ t \diamond_{tp}}\ \textsc{Def-Type-1} \qquad \frac{P\ \vdash\ c \diamond_{cl}}{P\ \vdash\ c \diamond_{tp}}\ \textsc{Def-Type-2}$$

Source-Level Type Judgements

$$\frac{P\ \vdash\ c \diamond_{cl}}{P,\ \Gamma\ \vdash\ \mathit{null} : c}\ \text{ST-}\textsc{Null} \qquad \frac{}{P,\ \Gamma\ \vdash\ \mathit{voidVal} : \mathit{void}}\ \text{ST-}\textsc{Void} \qquad \frac{z\ \in\ \{\ \mathit{this}\ \}\ \cup\ x}{P,\ \Gamma\ \vdash\ z : \Gamma\ (\ z\ )}\ \text{ST-}\textsc{ThisX}$$

$$\frac{P\ \vdash\ c \diamond_{cl}}{P,\ \Gamma\ \vdash\ \texttt{new c} : \mathrm{c}}\ \text{ST-}\textsc{New} \qquad \frac{\begin{array}{l}P,\ \Gamma\ \vdash\ e_1 : c\\ P,\ \Gamma\ \vdash\ e_2 : t\\ \mathcal{M}\ (\ P,\ c,\ \mathrm{m}\ )\ =\ t_r\ \mathrm{m}\ (\ t\ )\end{array}}{P,\ \Gamma\ \vdash\ e_1.\mathrm{m}\ (\ e_2\ ) : t_r}\ \text{ST-}\textsc{Inv} \qquad \frac{\begin{array}{c}P,\ \Gamma\ \vdash\ e_1 : t_1\\ P,\ \Gamma\ \vdash\ e_2 : t_2\end{array}}{P,\ \Gamma\ \vdash\ e_1\ \ ;\ \ e_2 : t_2}\ \text{ST-}\textsc{Seq}$$

Run-Time Type Judgements

$$\frac{P\ \vdash\ c \diamond_{cl}}{P,\ h\ \vdash\ \mathit{null} : c}\ \text{RT-}\textsc{Null} \qquad \frac{}{P,\ h\ \vdash\ \mathit{voidVal} : \mathit{void}}\ \text{RT-}\textsc{Void} \qquad \frac{h\ (\ \iota\ )\ =\ [\![\ c \mid\mid __\ ]\!]}{P,\ h\ \vdash\ \iota : c}\ \text{RT-}\textsc{Addr}$$

$$\frac{P\ \vdash\ c \diamond_{cl}}{P,\ h\ \vdash\ \texttt{new c} : \mathrm{c}}\ \text{RT-}\textsc{New} \qquad \frac{\begin{array}{l}P,\ h\ \vdash\ e_1 : c\\ P,\ h\ \vdash\ e_2 : t\\ \mathcal{M}\ (\ P,\ c,\ \mathrm{m}\ )\ =\ t_r\ \mathrm{m}\ (\ t\ )\end{array}}{P,\ h\ \vdash\ e_1.\mathrm{m}\ (\ e_2\ ) : t_r}\ \text{RT-}\textsc{Inv} \qquad \frac{\begin{array}{c}P,\ h\ \vdash\ e_1 : t_1\\ P,\ h\ \vdash\ e_2 : t_2\end{array}}{P,\ h\ \vdash\ e_1\ \ ;\ \ e_2 : t_2}\ \text{RT-}\textsc{Seq}$$

$$\frac{\begin{array}{l}P,\ h\ \vdash\ e : c\\ P{\downarrow_4}\ (\ \mathrm{c}\ )\ =\ c'\end{array}}{P,\ h\ \vdash\ e : c'}\ \text{RT-}\textsc{Sub-Cls} \qquad \frac{P,\ h\ \vdash\ e : \mathit{void}}{P,\ h\ \vdash\ e : \mathit{async}}\ \text{RT-}\textsc{Sub-Async} \qquad \frac{P\ \vdash\ t \diamond_{tp}}{P,\ h\ \vdash\ \mathit{nullPtrEx} : t}\ \text{RT-}\textsc{Ex}$$

Fig. 3. SCHOOL Type System