Correspondence Polymorphism for Object-Oriented Languages

Ran Rinat
Institute of Computer Science
Hebrew University, Israel
rinat@cs.huji.ac.il

Scott F. Smith*
Department of Computer Science
The Johns Hopkins University
scott@cs.jhu.edu

Menachem Magidor
Institute of Mathematics
Hebrew University, Israel
menachem@sunrise.huji.ac.il

Abstract

In this paper we propose a new form of polymorphism for object-oriented languages, called *correspondence polymorphism*. It lies in a different dimension than either parametric or subtype polymorphism. In correspondence polymorphism, some methods are declared to correspond to other methods, via a *correspondence relation*. With this relation, it is possible to reuse non-generic code in various type contexts—not necessarily subtyping or matching contexts—without having to plan ahead for this reuse. Correspondence polymorphism has advantages over other expressive object type systems in that programmer-declared types still may be simple, first-order types that are easily understood. We define a simple language LCP that reflects these new ideas, illustrating its behavior with multiple examples. We present formal type rules and an operational semantics for LCP, and establish soundness of the type system with respect to reduction.

1 Introduction

Polymorphism is an important component of object-oriented language design. The two foundational forms of polymorphism that may be used to give semantics to typed objects are *subtype polymorphism* and *parametric polymorphism*. The former may be used to model the manner in which a subclass member can be used where a superclass is the declared type. The latter allows code to be written where some types may be arbitrary, and later are instantiated.

In this paper we propose a new form of polymorphism for object-oriented languages, so-called *correspondence polymorphism*. The novelty of this new mechanism is that it allows programmers to reuse existing code in non-subtyping/subclassing contexts within a statically type-safe language. For example, given a class *matrix* of square matrices, its "power(n: *Nat*):*Matrix*" method (successive self multiplication) can be implemented by extracting method "power(n: *Nat*):*Integer*" from class *integer*, provided that the multiplication methods in both classes are related via a *correspondence relation*, $\overset{Corr}{\longleftrightarrow}$. The point is that matrices and integers are not related by subtyping/subclassing, nor do they share a common supertype/superclass.

This is the essence of correspondence polymorphism: it implicitly makes non-generic code usable for purposes beyond the specific instance for which it was originally developed. It differs from a programming style that relies on universal polymorphism, in that code need not be a-priori written for universal applicability. In other words, with respect to type declaration, programmers do not have to plan ahead for code reuse.

In [17], the first and third authors proposed a form of polymorphism motivated from metaphors of natural language. This paper makes those high-level ideas concrete by defining an actual language embodying the concepts. We define the language, type rules, reduction rules, and prove soundness of the type rules with respect to reduction.

The best way the concepts can be understood is via examples, so we immediately proceed in Section 2 with the definition of our language, LCP (Language with Correspondence Polymorphism), followed by a series of informal examples showing how the language may be used. In this section we also relate correspondence polymorphism to other approaches

*Partial funding provided by NSF grant CCR-9619843

OOPSLA '99 11/99 Denver, CO, USA

$$
\begin{array}{l}
e ::= \; x \mid n \mid b \mid \lambda x : \tau.e \mid e\ e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \\
\quad \mid \textbf{let } x : \tau = e \textbf{ in } e \mid e \leftarrow m \mid e.y \mid e.y := e \\
\quad \mid \textbf{obj}(s : \tau) \textbf{ inst } \overline{y_j = e_j} \textbf{ meth } \overline{m_k = e_k} \\
\quad \mid \textbf{class}(s : \tau) \textbf{ inst } \overline{y_j = e_j} \textbf{ meth } \overline{m_k = e_k} \\
\quad \mid e :: m \mid e :: y \mid \textbf{new } e
\end{array}
$$

$x \in Var$
$n ::= \; 0 \mid 1 \mid 2 \mid \ldots$
$b ::= \; \textbf{true} \mid \textbf{false}$

Figure 1: LCP expressions

for typing object-oriented programs, including [6, 7, 4, 15]. Section 3 then demonstrates that the concepts illustrated in the examples are well-founded by defining type and reduction rules, and by proving a subject reduction property.

2 The Language LCP

The syntax of LCP (Language with Correspondence Polymorphism) is shown in Figure 1. The language incorporates a simple notion of object and class. Writable instance variables are hidden in objects. Objects may be created either by **new**-ing a class, or directly by a primitive **obj** construct in the spirit of the Abadi-Cardelli object calculus [1]. Method bodies are typically functions $\lambda x : \tau.e$. Functions, objects, and classes all have explicitly declared types, but these declarations only assert the *original intention* of the code, and the code may in fact be used at other types, via the correspondence relation. These concepts will be clarified by the examples below.

LCP contains a fine-grain notion of inheritance drawing on a conception of classes as collections of pre-methods [1]: classes inherit by extracting methods from existing classes. Syntax $e :: m$ extracts method m from class e as a function from self. The extracted method can then be embedded in another class. This allows for multiple inheritance, and also allows for an inheriting class to only inherit some of the code; in the case where the code left behind was in fact needed, a type error will arise. The standard notion of inheritance is syntactic sugar that implicitly extracts all methods from the superclass into the current class.

Figure 2 gives the syntax of LCP types. Class and object types are "named" via an identifier T, taken from the set *ObjTyId* of *object type identifiers*. Inst-Meth types $\ll T,\ \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau_k} \gg$ are the bodies of object types with name T: in $\textbf{Obj } mt\ \tau$, τ must be an Inst-Meth type, in which mt may occur free. Object types are thus a combination of name and structure, much like in real

$$
\begin{array}{l}
\tau ::= \; mt \mid Bool \mid Nat \mid \tau \to \tau \\
\quad \mid \ll T,\ \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau_k} \gg \mid \textbf{Obj } mt\ \tau \\
\quad \mid \textbf{Class } \ll T,\ \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau'_k \mapsto \tau_k} \gg
\end{array}
$$

$mt \in TyVar$
$T \in ObjTyId$

Figure 2: LCP types

programming languages such as C++, Java and Eiffel. The type variable mt is analogous to Bruce's *MyType*, the external type of self in objects and classes. We omit the keyword **Inst** when there are no instance variables, so $\ll T,\ \textbf{Meth } \overline{m_k : \tau_k} \gg$ stands for $\ll T,\ \textbf{Inst Meth } \overline{m_k : \tau_k} \gg$.

Class types are similar to Inst-Meth types, except that methods are given types $\tau'_k \mapsto \tau_k$ for future extraction: τ'_k is the future self and τ_k is the future method result type. The purpose of these types will be made clear by the examples. Note that in constrast with object types, class types are not recursive.

Along with correspondence polymorphism, LCP has subtype polymorphism and what could be called "poor man's parametric polymorphism" – in the **let** construct, the expression placed in x is re-type-checked for every context it occurs in the **let** body. This is strictly more powerful than parametric let-polymorphism, but requires re-type-checking for each occurrence of x. **let** is used in LCP to express code reuse, as the examples below will show.

LCP expressions are evaluated and typed w.r.t. a fixed correspondence relation $\overset{Cor_\Gamma}{\longleftrightarrow}$. It is an equivalence relation on the set $\{T.a \mid T \in ObjTyId,\ a \in L\}$, where L is the set of instance variables and method labels. Intuitively, $T.m \overset{Cor_\Gamma}{\longleftrightarrow} S.n$ asserts that method name m relative to type name T "has the same meaning" as method name n relative to type name S. In LCP this relation is assumed given, but in a more realistic language, it would be induced by program structure. One specific example is found in [17], where the correspondence relation is induced by the inheritance hierarchy, and by an explicit construct for relating method names in different types.

Every object with declared type

$$\textbf{Obj } mt\ \ll T,\ \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau_k} \gg,$$

is labeled by the identifier T. This label is used in conjunction with the correspondence relation to interpret message sends at run-time: suppose that message send $e \leftarrow m$ is to be computed, that the originally intended (*i.e.*, declared) type of e is an object type named S, and that e evaluates to an

object o labeled T. Then, assuming $S.m \overset{Corr}{\longleftrightarrow} T.n$, the method n in o is invoked. A runtime type error is flagged if there is no n such that $S.m \overset{Corr}{\longleftrightarrow} T.n$. A typable expression, however, cannot run into type errors, as guaranteed by the subject reduction property of LCP's type system.

To ensure the determinism of the runtime semantics, we impose the following condition on the correspondence relation $\overset{Corr}{\longleftrightarrow}$: for every $T \in ObjTyId$, $a \in L$, and $S \in ObjTyId$ there is at most one $b \in L$ such that $T.a \overset{Corr}{\longleftrightarrow} S.b$.

We now present a series of examples to informally explain the language; the operational semantics and type rules presented in Section 3 will justify the informal assertions made below.

2.1 A First Example: Integers and Matrices

Consider the object types *IntType* and *MatType* (the type of square matrices) shown in Fig. 3(a).

Neither has fields that are a subset of the fields of the other, so no inheritance relationship could be defined in most object-oriented languages. In LCP, fine-grain inheritance is still possible, as we will show. *IntType* and *MatType* are notations in the meta-language for two object types named Integer and Matrix, respectively (*i.e.*, Integer and Matrix are elements of *ObjTyId*). Using these types, we define the classes *intClass* and *matClass* as in Fig. 3(b). *Notation:* we assume some standard built-in primitive functions, such as *isZero*(n), *pred*(n), in the set $B \subseteq Var$. If

$$c = \textbf{class}(s : \tau)\ \textbf{inst} \ldots \textbf{meth} \ldots$$

is a class expression then

letclass $x = c$ **in** e is syntactic sugar for

$$\textbf{let}\ x : \tau^{Class} = c\ \textbf{in}\ e$$

where τ^{Class} denotes the type of classes producing objects of type τ. The notation τ^{Class} is defined precisely in Sec. 3.

In the class *matClass*, the method powerMat is implemented by extracting power from *intClass* and applying it to self. We say that power is *extracted* from *intClass* and *embedded* in *matClass*. Now, assuming Integer.mult $\overset{Corr}{\longleftrightarrow}$ Matrix.multMat and Integer.power $\overset{Corr}{\longleftrightarrow}$ Matrix.powerMat, the method powerMat will typecheck and function as desired. We illustrate this with a sample execution:

```
letclass matrix = matClass
in let o:MatType = new matrix
     in o ← powerMat(5)
```

The object o created by "**new** matrix" is labeled at run-time by type identifier "Matrix" since this is the type identifier declared in its class. Then, evaluation of powerMat(5) proceeds as follows: integer::power applied to o and then to 5 is evaluated. Inside method power, in turn, the self variable s has originally intended type named "Integer". And, s is bound to o which is labeled "Matrix". Since Integer.mult $\overset{Corr}{\longleftrightarrow}$ Matrix.multMat, the message send s ← mult inside power will dispatch as multMat. Similarly, power message sends will dispatch as powerMult.

Thus, the code exploits the fact that the algorithm for matrix powering relates to matrix multiplication in exactly the same way integer powering relates to integer multiplication. In LCP, such reuse is possible even though no subtyping or matching holds between *IntType* and *MatType*, and moreover, the multiplication methods have different names. The choice of method name "multMat" rather than "mult", as in *IntType*, is artificial in this example, but we will give an example below that shows a case when name changes are useful.

By extracting the power method and embedding it in *matClass*, we have reused a non-generic piece of code in a new context. Similar reuse can be made of free-standing functions, not just of methods. Consider the following LCP program fragment (*Notation:* **let** $f(x : \tau) : \sigma = e'$ **in** e is syntactic sugar for **let** $f : \tau \to \sigma = \lambda x : \tau.e'$ **in** e.):

```
letclass integer = intClass
in letclass matrix = matClass
   in let square(i:IntType):IntType = i← mult(i)
      in... square(new integer) ...
         ... square(new matrix) ...
```

Here, the function "square", written with integers in mind, is used to square both integers and matrices. The typing rule for **let** allows for different typings of "square" after the **in**, so the code above will typecheck. In addition, the rule for **let** verifies that "square" has its declared type *IntType* → *IntType*, *regardless of what appears after the* **in** (that is, this would be required even if there were no application of "square" to integers after the **in**). Thus, functions can be typechecked at types different than their declarations, but declared typings must be verified to hold. Similarly, although the method "power" is embeddable in *matClass*, we are sure that integer::power has type *IntType* → *Nat* → *IntType* since it is part of a class whose declared type is *IntType*, in which power is declared to have type *Nat* → *IntType*. Therefore, programming in LCP has two levels of sophistication: a novice user can write programs that look like code in any other

IntType $\triangleq$ **Obj** mt $\ll$ Integer, **Meth** mult: mt$\rightarrow$ mt, power: *Nat*$\rightarrow$ mt, isPrime: *Bool* $\gg$
MatType $\triangleq$ **Obj** mt $\ll$ Matrix, **Meth** multMat: mt$\rightarrow$ mt, powerMat: *Nat*$\rightarrow$ mt, isSingular: *Bool* $\gg$

(a) The types *IntType* and *MatType*

```
intClass ≙ class (s:IntType) meth mult =...,
                                  power = λn:Nat. if isZero(n) then s
                                                              else s ← mult(s ← power(pred(n)),
                                  isPrime =...

matClass ≙ letclass integer = intClass
           in class (s:MatType) meth multMat =...,
                                     powerMat = integer::power(s),
                                     isSingular =...
```

(b) The classes *intClass* and *matClass*

Figure 3: The integer-matrix example

annotated language, where everything behaves according to declaration. A more sophisticated user can reuse code at other types, with the safety of such usage verified by the type checker.

2.2 Example: Points and ColorPoints

As a benchmark to compare with other object typing papers, we haul out the classic Point-ColorPoint example [6], shown in Fig. 4. We implicitly assume in this and the following examples that like-named methods correspond.

This example illustrates that in addition to code reuse in non-subtyping/matching contexts, LCP can also typecheck safe inheritance with binary methods (that is, inheritance in matching contexts). The method "usesEqual" (some random method which uses the "equal" method) is inheritable into class *cPointClass* because it can be retyped as *CPointType* $\rightarrow$ *CPointType*. This form of inheritance is impossible in Java and C++. It *is* possible in languages supporting *MyType* and matching-based inheritance, e.g. Bruce's PolyTOIL [7].

The example also shows how superclass methods may be referenced during method override in LCP (which in some languages is accomplished with a keyword **super**): the "equal" method of *cPointClass* calls the superclass equal method via syntax"point::equal(s)". This code in Fig. 4, and the following safe message sends

- aPoint$\leftarrow$ equal(anotherPoint)
- aColorPoint$\leftarrow$ equal(anotherColorPoint)
- aPoint$\leftarrow$ equal(aColorPoint)

all typecheck, whereas the unsafe case aColorPoint$\leftarrow$ equal(aPoint), which produces a run-time error, does not. Note especially the third safe case aPoint$\leftarrow$ equal(aColorPoint) which LCP will typecheck, but PolyTOIL and other matching-based languages will fail on (they will type the other safe cases). Precise declared typings can typecheck this particular message send as well; see Section 4.4 of [6].

It should be pointed out, however, that the languages mentioned above pursue a modular typechecking strategy, in which a method is typechecked once and for all for safe inheritability. LCP, in contrast, re-typechecks methods in every new context. This is a weakness of LCP relative to all the examples presented here. We discuss this issue in Sec. 4.

2.3 Example: Circles and ColorCircles

The code in Fig. 5 demonstrates that in addition to binary methods, LCP can also handle covariant redefinition of method parameters and instance variables. The example is reproduced from [5] with minor modifications. It uses the *PointType* and *CPointType* types of the previous example.

The inheritance of *cCircleClass* from *circleClass* is impossible in languages with subtype-based subclassing like C++ and Java, as well as in languages with match-based subclassing like TOOPLE [4] and PolyTOIL [7]. In PolyTOIL, however, a parametric structure which uses match-bounded polymorphism can be programmed instead [5]. The code does type check in Eiffel. Consider now the following code (we informally use sugar for a 2-argument function here, which is implemented in LCP via currying):

```
PointType  ≜ Obj mt ≪ Point, Inst xVal: Nat
                        Meth equal: mt→ Bool, usesEqual: mt→ mt, x: Nat ≫
CPointType ≜ Obj mt ≪ ColorPoint, Inst xVal: Nat, colorVal: Nat
                        Meth equal: mt→ Bool, usesEqual: mt→ mt, x: Nat, color: Nat ≫

pointClass ≜ class (s:PointType) inst xVal = 0
                        meth equal = λp:PointType.(if p←x == s←x then true else false),
                             usesEqual = λp:PointType.( ...s← equal(p) ... ),
                             x = s.xVal

cPointClass ≜ letclass point = pointClass
              in class (s:CPointType) inst xVal = point::xVal, colorVal = 0
                        meth equal = λp:CPointType.(point::equal(s)(p) &&
                                                    p←color == s←color),
                             usesEqual = point::usesEqual(s),
                             x = point::x(s),
                             color = s.colorVal
```

Figure 4: The point-colorPoint example

```
CircleType  ≜ Obj mt ≪ Circle, Inst center: PointType, radius: Nat
                        Meth getCenter: PointType, setCenter: PointType→ mt ≫
CCircleType ≜ Obj mt ≪ ColorCircle, Inst center: CPointType, radius: Nat, colorVal: Nat
                        Meth getCenter: CPointType, setCenter: CPointType→ mt, color: Nat ≫

circleClass ≜ class (s:CircleType) inst center = new pointClass, radius = 1
                        meth getCenter = s.center,
                             setCenter = λc:PointType.(s.center := c)

cCircleClass ≜ letclass circle = circleClass
               in class (s:CCircleType) inst center = new cPointClass, radius = circle::radius, colorVal = 0
                        meth getCenter = circle::getCenter(s),
                             setCenter = circle::setCenter(s),
                             color = s.colorVal
```

Figure 5: The circle-colorCircle example

let setCircleCenter(c:*CircleType*,
p:*PointType*):*CircleType*
= c← setCenter(p)

in... setCircleCenter(aCircle,aPoint) ...
... setCircleCenter(aColorCircle,aColorPoint) ...
... setCircleCenter(aCircle,aColorPoint) ...
... setCircleCenter(aColorCircle,aPoint) ...

The first call typechecks in any reasonable system, and the last two fail because they can lead to a runtime type error. The interesting call is the second one, which typechecks in LCP because the function "setCircleCenter" can be re-typechecked with argument types *CCircleType* and *CPointType*. The body of "setCircleCenter" will not type check in Eiffel because the message send c← setCenter(p) is a *polymorphic catcall* [15]. It would typecheck in earlier versions of Eiffel which defined a so-called *system level validity check* [14], which was never in fact implemented in a compiler. In PolyTOIL, the function can typecheck after reprogramming with appropriate match-bounded parametrization.

Covariant retyping of subclass method parameters can also be supported via *virtual types* as originated in Beta [13], and later proposed for Java [18], but both without static typechecking. A recent proposal [8] shows that similar functionality can be obtained and statically typechecked by using a natural generalization of matching to mutually recursive types. The circle-colorCircle inheritance could be programmed and typechecked there, and "setCircleCenter" could be programmed with match-bounded parameterization to make it callable with argument types *CCircleType*, *CPointType*. This proposal, however, requires the programmer to introduce a new level of a abstraction - *type groups* - defining several types at once by mutual recursion. This is somewhat complicated, and also, does not always correspond to intuition. For example, *PointType* and *CircleType* are not by intuition mutually recursive: it is just the latter which depends the former.

2.4 Example: Functions and Integers

This last example demonstrates how it may be useful for $\overset{Cor}{\longleftrightarrow}$ to relate methods with different names and meanings in order to achieve reuse. Consider the following object type *NatFuncType* of functions from *Nat* to *Nat*:

NatFuncType $\triangleq$
Obj mt $\ll$ NatFunc, **Meth** apply: *Nat*→ *Nat*,
compose: *mt*→ mt,
power: *Nat*→ mt$\gg$

The method "apply" returns the result of applying self to the argument, "compose" returns the composition of self and the argument, and "power" repeatedly composes self with itself.

Recall the Integer-Matrix example of Fig. 3, and suppose we define the correspondence Integer.mult $\overset{Cor}{\longleftrightarrow}$ NatFunc.compose. Then a class *natFuncClass*, producing objects of type *NatFuncType*, can implement "power" by extracting it from *intClass* and applying it to self.

This example shows that in LCP, an algorithm can be reused even when the underyling concepts are quite different: integer multiplication is nothing like function composition. An important property here is that even though the correspondence is between methods of completely different names and meanings, the extracted method implements a method with the same name ("power"). Thus, there is reasonable chance that programmers will consider this reuse potential. Reusing a concept ("power" in this example) from one domain (integers) to another (matrices) by mapping different lower-level concepts is in the essence of metaphors, which underly the concept of correspondence polymorphism [17].

2.5 The Role of Declared Types in LCP

An essential characteristic of correspondence polymorphism is that code can typecheck at types different than declared via a correspondence relation. However, declared types retain their documentary and client-contract roles as *programmer-declared interfaces* since they are always verified to hold. Typing of the same code at different types is accomplished in LCP via use of **let**:

let integer:$IntType^{Class}$ = *intClass*
in ... integer::power(s) ... integer::isPrime(s) ...

The two occurrences of "integer" in the body can have components re-type-checked at different types, for instance the "power" method being used inside the matrix class (and, being re-type-checked to work on matrices), and the "isPrime" method being similarly extracted, but for use in some third class. Regardless of what appears after the **in**, the type rules verify that *intClass* has type $IntType^{Class}$, as declared for it after the **let**.

Not less importantly, when retyping code, declared types provide the programmer's originally intended types, which are used in conjunction with the correspondence relation to dispatch methods. Thus, LCP can be viewed as a hybrid of an explicitly- and implicitly-typed language: the user type declarations are honored, but in retypings, these declarations are ignored as type assumptions, and instead

take on meaning as original intentions. For example, in the typing of "integer::power(s)" above, the body of the power method is re-type-checked for self s a matrix, *and*, the send of "mult" to s in the body of power requires the intended typing *IntType* of s, so that mult maps to multMat via the correspondence relation.

In summary, declared types have a dual role:

- They are verified interfaces, *and*
- They are used to dispatch methods via the correspondence relation

2.6 Correspondence polymorphism related to other forms

Correspondence polymorphism significantly differs from polymorphism in standard object-oriented languages, but there are nonetheless some relationships, which we discuss now.

The nature of object polymorphism in different languages may be partly modeled by different forms of $\overset{Cor\tau}{\longleftrightarrow}$. The modeling is partial in the sense that correspondence polymorphism will still allow more programs to typecheck; we will clarify what the gap is. The object polymorphism allowed in C++ and Java (ignoring casting and templates) is one where subclass objects may be used where a superclass object is declared. The method names used in the superclass must be identical to the corresponding method names in the subclass. This may be modeled in LCP by a correspondence relation with $T.m \overset{Cor\tau}{\longleftrightarrow} S.n$ whenever T is a subclass of S and $m = n$, that is, subclass methods correspond to the same method in their superclass.

This analogy is not precise in that LCP will still allow more programs to typecheck: suppose a C++ class A defining a method m_A is inherited by B, which only adds a method m_B. Suppose we write a global C++ function **void** $f(B\ aB)$ whose implementation uses m_A, but not m_B. Then $f(anA)$ would not typecheck in C++, but would in LCP since f is re-typable at type $A \to$ **void**. Thus, LCP can compensate for lack of typing precision on the part of the programmer by retyping.

PolyTOIL [7] has a richer type system, but the types that programmers must declare can be quite complex: bounded matching is a form of higher-order bounded polymorphic type [6]. LCP declared types, in contrast, are of a complexity similar to Java and C++ types. The difference in expressivity between LCP and PolyTOIL is not great. It is mainly one of programming style: PolyTOIL achieves code reuse through *generic* type declarations whereas LCP reuses by interpreting code with *specific* declared types in new contexts, ignoring the type declarations. PolyTOIL has the advantage over LCP of being easier to typecheck modularly – avoiding the need for re-type-checking method bodies – and of types that provide more precise code documentation.

Finally, if $\overset{Cor\tau}{\longleftrightarrow}$ is only reflexive ($T.m \overset{Cor\tau}{\longleftrightarrow} T.m$ only), LCP will have no subtype polymorphism, something roughly corresponding to C or Pascal.

3 Semantics of LCP

In this section we give the semantics of LCP by presenting type rules, operational semantics, and the proof of subject reduction. In the body of the paper we present only the type rules and state the Theorems, and relegate the operational semantics rules and proofs to Appendicies.

Notation: If

$$\tau = \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg$$

is an Inst-Meth type then τ^{Meth} denotes

$$\ll T,\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg$$

and $(\mathbf{Obj}\ mt\ \tau)^{Class}$ stands for the class type

$$\mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : (\mathbf{Obj}\ mt\ \tau) \mapsto \tau_k[\mathbf{Obj}\ mt\ \tau^{Meth}/mt]} \gg$$

3.1 Subtyping

The subtyping rules of LCP are standard except for incorporation of $\overset{Cor\tau}{\longleftrightarrow}$ in object and class subtyping. Figure 6 presents the subtyping rules. The system does not contain transitivity or reflexivity rules since they are derivable; see the Lemma below. The (Sub Object) rule is a simple form of object subtyping, in contrast with the more expressive [3, 10] which equate recursive types with their unfolding. The rule here is closer to the simple rules of [4]. In the (Sub InstMeth) rule, subtyping between objects can be seen to depend on the correspondence relation: given two Inst-Meth types

$$\begin{aligned}\tau = \ll T,\ & \mathbf{Inst}\ y_j : \tau_j^{\ j\in\{1\ldots n\}} \\ & \mathbf{Meth}\ m_k : \rho_k^{\ k\in\{1\ldots n'\}} \gg \\ \sigma = \ll S,\ & \mathbf{Inst}\ x_j : \sigma_j^{\ j\in\{1\ldots n+m\}} \\ & \mathbf{Meth}\ n_k : \mu_k^{\ k\in\{1\ldots n'+m'\}} \gg\end{aligned}$$

the longer σ is a subtype of the shorter τ when $T.y_j \overset{Cor\tau}{\longleftrightarrow} S.x_j$, $T.m_k \overset{Cor\tau}{\longleftrightarrow} S.n_k$, and (recursively) $\overline{\sigma_j <: \tau_j}$, $\overline{\tau_j <: \sigma_j}$, and $\overline{\mu_k <: \rho_k}$. Thus, this rule is a combination of name-based subtyping and structural subtyping of record types. Note that with

(Sub Env) $$\frac{}{C, \sigma <: \tau \vdash \sigma <: \tau}$$

(Sub Nat) $$\frac{}{C \vdash \mathit{Nat} <: \mathit{Nat}}$$

(Sub Bool) $$\frac{}{C \vdash \mathit{Bool} <: \mathit{Bool}}$$

(Sub Func) $$\frac{C \vdash \tau_1 <: \tau_2 \quad C \vdash \sigma_1 <: \sigma_2}{C \vdash \tau_2 \to \sigma_1 <: \tau_1 \to \sigma_2}$$

(Sub InstMeth) (where $\tau = \ll T,\ \mathbf{Inst}\ y_j : \tau_j{}^{j \in \{1 \ldots n\}}\ \mathbf{Meth}\ m_k : \rho_k{}^{k \in \{1 \ldots n'\}} \gg$
$\sigma = \ll S,\ \mathbf{Inst}\ x_j : \sigma_j{}^{j \in \{1 \ldots n+m\}}\ \mathbf{Meth}\ n_k : \mu_k{}^{k \in \{1 \ldots n'+m'\}} \gg$)

$$\frac{(T.y_j \overset{Corr}{\longleftrightarrow} S.x_j \quad C \vdash \sigma_j <: \tau_j \quad C \vdash \tau_j <: \sigma_j)^{\,j \in \{1 \ldots n\}} \quad (T.m_k \overset{Corr}{\longleftrightarrow} S.n_k \quad C \vdash \mu_k <: \rho_k)^{\,k \in \{1 \ldots n'\}}}{C \vdash \sigma <: \tau}$$

(Sub Obj) (τ, σ as above)

$$\frac{C, \mathbf{Obj}\ mt\ \sigma^{Meth} <: \mathbf{Obj}\ mt\ \ \tau^{Meth} \vdash \sigma[\mathbf{Obj}\ mt\ \sigma^{Meth}/mt]\ <:\ \tau[\mathbf{Obj}\ mt\ \tau^{Meth}/mt]}{C \vdash \mathbf{Obj}\ mt\ \sigma <: \mathbf{Obj}\ mt\ \ \tau}$$

(Sub Class) (where $\tau = \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ y_j : \tau_j{}^{j \in \{1 \ldots n\}}\ \mathbf{Meth}\ m_k : \rho'_k \mapsto \rho_k{}^{k \in \{1 \ldots n'\}} \gg$
$\sigma = \mathbf{Class}\ \ll S,\ \mathbf{Inst}\ x_j : \sigma_j{}^{j \in \{1 \ldots n+m\}}\ \mathbf{Meth}\ n_k : \mu'_k \mapsto \mu_k{}^{k \in \{1 \ldots n'+m'\}} \gg$)

$$\frac{(T.y_j \overset{Corr}{\longleftrightarrow} S.x_j \quad C \vdash \sigma_j <: \tau_j \quad C \vdash \tau_j <: \sigma_j)^{\,j \in \{1 \ldots n\}} \quad (T.m_k \overset{Corr}{\longleftrightarrow} S.n_k \quad C \vdash \mu'_k \to \mu_k <: \rho'_k \to \rho_k)^{\,k \in \{1 \ldots n'\}}}{C \vdash \sigma <: \tau}$$

Figure 6: Subtyping rules for LCP

the trivial correspondence relation $T.m \overset{Corr}{\longleftrightarrow} S.n$ iff $T = S$ and $m = n$, we obtain trivial subtyping $\vdash \tau <: \sigma$ iff $\tau = \sigma$.

Lemma 3.1: The following properties of subtyping hold:

1. It is reflexive on closed types: $\vdash \tau <: \tau$ for any closed τ.
2. It is transitive on closed types: $\vdash \sigma <: \tau$, $\vdash \tau <: \rho$ implies $\vdash \sigma <: \rho$ (for any τ, σ, ρ).
3. Let τ be an Inst-Meth type with no free variables other than mt. Then $\vdash$ **Obj** mt τ $<:$ **Obj** mt τ^{Meth}.
4. Let $\tau =$ **Obj** mt $\ll T$, **Inst** $y_j : \tau_j{}^{j\in\{1\ldots n\}}$ **Meth** $m_k : \rho_k{}^{k\in\{1\ldots n'\}} \gg$, $\sigma =$ **Obj** mt $\ll S$, **Inst** $x_i : \sigma_i{}^{i\in\{1\ldots m\}}$ **Meth** $n_l : \mu_l{}^{l\in\{1\ldots m'\}} \gg$ be object types, and suppose $\vdash \sigma <: \tau$, $T.y_j \overset{Corr}{\longleftrightarrow} S.x_i$, and $T.m_k \overset{Corr}{\longleftrightarrow} S.n_l$. Then $\vdash \sigma_i <: \tau_j$, $\vdash \tau_j <: \sigma_i$, and $\vdash \sigma_l[$**Obj** mt $\sigma^{Meth}/mt] <: \tau_k[$**Obj** mt $\tau^{Meth}/mt]$.
5. Let $\tau =$ **Class** $\ll T$, **Inst** $y_j : \tau_j{}^{j\in\{1\ldots n\}}$ **Meth** $m_k : \rho'_k \mapsto \rho_k{}^{k\in\{1\ldots n'\}} \gg$, $\sigma =$ **Class** $\ll S$, **Inst** $x_i : \sigma_i{}^{i\in\{1\ldots m\}}$ **Meth** $n_l : \mu'_l \mapsto \mu_l{}^{l\in\{1\ldots m'\}} \gg$ be class types, and suppose $\vdash \sigma <: \tau$, $T.y_j \overset{Corr}{\longleftrightarrow} S.x_i$, and $T.m_k \overset{Corr}{\longleftrightarrow} S.n_l$. Then $\vdash \sigma_i <: \tau_j$, $\vdash \tau_j <: \sigma_i$, and $\vdash \mu'_l \to \mu_l <: \rho'_k \to \rho_k$.
6. Let τ and σ be InstMeth types. Then **Obj** mt σ $<:$ **Obj** mt τ if and only if $($**Obj** mt $\sigma)^{Class} <: ($**Obj** mt $\tau)^{Class}$.

In what follows we write $\sigma <: \tau$ for $\vdash \sigma <: \tau$.

3.2 Originally Intended Types

Both the operational semantics and the typing rules for LCP rely on the notion of *originally intended type*. In this section we present rules for deriving intended types. Informally, the intended type of a given occurrence of an expression e is the type of the value to which the programmer intended e to evaluate when he or she originally wrote it. For LCP, the intended types are precisely the programmer-declared types annotating a program.

Figure 7 presents a system to extract intended typings from LCP programs. We write $I \vdash\!\sim e : \tau$ to express that from expression e, intended type τ is derived under the type assignment I. This system should not be confused with a standard type system; it only extracts the declared type information from terms and asserts no relationship between the declarations and the program execution.

The system is very simple and deterministic, so it immediately implies a (linear) inference algorithm.

Lemma 3.2: The intended type extraction rules of Figure 7 have a decidable type inference property.

Intended typing, when it exists, is unique. This guarantees that the semantics is deterministic.

Lemma 3.3: If $I \vdash\!\sim \tau$ and $I \vdash\!\sim \tau'$ then $\tau = \tau'$.

The notion of originally intended type in LCP is weaker than the ideal notion. Recall the Integer-Matrix example of Section 2.1 and the auxiliary function *square* (which is not a method of *MatType* or *IntType*):

$$square \triangleq \lambda o{:}IntType.o \leftarrow \text{mult}(o)$$

and consider the message send "*square*(aMatrix)$\leftarrow$ isSingular". This perfectly sensible expression will not type check in LCP: square has intended type $IntType \to IntType$, so "square(aMatrix)" has intended type *IntType*, and *IntType* does not have a method "isSingular", and nothing corresponds to Integer.isSingular. Thus, typechecking fails. Note that this problem would *not* arise if square were a method inside *IntType*, where it really belongs: then it could be inherited into *matClass* with type $mt \to mt$.

Stronger intended type derivation rules could potentially infer that *square*(aMatrix) has intended type *MatType*, because the programmer was using *square* at a non-declared type, and expected *square*(aMatrix) to return a matrix. A topic of future work is to improve on the intended typing rules to incorporate such a context-sensitive notion of intention. There is in fact a way of circumventing this problem by breaking the above computation into two steps using local variables: declare a variable x of type *MatType*, assign x:= *square*(aMatrix), then write x$\leftarrow$ isSingular. This has the same semantics as the above, but will work since x has intended type *MatType*.

3.3 Typing Rules

Figure 8 presents the typing rules for LCP. Judgements are sequents of the form $I, A \vdash e : \tau$. I and A are both environments mapping variables to types. Environment I gives the declared types of variables, and is used to determine the originally intended types of expressions. I is used only in conjunction with the correspondence relation. Environment A gives the actual types of variables, and corresponds to the "normal" kind of type environment found in typing rules.

The rule (mClosure), used only to type closures in subject-reduction, uses the notation $A \vdash M$, defined as follows.

Let $\tau = \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg$ $\quad \tau^{Meth} = \ll T,\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg$ in all the rules

(iVar) $\dfrac{I(x) = \sigma}{I \vdash x : \sigma}$

(iBool) $\dfrac{}{I \vdash b : Bool}$ (iNum) $\dfrac{}{I \vdash n : Nat}$

(iAbs) $\dfrac{I \cup \{x : \sigma\} \vdash e : \sigma'}{I \vdash \lambda x : \sigma.e : \sigma \to \sigma'}$ (iApp) $\dfrac{I \vdash e_1 : \sigma \to \sigma'}{I \vdash e_1\ e_2 : \sigma'}$

(iCond) $\dfrac{I \vdash e_3 : \sigma}{I \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \sigma}$

(iLet) $\dfrac{I \cup \{x : \sigma\} \vdash e' : \sigma'}{I \vdash \mathbf{let}\ x : \sigma = e\ \mathbf{in}\ e' : \sigma'}$

(iMsg) $\dfrac{I \vdash e :\ \mathbf{Obj}\ mt\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg}{I \vdash e \leftarrow m_k : \tau_k[\mathbf{Obj}\ mt\ \ll T,\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg\ /\ mt]}$

(iRead) $\dfrac{I \vdash e :\ \mathbf{Obj}\ mt\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg}{I \vdash e.y_j : \tau_j}$

(iWrite) $\dfrac{I \vdash e :\ \mathbf{Obj}\ mt\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg}{I \vdash e_1.y_j := e_2 :\ \mathbf{Obj}\ mt\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg}$

(iObj) $\dfrac{}{I \vdash \mathbf{obj}(s : \mathbf{Obj}\ mt\ \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k}\ :\ \mathbf{Obj}\ mt\ \tau}$

(iClass) $\dfrac{}{I \vdash \mathbf{class}(s : \mathbf{Obj}\ mt\ \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k}\ : \rho}$

where $\rho = \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : (\mathbf{Obj}\ mt\ \tau) \mapsto \tau_k[\mathbf{Obj}\ mt\ \tau^{Meth}/mt]} \gg$

(iNew) $\dfrac{I \vdash e : \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg}{I \vdash \mathbf{new}\ e :\ \mathbf{Obj}\ mt\ \ll T,\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg}$

(iMExt) $\dfrac{I \vdash e : \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg}{I \vdash e :: m_k : \tau'_k \to \tau_k}$

(iIExt) $\dfrac{I \vdash e : \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg}{I \vdash e :: y_j : \tau_j}$

Figure 7: The intended type extraction rules for LCP

Definition 3.4: A type assignment A is *adequate* for an environment M, written $A \vdash M$, if A and M list exactly the same variables, and $M(x) = v$ implies $A(x) = \tau$ for some τ such that $\vdash v : \tau$.

One unusual feature of LCP is the nature of type declarations. Code may be typechecked and used at a type other than the type it was declared at. This is apparent in the rules. By the (mVar) rule, variables are taken to have the type listed in A, not in I. The (mAbs) rule in effect ignores the declared type of the argument for its conclusion, but inserts it into I when typechecking the body. I is only used in conjunction with $\overset{Corr}{\longleftrightarrow}$ in rules (mMsg), (mRead), (mWrite), (mMext) and (mIExt).

It may appear that the types a programmer places in a program could be very far from the truth, but this is not the case: the (mObj) and (mClass) rules force objects to typecheck at the declared type of their self variable, and the (mLet) rule requires the expression e' to have the type it was declared to have. Still, the key difference of correspondence polymorphism is the fact that programs *can* be given types other than their declared type, much as in an implicitly typed language (a language where programs are not decorated with any type information). So, LCP is a hybrid of an explicitly- and implicitly-typed language.

The (mMsg), (mRead), and (mWrite) rules use the correspondence relation to determine method and instance variable lookup. In (mMsg), for instance, the message send $e \leftarrow m_k$ on an expression e of originally intended type T may currently be typechecking against an actual object named S. If that type has a method n_l, and $T.m_k \overset{Corr}{\longleftrightarrow} S.n_l$, then the message send typechecks thanks to the correspondence. Similarly, the rules (mMExt) and (MIExt) use the correspondence relation to determine which method is extracted.

In (mObj), a type **Obj** mt τ' can be proved for an object by typing its components according to τ', assuming self has intended type as declared (**Obj** mt τ) and actual type **Obj** mt τ'. Thus, the declared types of instance variables and methods can be ignored, much like the argument type in (mAbs). Note, however, that the conclusion type τ' and the declared type τ must have the same identifier T and the same instance variables and methods. The two premises in the second line of (mObj) force the object to typecheck at its declared type, even if this is not the actual consequence of the judgment.

The (mClass) rule spotlights the dual role of classes, as object factories and as collections of extractable pre-methods. Each method m_k in a class is typed as $m_k : \tau'_k \mapsto \tau_k$, where τ'_k is an arbitrary self, for use on extraction as a function from self. This is again analogous to (mAbs), where the parameter can be arbitrary. As in (mObj), the two premises in the second line force the class to typecheck at its declared type.

The (mNew) rule proves **new** e has the type **Obj** mt τ^{Meth} after verifying that e has a class type in which the type of self is **Obj** mt τ in all methods. The premise of (mNew) could also be written as $I, A \vdash e : (\textbf{Obj}\ mt\ \tau)^{Class}$. Note that the type of objects created via **new** hides their instance variables, whereas the type of objects created directly via **obj** exposes them (see (mObj)).

The rule (mExt) just turns $\tau'_k \mapsto \tau_k$ into a function type $\tau'_k \to \tau_k$. Instance variables behave similarly, except that they are extracted as is – not as functions from self.

The (mLet) rule copies the argument e' into the body, so the body will be re-type-checked at each occurrence. This is "poor mans let polymorphism", it requires code be re-type-checked instead of inferring a most general type once for e'. We plan on improving on this in future work. The third premise of this rules forces e' to typecheck at its declared type.

The system also contains a standard subsumption rule (mSub) which allows for subtype polymorphism.

Typing the integer-matrix example. Consider again the types *IntType* and *MatType* and the classes *intClass* and *matClass* of Fig. 3. We sketch how to prove $\vdash$ *matClass* : $MatType^{Class}$. Since the class *matClass* is defined as **let** integer:$IntType^{Class}$ = *intClass* **in**..., we should first establish the declared typing $\vdash$ *intClass* : $IntType^{Class}$ by (mLet), then verify that $\not\vdash$ *intClass* : $IntType^{Class}$, and finally prove that the **let** body, with the code for class *intClass* substituted for the identifier "integer", has type $MatType^{Class}$. The first requirement is straightforward, and the second follows immediately from (iClass), so we concentrate on the third. We need to show that

class (s:*MatType*)
  meth multMat =...,
    powerMat = *intClass*::power(s),
    isSingular =...

has type $MatType^{Class}$. For this we need to typecheck the methods assuming s has type *MatType* in both I and A. This is enough here since $MatType^{Class}$ is also the intended type of the class expression, so the premises for intended typing in (mClass) are the same as those for actual typing. We concentrate on proving $\{s : MatType\}, \{s : MatType\} \vdash intClass :: power(s) : Nat \to MatType$. By (mApp), this will be obtained if we show that $\vdash$ *intClass* :: *power* :

Let $\tau = \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg \qquad \tau^{Meth} = \ll T,\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg$ in all the rules

(mSub) $\dfrac{I, A \vdash e : \sigma \quad \sigma <: \sigma'}{I, A \vdash e : \sigma'}$ (mClosure) $\dfrac{I', A' \vdash e : \sigma \quad A' \vdash M'}{I, A \vdash (e, I', M') : \sigma}$

(mVar) $\dfrac{A(x) = \sigma}{I, A \vdash x : \sigma}$ (mBool) $\dfrac{}{I, A \vdash b : Bool}$ (mNum) $\dfrac{}{I, A \vdash n : Nat}$

(mAbs) $\dfrac{I \cup \{x : \rho\}, A \cup \{x : \sigma\} \vdash e : \sigma'}{I, A \vdash \lambda x : \rho . e : \sigma \to \sigma'}$ (mApp) $\dfrac{I, A \vdash e_1 : \sigma \to \sigma' \quad I, A \vdash e_2 : \sigma}{I, A \vdash e_1\ e_2 : \sigma'}$

(mCond) $\dfrac{I, A \vdash e_1 : Bool \quad I, A \vdash e_2 : \sigma \quad I, A \vdash e_3 : \sigma}{I, A \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \sigma}$

(mLet) $\dfrac{I \vdash\sim e' : \sigma' \quad I, A \vdash e[e'/x] : \sigma \quad I, A \vdash e' : \sigma'}{I, A \vdash \mathbf{let}\ x : \sigma' = e'\ \mathbf{in}\ e : \sigma}$

(mMsg) $\dfrac{\begin{array}{l} I \vdash\sim e :\ \mathbf{Obj}\ mt\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg \\ I, A \vdash e :\ \mathbf{Obj}\ mt\ \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg \quad T.m_k \overset{Corr}{\longleftrightarrow} S.n_l \end{array}}{I, A \vdash e \leftarrow m_k : \sigma_l[\mathbf{Obj}\ mt\ \ll S,\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg\ /\ mt]}$

(mRead) $\dfrac{\begin{array}{l} I \vdash\sim e :\ \mathbf{Obj}\ mt\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg \\ I, A \vdash e :\ \mathbf{Obj}\ mt\ \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg \quad T.y_j \overset{Corr}{\longleftrightarrow} S.x_i \end{array}}{I, A \vdash e.y_j : \sigma_i}$

(mWrite) $\dfrac{\begin{array}{l} I \vdash\sim e :\ \mathbf{Obj}\ mt\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg \\ I, A \vdash e :\ \mathbf{Obj}\ mt\ \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg \quad T.y_j \overset{Corr}{\longleftrightarrow} S.x_i \quad I, A \vdash e_2 : \sigma_i \end{array}}{I, A \vdash e_1.y_j := e_2 :\ \mathbf{Obj}\ mt\ \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg}$

(mObj) $\dfrac{\begin{array}{l} I, A \vdash \overline{e_j : \tau'_j} \quad I \cup \{s : \mathbf{Obj}\ mt\ \tau\}, A \cup \{s : \mathbf{Obj}\ mt\ \tau'\} \vdash \overline{e_k : \tau'_k[\mathbf{Obj}\ mt\ \tau'^{Meth}/mt]} \\ I, A \vdash \overline{e_j : \tau_j} \quad I \cup \{s : \mathbf{Obj}\ mt\ \tau\}, A \cup \{s : \mathbf{Obj}\ mt\ \tau\} \vdash \overline{e_k : \tau_k[\mathbf{Obj}\ mt\ \tau^{Meth}/mt]} \end{array}}{I, A \vdash \mathbf{obj}(s : \mathbf{Obj}\ mt\ \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k}\ :\ \mathbf{Obj}\ mt\ \tau'}$

where $\tau' = \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau'_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k} \gg$

(mClass) $\dfrac{\begin{array}{l} I, A \vdash \overline{e_j : \tau'_j} \quad \overline{I \cup \{s : \mathbf{Obj}\ mt\ \tau\}, A \cup \{s : \tau''_k\} \vdash e_k : \tau'_k} \\ I, A \vdash \overline{e_j : \tau_j} \quad I \cup \{s : \mathbf{Obj}\ mt\ \tau\}, A \cup \{s : \mathbf{Obj}\ mt\ \tau\} \vdash \overline{e_k : \tau_k[\mathbf{Obj}\ mt\ \tau^{Meth}/mt]} \end{array}}{I, A \vdash \mathbf{class}(s : \mathbf{Obj}\ mt\ \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k}\ :\ \rho}$

where $\rho = \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau'_j}\ \mathbf{Meth}\ \overline{m_k : \tau''_k \mapsto \tau'_k} \gg$

(mNew) $\dfrac{I, A \vdash e : \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \mathbf{Obj}\ mt\ \tau \mapsto \tau_k[\mathbf{Obj}\ mt\ \tau^{Meth}/mt]} \gg}{I, A \vdash \mathbf{new}\ e :\ \mathbf{Obj}\ mt\ \tau^{Meth}}$

(mMExt) $\dfrac{\begin{array}{l} I, A \vdash\sim e : \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg \\ I, A \vdash e : \mathbf{Class}\ \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma'_l \mapsto \sigma_l} \gg \quad T.m_k \overset{Corr}{\longleftrightarrow} S.n_l \end{array}}{I, A \vdash e :: m_k : \sigma'_l \to \sigma_l}$

(mIExt) $\dfrac{\begin{array}{l} I, A \vdash\sim e : \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg \\ I, A \vdash e : \mathbf{Class}\ \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma'_l \mapsto \sigma_l} \gg \quad T.y_j \overset{Corr}{\longleftrightarrow} S.x_i \end{array}}{I, A \vdash e :: y_j : \sigma_i}$

Figure 8: The typing rules for LCP

MatType → *Nat* → *MatType*. By (mMExt), this holds if we prove a class type for *intClass*, with power:*MatType* ↦ (*Nat*→*MatType*) inside. By the (mClass) rule, this requires showing that

λ(n:*Nat*). **if** *isZero*(n)
then s
else s ← mult(s ← power(*pred*(n))

has type *Nat*→*MatType* with assumptions s:*IntType* in I, and s:*MatType* in A (the (mClass) rule also requires other judgments to be proved, but they are all standard and straight forward, so we concentrate of this one). This in turn invokes (mAbs), which needs to typecheck the body with assumption n:*Nat* in both I and A. In the body, the message send s ← mult types as *MatType*→*MatType* by (mMsg) because s has intended type *IntType*, actual type *MatType*, Integer.mult $\overset{Corr}{\longleftrightarrow}$ Matrix.multMat, and multMat has type *MatType*→*MatType* in *MatType* (after substituting for mt). The rest of the body is typechecked similarly (recall that also Integer.power $\overset{Corr}{\longleftrightarrow}$ Matrix.powerMat), and we are done.

Having shown that $\vdash$ *matClass* : $MatType^{Class}$, proving

let matrix:$MatType^{Class}$ = *matClass*
in let o:*MatType* = **new** matrix
in o ← powerMat(5)

is typable is straightforward.

3.4 Operational Semantics and Soundness of the Type System

The type system importantly has a *subject reduction* property. The reduction system is given in natural semantics ("big-step") form in Appendix A, defining a relation $e \Downarrow_M^I v$ reducing expressions to values. The irreducible values of LCP are naturals, booleans, function closures, object closures, and class closures. In $e \Downarrow_M^I v$, the parameter I propagates the originally intended type information, mapping free variables to their intended types, and M is a standard environment which maps free variables in e to their values.

The reduction system uses the intended typings and the correspondence relation to compute appropriate message dispatch. In particular, if message send $e \leftarrow m$ is being computed and e is an object with identifier S but of originally intended object identifier T, and $T.m \overset{Corr}{\longleftrightarrow} S.n$, then $e \leftarrow m$ dispatches to method n in e. To perform this dispatch, each message send $e \leftarrow m$ for e with intended object identifier T is compiled with a lookup table, which maps S to n whenever $S.n \overset{Corr}{\longleftrightarrow} T.m$. At run-time, each object is labeled with its object type identifier S, and $e \leftarrow m$ looks up S in its table, finds n and invokes it. So, this notion of dispatch is in fact not significantly more inefficient than traditional dispatch. Besides the issue of method/instance dispatch via the correspondence relation, and method/instance extraction which also uses $\overset{Corr}{\longleftrightarrow}$, the operational semantics is completely standard.

The subject reduction property is as follows.

Theorem 3.5 (Subject Reduction): If $e \Downarrow_M^I v$ and $I, A \vdash e : \tau$ for some A such that $A \vdash M$ then $\vdash v : \tau$.

The proof appears in Appendix B.

4 Conclusions

We have defined a new class of polymorphism, correspondence polymorphism, and defined a simple language LCP to concretely illustrate the expressiveness of the concept. This paper represents a feasability study—LCP is still a toy language, but it illustrates well the fundamental new concepts proposed. Three contributions were made here to build a feasible programming language based on metaphors:

1. The notion of a correspondence relation, $\overset{Corr}{\longleftrightarrow}$: methods are considered "the same" when they are related via $\overset{Corr}{\longleftrightarrow}$, not when they circumstantially have the same name. Consequently, typing, subtyping, and the semantics of method dispatch all use $\overset{Corr}{\longleftrightarrow}$.

2. Type declarations as nonrestrictive but verified interfaces. Even though the language looks – and in many respects also behaves – like an explicitly-typed language, it can also be viewed as an implicitly-typed language in which declared types are verified to be valid (and, in which declared types are required in certain locations such as on all object and class declarations).

3. Fine-grained reuse: code is reused at the granularity of the method level rather than class level. This creates more opportunity for code reuse.

These three, plus a reuse mechanism (here as the less than ideal poor-man's let-polymorphism), work together to achieve type-safe reuse in non subtyping/subclassing contexts.

In the context of object-oriented languages with programmer type declarations, better code reuse can often be obtained either via universal polymorphic type declarations, or via correspondence polymorphism. There is however an essential difference between them: universal polymorphism con-

sists in writing *generic* code for universal applicability, foreseeing, so to speak, all its potential instantiations. Correspondence polymorphism, on the other hand, makes *specific* code reusable in new contexts. This dichotomy bears some resemblance to the difference between class-based and prototype-based languages. There is also an important cognitive analogy: universal polymorphism represents a cognitive activity of *abstraction*, whereas correspondence polymorphism is related to (and in fact motivated from) *metaphors* (see [17] where correspondence polymorphism was called "metaphoric polymorphism").

Correspondence polymorphism has practical advantages over expressive systems with declared universal polymorphic types [7, 12], in that declared types may still be simple (first-order) types that programmers can easily understand, and there is no need to plan ahead for reuse. It is also more flexible in that code can be reused even if some method names are inappropriate for the current context. Universal types have the advantages of straightforward modular typechecking, and of giving more precise programmer-defined interfaces. The latter serve as good documentation, and give a precise contract between clients and server objects. In particular, implementations can be altered as long as they still conform to the interfaces.

This last point appears to be one unavoidable disadvantage of correspondence polymorphism: it lessens the ability to "implement to interfaces", since an implementation needs to conform not only to its declared type, but also to other types, as required by the contexts in which it is embedded. For example, if we change the implementation of "power" in *intClass*, still preserving its declared interface, it may surprisingly affect *matClass* since the new implementation may not be appropriate in the matrix context. In other words, in LCP declared interfaces are sound client-side contracts, but server classes may need to do better. Still, such problems will be caught by the compiler, flagging them as type errors, and so this is not a new source of run-time errors.

LCP uses a nonstandard notion of run-time dispatch that requires run-time type information be kept, but this is mostly orthogonal to other generalized notions of dispatch such as that found in multimethods. The correspondence relation we define here is "context-free" in the sense that two methods in different objects are related always or never. If this were generalized to a correspondence relation in which correspondences could depend also on the (dynamic) types of *arguments* supplied to the method, features of multimethod dispatch (of the "encapsulated" variety [9]) could perhaps be modeled in the framework of correspondence polymorphism.

The most significant shortcoming of the current incarnation of LCP is the lack of a good type inference algorithm. We expect that a feasible algorithm exists for a variant of LCP, based on recursively constrained types, i.e. types which include sets of subtyping constraints [11, 2]. This is currently the most widespread school for type inference over languages with subtyping (flow analysis-based views [16] are also constraint-based). In this style of type system, let-polymorphism allows a least type to be given to reusable expressions. This should avoid the current need to re-typecheck existing code when using **let** in LCP. This proposed variant would thus have true parametric, subtype, and correspondence polymorphism. Users in such a language would still program with first-order types only: the inferred types would be verified to be more general than the declared types, and users would not need to see the inferred types (which in the case of constrained types can be quite hard to read).

References

[1] A. Abadi and L. Cardelli. *A theory of objects.* Springer-Verlag, 1996.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[3] R. Amadio and L. Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[4] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 285–298, 1993.

[5] K. Bruce. Typing in object-oriented languages: achieving expressiveness and safety. *Technical report, Williams College*, 1996.

[6] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group (Jonathan Eifrig, Scott Smith, and Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[7] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP '95*, 1995.

[8] Kim B. Bruce and Joseph C. Vanderwaart. Semantic-driven language design: type-safe virtual types in object-oriented languages. In *Proceedings of MFPS 99*, Electronic Notes in Theoretical Computer Science, to appear.

[9] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3), 1995.

[10] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation*, 8(4):357–397, 1995.

[11] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95*, pages 169–184, 1995.

[12] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico. Application of OOP type theory: State, decidability, integration. In *OOPSLA '94*, pages 16–30, 1994.

[13] O. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89 conference proceedings, ACM SIGPLAN Notices, 24(10)*, 1998.

[14] B. Meyer. *Eiffel, the Language.* Prentice Hall, 1992.

[15] B. Meyer. *Object-Oriented Software Construction, 2nd edition.* Prentice Hall, 1997.

[16] J. Palsberg and M. Schwartzbach. *Object-Oriented Type Systems.* Wiley, 1994.

[17] R. Rinat and M. Magidor. Metaphoric polymorphism: taking code reuse one step further. In *LNCS Vol. 1098: ECOOP '96 conference proceedings.* Springer-Verlag, 1996.

[18] Kresten Krab Thorup. Genericity in java with virtual types. In *LNCS Vol. 1241: ECOOP '97 conference proceedings.* Springer-Verlag, 1997.

A Operational Semantics

The operational semantics of LCP, presented in a natural semantics style, is shown in Fig. 9 and 10. Reductions have the form $e \Downarrow^I_M v$, where I is an intended type assignment, and M is an environment mapping variables to irreducible values. The rules also implicitly depend on a fixed correspondence relation $\overset{C \, or \, \tau}{\longleftrightarrow}$.

The rules for objects are in the spirit of Abadi and Cardelli's object calculus. Objects, classes and functions are represented as closures. The most novel aspect of the rules is how the correspondence relation is used to determine method dispatch in (Red Msg), (Red Read), and (Red Write). Intended type information I is carried along in this reduction system, but it is in fact statically determinable and the only information that needs to be carried at run-time is the actual type identifier of objects; with this information, none of the proof obligations below require actual proof at run-time, and dispatch can be computed in near-constant time.

In the operational semantics, the irreducible values v are precisely numbers, booleans, function closures, class closures, and object closures, where in the latter the component instances are also values.

B Subject Reduction

In this Appendix we establish the soundness of the type system by showing a subject reduction property holds.

Lemma B.1: if $I, A \vdash e : \tau$, $I \subseteq I'$, $A \subseteq A'$ then $I', A' \vdash e : \tau$.

Lemma B.2: Suppose $e' \Downarrow v'$ and $I \mid\sim e' : \tau'$. Then $e[e'/x] \Downarrow^I_M v$ iff $e \Downarrow^{I, x:\tau'}_{M, x \mapsto v'} v$.

Theorem B.3 (Subject Reduction): If $e \Downarrow^I_M v$ and $I, A \vdash e : \tau$ for some A such that $A \vdash M$ then $\vdash v : \tau$.

Proof. Suppose $e \Downarrow^I_M v$, $A \vdash M$, and $I, A \vdash e : \tau$. The proof is by induction on the derivation of $e \Downarrow^I_M v$. There is a case for each possible last step of the derivation.

(Red Var) Here $e = x$, so it must be that $M(x) = v$. The judgment $I, A \vdash x : \tau$ must have come from (mVar) with premise $A(x) = \sigma$ and conclusion $I, A \vdash x : \sigma$, followed by a number of subsumption steps implying $\sigma <: \tau$. Since $A \vdash M$ it must be that $\vdash v : \sigma$, so $\vdash v : \tau$ by subsumption.

(Red Abs) Here $e = \lambda x : \tau'.e' \Downarrow^I_M v$ for $(\lambda x : \tau'.e', I, M) = v$, so applying (mClosure) we obtain $\vdash (\lambda x : \tau'.e', I, M) : \tau$.

(Red App) In this case $e = e_1 \; e_2$, $e_1 \Downarrow^I_M (\lambda x : \tau'.e', I', M')$, $e_2 \Downarrow^I_M v'$, and $e' \Downarrow^{I', x:\tau'}_{M', x \mapsto v'} v$. The judgment $I, A \vdash e_1 \; e_2 : \tau$ must have come from (mApp) with premises $I, A \vdash e_1 : \sigma \to \sigma'$

(Red Obj) (where $o = \mathbf{obj}(s : \tau)\ \mathbf{inst}\ \overline{y_j = v_j}\ \mathbf{meth}\ \overline{m_k = e_k}$)

$$\frac{\overline{e_j \Downarrow_M^I v_j}}{\mathbf{obj}(s : \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k} \Downarrow_M^I (o, I, M)}$$

(Red Msg) (where $o = \mathbf{obj}(s : \mathbf{Obj}\ mt \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg)\ \mathbf{inst}\ \overline{x_i = v_i}\ \mathbf{meth}\ \overline{n_l = e_l}$)

$$\frac{\begin{array}{l} I \vdash e : \mathbf{Obj}\ mt \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg \\ e \Downarrow_M^I (o, I', M') \\ T.m_k \overset{Corr}{\longleftrightarrow} S.n_l \\ e_l \Downarrow_{M', s \mapsto (o, I', M')}^{I', s : \mathbf{Obj}\ mt \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg} v \end{array}}{e \leftarrow m_k \Downarrow_M^I v}$$

(Red Read) (where $o = \mathbf{obj}(s : \mathbf{Obj}\ mt \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg)\ \mathbf{inst}\ \overline{x_i = v_i}\ \mathbf{meth}\ \overline{n_l = e_l}$)

$$\frac{\begin{array}{l} I \vdash e : \mathbf{Obj}\ mt \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg \\ e \Downarrow_M^I (o, I', M') \\ T.y_j \overset{Corr}{\longleftrightarrow} S.x_i \end{array}}{e.y_j \Downarrow_M^I v_i}$$

(Red Write) (where $o = \mathbf{obj}(s : \sigma)\ \mathbf{inst}\ \overline{x_i = v_i}\ \mathbf{meth}\ \overline{n_l = e_l}$,
$\sigma = \mathbf{Obj}\ mt \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg$)

$$\frac{\begin{array}{l} I \vdash e_1 : \mathbf{Obj}\ mt \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg \\ e_1 \Downarrow_M^I (o, I', M') \\ T.y_j \overset{Corr}{\longleftrightarrow} S.x_i \\ e_2 \Downarrow_M^I v' \end{array}}{e_1.y_j := e_2 \Downarrow_M^I (o[x_i = v'], I', M')}$$

(Red Class) (where $c = \mathbf{class}(s : \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k}$)

$$\frac{}{c \Downarrow_M^I (c, I, M)}$$

(Red New)

$$\frac{\begin{array}{l} e \Downarrow_M^I (\mathbf{class}(s : \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k}, I', M') \\ \overline{e_j \Downarrow_{M'}^{I'} v_j} \end{array}}{\mathbf{new}\ e \Downarrow_M^I (\mathbf{obj}(s : \tau)\ \mathbf{inst}\ \overline{y_j = v_j}\ \mathbf{meth}\ \overline{m_k = e_k}, I', M')}$$

Figure 9: Reduction rules for LCP

(Red MExt) (where $c = \mathbf{class}(s : \mathbf{Obj}\ mt\ \sigma)\ \mathbf{inst}\ \overline{x_i = e_i}\ \mathbf{meth}\ \overline{n_l = e_l}$,
$\sigma = \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg$)

$$\frac{\begin{array}{l} I \vdash e : \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg \\ e \Downarrow^I_M (c, I', M') \\ T.m_k \overset{Corr}{\longleftrightarrow} S.n_l \end{array}}{e :: m_k \Downarrow^I_M (\lambda s : (\mathbf{Obj}\ mt\ \sigma).e_l, I', M')}$$

(Red IExt) (where $c = \mathbf{class}(s : \mathbf{Obj}\ mt\ \sigma)\ \mathbf{inst}\ \overline{x_i = e_i}\ \mathbf{meth}\ \overline{n_l = e_l}$,
$\sigma = \ll S,\ \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg$)

$$\frac{\begin{array}{l} I \vdash e : \mathbf{Class}\ \ll T,\ \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg \\ e \Downarrow^I_M (c, I', M') \\ T.y_j \overset{Corr}{\longleftrightarrow} S.x_i \\ e_i \Downarrow^{I'}_{M'} v \end{array}}{e :: y_j \Downarrow^I_M v}$$

(Red Abs) $$\frac{}{\lambda x : \tau.e \Downarrow^I_M (\lambda x : \tau.e, I, M)}$$

(Red App) $$\frac{e_1 \Downarrow^I_M (\lambda x : \tau.e, I', M') \quad e_2 \Downarrow^I_M v' \quad e \Downarrow^{I', x:\tau}_{M', x \mapsto v'} v}{e_1\ e_2 \Downarrow^I_M v}$$

(Red Let) $$\frac{e' \Downarrow^I_M v' \quad e \Downarrow^{I, x:\tau}_{M, x \mapsto v'} v}{\mathbf{let}\ x : \tau = e'\ \mathbf{in}\ e \Downarrow^I_M v}$$

(Red Var) $$\frac{M(x) = v}{x \Downarrow^I_M v}$$

(Red CTrue) $$\frac{e_1 \Downarrow^I_M true \quad e_2 \Downarrow^I_M v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow^I_M v}$$

(Red CFalse) $$\frac{e_1 \Downarrow^I_M false \quad e_3 \Downarrow^I_M v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow^I_M v}$$

(Red Nat) (where $n = 0, 1, 2, \ldots$)

$$\frac{}{n \Downarrow^I_M n}$$

(Red Bool) (where $b = true, false$)

$$\frac{}{b \Downarrow^I_M b}$$

Figure 10: Reduction rules for LCP - continued

and $I, A \vdash e_2 : \sigma$, and conclusion $I, A \vdash \sigma'$, followed by a number of subsumption steps implying $\sigma' <: \tau$. By the induction hypothesis we have $\vdash (\lambda x : \tau'.e', I', M') : \sigma \to \sigma'$. This must have come from (mClosure) with premise $I', A' \vdash \lambda x : \tau'.e' : \rho \to \rho'$ for some $A' \vdash M'$ and conclusion $\vdash (\lambda x : \tau'.e', I', M') : \rho \to \rho'$, followed by some subsumptions implying $\sigma <: \rho$ and $\rho' <: \sigma'$. This in turn was obtained by (mAbs) with premise $I' \cup \{x : \tau'\}, A' \cup \{x : \mu\} \vdash e' : \mu'$ and conclusion $I', A' \vdash \lambda x : \tau'.e' : \mu \to \mu'$, followed by some subsumptions implying $\rho <: \mu$ and $\mu' <: \rho'$. By the induction hypothesis we have $\vdash v' : \sigma$, so by subsumption also $\vdash v' : \rho$. Thus $A', x : \rho \vdash M', x : v'$. From the induction hypothesis it follows that $\vdash v : \mu'$, and since $\mu' <: \rho' <: \sigma' <: \tau$ we obtain $\vdash v : \tau$ by subsumption.

(Red Let) Here $e = \textbf{let } x : \sigma' = e_2 \textbf{ in } e_1$, $e_2 \Downarrow^I_M v'$, and $e_1 \Downarrow^{I, x:\tau'}_{M, x \mapsto v'} v$. The judgment $I, A \vdash e : \tau$ must have come from (mLet) with premises $I \vdash\sim e_2 : \sigma'$ and $I, A \vdash e_1[e_2/x] : \sigma$, and conclusion $I, A \vdash e : \sigma$, followed by some subsumptions implying $\sigma <: \tau$. By Lemma B.2 we have $e_1[e_2/x] \Downarrow^I_M v$, so it follows from the induction hypothesis that $\vdash v : \sigma$. Thus $\vdash v : \tau$ by subsumption.

(Red Obj) Here $e = \textbf{obj}(s : \textbf{Obj } mt\ \sigma)\ \textbf{inst } \overline{y_j = e_j}\ \textbf{meth } \overline{m_k = e_k} \Downarrow^I_M (o, I, M) = v$ and $e_j \Downarrow^I_M v_j$, where $o = \textbf{obj}(s : \textbf{Obj } mt\ \sigma)\ \textbf{inst } \overline{y_j = v_j}\ \textbf{meth } \overline{m_k = e_k}$ and $\sigma = \ll T, \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau_k} \gg$. The judgment $I, A \vdash e : \tau$ must have come from (mObj) with premises (among others) $I, A \vdash \overline{e_j : \tau'_j}$ and $I \cup \{s : \textbf{Obj } mt\ \sigma\}, A \cup \{s : \textbf{Obj } mt\ \sigma'\} \vdash \overline{e_k : \tau'_k[\textbf{Obj } mt\ \sigma'^{Meth}/mt]}$, and the conclusion $I, A \vdash e : \textbf{Obj } mt\ \sigma'$, followed by a number of subsumption steps implying $\textbf{Obj } mt\ \sigma' <: \tau$ (where $\sigma' = \ll T, \textbf{Inst } \overline{y_j : \tau'_j}\ \textbf{Meth } \overline{m_k : \tau'_k} \gg$). By the hypothesis we have $\vdash \overline{v_j : \tau'_j}$, so by Lemma B.1 also $I, A \vdash \overline{v_j : \tau'_j}$. Thus $I, A \vdash o : \textbf{Obj } mt\ \sigma'$ by (mObj). By subsumption we have $I, A \vdash o : \tau$, so applying (mClosure) we obtain $\vdash v : \tau$.

(Red Msg) In this case $e = e' \leftarrow m_k \Downarrow^I_M v$, $I \vdash\sim e' : \textbf{Obj } mt \ll T, \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau_k} \gg$, $e' \Downarrow^I_M (o, I', M')$, $T.m_k \overset{Corr}{\longleftrightarrow} S.n_l$, and $e_l \Downarrow^{I', s:\textbf{Obj } mt\ \sigma}_{M', s \mapsto (o, I', M')} v$, where $o =$

$\textbf{obj}(s : \textbf{Obj } mt\ \sigma)\ \textbf{inst } \overline{x_i = v_i}\ \textbf{meth } \overline{n_l = e_l}$,

$\sigma = \ll S, \textbf{Inst } \overline{x_i : \sigma_i}\ \textbf{Meth } \overline{n_l : \sigma_l} \gg$. The judgment $I, A \vdash e' \leftarrow m_k : \tau$ must have come from (mMsg) with premises $I \vdash\sim e' : \textbf{Obj } mt \ll T, \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau_k} \gg$, $I, A \vdash e' : \textbf{Obj } mt\ \sigma'$, and $T.m_k \overset{Corr}{\longleftrightarrow} S'.n'_{l'}$, and conclusion $e' \leftarrow m_k : \sigma'_{l'}[\textbf{Obj } mt\ \sigma'^{Meth}/mt]$, followed by a number of subsumptions implying $\sigma'_{l'}[\textbf{Obj } mt\ \sigma'^{Meth}/mt] <: \tau$ (where $\sigma' = \ll S', \textbf{Inst } \overline{x'_{i'} : \sigma'_{i'}}\ \textbf{Meth } \overline{n'_{l'} : \sigma'_{l'}} \gg$). By the induction hypothesis we have $\vdash (o, I', M') : \textbf{Obj } mt\ \sigma'$. This came from (mClosure) with premise $I', A' \vdash o : \textbf{Obj } mt\ \sigma''$ for some $A' \vdash M'$, followed by a number of subsumptions implying $\textbf{Obj } mt\ \sigma'' <: \textbf{Obj } mt\ \sigma'$ (where $\sigma'' = \ll S''$, $\textbf{Inst } \overline{x''_{i''} : \sigma''_{i''}}\ \textbf{Meth } \overline{n''_{l''} : \sigma''_{l''}} \gg$). Since o explicitly mentions σ, this must have come from (mObj) with premise (among others) $I' \cup \{s : \textbf{Obj } mt\ \sigma\}, A' \cup \{s : \textbf{Obj } mt\ \tilde{\sigma}\} \vdash e_l : \tilde{\sigma}_l[\textbf{Obj } mt\ \tilde{\sigma}^{Meth}/mt]$ and conclusion $I', A' \vdash o : \textbf{Obj } mt\ \tilde{\sigma}$, followed by some subsumptions implying $\textbf{Obj } mt\ \tilde{\sigma} <: \textbf{Obj } mt\ \sigma''$, where $\tilde{\sigma} = \ll S$, $\textbf{Inst } \overline{x_i : \tilde{\sigma}_i}\ \textbf{Meth } \overline{n_l : \tilde{\sigma}_l} \gg$. By (mClosure) we have $\vdash (o, I', M') : \textbf{Obj } mt\ \tilde{\sigma}$, so $A', s : \textbf{Obj } mt\ \tilde{\sigma} \vdash M', s \mapsto (o, I', M')$. By the induction hypothesis we have $\vdash v : \tilde{\sigma}_l[\textbf{Obj } mt\ \tilde{\sigma}^{Meth}/mt]$. Since $\textbf{Obj } mt\ \tilde{\sigma} <: \textbf{Obj } mt\ \sigma'' <: \textbf{Obj } mt\ \sigma'$ and $S.n_l \overset{Corr}{\longleftrightarrow} T.m_k \overset{Corr}{\longleftrightarrow} S'.n'_{l'}$ we have $\tilde{\sigma}_l[\textbf{Obj } mt\ \tilde{\sigma}^{Meth}/mt] <: \sigma'_{l'}[\textbf{Obj } mt\ \sigma'^{Meth}/mt]$ by Lemma 3.1. Combining it with $\sigma'_{l'}[\textbf{Obj } mt\ \sigma'^{Meth}/mt] <: \tau$ we obtain $\vdash v : \tau$ by subsumption.

(Red Read) In this case $e = e'.y_j \Downarrow^I_M v_i = v$, $I \vdash\sim e' : \textbf{Obj } mt \ll T, \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau_k} \gg$, $e' \Downarrow^I_M (o, I', M')$, and $T.y_j \overset{Corr}{\longleftrightarrow} S.x_i$, where o is

$\textbf{obj}(s : \textbf{Obj } mt\ \sigma)\ \textbf{inst } \overline{x_i = v_i}\ \textbf{meth } \overline{n_l = e_l}$,

$\sigma = \ll S, \textbf{Inst } \overline{x_i : \sigma_i}\ \textbf{Meth } \overline{n_l : \sigma_l} \gg$. The judgment $I, A \vdash e'.y_j : \tau$ must have come from (mRead) with premises $I \vdash\sim e' : \textbf{Obj } mt \ll T, \textbf{Inst } \overline{y_j : \tau_j}\ \textbf{Meth } \overline{m_k : \tau_k} \gg$, $I, A \vdash e' : \textbf{Obj } mt\ \sigma'$, and $T.y_j \overset{Corr}{\longleftrightarrow} S'.x'_{i'}$, and conclusion $e'.y_j : \sigma'_{i'}$, followed by a number of subsumptions implying $\sigma'_{i'} <: \tau$ (where $\sigma' = \ll S'$, $\textbf{Inst } \overline{x'_{i'} : \sigma'_{i'}}\ \textbf{Meth } \overline{n'_{l'} : \sigma'_{l'}} \gg$). By the induction hypothesis we have $\vdash (o, I', M') : \textbf{Obj } mt\ \sigma'$. This came from (mClosure) with premise $I', A' \vdash o : \textbf{Obj } mt\ \sigma''$ for some $A' \vdash M'$, followed by a number of subsumptions implying $\textbf{Obj } mt\ \sigma'' <: \textbf{Obj } mt\ \sigma'$ (where $\sigma'' = \ll S''$, $\textbf{Inst } \overline{x''_{i''} : \sigma''_{i''}}\ \textbf{Meth } \overline{n''_{l''} : \sigma''_{l''}} \gg$). Since o explicitly mentions σ, this must have come from (mObj) with premise (among others) $I', A' \vdash e_i : \tilde{\sigma}_i$ and conclusion $I', A' \vdash o : \textbf{Obj } mt\ \tilde{\sigma}$, followed by some subsumptions

implying $\mathbf{Obj}\ mt\ \tilde{\sigma} <: \mathbf{Obj}\ mt\ \sigma''$, where $\tilde{\sigma} =\ll S, \mathbf{Inst}\ \overline{x_i : \tilde{\sigma}_i}\ \mathbf{Meth}\ \overline{n_l : \tilde{\sigma}_l} \gg$. By the induction hypothesis we have $\vdash v_i : \tilde{\sigma}_i$. Since $\mathbf{Obj}\ mt\ \tilde{\sigma} <: \mathbf{Obj}\ mt\ \sigma'' <: \mathbf{Obj}\ mt\ \sigma'$ and $S.x_i \overset{Corr}{\longleftrightarrow} T.y_j \overset{Corr}{\longleftrightarrow} S'.x'_{i'}$ we have $\tilde{\sigma}_i <: \sigma'_{i'}$ by Lemma 3.1. Combining it with $\sigma'_{i'} <: \tau$ we obtain $\vdash v = v_i : \tau$ by subsumption.

(Red Write) In this case $e = e_1.y_j := e_2 \Downarrow^I_M (o[x_i = v']), I', M') = v$,

$$I \vdash\sim e_1 : \mathbf{Obj}\ mt \ll T, \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg,$$

$e_1 \Downarrow^I_M (o, I', M')$, $T.y_j \overset{Corr}{\longleftrightarrow} S.x_i$, and $e_2 \Downarrow^I_M v'$, where o is

$$\mathbf{obj}(s : \mathbf{Obj}\ mt\ \sigma)\ \mathbf{inst}\ \overline{x_i = v_i}\ \mathbf{meth}\ \overline{n_l = e_l},$$

$\sigma =\ll S, \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg$. The judgment $I, A \vdash e_1.y_j := e_2 : \tau$ must have come from (mWrite) with premises $I \vdash\sim e_1 : \mathbf{Obj}\ mt \ll T, \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg$, $I, A \vdash e_1 : \mathbf{Obj}\ mt\ \sigma'$, $T.y_j \overset{Corr}{\longleftrightarrow} S'.x'_{i'}$, and $I, A \vdash e_2 : \sigma'_{i'}$, and conclusion $e_1.y_j := e_2 : \mathbf{Obj}\ mt\ \sigma'$, followed by a number of subsumptions implying $\mathbf{Obj}\ mt\ \sigma' <: \tau$ (where $\sigma' =\ll S', \mathbf{Inst}\ \overline{x'_{i'} : \sigma'_{i'}}\ \mathbf{Meth}\ \overline{n'_{l'} : \sigma'_{l'}} \gg$). By the induction hypothesis we have $\vdash (o, I', M') : \mathbf{Obj}\ mt\ \sigma'$ and also $\vdash v' : \sigma'_{i'}$. The former must have come from (mClosure) with premise $I', A' \vdash o : \mathbf{Obj}\ mt\ \sigma''$ for some $A' \vdash M'$, followed by a number of subsumptions implying $\mathbf{Obj}\ mt\ \sigma'' <: \mathbf{Obj}\ mt\ \sigma'$ (where $\sigma'' =\ll S'', \mathbf{Inst}\ \overline{x''_{i''} : \sigma''_{i''}}\ \mathbf{Meth}\ \overline{n''_{l''} : \sigma''_{l''}} \gg$). Since o explicitly mentions σ, this must have come from (mObj) with premise (among others) $I', A' \vdash e_i : \tilde{\sigma}_i$ and conclusion $I', A' \vdash o : \mathbf{Obj}\ mt\ \tilde{\sigma}$, followed by some subsumptions implying $\mathbf{Obj}\ mt\ \tilde{\sigma} <: \mathbf{Obj}\ mt\ \sigma''$, where $\tilde{\sigma} =\ll S, \mathbf{Inst}\ \overline{x_i : \tilde{\sigma}_i}\ \mathbf{Meth}\ \overline{n_l : \tilde{\sigma}_l} \gg$. By the induction hypothesis we have $\vdash v_i : \tilde{\sigma}_i$. Since $\mathbf{Obj}\ mt\ \tilde{\sigma} <: \mathbf{Obj}\ mt\ \sigma'' <: \mathbf{Obj}\ mt\ \sigma'$ and $S.x_i \overset{Corr}{\longleftrightarrow} T.y_j \overset{Corr}{\longleftrightarrow} S'.x'_{i'}$ we have $\sigma'_{i'} <: \tilde{\sigma}_i$ by Lemma 3.1, so from $\vdash v' : \sigma'_{i'}$ we obtain $\vdash v' : \tilde{\sigma}_i$ by subsumption. Applying (mObj) we obtain $I', A' \vdash o[x_i = v'] : \mathbf{Obj}\ mt\ \tilde{\sigma}$, and so $I', A' \vdash o[x_i = v'] : \tau$ by subsumption. Applying (mClosure) we obtain $\vdash v = (o[x_i = v'], I', M') : \tau$.

(Red Class) Here $e = \mathbf{class}(s : \tau')\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k} \Downarrow^I_M (\mathbf{class}(s : \tau')\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k}, I, M) = v$. By applying (mClosure) we obtain $\vdash v : \tau$.

(Red New) Here $e = \mathbf{new}\ e' \Downarrow^I_M (o, I', M') = v$, $e' \Downarrow^I_M c$, and $\overline{e_j \Downarrow^{I'}_{M'} v_j}$, where $o = \mathbf{obj}(s : \mathbf{Obj}\ mt\ \sigma)\ \mathbf{inst}\ \overline{y_j = v_j}\ \mathbf{meth}\ \overline{m_k = e_k}$, $c = \mathbf{class}(s : \mathbf{Obj}\ mt\ \sigma)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = e_k}$, and $\sigma =\ll T, \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau_k} \gg$. The judgment $I, A \vdash \mathbf{new}\ e' : \tau$ must have come from (mNew) with premise $I, A \vdash e' : (\mathbf{Obj}\ mt\ \nu)^{Class}$ and conclusion $I, A \vdash \mathbf{new}\ e' : \mathbf{Obj}\ mt\ \nu^{Meth}$, followed by some subsumptions implying $\mathbf{Obj}\ mt\ \nu^{Meth} <: \tau$, where ν is some InstMeth type. By the induction hypothesis we have $\vdash (c, I', M') : (\mathbf{Obj}\ mt\ \nu)^{Class}$. This must have come from (mClosure) with premise $I', A' \vdash c : \mu$ for some $A' \vdash M'$ and class type μ, followed by a number of subsumptions implying $\mu <: (\mathbf{Obj}\ mt\ \nu)^{Class}$. This must have come from (mClass) with premises (among others)

$$I', A' \vdash \overline{e_j : \tau'_j}, \qquad I' \cup \{s : \mathbf{Obj}\ mt\ \sigma\},\ A' \cup \{s : \mathbf{Obj}\ mt\ \sigma'\} \vdash \overline{e_k : \tau'_k[\mathbf{Obj}\ mt\ \sigma'^{Meth}/mt]}$$

and conclusion $I', A' \vdash c : (\mathbf{Obj}\ mt\ \sigma')^{Class}$ where $\sigma' =\ll T, \mathbf{Inst}\ \overline{y_j : \tau'_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k} \gg$, followed by some subsumptions implying $(\mathbf{Obj}\ mt\ \sigma')^{Class} <: \mu$. By the induction hypothesis we have $I', A' \vdash \overline{v_j : \tau'_j}$, so $I', A' \vdash o : \mathbf{Obj}\ mt\ \sigma'$ by (mObj). From

$$(\mathbf{Obj}\ mt\ \sigma')^{Class} <: \mu <: (\mathbf{Obj}\ mt\ \nu)^{Class}$$

and Lemma 3.1 it follows that

$$\mathbf{Obj}\ mt\ \sigma' <: \mathbf{Obj}\ mt\ \nu <: \mathbf{Obj}\ mt\ \nu^{Meth},$$

and combining it with $\mathbf{Obj}\ mt\ \nu^{Meth} <: \tau$ we obtain $I', A' \vdash o : \tau$ by subsumption. Applying (mClosure) we obtain $\vdash v : \tau$.

(Red MExt) Here $e = e' :: m_k \Downarrow^I_M \lambda s : (\mathbf{Obj}\ mt\ \sigma).e_l, I', M') = v, I \vdash\sim e : \mathbf{Class} \ll T, \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg$, $e' \Downarrow^I_M (c, I', M')$, and $T.m_k \overset{Corr}{\longleftrightarrow} S.n_l$, where $c = \mathbf{class}(s : \mathbf{Obj}\ mt\ \sigma)\ \mathbf{inst}\ \overline{x_i = e_i}\ \mathbf{meth}\ \overline{n_l = e_l}$, $\sigma =\ll S, \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg$. The judgment $I, A \vdash e' :: m_k : \tau$ must have come from (mMExt) with premises $I \vdash\sim e' : \mathbf{Class} \ll T, \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg$, $I, A \vdash e' : \sigma'$, and $T.m_k \overset{Corr}{\longleftrightarrow} S'.n'_{l'}$, where $\sigma' =$

$$\mathbf{Class} \ll S', \mathbf{Inst}\ \overline{x'_{i'} : \sigma'_{i'}}\ \mathbf{Meth}\ \overline{n'_{l'} : \sigma''_{l'} \mapsto \sigma'_{l'}} \gg,$$

and conclusion $I, A \vdash e' :: m_k : \sigma''_{l'} \to \sigma'_{l'}$ followed by some subsumptions implying $\sigma''_{l'} \to \sigma'_{l'} <: \tau$. From the induction hypothesis it follows that $\vdash (c, I', M') : \sigma'$. This must have come from (mClosure) with premises $I', A' \vdash$

$c : \mu$ for some $A' \vdash M'$, where μ is some class type, followed by a number of subsumptions implying $\mu <: \sigma'$. This must have come from (mClass) with premise (among others) $I' \cup \{s : \mathbf{Obj}\ mt\ \sigma\}, A' \cup \{s : \nu'_l\} \vdash e_l : \nu_l$, and conclusion $I', A' \vdash c : \nu$, where $\nu = \mathbf{Class} \ll S, \mathbf{Inst}\ \overline{x_i : \nu_i}\ \mathbf{Meth}\ \overline{n_l : \nu'_l \mapsto \nu_l} \gg$ followed by some subsumptions implying $\nu <: \mu$. Thus we have $I', A' \vdash \lambda s : (\mathbf{Obj}\ mt\ \sigma).e_l : \nu'_l \to \nu_l$ by (mAbs). Since $\nu <: \mu <: \sigma'$ and $S.n_l \overset{Corr}{\longleftrightarrow} T.m_k \overset{Corr}{\longleftrightarrow} S'.n'_{l'}$ we have $\nu'_l \to \nu_l <: \sigma'_{l'} \to \sigma_{l'}$ by Lemma 3.1, and combining it with $\sigma'_{l'} \to \sigma_{l'} <: \tau$ we obtain $\lambda s : (\mathbf{Obj}\ mt\ \sigma).e_l : \tau$ by subsumption. By applying (mClosure) we obtain $\vdash v : \tau$.

(Red IExt) Here $e = e' :: y_j \Downarrow^I_M v$, $I \mid\sim e : \mathbf{Class} \ll T, \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg$, $e' \Downarrow^I_M (c, I', M')$, $T.y_j \overset{Corr}{\longleftrightarrow} S.x_i$, and $e_i \Downarrow^I_M v$, where $c = \mathbf{class}(s : \mathbf{Obj}\ mt\ \sigma)\ \mathbf{inst}\ \overline{x_i = e_i}\ \mathbf{meth}\ \overline{n_l = e_l}$,

$$\sigma = \ll S, \mathbf{Inst}\ \overline{x_i : \sigma_i}\ \mathbf{Meth}\ \overline{n_l : \sigma_l} \gg .$$

The judgment $I, A \vdash e' :: y_j : \tau$ must have come from (mIExt) with premises $I \mid\sim e' : \mathbf{Class} \ll T, \mathbf{Inst}\ \overline{y_j : \tau_j}\ \mathbf{Meth}\ \overline{m_k : \tau'_k \mapsto \tau_k} \gg$, $I, A \vdash e' : \sigma'$, and $T.y_j \overset{Corr}{\longleftrightarrow} S'.x'_{i'}$, where $\sigma' =$

$$\mathbf{Class} \ll S', \mathbf{Inst}\ \overline{x'_{i'} : \sigma'_{i'}}\ \mathbf{Meth}\ \overline{n'_{l'} : \sigma''_{l'} \mapsto \sigma'_{l'}}] \gg$$

followed by some subsumptions implying $\sigma'_{i'} <: \tau$. From the induction hypothesis it follows that $\vdash (c, I', M') : \sigma'$. This must have come from (mClosure) with premises $I', A' \vdash c : \mu$ for some $A' \vdash M'$, where μ is some class type, followed by a number of subsumptions implying $\mu <: \sigma'$. This must have come from (mClass) with premise (among others) $I', A' \vdash e_i : \nu_i$, and conclusion $I', A' \vdash c : \nu$, where $\nu = \mathbf{Class} \ll S, \mathbf{Inst}\ \overline{x_i : \nu_i}\ \mathbf{Meth}\ \overline{n_l : \nu'_l \mapsto \nu_l} \gg$ followed by some subsumptions implying $\nu <: \mu$. By the induction hypothesis we have $\vdash v : \nu_i$. Since $\nu <: \mu <: \sigma'$ and $S.x_i \overset{Corr}{\longleftrightarrow} T.y_j \overset{Corr}{\longleftrightarrow} S'.x'_{i'}$ we have $\nu_i <: \sigma'_{i'}$ by Lemma 3.1, and combining it with $\sigma'_{i'} <: \tau$ we obtain $\vdash v : \tau$ by subsumption.

(Red CTrue) Here $e = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$, $e_1 \Downarrow^I_M true$, and $e_2 \Downarrow^I_M v$. The judgment $I, A \vdash e : \tau$ must have come from (mCond) with premise (among others) $I, A \vdash e_2 : \sigma$ and conclusion $I, A \vdash e : \sigma$, followed by some subsumptions implying $\sigma <: \tau$. It follows from the induction hypothesis that $\vdash v : \sigma$, so $\vdash v : \tau$ by subsumption.

(Red CFalse) Similar to (Red CTrue).

(Red Nat) Trivial.

(Red Bool) Trivial.