Type-Safe Covariant Specialization with Generalized Matching

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

December 10, 1999

Abstract

Matching is a relation between object types originally designed to support type-safe subclassing with binary methods. In this paper we generalize it to deal with general covariance by allowing field and method parameter types to change covariantly in matching types. The result is a significant increase in the flexibility of subclassing. The generalization is in a different direction than the recent proposal for supporting type-safe virtual types via matching between mutually recursive types. The new definition is accompanied by appropriate machinery, allowing the loss of subsumption to be compensated by match-bounded parameterization and hash types, like in the language $\mathcal{LOOM}$. As a bonus of the generalized approach to matching we also obtain a more flexible typing for binary methods, allowing instances of a subclass to be supplied as argument. We define a language LGM supporting generalized matching, its typing rules, and its operational semantics, and state a subject reduction theorem showing the soundness of the system.

1 Introduction

The problem of covariant specialization in object-oriented languages has been a research issue for about a decade now. Both common sense and practical experience suggest that a specialized class should have specialized fields and (signature-wise) specialized methods. Yet, in a static typing setting, there seems to be some conflict between this rationale and what may be called "subclass polymorphism": the principle that instances of a subclass be accepted where the type of objects generated by the class itself is expected. The essential problem is that these expectations include the range of values on which the methods can operate, and specializing their argument types covariantly in subclasses means reducing that range, thereby violating the expectations. In type-theoretic terminology, such subclasses do not generate subtypes.

Two principled approaches have emerged to deal with this difficulty: Castagna's encapsulated multi-methods [9, 5, 10] (which are the object-oriented version of multi-methods *à la CLOS*, see [5]) and Bruce's matching [3, 7, 5] (an approach of *precise typing* has also been proposed, see [5]). Multi-methods settle the conflict by distinguishing between subtyping and specialization. The approach argues that method specialization does not consist in *replacing* old implementations by new ones, but in *adding* new covariantly-specialized code as branches of the same (multi-)method. Consequently, a specialized subclass retains all the capabilities of the class itself, and one obtains a system in which subclasses generate subtypes. The multi-methods approach therefore fully supports covariant specialization and subclass substitutability within a type-safe language. It does not, however, support field specialization. The approach also implies a significant shift in the semantics of method dispatch (i.e. from single- to multi-dispatch) vis-a-vis common OO languages such as C++, Java and Eiffel. Boyland and Castagna [2] have shown how to typecheck such languages, assuming the necessary alterations to their semantics have been made.

The matching approach, on the other hand, remains within the school of traditional single-dispatch. The essential wisdom here is to support a degree of covariant specialization at the price of some loss in subclass substitutability. The idea is to define a weaker relation than subtyping – namely matching – and to require that subclasses generate matching types rather than subtypes. The difference between matching and subtyping is that the former can still hold between object types containing binary methods ([5]). Consequently, it becomes possible to extend classes containing binary methods, which is not the case if subclasses are required to generate subtypes. As as penalty, the typechecker can no longer allow subclass substitutability in general. This can, however, be compensated by the use of match-bounded parameterization as in PolyToil [7] and $\mathcal{LOOM}$ [6]. The latter also supports *hash types*, which provide some more compensation up to the degree that matching could perhaps replace subtyping.

A recent development in this direction generalizes matching to groups of mutually recursive types [8]. It requires the programmer to introduce a new level of abstraction – *type groups* – mutually defining several types, possibly using one-another in negative positions. These groups can then be specialized as a whole, thereby covariantly specializing the participating types all at once. This construct was conceived as a type-safe counterpart of Beta's virtual classes [11], recently also proposed for Java as virtual *types* [14]. As such, this solution does not directly address the general covariance problem because many cases in which one type appears negatively in another are not cases of mutually recursive types, so defining them as a group is not natural. This point is recognized by the authors of [8], who suggest to use parameterized classes and types for these (rather frequent) cases.

In this paper we directly address the general covariance problem within the school of single dispatch. We generalize matching in a different direction than [8] by allowing fields and method parameters to change covariantly in matching types. The result is a significant increase in the flexibility of subclassing. Using appropriate machinery - variance annotations and role types – we are able to compensate for the loss of subclass substitutability via match-bounded parameterization and hash types, as in $\mathcal{LOOM}$. Variance annotations were introduced by Abadi and Cardelli [1] in the context of subtyping, and role types generalize Eiffel's "anchored types" [12], but here we put them together to work in a new framework. As a bonus of the generalized approach to matching, we are also able to obtain a more flexible typing for binary methods, allowing instances of a subclass to be supplied as argument. We define a language LGM supporting generalized matching, its typing rule, and its operational semantics. We then state a subject reduction property for the system, showing it is sound.

The paper is organized as follows: Sec. 2 recalls binary methods and matching, and presents the general covariance problem addressed in this paper, Sec. 3 explains the ideas and introduces the essential judgment rules informally via examples, Sec. 4 defines the language, its type system, and operational semantics, followed by a statement of subject reduction, and Sec. 5 concludes with future work.

2 The Problem: From Binary Methods to General Covariance

Binary methods, i.e. methods that require an argument of the same type as that of self, are known to be problematic in statically-typed object-oriented languages (see [5]). The following point and color-point types are a standard example (they are given here in an informal version of LGM. In **Obj** st ..., the type variable st stands for the type of self (sometimes denoted by a special keyword like "SelfType" or "MyType")):

$$\begin{array}{lll} \mathit{Point} & \triangleq & \mathbf{Obj}\ \mathrm{st}\ \mathbf{Inst}\ \mathrm{y}\colon \mathbf{Nat} \\ & & \qquad\quad \mathbf{Meth}\ \mathrm{equal}\colon \mathrm{st} \to \mathbf{Bool},\ \mathrm{move}\colon \mathbf{Nat} \to \mathrm{st} \\ \mathit{ColorPoint} & \triangleq & \mathbf{Obj}\ \mathrm{st}\ \mathbf{Inst}\ \mathrm{y}\colon \mathbf{Nat},\ \mathrm{color}\colon \mathbf{Nat} \\ & & \qquad\quad \mathbf{Meth}\ \mathrm{equal}\colon \mathrm{st} \to \mathbf{Bool},\ \mathrm{move}\colon \mathbf{Nat} \to \mathrm{st} \end{array}$$

The subtyping *ColorPoint* <: *Point* does not hold because of the binary method "equal", which requires a smaller argument type in *ColorPoint*. Indeed, one can easily construct an example, showing that assuming this subtyping may lead to a runtime type error (see e.g [5]). This has two unfortunate consequences:

1. If subclasses are required to generate subtypes then the class for *Point*s cannot be a subclass of the class for *ColorPoint*s.

2. Objects of type *ColorPoint* cannot be supplied where objects of type *Point* are required.

These consequences are unfortunate because in many cases, code for *Point*s could in fact be safely reused for *ColorPoint*s either via subclassing (i.e. via inheritance), or via supplying *ColorPoint*s where *Point*s are required. For example, if the code did not use "equal" at all, or it did, but was always careful to supply an argument of the relative Selftype (st).

These problems can be handled by basing subclassing and polymorphism on matching ($<\!\#$) – a relation between object types which is weaker than subtyping (and in particular, the matching *ColorPoint* $<\!\#$ *Point* holds). In matching-based languages (e.g. PolyToil [7], $\mathcal{LOOM}$ [6], O-3 [1]), subclasses are required to generate matching types, not subtypes, and typechecking proceeds accordingly. This means that when typechecking the class for *Point*s, the Selftype variable is assumed to only match *Point*.

Still, supplying *ColorPoint*s where *Point*s are required is impossible in such languages because matching does not have a subsumption property (i.e. it is not true that $e : \tau$ and $\tau <\!\# \ \sigma$ implies $e : \sigma$). This can be compensated by writing generic match-bounded code that works for both types (see [6]). Since this may sometimes be complicated, the language $\mathcal{LOOM}$ introduces additional machinery to cope with the lack of subsumption: *hash types*. The hash type $\#\tau$ stands for $(\exists t <\!\# \ \tau)t$, so, for example, a variable of type $\#Point$ may hold an object of any type that matches *Point* (and *ColorPoint* in particular). Thus, a function f expecting an argument of type $\#Point$ may run on *ColorPoint*s. Hash types are therefore quite useful, but they also have limitations, namely that one cannot call binary methods on them. Thus, f above may not manipulate its argument as freely as it could, had it assumed an argument of type *Point*.

Matching can also hold between types containing "binary fields", i.e instance variables of type st. For example, the types *Point* and *ColorPoint* above could both have an instance variable (listed in the **Inst** part) "adjacentPoint: st". Of course, st means something else in each type. Thus, matching-based languages can support this as well.

Binary methods and fields, however, are only a special case of the more general case in which the type of fields and method parameters are required to change covariantly, i.e. "downwards". Consider the following circles and color-circles (the superscript *in* for "informal" distinguishes the types below from the superscript-free ones used later):

$$\begin{array}{lll} Circle^{in} & \triangleq & \textbf{Obj}\ \text{st}\ \textbf{Inst}\ \text{center: } Point, \text{ radius: } \textbf{Nat} \\ & & \qquad\qquad \textbf{Meth}\ \text{equal: st} \to \textbf{Bool}, \text{ relocate: } Point \to \text{st} \\ ColorCircle^{in} & \triangleq & \textbf{Obj}\ \text{st}\ \textbf{Inst}\ \text{center: } ColorPoint, \text{ radius: } \textbf{Nat}, \text{ color: } \textbf{Nat} \\ & & \qquad\qquad \textbf{Meth}\ \text{equal: st} \to \textbf{Bool}, \text{ relocate: } ColorPoint \to \text{st}, \text{ setColor: } \textbf{Nat} \to \text{st} \end{array}$$

Neither the subtyping $Circle^{in} <: ColorCircle^{in}$ nor the matching $Circle^{in} <\!\# \ ColorCircle^{in}$ hold due to the covariant change of the field "center" from *Point* to *colorPoint* and the method "relocate" from $Point \to$ st to $ColorPoint \to$ st (each suffices to invalidate subtyping and matching). Consequently, the languages mentioned above cannot define the class for color-circles as a subclass of that for circles. Similarly, match-bounded polymorphism and hash types cannot be used. One can still obtain reuse in this context by reformulating the above types as match-bounded parameterized types, and writing match-bounded parameterized classes realizing them (see e.g. [1] Sec. 3.3.). But this is fairly complicated.

Our goal in this paper is to generalize matching to allow covariant change of fields and method parameters. Then, basing subclassing on (generalized) matching, and using match-bounded polymorphism and hash types, we obtain a language in which subclassing is significantly more flexible, and reuse can be applied to a broader range of related types. To our experience, cases of general (non-binary) covariance are far more frequent in real systems than binary methods: the latter seem to arise mostly in mathematical domains. The need for general covariance, in contrast, often arises in real-life domains. Here is an example (assume $Pediatrician <\!\# \ Doctor$):

$$\begin{array}{lll} Person & \triangleq & \textbf{Obj}\ \text{st}\ \textbf{Inst}\ \text{age: } \textbf{Nat}, \text{ physician: } Doctor, \ \ldots \\ & & \qquad\qquad \textbf{Meth}\ \text{changePhysician: } Doctor \to \text{st}, \text{ getSecondOpinion: } Doctor \to \textbf{Bool}, \ \ldots \\ Child & \triangleq & \textbf{Obj}\ \text{st}\ \textbf{Inst}\ \text{age: } \textbf{Nat}, \text{ physician: } Pediatrician, \text{ school: String}, \ \ldots \\ & & \qquad\qquad \textbf{Meth}\ \text{changePhysician: } Pediatrician \to \text{st}, \text{ getSecondOpinion: } Pediatrician \to \textbf{Bool}, \ \ldots \end{array}$$

3 Generalized Matching

In its current view, matching is a relation between object types defined in terms of subtyping: an object type $\sigma \equiv \textbf{Obj}\ st\ \textbf{Inst}\ y_j : \sigma_j{}^{j\in\{1\ldots n+m\}}\ \textbf{Meth}\ m_k : \mu_k{}^{k\in\{1\ldots \bar{n}+\bar{m}\}}$ matches a shorter one $\tau \equiv \textbf{Obj}\ st\ \textbf{Inst}\ y_j : \tau_j{}^{j\in\{1\ldots n\}}\ \textbf{Meth}\ m_k : \rho_k{}^{k\in\{1\ldots \bar{n}\}}$, denoted $\sigma <\!\# \ \tau$, when $(\sigma_j <: \tau_j \ \ , \tau_j <: \sigma_j)^{j\in\{1\ldots n\}}$ and $(\mu_k <: \rho_k)^{k\in\{1\ldots \bar{n}\}}$. Our goal is to extend matching to a relation between arbitrary types, and then redefine it for object types in terms of itself by requiring $(\sigma_j <\!\# \ \tau_j)^{j\in\{1\ldots n\}}$ and $(\mu_k <\!\# \ \rho_k)^{k\in\{1\ldots \bar{n}\}}$. The problem with

$$
\begin{array}{lll}
\textit{Point} & \triangleq & \textbf{Obj}\ \mathrm{st}\ \textbf{Inst}\ \mathrm{y}\colon \textbf{Nat} \\
 & & \qquad\quad \textbf{Meth}\ \mathrm{equal}\colon \mathrm{st} \to \textbf{Bool},\ \mathrm{move}\colon \textbf{Nat} \to \mathrm{st} \\
\textit{ColorPoint} & \triangleq & \textbf{Obj}\ \mathrm{st}\ \textbf{Inst}\ \mathrm{y}\colon \textbf{Nat},\ \mathrm{color}\colon \textbf{Nat} \\
 & & \qquad\quad \textbf{Meth}\ \mathrm{equal}\colon \mathrm{st} \to \textbf{Bool},\ \mathrm{move}\colon \textbf{Nat} \to \mathrm{st}
\end{array}
$$

$$
\begin{array}{lll}
\textit{point} & \triangleq & \textbf{class}\ (\mathrm{s}\colon \mathrm{st} \lessdot\!\#\ \textit{Point})\ \textbf{inst}\ \mathrm{y} = 0 \\
 & & \qquad \textbf{meth}\ \mathrm{equal} = \lambda(\mathrm{p}\colon \mathrm{st})\ \textbf{if}\ \mathrm{p.y} = \mathrm{s.y}\ \textbf{then true else false}, \\
 & & \qquad\quad \mathrm{move} = \lambda(\mathrm{x}\colon \textbf{Nat})\ \mathrm{s.y} := \mathrm{x}
\end{array}
$$

$$
\begin{array}{lll}
\textit{colorPoint} & \triangleq & \textbf{class}\ (\mathrm{s}\colon \mathrm{st} \lessdot\!\#\ \textit{ColorPoint})\ \textbf{extends}\ \textit{point} \\
 & & \qquad \textbf{inst}\ \mathrm{color} = 1 \\
 & & \qquad \textbf{meth}\ \mathrm{equal} = \lambda(\mathrm{p}\colon \mathrm{st})\ \textbf{if}\ (\mathrm{p.y} = \mathrm{s.y})\ \&\ (\mathrm{p.color} = \mathrm{s.color}) \\
 & & \qquad\qquad \textbf{then true else false}
\end{array}
$$

Figure 1: Point-related types and classes

just adopting this definition is that matching knowledge would then become virtually worthless because it could only result in more matching knowledge, and starting from a matching assumption, it would be impossible to ever derive an exact type (at least in a pure object system). This would make match-bounded parameterization and hash types futile, and there would be no way to obtain effective matching-based (and hence subclass-based) polymorphism. It is therefore necessary to introduce some extra machinery to achieve our goal.

In the rest of this section, we introduce and explain the ideas informally through examples written in the language LGM (Language with Generalized Matching). The language, its typing rules, and its semantics are formally defined in the next section.

3.1 Matching Object Types: Variance Annotations and Role Types

Consider the types *Circle* and *ColorCircle* defined as follows:

$$
\begin{array}{lll}
\textit{Circle} & \triangleq & \textbf{Obj}\ \mathrm{st}\ \textbf{Inst}\ \mathrm{center}^{+}\colon \textit{Point},\ \mathrm{radius}\colon \textbf{Nat} \\
 & & \qquad\quad \textbf{Meth}\ \mathrm{equal}\colon \mathrm{st} \to \textbf{Bool},\ \mathrm{relocate}\colon \mathrm{st.center} \to \mathrm{st} \\
\textit{ColorCircle} & \triangleq & \textbf{Obj}\ \mathrm{st}\ \textbf{Inst}\ \mathrm{center}^{+}\colon \textit{ColorPoint},\ \mathrm{radius}\colon \textbf{Nat},\ \mathrm{color}\colon \textbf{Nat} \\
 & & \qquad\quad \textbf{Meth}\ \mathrm{equal}\colon \mathrm{st} \to \textbf{Bool},\ \mathrm{relocate}\colon \mathrm{st.center} \to \mathrm{st},\ \mathrm{setColor}\colon \textbf{Nat} \to \mathrm{st}
\end{array}
$$

The code above uses the standard types *Point* and *ColorPoint* mentioned in Sec. 2, and shown again in Fig. 1, for which we have *ColorPoint* $\lessdot\!\#$ *Point*. LGM's matching rules also imply that *ColorCircle* $\lessdot\!\#$ *Circle*, which is the focus of this example. The superscript "+" of the instance variable "center" is a *variance annotation*, indicating that its type can change covariantly in a matching type. Therefore, the change from *Point* to *ColorPoint* is sound for matching.

Apart from "+", there is one more variance annotation: "$\circ$". It indicates invariance, meaning that the type remains unchanged in matching types (up to type equality). Variance annotations decorate *roles* in object and function types. The roles of an object type are its method and instance variable labels (like "center" above). All function types have exactly two roles: the argument and the return value, and variance annotations are indicated on the appropriate side of the arrow, e.g. $\tau\ {}^{+}\!\!\to^{\circ} \sigma$. Every role in an object and function type is annotated by either "$\circ$" or "+", but we informally omit "$\circ$"–annotations for the sake of clarity. Thus, all roles in the current examples, except "center", are annotated with "$\circ$". An example which uses "+"–annotations on function types is given in Sec. 3.2 below.

The concept of, and the notation for, variance annotations is borrowed from Abadi and Cardelli [1], but here it is with relation to matching (as opposed to subtyping there), and the judgment rules treat them differently.

The other new feature in this example is the use of *role types*: st.center, which is used to type the method "relocate" as st.center $\to$ st, is a role type. This typing means that "relocate" is a function expecting an argument having the same type as the instance variable "center" in self. Role types formalize and generalize Eiffel's *anchored types* [12]. They are formed by qualifying a type with a role, as in st.center, *Circle*.equal, *ColorCircle*.color and so on. Role types make it possible to manipulate covariant attributes – and to apply functions with a covariant argument – given matching knowledge about the type of self. Besides typing methods such as "relocate", they are also useful for typing (free-standing) match-bounded parameterized functions, as we will show later.

The matching rule for object types reads as follows:

$$\text{(Ma Obj)} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ y_j^{v_j} : \tau_j{}^{j\in\{1\ldots n\}}\ \mathbf{Meth}\ m_k^{u_k} : \rho_k{}^{k\in\{1\ldots \bar{n}\}}$$
$$\sigma \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ y_j^{v'_j} : \sigma_j{}^{j\in\{1\ldots n+m\}}\ \mathbf{Meth}\ m_k^{u'_k} : \mu_k{}^{k\in\{1\ldots \bar{n}+\bar{m}\}})$$

$$\frac{\begin{array}{l}(C \vdash \sigma_j\ \mathcal{R}(v_j)\ \tau_j \quad v'_j \prec v_j)^{\ j\in\{1\ldots n\}}\\ (C \vdash \mu_k\ \mathcal{R}(u_k)\ \rho_k \quad u'_k \prec u_k)^{\ k\in\{1\ldots \bar{n}\}}\end{array}}{C \vdash \sigma \mathrel{<\!\!\#} \tau} \qquad \mathcal{R}(v) \triangleq \begin{cases} = & \text{if } v = \circ \\ \mathrel{<\!\!\#} & \text{if } v = + \end{cases}$$

$$v_1 \prec v_2 \text{ iff } v_1 = \circ \text{ or } v_2 = +$$
$$st \text{ not free in } C$$

That is, invariant attributes must be equal and covariant attributes must match. Variance annotations can change from covariance to invariance. Like the language $\mathcal{LOOM}$ [6], LGM does not support subtyping, so the special case in which all attributes are invariant amounts to matching as in $\mathcal{LOOM}$. Note that the conclusion judgment captures one variable *st* which appears free in both premises. This is a characteristic property of matching between object types. Role types are useful because they can be fixed in invariant roles, while their meaning implicitly changes covariantly. For example, the meaning of st.center in the invariant method "relocate" changes implicitly from *Point* to *ColorPoint*, but matching holds because it is the same role type. This is analogous to the implicit change of meaning of the Selftype variable st from *Circle* to *ColorCircle*.

Consider the classes *circle* and *colorCircle* producing objects of types *Circle* and *ColorCircle* respectively:

```
circle      ≜ class (s: st<# Circle) inst center = new point, radius = 1
                                     meth equal = λ(c: st) if (s.center ← equal(c.center)) &
                                                              (s.radius = c.radius)
                                                           then true else false
                                          relocate = λ(p: st.center) s.center := p

colorCircle ≜ class (s: st<# ColorCircle) extends circle
                                     inst center = new colorPoint, color = 1
                                     meth relocate = λ(p: st.center) if (s.center.color = p.color)
                                                                    then s.center := p else s
                                          setColor = λ(nc: Nat)
                                                   (s.color := nc).center := (s.center.color := nc)
```

The header "**class** (s: st<# *Circle*)" of *circle* means that it is typechecked to generate objects of type *Circle*, that s stands for self inside the class, and that typechecking will be done under the assumption that s has type st, where st<# *Circle*. LGM's typing rules require that subclasses generate matching types, so these assumptions guarantee safe method inheritance without re-typechecking. In *colorCircle*, the idea is that the center's color is kept consistent with the circle's color. This is guaranteed by "setColor" and by "relocate", which updates the center with the given new point only if its color is the same as the circle's. The method "equal" in *circle* is inherited into *colorCircle* relying on this consistency, knowing that "equal" in *colorPoint* tests for color equality.

Message sends with matching knowledge are typechecked in LGM by first introducing role types and then resolving them according to the variance annotations:

$$(\text{Msg}\mathrel{<\!\!\#}) \qquad \frac{C, A \vdash e : \sigma \quad C \vdash \sigma \mathrel{<\!\!\#} \tau}{C, A \vdash e \leftarrow m_k : \sigma.m_k} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

$$\text{(Eq MatchMeth}^\circ\text{)}\quad \frac{C \vdash \sigma \lessdot\!\!\# \ \textbf{Obj}\ st\ \textbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \textbf{Meth}\ \overline{m_k^{u_k} : \rho_k} \qquad u_k = \circ}{C \vdash \sigma.m_k = \rho_k[\sigma/st]}$$

$$\text{(Ma MatchMeth}^+\text{)}\quad \frac{C \vdash \sigma \lessdot\!\!\# \ \textbf{Obj}\ st\ \textbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \textbf{Meth}\ \overline{m_k^{u_k} : \rho_k} \qquad u_k = +}{C \vdash \sigma.m_k \lessdot\!\!\# \ \rho_k[\sigma/st]}$$

The first rule is a typing rule, the second a type equality rule, and the third a matching rule. The combination of (Msg$\lessdot\!\!\#$) and (Eq MatchMeth°) provides the same effect as $\mathcal{LOOM}$'s message send rule, which directly derives $\rho_k[\sigma/st]$ given the premises of (Msg$\lessdot\!\!\#$). The rules for reading instance variables – (Read$\lessdot\!\!\#$), (Eq MatchInst°), and (Ma MatchInst$^+$) – are analogous.

Back to the example, the method "equal" in *circle* typechecks as follows: the function's body is examined under the assumptions s: st, c: st and st $\lessdot\!\!\#$ *Circle*. Looking at the first condition after the **if**, (s.center $\leftarrow$ equal(c.center)), it follows from (Read$\lessdot\!\!\#$) that s.center has role type st.center. Since st $\lessdot\!\!\#$ *Circle* and "center" is a covariant attribute, we have that st.center $\lessdot\!\!\#$ *Point* by (Ma MatchInst$^+$). From s.center: st.center $\lessdot\!\!\#$ *Point* we may conclude that s.center$\leftarrow$ equal has type st.center$\rightarrow$ **Bool** by (Msg$\lessdot\!\!\#$) and (Eq MatchMeth°) above. The actual argument c.center has role type st.center by by (Read$\lessdot\!\!\#$), so the application typechecks as **Bool**. For the second condition (s.radius = c.radius), we conclude that both sides of the equation have type st.radius, which by (Eq MatchInst°) equals **Nat** because "radius" is an invariant attribute. So the test for equality between naturals ("=") typechecks too.

The method "relocate" in *circle* contains an assignment to the covariant instance variable "center". The rule for writing instance variables with matching knowledge reads:

$$\text{(Write}\lessdot\!\!\#\text{)}\quad \frac{C, A \vdash e_1 : \sigma \quad C \vdash \sigma \lessdot\!\!\# \ \tau \quad C, A \vdash e_2 : \sigma.y_j}{C, A \vdash e_1.y_j := e_2 : \sigma} \quad (\text{where } \tau \equiv \textbf{Obj}\ st\ \textbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \textbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

It follows that in case of a covariant instance variable, only a role type can be assigned (in the invariant case, a concrete type could be proven equal to the role type). The assignment s.center := p in "relocate" typechecks by (Write$\lessdot\!\!\#$) because s has type st, st $\lessdot\!\!\#$ *Circle*, and p has type st.center. The other methods in this example typecheck using similar considerations.

Variance annotations make a difference only in the context of matching: if we know the exact object type of an expression, not just what it matches, then the variance annotations are not important. For example, the following rules are indifferent to variance annotations:

$$\text{(Msg)}\quad \frac{C, A \vdash e : \tau}{C, A \vdash e \leftarrow m_k : \rho_k[\tau/st]} \quad (\text{where } \tau \equiv \textbf{Obj}\ st\ \textbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \textbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

$$\text{(Eq RoleMeth)}\quad \frac{}{C \vdash \tau.m_k = \rho_k[\tau/st]} \quad (\text{where } \tau \equiv \textbf{Obj}\ st\ \textbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \textbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

The rules (Read) and (Eq RoleInst) for instance variables are analogous. Unlike Abadi and Cardelli [1], the rule for writing instance variables knowing the exact type is also indifferent to variance annotations:

$$\text{(Write)}\quad \frac{C, A \vdash e_1 : \tau \quad C, A \vdash e_2 : \tau_j[\tau/st]}{C, A \vdash e_1.y_j := e_2 : \tau} \quad (\text{where } \tau \equiv \textbf{Obj}\ st\ \textbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \textbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

Thus, if we know that e : *Circle*, we may conclude by (Read) that e.center: *Point*, and we may assign a concrete point to it by (Write), even though "center" is covariant. Also, using the rules above we may conclude that e : *Circle* implies $e \leftarrow$ relocate: *Point* $\rightarrow$ *Circle*. Therefore, the following expressions all typecheck: *aCircle*.center := *aPoint*, *aColorCircle*.center := *aColorPoint*, *aCircle*.radius := 4, *aColorCircle*.radius := 4, *aCircle* $\leftarrow$ relocate(*aPoint*), *aColorCircle* $\leftarrow$ relocate(*aColorPoint*), *aCircle* $\leftarrow$ equal(*anotherCircle*), *aColorCircle* $\leftarrow$ equal(*anotherColorCircle*). We show in Sec. 3.3 below that the typing of binary methods can be improved to allow typechecking of the safe case *aCircle* $\leftarrow$ equal(*aColorCircle*) as well.

Role types make it possible to write generic code manipulating covariant attributes, via match-bounded parameterization (we write a multi-argument function for clarity):

relocateCircle $\triangleq$ λ(t$\lessdot\!\!\#$ *Circle*, c: t, p: t.center) c $\leftarrow$ relocate(p)

The function *relocateCircle* has type $\forall$(t<# *Circle*) t $\rightarrow$ t.center $\rightarrow$ t, and can be instantiated for both *Circle* and *ColorCircle*. By the rules, *relocateCircle*[*Circle*] has type *Circle* $\rightarrow$ *Point* $\rightarrow$ *Circle*, and *relocateCircle*[*ColorCircle*] has type *ColorCircle* $\rightarrow$ *ColorPoint* $\rightarrow$ *ColorCircle*. This insures that only safe applications of the function will typecheck.

3.2 Matching Function Types

The following celebrated example illustrates a covariant change in the argument type of a method:

$$\begin{array}{lcl}
\mathit{Animal} & \triangleq & \textbf{Obj}\ \mathrm{st}\ \textbf{Meth}\ \mathrm{eat}^{+}\mathrm{:}\ \mathit{Food}\ {}^{+}\!\!\rightarrow \mathrm{st},\ \mathrm{isHappy:}\ \textbf{Bool},\ \mathrm{likesFood:}\ \mathrm{st.eat.}\textbf{arg} \rightarrow \textbf{Bool} \\
\mathit{Herbivore} & \triangleq & \textbf{Obj}\ \mathrm{st}\ \textbf{Meth}\ \mathrm{eat}^{+}\mathrm{:}\ \mathit{PlantFood}\ {}^{+}\!\!\rightarrow \mathrm{st},\ \mathrm{isHappy:}\ \textbf{Bool},\ \mathrm{likesFood:}\ \mathrm{st.eat.}\textbf{arg} \rightarrow \textbf{Bool}
\end{array}$$

$$\begin{array}{lcl}
\mathit{animal} & \triangleq & \textbf{class}\ (\mathrm{s:\ st} \lessdot\!\#\ \mathit{Animal})\ \textbf{meth}\ \mathrm{eat} = \lambda(\mathrm{f:}\ \mathit{Food})\ldots \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \mathrm{isHappy} = \ldots \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \mathrm{likesFood} = \lambda(\mathrm{f:\ st.eat.}\textbf{arg})\ \mathrm{s} \leftarrow \mathrm{eat(f)} \leftarrow \mathrm{isHappy}
\end{array}$$

$$\begin{array}{lcl}
\mathit{herbivore} & \triangleq & \textbf{class}\ (\mathrm{s:\ st} \lessdot\!\#\ \mathit{Herbivore})\ \textbf{extends}\ \mathit{animal} \\
 & & \qquad\qquad\qquad\qquad\qquad \textbf{meth}\ \mathrm{eat} = \lambda(\mathrm{f:}\ \mathit{PlantFood})\ldots
\end{array}$$

Recall that function types have two roles, the argument and the return value, and variance annotations for these roles are indicated on the appropriate side of the arrow. Thus, the function in "eat" has a covariant argument (and invariant return value). Assuming that *PlantFood* <# *Food*, we have that *Herbivore* <# *Animal* because *PlantFood* $^{+}\!\!\rightarrow^{\circ}$ st <# *Food* $^{+}\!\!\rightarrow^{\circ}$ st by the following matching rule for function types:

$$\text{(Ma Func)}\quad \frac{\begin{array}{cc} u \prec v & u' \prec v' \\ C \vdash \tau\ \mathcal{R}(v)\ \sigma & C \vdash \tau'\ \mathcal{R}(v')\ \sigma' \end{array}}{C \vdash \tau\ {}^{u}\!\!\rightarrow^{u'}\ \tau' \lessdot\!\#\ \sigma\ {}^{v}\!\!\rightarrow^{v'}\ \sigma'} \qquad \mathcal{R}(v) \triangleq \begin{cases} = & \text{if } v = \circ \\ \lessdot\!\# & \text{if } v = + \end{cases}$$

$$v_1 \prec v_2 \text{ iff } v_1 = \circ \text{ or } v_2 = +$$

As with assignment to covariant instance variables, the only way to type-safely apply a function with a covariant argument is via role types. The argument and return value roles of a function type are denoted in LGM by the keywords **arg** and **ret**. The typechecking rule for function application with matching knowledge reads as follows:

$$\text{(App}\lessdot\!\#\text{)}\quad \frac{C, A \vdash e_1 : \rho \quad C \vdash \rho \lessdot\!\#\ \sigma\ {}^{v}\!\!\rightarrow^{v'}\ \sigma' \quad C, A \vdash e_2 : \rho.\mathbf{arg}}{C, A \vdash e_1(e_2) : \rho.\mathbf{ret}}$$

The method "likesFood" in *animal*, which is inherited into *herbivore*, calls "eat" and then checks whether or not the animal is happy. It typechecks as follows: the body s $\leftarrow$ eat(f) $\leftarrow$ isHappy is examined under the assumptions s: st <# *Animal* and f: st.eat.**arg**. By (Msg<#) we have that s $\leftarrow$ eat: st.eat. Since f: st.eat.**arg**, we have that s $\leftarrow$ eat(f): st.eat.**ret** by (App<#). Since st <# *Animal* and "eat" is covariant, we have that st.eat <# *Food* $^{+}\!\!\rightarrow^{\circ}$ *Animal*, and since the return value is invariant we have by an appropriate equality rule that st.eat.**ret** = *Animal*, so s $\leftarrow$ eat(f): *Animal*. Finally, the message send s $\leftarrow$ eat(f) $\leftarrow$ isHappy typechecks as **Bool** by (Msg).

Function role types are also useful for writing generic match-bounded functions:

$$\mathit{animalLikesFood} \triangleq \lambda(\mathrm{t} \lessdot\!\#\ \mathit{Animal},\ \mathrm{a:\ t,\ f:\ t.eat.}\textbf{arg})\ \mathrm{a} \leftarrow \mathrm{eat(f)} \leftarrow \mathrm{isHappy}$$

We have that *animalLikesFood*[*Animal*] has type *Animal* $\rightarrow$ *Food* $\rightarrow$ **Bool**, and *animalLikesFood*[*Herbivore*] has type *Herbivore* $\rightarrow$ *PlantFood* $\rightarrow$ **Bool**, as expected.

3.3 Hash Types and Flexible Binary Methods

LGM also has hash types to partly compensate for the lack of a subsumption property for matching. They are interpreted as in $\mathcal{LOOM}$, that is $\#\tau$ is interpreted as the existential type $\exists(t \lessdot\!\#\ \tau)t$. Thus, an object of type $\#\tau$ may be an instance of a class generating objects of type τ, or of any subclass. The types *Animal* and *Herbivore* could be rephrased with #*Food* and #*PlantFood* as the argument of "eat", to allow animals to eat plant food as well:

$Animal' \triangleq$ **Obj** st **Meth** eat$^+$: $\#Food \overset{+}{\rightarrow}$ st, isHappy: **Bool**, likesFood: st.eat.**arg** $\rightarrow$ **Bool**
$Herbivore' \triangleq$ **Obj** st **Meth** eat$^+$: $\#PlantFood \overset{+}{\rightarrow}$ st, isHappy: **Bool**, likesFood: st.eat.**arg** $\rightarrow$ **Bool**

The matching $Herbivore' <\!\!\# \ Animal'$ holds because we have $\#PlantFood <\!\!\# \ \#Food$ by the following monotonicity rule for hashing:

$$\text{(Ma Mon)} \qquad \frac{C \vdash \tau <\!\!\# \ \sigma}{C \vdash \#\tau <\!\!\# \ \#\sigma}$$

In the corresponding classes *animal′* and *herbivore′*, the parameter declaration in "eat" should be λ(f: #*Food*) in *animal′* and λ(f: #*PlantFood*) in *herbivore′*, and the implementations must be written according to these assumptions. The method "likesFood" remains as in *animal*. Hash types are also useful for typing heterogeneous collections, for example we could define a linked list of #*Animal*s holding and manipulating animals and herbivores in a single data structure.

Hash types do provide matching-based substitutability, but they have significant limitations: it is impossible to call methods with a covariant argument (like "eat" above) on a hash type, or to call methods (even invariant) mentioning the Selftype variable in a negative position. This includes binary methods and methods whose argument is a role type built around the Selftype variable. For example, if c has type #*Circle*, then one cannot call "equal" or "relocate" on c because there is no knowledge what argument is exactly required. It is also impossible to assign anything to the covariant c.center. Still, one may deduce that c.center has type #*Point*. Invariant attributes not mentioning the Selftype variable in a negative position can be manipulated freely, so c.radius has type **Nat**, c.radius := 4 has type #*Circle*, and c.center ← move(5) has type #*Point*.

Hash types in LGM are handled by the two following typing rules:

$$\text{(\#Assume)} \qquad \frac{C \cup \{t <\!\!\# \ \sigma\}, A \cup \{x : t\} \vdash e : \sigma'}{C, A \cup \{x : \#\sigma\} \vdash e : \sigma'} \qquad t \text{ not free in } C, A, \sigma'$$

$$\text{(\#Conclude)} \qquad \frac{C, A \vdash e : \sigma \quad C \vdash \sigma <\!\!\# \ \sigma'}{C, A \vdash e : \#\sigma'}$$

No other typing rules are concerned with hash types. The rule (#Assume) says that a $\#\sigma$ assumption is handled by opening the hash, that is by explicitly introducing a variable t to stand for the type that matches σ. The rule (#Conclude) packs a matching type in a hash type. To typecheck e.g. a message send on a hash type, one opens it with (#Assume), and then uses the same (Msg<#) above, which is used to typecheck all message sends, given matching knowledge.

As a bonus of the generalized approach to matching, we are also able to obtain a more flexible typing for binary methods by typing the argument as #st rather than st. Consider the following improved points and color points:

$Point' \triangleq$ **Obj** st **Inst** y: **Nat**
Meth equal: #st → **Bool**, move: **Nat** → st
$ColorPoint' \triangleq$ **Obj** st **Inst** y: **Nat**, color: **Nat**
Meth equal: #st → **Bool**, move: **Nat** → st

The corresponding classes *point′* and *colorPoint′* are identical to *point* and *colorPoint* except the parameter declaration in "equal", which should be hashed. In this version, the safe method call $aPoint' \leftarrow$ equal($aColorPoint'$) will typecheck in LGM. The technique of *precise typing* [5] can also typecheck this case, but it is considerably more complicated. Note that the return type of "move" did not change, although it could, because hashing it would only weaken the typing. This flexibility carries on to types using *Point′* and *ColorPoint′*: we can obtain improved circles and color circles by changing equal's argument to #st:

$Circle' \triangleq$ **Obj** st **Inst** center$^+$: *Point′*, radius: **Nat**
Meth equal: #st → **Bool**, relocate: st.center → st
$ColorCircle' \triangleq$ **Obj** st **Inst** center$^+$: *ColorPoint′*, radius: **Nat**, color: **Nat**
Meth equal: #st → **Bool**, relocate: st.center → st

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

$$
\begin{array}{l}
x \in Var \\
t, st \in TypeVar \\
\overline{y_j}, \overline{m_k} \in Labels \\
n ::= \; 0 \mid 1 \mid 2 \mid \ldots \\
b ::= \; \textbf{true} \mid \textbf{false}
\end{array}
$$

$$
\begin{array}{l}
\tau ::= \; t \mid \textbf{Nat} \mid \textbf{Bool} \mid \tau \; {}^{v}\!\!\rightarrow^{v'} \tau \mid \forall(t <\# \tau)\tau \mid \#\tau \\
\quad \mid \textbf{Obj } st \textbf{ Inst } \overline{y_j^{v_j} : \tau_j} \textbf{ Meth } \overline{m_k^{v_k} : \tau_k} \mid \textbf{Class}(\tau) \mid \tau.role
\end{array}
$$

$$
\begin{array}{l}
v, v', \overline{v_j}, \overline{v_k} \in \{\circ, +\} \\
role \in Labels \cup \{\textbf{arg}, \textbf{ret}\}
\end{array}
$$

Figure 2: LGM expressions and types

The corresponding classes $circle'$ and $colorCircle'$ are unchanged except the parameter declaration in "equal". They will typecheck (using also the rule (Ma Role), see Fig. 4), as will the safe call $aCircle' \leftarrow$ equal$(aColorCircle')$.

4 The Language LGM

This section formally specifies the language LGM (Language with Generalized Matching), including type equality and matching rules, typing rules, and operational semantics. It concludes with a statement of subject reduction, establishing the soundness of the system.

Figure 2 shows the syntax for LGM expressions and types. The **obj** construct defines objects explicitly, where $\overline{y_j = e_j}$ are the instance variables and $\overline{m_k = e_k}$ are the methods. $e \leftarrow m$ is a message send (method call), $e.y$ reads an instance variable, and $e_1.y := e_2$ updates an instance variable. The semantics is functional, so a copy of e_1 with y updated is actually returned. The **class** construct defines classes which can then be instantiated with **new**. The header "**class**$(s : st <\# \tau)$" declares a variable (s) to stand for self within the body, a type variable (st) to stand for the type of self, and a matching assumption ($st <\# \tau$), which also declares the type of objects (τ) generated from the class. The **extends** keyword indicates subclassing, and **root** represents the root class. Overriding is not indicated by a keyword, but simply by including the overridden attributes among the new ones (as in C++ and Java). $\lambda[t <\# \tau]e$ is a match-bounded type abstraction, and $e[\tau]$ is a type application. These constructs are used to write match-bounded parameterized functions or classes.

Object types and function types contain *variance annotations*, which can be either "+" or "$\circ$". As explained in the previous section, these annotations decorate roles, which are attribute (i.e. instance variable or method) labels in object types, and the argument and return value in function types. "+" and "$\circ$" indicate covariance and invariance change of the role in matching types. $\#\tau$ is a hash type, and since matching is defined on all types, τ can be any type. **Class**(τ) is the type of classes generating objects of type τ. $\tau.role$ is a *role type* as explained in the previous section. In a role type, *role* is either an attribute label or one of the keywords **arg** and **ret**. $\forall(t <\# \tau)\tau$ is the type of match-bounded type abstractions.

In the examples of the previous section we assumed some standard built-in primitive functions such as natural numbers equality test (=). We also omitted "$\circ$" annotations, and **extends** clauses when subclassing **root**.

4.1 Type Equality and Matching Rules

The type equality and matching rules have the form $C \vdash J$. C contains matching assumptions of the form $t <\# \tau$, where t is a type variable, and J is a type equality or matching judgment. The rules are interrelated:

there are matching rules with equality judgments as premises and vice-versa. These two related sets of rules are the heart of the "matching game": they propagate top-level matching knowledge through the type structure until concrete equality knowledge is reached. The typing rules for matching just derive "symbolic" role types, which are resolved to concrete types using the equality and matching rules.

For simplicity, LGM does not support subtyping. We mention the possibility of adding subtyping in Sec. 5.

4.1.1 Type Equality Rules

The type equality rules are shown in Fig 3. Formal equality is denoted by "$=$", and syntactical identity is denoted by "$\equiv$". Types which can be obtained from one-another by renaming bound variables are considered syntactically equivalent (i.e. they are the same type).

The four rules (Eq RoleArg) – (Eq RoleInst) equate role types with the types playing the role. They are indifferent to variance annotations. The last four rules (Eq MatchArg$^{\circ}$) – (Eq MatchInst$^{\circ}$) equate role types matching a given type with types playing invariant roles in that type. These rules define the meaning of "$\circ$" annotations.

The following lemma states that equality is well behaved in the sense that equal non-role types have the same structure and are deeply equal, or in other words, role types are faithful mediators.

Lemma 4.1: Suppose $\vdash \sigma = \tau$, where σ is not a role type.

- If $\tau \equiv \mathbf{Nat}$ then $\sigma \equiv \mathbf{Nat}$.
- If $\tau \equiv \mathbf{Bool}$ then $\sigma \equiv \mathbf{Bool}$.
- If $\tau \equiv \tau' \ {}^{v}{\rightarrow}^{v''}\ \tau''$ then $\sigma \equiv \sigma' \ {}^{v'}{\rightarrow}^{v''}\ \sigma''$ and $\vdash \tau' = \sigma'$, $\vdash \tau'' = \sigma''$.
- If $\tau \equiv \mathbf{Obj}\ \mathit{st}\ \mathbf{Inst}\ {y_j^{v_j} : \tau_j}^{j \in \{1 \ldots n\}}\ \mathbf{Meth}\ m_k^{u_k} : \rho_k\ {}^{k \in \{1 \ldots \bar{n}\}}$ then $\sigma \equiv \mathbf{Obj}\ \mathit{st}\ \mathbf{Inst}\ {y_j^{v_j} : \sigma_j}^{j \in \{1 \ldots n\}}\ \mathbf{Meth}\ m_k^{u_k} : \mu_k\ {}^{k \in \{1 \ldots \bar{n}\}}$ and $(\vdash \tau_j = \sigma_j)\ {}^{j \in \{1 \ldots n\}}$, $(\vdash \rho_k = \mu_k)\ {}^{k \in \{1 \ldots \bar{n}\}}$.
- If $\tau \equiv \forall(t \mathrel{\lessdot\!\#} \tau')\tau''$ then $\sigma \equiv \forall(t \mathrel{\lessdot\!\#} \sigma')\sigma''$ and $\vdash \tau' = \sigma'$, $\vdash \tau'' = \sigma''$.
- If $\tau \equiv \#\tau'$ then $\sigma \equiv \#\sigma'$ and $\vdash \tau' = \sigma'$.
- If $\tau \equiv \mathbf{Class}(\tau')$ then $\sigma \equiv \mathbf{Class}(\sigma')$ and $\vdash \tau' = \sigma'$.

4.1.2 Matching Rules

The matching rules are shown in Fig. 4. The rule (Ma Eq) says that equal types match, and (Ma Trans) asserts that matching is transitive. The rule (Ma Mon) asserts the monotonicity of hashing, allowing hash type specialization, as in *Animal$'$* and *Herbivore$'$* in Sec. 3.3. The rules for matching object and function types behave according to the variance annotations decorating the roles. Object type matching with all roles being invariant amounts to matching as in $\mathcal{LOOM}$. The last four rules (Ma MatchArg^{+}) – (Ma MatchInst^{+}) propagate matching knowledge for covariant attributes via role types. They define the meaning of "$+$" annotations.

The next lemma states that matching types preserve structure, with component types behaving according to variance annotations.

Lemma 4.2: Suppose $\vdash \sigma \mathrel{\lessdot\!\#} \tau$.

- If $\vdash \tau = \tau' \ {}^{v'}{\rightarrow}^{v''}\ \tau''$ then $\vdash \sigma = \sigma' \ {}^{u'}{\rightarrow}^{u''}\ \sigma''$ and $\vdash \sigma'\ \mathcal{R}(v')\ \tau'$, $\vdash \sigma''\ \mathcal{R}(v'')\ \tau''$, $u' \prec v'$, $u'' \prec v''$.
- If $\vdash \tau = \mathbf{Obj}\ \mathit{st}\ \mathbf{Inst}\ {y_j^{v_j} : \tau_j}^{j \in \{1 \ldots n\}}\ \mathbf{Meth}\ m_k^{u_k} : \rho_k\ {}^{k \in \{1 \ldots \bar{n}\}}$ then $\vdash \sigma = \mathbf{Obj}\ \mathit{st}\ \mathbf{Inst}\ {y_j^{v'_j} : \sigma_j}^{j \in \{1 \ldots n+m\}}\ \mathbf{Meth}\ m_k^{u'_k} : \mu_k\ {}^{k \in \{1 \ldots \bar{n}+\bar{m}\}}$ and $(\vdash \sigma_j\ \mathcal{R}(v_j)\ \tau_j,\ v'_j \prec v_j)\ {}^{j \in \{1 \ldots n\}}$, $(\vdash \mu_k\ \mathcal{R}(u_k)\ \rho_k,\ u'_k \prec u_k)\ {}^{k \in \{1 \ldots \bar{n}\}}$.
- If $\vdash \tau = \#\tau'$ then $\vdash \sigma = \#\sigma'$ and $\vdash \sigma' \mathrel{\lessdot\!\#} \tau'$.
- If $\vdash \tau = \mathbf{Nat}$ or $\vdash \tau = \mathbf{Bool}$ or $\vdash \tau = \forall(t \mathrel{\lessdot\!\#} \tau')\tau''$ or $\vdash \tau = \mathbf{Class}(\tau')$ then $\vdash \sigma = \tau$.

(Eq Refl) $$\frac{}{C \vdash \tau = \tau}$$

(Eq Sym) $$\frac{C \vdash \tau = \sigma}{C \vdash \sigma = \tau}$$

(Eq Trans) $$\frac{C \vdash \tau = \sigma \qquad C \vdash \sigma = \rho}{C \vdash \tau = \rho}$$

(Eq Func) $$\frac{C \vdash \tau = \sigma \qquad C \vdash \tau' = \sigma'}{C \vdash \tau \ {}^{v}\!\!\rightarrow^{v'} \tau' = \sigma \ {}^{v}\!\!\rightarrow^{v'} \sigma'}$$

(Eq Obj) $$\frac{\overline{C \vdash \tau_j = \tau'_j} \qquad \overline{C \vdash \rho_k = \rho'_k}}{C \vdash \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k} = \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau'_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho'_k}}$$

(Eq Class) $$\frac{C \vdash \tau = \tau'}{C \vdash \mathbf{Class}(\tau) = \mathbf{Class}(\tau')}$$

(Eq PolyFunc) $$\frac{C \vdash \tau = \tau' \qquad C \vdash \sigma = \sigma'}{C \vdash \forall(t \mathrel{<\!\!\#} \sigma)\tau = \forall(t \mathrel{<\!\!\#} \sigma')\tau'}$$

(Eq Hash) $$\frac{C \vdash \tau = \sigma}{C \vdash \#\tau = \#\sigma}$$

(Eq Role) $$\frac{C \vdash \tau = \sigma \qquad role \in Labels \cup \{\mathbf{arg}, \mathbf{ret}\}}{C \vdash \tau.role = \sigma.role}$$

(Eq RoleArg) $$\frac{}{C \vdash (\tau \ {}^{v}\!\!\rightarrow^{v'} \tau').\mathbf{arg} = \tau}$$

(Eq RoleRet) $$\frac{}{C \vdash (\tau \ {}^{v}\!\!\rightarrow^{v'} \tau').\mathbf{ret} = \tau'}$$

(Eq RoleMeth) $$\frac{}{C \vdash \tau.m_k = \rho_k[\tau/st]} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

(Eq RoleInst) $$\frac{}{C \vdash \tau.y_j = \tau_j[\tau/st]} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

(Eq MatchArg°) $$\frac{C \vdash \sigma \mathrel{<\!\!\#} \rho \ {}^{\circ}\!\!\rightarrow^{v'} \rho'}{C \vdash \sigma.\mathbf{arg} = \rho}$$

(Eq MatchRet°) $$\frac{C \vdash \sigma \mathrel{<\!\!\#} \rho \ {}^{v}\!\!\rightarrow^{\circ} \rho'}{C \vdash \sigma.\mathbf{ret} = \rho'}$$

(Eq MatchMeth°) $$\frac{C \vdash \sigma \mathrel{<\!\!\#} \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k} \qquad u_k = \circ}{C \vdash \sigma.m_k = \rho_k[\sigma/st]}$$

(Eq MatchInst°) $$\frac{C \vdash \sigma \mathrel{<\!\!\#} \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k} \qquad v_j = \circ}{C \vdash \sigma.y_j = \tau_j[\sigma/st]}$$

Figure 3: Type equality rules for LGM

(Ma Env)
$$\frac{}{C \cup \{t \mathrel{<\#} \tau\} \vdash t \mathrel{<\#} \tau}$$

(Ma Eq)
$$\frac{C \vdash \tau = \sigma}{C \vdash \tau \mathrel{<\#} \sigma}$$

(Ma Trans)
$$\frac{C \vdash \tau \mathrel{<\#} \sigma \quad C \vdash \sigma \mathrel{<\#} \rho}{C \vdash \tau \mathrel{<\#} \rho}$$

(Ma Mon)
$$\frac{C \vdash \tau \mathrel{<\#} \sigma}{C \vdash \#\tau \mathrel{<\#} \#\sigma}$$

(Ma Role)
$$\frac{C \vdash \tau \mathrel{<\#} \sigma}{C \vdash \tau.role \mathrel{<\#} \sigma.role}$$

(Ma Func)
$$\frac{\begin{array}{l} u \prec v \quad u' \prec v' \\ C \vdash \tau\ \mathcal{R}(v)\ \sigma \quad C \vdash \tau'\ \mathcal{R}(v')\ \sigma' \end{array}}{C \vdash \tau\ {}^{u}{\rightarrow}^{u'}\ \tau' \mathrel{<\#} \sigma\ {}^{v}{\rightarrow}^{v'}\ \sigma'} \qquad \mathcal{R}(v) \triangleq \begin{cases} = & \text{if } v = \circ \\ \mathrel{<\#} & \text{if } v = + \end{cases}$$

$v_1 \prec v_2$ iff $v_1 = \circ$ or $v_2 = +$

(Ma Obj)
$$\begin{array}{l}(\text{where } \tau \equiv \textbf{Obj}\ st\ \textbf{Inst}\ y_j^{v_j} : \tau_j{}^{j \in \{1 \ldots n\}}\ \textbf{Meth}\ m_k^{u_k} : \rho_k\ {}^{k \in \{1 \ldots \bar{n}\}} \\ \qquad \sigma \equiv \textbf{Obj}\ st\ \textbf{Inst}\ y_j^{v'_j} : \sigma_j{}^{j \in \{1 \ldots n+m\}}\ \textbf{Meth}\ m_k^{u'_k} : \mu_k\ {}^{k \in \{1 \ldots \bar{n}+\bar{m}\}})\end{array}$$

$$\frac{\begin{array}{l} (C \vdash \sigma_j\ \mathcal{R}(v_j)\ \tau_j \quad v'_j \prec v_j)\ {}^{j \in \{1 \ldots n\}} \\ (C \vdash \mu_k\ \mathcal{R}(u_k)\ \rho_k \quad u'_k \prec u_k)\ {}^{k \in \{1 \ldots \bar{n}\}} \end{array}}{C \vdash \sigma \mathrel{<\#} \tau} \qquad st \text{ not free in } C,\ \mathcal{R} \text{ as above}$$

(Ma MatchArg$^+$)
$$\frac{C \vdash \sigma \mathrel{<\#} \rho\ {}^{+}{\rightarrow}^{v'}\ \rho'}{C \vdash \sigma.\mathbf{arg} \mathrel{<\#} \rho}$$

(Ma MatchRet$^+$)
$$\frac{C \vdash \sigma \mathrel{<\#} \rho\ {}^{v}{\rightarrow}^{+}\ \rho'}{C \vdash \sigma.\mathbf{ret} \mathrel{<\#} \rho'}$$

(Ma MatchMeth$^+$)
$$\frac{C \vdash \sigma \mathrel{<\#} \textbf{Obj}\ st\ \textbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \textbf{Meth}\ \overline{m_k^{u_k} : \rho_k} \quad u_k = +}{C \vdash \sigma.m_k \mathrel{<\#} \rho_k[\sigma/st]}$$

(Ma MatchInst$^+$)
$$\frac{C \vdash \sigma \mathrel{<\#} \textbf{Obj}\ st\ \textbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \textbf{Meth}\ \overline{m_k^{u_k} : \rho_k} \quad v_j = +}{C \vdash \sigma.y_j \mathrel{<\#} \tau_j[\sigma/st]}$$

Figure 4: Matching rules for LGM

4.2 Typing Rules

The typing rules are shown in Fig. 5. The obvious rules for variables, naturals, booleans, **if** statements, and **root** have been omitted to fit the rules into one figure. All the other rules are present in Fig. 5. The rule (SubsEq) assimilates equality into the system. (#Assume) handles hash types by introducing a variable t to stand for the packed matching type. (#Conclude) concludes hash types for expressions by packing a matching type. No other rule is concerned with hash types.

The rules for typing expressions with matching knowledge are (App$<\#$), (Msg$<\#$), (Read$<\#$), and (Write$\lessdot\#$). All but (Write$\lessdot\#$) derive role types, which may then be used directly (as in typing (s.center $\leftarrow$ equal(c.center)) inside *circle*'s "equal"), or otherwise be resolved to concrete types via the equality and matching rules, and using (SubsEq) (as in typing (s.radius = c.radius) in the same method). In (App$\lessdot\#$), the parameter to the function must have the appropriate role type, which may be resolved to a concrete type in case of an invariant argument. Similarly, in (Write$\lessdot\#$), the assigned expression must have a role type. The corresponding non-matching rules – (App), (Msg), (Read) and (Write) – are standard. They are indifferent to variance annotations (which is why they are not implied by the $\lessdot\#$ -rules).

The (Class) rule requires that a class be extended by a subclass producing objects of a matching type, and the body is typechecked under the assumption that the Selftype variable matches the generated object type. This guarantees that methods can be safely inherited without re-typechecking.

The lemmas below summarize a few properties of the typing rules.

Lemma 4.3: Suppose $C, A \cup \{x : \tau\} \vdash e : \sigma$ and $C, A \vdash e' : \tau$. Then $C, A \vdash e[e'/x] : \sigma$.

Lemma 4.4: Suppose $C \vdash \tau' \lessdot\# \; \tau$ and $C \cup \{t \lessdot\# \; \tau\}, A \cup \{x : t\} \vdash e : \sigma$, where t is not free in C, A. Then $C, A \cup \{x : \tau'\} \vdash e : \sigma[\tau'/t]$.

Lemma 4.5: Suppose $C \vdash \tau' <\# \; \tau$ and $C \cup \{t <\# \; \tau\}, A \vdash e : \sigma$, where t is not free in C. Then $C \cup \{t \lessdot\# \; \tau'\}, A \vdash e : \sigma$.

4.3 Operational Semantics and Subject Reduction

The operational semantics of LGM can be found in Appendix A. It is a standard natural semantics defining a reduction relation $e \Downarrow v$, where e is an LGM expression and v is an irreducible value. The irreducible values of LGM are natural, booleans, **root**, classes extending **root**, lambda abstractions, type abstractions, and objects with evaluated instance variables.

We have a subject reduction property:

Theorem 4.6 (Subject Reduction): Suppose $e \Downarrow v$ and $\vdash e : \tau$. Then $\vdash v : \tau$.

The proof is rather routine given Lemma 4.1 – 4.5.

5 Conclusions

The generalization of matching developed in this paper allows covariant change in the types of fields and method parameters, thereby supporting type-safe flexible subclassing in an ordinary single-dispatch language. Variance annotations control how specialized a matching type can get, and role types provide a mechanism to manipulate covariant attributes via matching knowledge. With role types, match-bounded parameterization can compensate for the loss of subclass substitutability, as it does for regular matching. Like in $\mathcal{LOOM}$, hash types provide a degree of direct "subclass polymorphism", but they also have limitations generalizing those of hash types w.r.t. regular matching.

In the introduction, multi-methods and matching were contrasted as two major approaches for dealing with the covariance problem, each going in a different direction. Putting it this way faithfully describes the state of things in current research, and serves to put generalized matching in context. However, in our opinion, these two approaches could be combined to form a flexible and appealing framework. In [9, 10], Castagna distinguishes between two kinds of mechanisms: *specialization*, in which an existing method is extended with a new specialized branch, and *substitutivity*, in which new code replaces old code. He then goes on to conclude that covariance is the appropriate rule for the former, and contravariance – which is consistent with subtyping – for the latter.

While the distinction between two possible ways to treat newly introduced code in subclasses – that is, either as an additional branch or as replacing code – is certainly important and fundamental, their

(SubsEq) $$\frac{C, A \vdash e : \sigma \quad C \vdash \sigma = \sigma'}{C, A \vdash e : \sigma'}$$

(#Assume) $$\frac{C \cup \{t \mathrel{<\!\!\#} \sigma\}, A \cup \{x : t\} \vdash e : \sigma'}{C, A \cup \{x : \#\sigma\} \vdash e : \sigma'} \qquad t \text{ not free in } C, A, \sigma'$$

(#Conclude) $$\frac{C, A \vdash e : \sigma \quad C \vdash \sigma \mathrel{<\!\!\#} \sigma'}{C, A \vdash e : \#\sigma'}$$

(Abs) $$\frac{C, A \cup \{x : \sigma\} \vdash e : \sigma'}{C, A \vdash \lambda(x : \sigma)e : \sigma \ {}^{v}\!\!\to^{v'} \sigma'}$$

(App) $$\frac{C, A \vdash e_1 : \sigma \ {}^{v}\!\!\to^{v'} \sigma' \quad C, A \vdash e_2 : \sigma}{C, A \vdash e_1(e_2) : \sigma'}$$

(App<#) $$\frac{C, A \vdash e_1 : \rho \quad C \vdash \rho \mathrel{<\!\!\#} \sigma \ {}^{v}\!\!\to^{v'} \sigma' \quad C, A \vdash e_2 : \rho.\mathbf{arg}}{C, A \vdash e_1(e_2) : \rho.\mathbf{ret}}$$

(TypeAbs) $$\frac{C \cup \{t \mathrel{<\!\!\#} \sigma\}, A \vdash e : \sigma'}{C, A \vdash \lambda[t \mathrel{<\!\!\#} \sigma]e : \forall(t \mathrel{<\!\!\#} \sigma)\sigma'} \qquad t \text{ not free in } C, A$$

(TypeApp) $$\frac{C, A \vdash e : \forall(t \mathrel{<\!\!\#} \sigma)\sigma' \quad C \vdash \rho \mathrel{<\!\!\#} \sigma}{C, A \vdash e[\rho] : \sigma'[\rho/t]}$$

(Msg) $$\frac{C, A \vdash e : \tau}{C, A \vdash e \leftarrow m_k : \rho_k[\tau/st]} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

(Msg<#) $$\frac{C, A \vdash e : \sigma \quad C \vdash \sigma \mathrel{<\!\!\#} \tau}{C, A \vdash e \leftarrow m_k : \sigma.m_k} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

(Read) $$\frac{C, A \vdash e : \tau}{C, A \vdash e.y_j : \tau_j[\tau/st]} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

(Read<#) $$\frac{C, A \vdash e : \sigma \quad C \vdash \sigma \mathrel{<\!\!\#} \tau}{C, A \vdash e.y_j : \sigma.y_j} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

(Write) $$\frac{C, A \vdash e_1 : \tau \quad C, A \vdash e_2 : \tau_j[\tau/st]}{C, A \vdash e_1.y_j := e_2 : \tau} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

(Write<#) $$\frac{C, A \vdash e_1 : \sigma \quad C \vdash \sigma \mathrel{<\!\!\#} \tau \quad C, A \vdash e_2 : \sigma.y_j}{C, A \vdash e_1.y_j := e_2 : \sigma} \qquad (\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

(Obj) $$(\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{u_k} : \rho_k}\ )$$

$$\frac{\begin{array}{l}\overline{C, A \vdash e_j : \tau_j[\tau/st]} \\ \overline{C, A \cup \{s : \tau\} \vdash f_k : \rho_k[\tau/st]}\end{array}}{C, A \vdash \mathbf{obj}(s : \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = f_k}\ : \tau}$$

(Class) $$\begin{array}{l}(\text{where } \tau \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ y_j^{v_j} : \tau_j{}^{j \in \{1 \ldots n\}}\ \mathbf{Meth}\ m_k^{u_k} : \rho_k{}^{k \in \{1 \ldots \bar{n}\}} \\ \qquad \sigma \equiv \mathbf{Obj}\ st\ \mathbf{Inst}\ y_j^{v'_j} : \sigma_j{}^{j \in \{1 \ldots n+m\}}\ \mathbf{Meth}\ m_k^{u'_k} : \mu_k{}^{k \in \{1 \ldots \bar{n}+\bar{m}\}} \\ \qquad IOvr \subseteq \{1 \ldots n\},\ MOvr \subseteq \{1 \ldots \bar{n}\}\ )\end{array}$$

$$\frac{\begin{array}{l}C, A \vdash e : \mathbf{Class}(\tau) \\ C \vdash \sigma \mathrel{<\!\!\#} \tau \\ (C \vdash \sigma_j = \tau_j)^{j \in \{1 \ldots n\} \setminus IOvr} \\ (C \vdash \mu_k = \rho_k)^{k \in \{1 \ldots \bar{n}\} \setminus MOvr} \\ (C \cup \{st \mathrel{<\!\!\#} \sigma\}, A \vdash e_j : \sigma_j)^{j \in IOvr \cup \{n+1 \ldots n+m\}} \qquad\qquad st \text{ not free in } C, A \\ (C \cup \{st \mathrel{<\!\!\#} \sigma\}, A \cup \{s : st\} \vdash f_k : \mu_k)^{k \in MOvr \cup \{\bar{n}+1 \ldots \bar{n}+\bar{m}\}}\end{array}}{\begin{array}{l}C, A \vdash \mathbf{class}(s : st \mathrel{<\!\!\#} \sigma)\ \mathbf{extends}\ e \\ \qquad \mathbf{inst}\ (y_j = e_j)^{j \in IOvr \cup \{n+1 \ldots n+m\}}\ \mathbf{meth}\ (m_k = f_k)^{k \in MOvr \cup \{\bar{n}+1 \ldots \bar{n}+\bar{m}\}} : \mathbf{Class}(\sigma)\end{array}}$$

(New) $$\frac{C, A \vdash e : \mathbf{Class}(\tau)}{C, A \vdash \mathbf{new}\ e : \tau}$$

Figure 5: Typing rules for LGM

identification with covariance versus contravariance should perhaps be reconsidered in view of this paper. That is to say, it is exactly our goal here to show that covariant substitutivity can be supported via generalized matching. It seems that the best of the two worlds could be combined in a language that supported covariant field overriding and allowed the programmer to specify, for each method, whether it is multi- or single-branched, where single-methods may be overridden covariantly. This would improve on multi-methods because the latter cannot support covariant field overriding, and consequently, cannot support covariant overriding of mutater methods that change the value of such fields. It would improve on matching-based single-dispatch languages (like LGM) because in many cases, multi-dispatch is just the *right* semantics. For example, accesser functions like "equal" are conceptually multi-branched: it makes perfect sense that a call $aColorPoint \leftarrow \mathrm{equal}(aPoint)$ should work, simply by ignoring the receiver's color, that is by invoking a less specific branch ([2]). Another example is the method "getSecondOpinion" at the end of Sec. 2: the call $aChild \leftarrow \mathrm{getSecondOpinion}(aDoctor)$ is sensible because if only a general doctor is available then his or her second opinion would be useful, even though a pediatrician could do a more specialized job. An appropriate language should define the concept of matching between multi-methods, and this seems to be an interesting and potentially beneficial line to pursue.

Moving on to a different issue, we noted that role types formalize and generalize Eiffel's "anchored types". Curiously enough, it seems that the rationale behind Meyer's "no polymorphic catcall" policy [13], designed to remedy Eiffel's type insecurities, can be roughly stated in our terms as "no calls to covariant methods, or to invariant methods mentioning Selftype in a positive position, on a hash type". Meyer's "catcalls" are not very far from such calls, and "polymorphic entities" recall hash typed variables. A precise comparison is hard to do here, so we just state this as an interesting point. At any rate, it seems that Eiffel's policy is (perhaps significantly) more restrictive since the potential of role types and match-bounded parameterization is not really exploited. The approach of this paper could be adopted to provide type-safe covariant specialization for Eiffel as envisioned by Meyer.

Supporting covariant specialization via generalized matching in practical languages such as Eiffel, C++ and Java implies minor additions to their syntax. The additions needed to incorporate Selftype (and hence subclassing with binary methods) and match-bounded parameterization in Java are described in [4]. For generalized matching, we need only add variance annotations and role types. The former can be expressed via a keyword such as **covariant** prefixing a field or method declaration (where keywords like **virtual** or **static** usually go). Declaring a method as **covariant** would also mean that the arguments are covariant. For role types, one could adopt Eiffel's syntax for anchored types "**like** a", where a is a field or method label of the enclosing class, instead of the explicit role type "Selftype.a". For the function type roles **arg** and **ret**, and for match-bounded parameterization, one should use explicit qualification as in LGM.

Two traditional features of theoretic OO languages are missing in LGM: the ability to invoke methods in the superclass, and hiding instance variables in object types. The former was not included for brevity, and adding a **super** construct should be straight forward. However, letting the Selftype variable range only over the methods is problematic because it is important to be able to refer to instance variables via role types (as in *Circle*'s "relocate"). We do not consider this a serious problem because there are ways to restrict access to object components not through the type system, e.g. by declaring them **private**.

LGM could be made more powerful by supporting subtyping (and subsumption) in addition to matching. There would then be interesting interactions between the two relations. For example, it looks sound to make the monotonicity rule for hashing a subtyping rule: $C \vdash \tau <\# \ \sigma$ implies $C \vdash \#\tau <: \#\sigma$. Another plausible rule is the axiom $\vdash \tau <: \#\tau$, which together with the previous one entails that $C \vdash \tau <\# \ \sigma$ implies $C \vdash \tau <: \#\sigma$. This latter rule could allow the substitution of a hash type for the positive occurrences of a matching type in any given typing conclusion, by subsumption. Assuming that subtyping should entail matching, it is also reasonable that the presence of subtyping should influence the definition of matching between object and function types.

More issues not yet explored include a typechecking algorithm and a denotational semantics. We do not foresee significant difficulties with the algorithm, but a semantics may be more complicated. Finally, we remark that matching could probably be defined between general recursive types as $C \vdash \sigma <\# \ \tau$ implies $C \vdash \mu t.\sigma <\# \ \mu t.\tau$, where t is not free in C. Such system would have record-like object types $\mathbf{Inst}\ \overline{y_j^{v_j} : \tau_j}\ \mathbf{Meth}\ \overline{m_k^{v_k} : \tau_k}$ instead of LGM's object types, with matching between them defined according to the variance annotations. Matching between recursive object types as well as between recursive function types would then be supported.

References

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

[2] J. Boyland and G. Castagna. Type-safe compilation of covariant specialization: a practical case. In *LNCS Vol. 1098: ECOOP '96 conference proceedings.* Springer-Verlag, 1996.

[3] K. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4:127–206, 1994.

[4] K. Bruce. Increasing java's expressiveness with thistype and match-bounded polymorphism. draft, http://www.cs.williams.edu/ kim, 1997.

[5] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G.T Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3), 1996.

[6] K. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good match for object-oriented languages. In *LNCS Vol. 1241: ECOOP '97 conference proceedings.* Springer-Verlag, 1997.

[7] K. Bruce, A. Schuett, and R. van Gent. Polytoil: A type-safe polymorphic object-oriented language, extended abstract. In *LNCS Vol. 952: ECOOP '95 conference proceedings.* Springer-Verlag, 1995.

[8] K. Bruce and J.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, http://www.elsevier.nl/locate/entcs, 1999.

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

[10] G. Castagna. *Object-oriented programming, a unified approach.* Birkhauser, 1997.

[11] 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)*, 1989.

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

[13] B. Meyer. Static typing. *OOPS Messenger*, 6(4), 1995.

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

A Operational Semantics of LGM

This appendix specifies the operational semantics of LGM. It is a standard natural semantics defining a reduction relation $e \Downarrow v$, where e is an LGM expression and v is an irreducible value. The irreducible values of LGM are natural, booleans, **root**, classes extending **root**, lambda abstractions, type abstractions, and objects with evaluated instance variables. The rules are given below.

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

$$\frac{\overline{e_j \Downarrow v_j}}{\mathbf{obj}(s : \tau)\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = f_k} \Downarrow o}$$

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

$$\frac{e \Downarrow o \qquad f_k[o/s] \Downarrow v}{e \Leftarrow m_k \ \Downarrow v}$$

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

$$\frac{e \Downarrow o}{e.y_j \ \Downarrow v_j}$$

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

$$\frac{e_1 \Downarrow o \quad e_2 \Downarrow v}{e_1.y_j := e_2 \ \Downarrow o[x_j = v]}$$

(Red Class) (where $c = \mathbf{class}(s : st <\!\!\# \ \tau)\ \mathbf{extends}\ \mathbf{root}\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = f_k}$)

$$\frac{e \Downarrow \mathbf{root}}{\mathbf{class}(s : st <\!\!\# \ \tau)\ \mathbf{extends}\ e\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = f_k} \Downarrow c}$$

(Red Subclass) (where $c = \mathbf{class}(s : st <\!\!\# \ \sigma)\ \mathbf{extends}\ \mathbf{root}$
$\mathbf{inst}\ y_j = e'_j{}^{\ j \in \{1 \ldots n\} \setminus IOvr},\ y_j = e_j{}^{\ j \in IOvr \cup \{n+1 \ldots n+m\}}$
$\mathbf{meth}\ m_k = f'_k{}^{\ k \in \{1 \ldots \bar{n}\} \setminus MOvr},\ m_k = f_k{}^{\ k \in MOvr \cup \{\bar{n}+1 \ldots \bar{n}+\bar{m}\}}$
$IOvr \subseteq \{1 \ldots n\},\ MOvr \subseteq \{1 \ldots \bar{n}\}$)

$$\frac{e \Downarrow \mathbf{class}(s : st <\!\!\# \ \tau)\ \mathbf{extends}\ \mathbf{root}\ \mathbf{inst}\ y_j = e'_j{}^{\ j \in \{1 \ldots n\}}\ \mathbf{meth}\ m_k = f'_k{}^{\ k \in \{1 \ldots \bar{n}\}}}{\mathbf{class}(s : st <\!\!\# \ \sigma)\ \mathbf{extends}\ e\ \mathbf{inst}\ y_j = e_j{}^{\ j \in IOvr \cup \{n+1 \ldots n+m\}}\ \mathbf{meth}\ m_k = f_k{}^{\ k \in MOvr \cup \{\bar{n}+1 \ldots \bar{n}+\bar{m}\}} \Downarrow c}$$

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

$$\frac{\begin{array}{l} e \Downarrow \mathbf{class}(s : st <\!\!\# \ \tau)\ \mathbf{extends}\ \mathbf{root}\ \mathbf{inst}\ \overline{y_j = e_j}\ \mathbf{meth}\ \overline{m_k = f_k} \\ \overline{e_j \Downarrow v_j} \end{array}}{\mathbf{new}\ e \Downarrow o}$$

(Red Abs) $\dfrac{}{\lambda(x : \tau)e \Downarrow \lambda(x : \tau)e}$

(Red App) $\dfrac{e_1 \Downarrow \lambda(x : \tau)e \quad e_2 \Downarrow v' \quad e[v'/x] \Downarrow v}{e_1(e_2) \Downarrow v}$

(Red TypeAbs) $\dfrac{}{\lambda[t <\!\!\# \ \tau]e \Downarrow \lambda[t <\!\!\# \ \tau]e}$

(Red TypeApp) $\dfrac{e' \Downarrow \lambda[t <\!\!\# \ \tau]e \quad e[\sigma/t] \Downarrow v}{e'[\sigma] \Downarrow v}$

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

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

(Red Nat) $\dfrac{}{n \Downarrow n}$ (where $n = 0, 1, 2, \ldots$)

(Red Bool) $\dfrac{}{b \Downarrow b}$ (where $b = \mathbf{true}, \mathbf{false}$)

(Red Root) $\dfrac{}{\mathbf{root} \Downarrow \mathbf{root}}$