Unifying Object-Oriented Programming with Typed Functional Programming

Hongwei Xi*
Boston University
hwxi@cs.bu.edu

ABSTRACT

The wide practice of object-oriented programming in current software construction is evident. Despite extensive studies on typing programming objects, it is still undeniably a challenging research task to design a type system for object-oriented programming that is both effective in capturing program errors and unobtrusive to program construction. In this paper, we present a novel approach to typing objects that makes use of a recently invented notion of guarded dependent datatypes. We show that our approach can address various difficult issues (e.g., handling "self" type, typing binary methods, etc.) in a simple and natural type-theoretical manner, remedying the deficiencies in many existing approaches to typing objects.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages

General Terms

Languages

Keywords

DML, dependent types, object-oriented

1. INTRODUCTION

The popularity of object-oriented programming in current software practice is evident. While this popularity may result in part from the tendency to chase after the latest "fads" in programming languages, there is undeniably some real substance in the growing use of object-oriented programming. In particular, objected-oriented programming can significantly facilitate software organization and reuse through encapsulation, inheritance and polymorphism. Building on our previous experience with Dependent ML [20, 18], we are naturally interested in combining object-oriented programming with dependent types. However, a straightforward combination of dependent types with object-oriented programming (e.g., following a Java-like approach) is largely unsatisfactory, as such an approach often requires a substantial use of run-time type downcasting. In search for a more satisfactory approach, we have noticed that a recently invented notion of guarded recursive datatype constructors [19] can be combined with dependent types to enable the construction of a type system for programming objects that needs no use of type downcasting. This is highly desirable as type downcasting is probably one of the most common causes of program errors in object-oriented languages like Java.

We briefly outline the basic idea behind our approach to typing programming objects. The central idea of objected-oriented programming is, of course, the programming objects. But what is really a programming object? Unfortunately, there is currently no simple answer to this question (and there unlikely will). In this paper, we take a view of programming objects in the spirit of Smalltalk [12, 14]; we suggest to conceptualize a programming object as a little intelligent being that is capable of performing actions according to the messages it receives; we suggest not to think of a programming object as a record of fields and methods in this paper.

We now present an example to illustrate how such a view of objects can be formulated in a typed setting. We assume the existence of a type constructor MSG that takes a type τ and forms the message type $(\tau)MSG$; after receiving a message of type $(\tau)MSG$, an object is supposed to return a value of type τ; therefore, we assign the following type OBJ to objects:

$$OBJ = \forall\alpha.(\alpha)MSG \rightarrow \alpha$$

Suppose that we have declared that $MSGgetfst$, $MSGgetsnd$, $MSGsetfst$ and $MSGsetsnd$ are message constructors of the following types, where **1** stands for the unit type.

$$\begin{array}{rcl} MSGgetfst & : & (\texttt{int})MSG \\ MSGgetsnd & : & (\texttt{int})MSG \\ MSGsetfst & : & \texttt{int} \rightarrow (\mathbf{1})MSG \\ MSGsetsnd & : & \texttt{int} \rightarrow (\mathbf{1})MSG \end{array}$$

In Figure 1, we implement integer pairs in a message-passing style, where the `withtype` clause is a type annotation that assigns the type $\texttt{int} \rightarrow \texttt{int} \rightarrow OBJ$ to the defined function $newIntPair$.[1] Note that such ML-like syntax is used to present examples throughout the paper. Given integers x

*Partially supported by the NSF Grants No. CCR-0081316 and No. CCR-0092703

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASIA-PEPM'02, September 12-14, 2002, Aizu, Japan.

[1]The reason for $newIntPair$ being well-typed is to be explained in Section 2.

$$
\begin{array}{rcl}
\mathit{HOAStup} & : & \forall\alpha_1\forall\alpha_2.(\alpha_1)\mathit{HOAS} * (\alpha_2)\mathit{HOAS} \rightarrow (\alpha_1 * \alpha_2)\mathit{HOAS} \\
\mathit{HOASlam} & : & \forall\alpha_1\forall\alpha_2.((\alpha_1)\mathit{HOAS} \rightarrow (\alpha_2)\mathit{HOAS}) \rightarrow (\alpha_1 \rightarrow \alpha_2)\mathit{HOAS} \\
\mathit{HOASapp} & : & \forall\alpha_1\forall\alpha_2.(\alpha_1 \rightarrow \alpha_2)\mathit{HOAS} * (\alpha_1)\mathit{HOAS} \rightarrow (\alpha_2)\mathit{HOAS} \\
\mathit{HOASlift} & : & \forall\alpha_1.\alpha_1 \rightarrow (\alpha_1)\mathit{HOAS}
\end{array}
$$

Figure 2: The value constructors associated with the g.r. datatype constructor *HOAS*

```
fun newIntPair x y = let
  val xref = ref x
  val yref = ref y
  fun dispatch MSGgetfst = !xref
    | dispatch MSGgetsnd = !yref
    | dispatch (MSGsetfst x') = (xref := x')
    | dispatch (MSGsetsnd y') = (yref := y')
    | dispatch msg = raise UnknownMessage
in dispatch end
withtype int -> int -> OBJ
```

Figure 1: An implementation of integer pairs

and y, we can construct an integer pair *anIntPair* by calling $\mathit{newIntPair}(x)(y)$; we can send the message *MSGgetfst* to the pair to obtain its first component: $\mathit{anIntPair}(\mathit{MSGgetfst})$; we can also reset its first component to x' by sending it the message $\mathit{MSGsetfst}(x')$: $\mathit{anIntPair}(\mathit{MSGsetfst}(x'))$; operations on the second component of the pair can be performed similarly; an exception is raised at run-time if *anIntPair* cannot interpret a message sent to it.

Obviously, there exists a serious problem with the above approach to implementing objects. Since every object is currently assigned the type *OBJ*, we cannot use types to differentiate objects. For instance, suppose that *MSGfoo* is another declared message constructor of type $(\mathbf{1})\mathit{MSG}$; then $\mathit{anIntPair}(\mathit{MSGfoo})$ is well-typed, but its execution leads to an uncaught exception *UnknownMessage* at run-time. This is clearly undesirable: $\mathit{anIntPair}(\mathit{MSGfoo})$ should be rejected at compile-time as an ill-typed expression. We will address this issue and many other ones in objected-oriented programming by making use of a restricted form dependent types developed in Dependent ML [20, 18].

The type constructor *MSG* is what we call a guarded recursive (g.r.) datatype constructor. The notion of g.r. datatype constructors, which extends the notion of datatypes in ML, is recently invented in the setting of functional programming for handling intentional polymorphism and run-time type passing [19]. We write $\exists\Delta.\tau$ for a guarded type, where Δ is a type variable context that may contain some type constraints. For instance, $\exists\Delta_1.\tau$ is a guarded type, where $\Delta_1 = (\alpha_1, \alpha_2, \alpha_1 * \alpha_2 \equiv \texttt{int} * \texttt{bool})$ and $\tau = \alpha_1 * \alpha_1$; this type is equivalent to $\texttt{int} * \texttt{int}$ since we must map α_1 to `int` in order to satisfy the type constraint $\alpha_1 * \alpha_2 \equiv \texttt{int} * \texttt{bool}$. The type $\exists\Delta_2.\tau$ is also a guarded type, where $\Delta_2 = (\alpha_1, \alpha_2, \alpha_1 * \alpha_2 \equiv \texttt{int})$; this type is equivalent to the type `void`, i.e., the type in which there is no element, since the type constraint $\alpha_1 * \alpha_2 \equiv \texttt{int}$ cannot be satisfied. If $\Delta = (\alpha_1, \alpha_2, \alpha_1 * \alpha_2 \equiv \alpha)$, we notice the type $\forall\alpha\exists\Delta.\tau$ has the following interesting feature: instantiating α with a type τ_0, we obtain a type that is equivalent to $\tau_1 * \tau_1$ if τ_0 is of the form $\tau_1 * \tau_2$, or `void` if τ_0 is of other forms.

A guarded recursive datatype constructor is a recursively defined type constructor for constructing guarded datatypes,

```
typecon (type) HOAS =
  {'a1,'a2}.
    ('a1 * 'a2) HOAStup of 'a1 HOAS * 'a2 HOAS
| {'a1,'a2}.
    ('a1 -> 'a2) HOASlam of 'a1 HOAS -> 'a2 HOAS
| {'a1,'a2}.
    ('a2) HOASapp of ('a1 -> 'a2) HOAS * 'a1 HOAS
| {'a1}. ('a1) HOASlift of 'a1
```

Figure 3: An example of g.r. datatype constructor

which are a special form of sum types in which each component is a guarded type. We present a short example of g.r. datatype constructor as follows for illustrating the notion. More details and examples can be found at [19]. The syntax in Figure 3 essentially declares a type constructor *HOAS*, which can take a type τ and then form another type $(\tau)\mathit{HOAS}$. Intuitively, a value of type $(\tau)\mathit{HOAS}$ represents a higher-order abstract syntax tree [9, 16] for a value of type τ. The value constructors associated with *HOAS* are given the types in Figure 2. Note the type constructor *HOAS* cannot be defined in ML. Because of the negative occurrence of *HOAS* in the argument type of *HOASlam*, *HOAS* cannot be inductively defined, either. The reason for calling *HOAS* a guarded recursive datatype constructor is that *HOAS* can be defined as follows through a fixed-point operator, where $*$ is the kind for types:

$$
\begin{array}{l}
\mu T : * \rightarrow *.\lambda\alpha : *. \\
\qquad \exists(\alpha_1 : *, \alpha_2 : *, \alpha_1 * \alpha_2 \equiv \alpha).(\alpha_1)T * (\alpha_2)T \\
\quad + \exists(\alpha_1 : *, \alpha_2 : *, \alpha_1 \rightarrow \alpha_2 \equiv \alpha).(\alpha_1)T \rightarrow (\alpha_2)T \\
\quad + \exists(\alpha_1 : *, \alpha_2 : *, \alpha_2 \equiv \alpha).(\alpha_1 \rightarrow \alpha_2)T * (\alpha_1)T \\
\quad + \exists(\alpha_1 : *, \alpha_1 \equiv \alpha).\alpha_1
\end{array}
$$

Then the value constructors associated with *HOAS* can be readily defined through the use of fold/unfold (for recursive types) and injection (for sum types). We can now define an evaluation function as follows that computes the value represented by a given higher-order abstract syntax tree.

```
fun eval(HOAStup (x1, x2)) = (eval x1, eval x2)
  | eval(HOASlam f) = fn x => eval (f (HOASlift x))
  | eval(HOASapp (x1, x2)) = (eval x1) (eval x2)
  | eval(HOASlift c) = c
withtype {'a}. 'a HOAS -> 'a
```

Note the `withtype` clause is a type annotation provided by the user, which indicates that *eval* is a function of type $\forall\alpha.(\alpha)\mathit{HOAS} \rightarrow \alpha$. In other words, the evaluation function *eval* is type-preserving.

In the rest of the paper, we are to present a type system to support g.r. datatype constructors. We then outline an approach to implementing programming objects, explaining how various issues in object-oriented programming can be addressed.

types	τ	::=	$\alpha \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2 \mid$
			$(\vec{\tau})T \mid \forall\alpha.\tau$
patterns	p	::=	$x \mid \langle\rangle \mid \langle p_1, p_2\rangle \mid c[\vec{\alpha}](p)$
clauses	ms	::=	$(p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)$
expressions	e	::=	$x \mid f \mid c[\vec{\tau}](e)$
			$\langle\rangle \mid \langle e_1, e_2\rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid$
			$\lambda x:\tau.e \mid e_1(e_2) \mid \Lambda\alpha.v \mid e[\tau] \mid$
			$\mathbf{fix}\ f:\tau.v \mid \mathbf{case}\ e\ \mathbf{of}\ ms \mid$
			$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}$
values	v	::=	$x \mid c[\vec{\tau}](v) \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid$
			$\lambda x:\tau.e \mid \Lambda\alpha.v$
exp. var. ctx.	Γ	::=	$\cdot \mid \Gamma, x:\tau$
typ. var. ctx.	Δ	::=	$\cdot \mid \Delta, \alpha \mid \Delta, \tau_1 \equiv \tau_2$

Figure 4: Syntax for the internal language $\lambda_{2,G\mu}$

2. THE LANGUAGE $\lambda_{2,G\mu}$

We present a language $\lambda_{2,G\mu}$ based on the explicitly typed second-order polymorphic λ-calculus. We present both static and dynamic semantics for $\lambda_{2,G\mu}$ and then show that the type system of $\lambda_{2,G\mu}$, which supports g.r. datatype constructors, is sound.

2.1 Syntax

We present the syntax for $\lambda_{2,G\mu}$ in Figure 4, which is mostly standard. We use α for type variables, $\mathbf{1}$ for the unit type and $\vec{\tau}$ for a (possibly empty) sequence of types $\tau_1, \ldots, \tau_n$. We have two kinds of expression variables: x for lam-variables and f for fix-variables. We use xf for either a lam-variable or a fix-variable. We can only form a λ-abstraction over a lam-variable and a fixed-point expression over a fix-variable. Note that a lam-variable is a value but a fix-variable is not. We use c for constructors and assume that every constructor is unary.[2] Also, we require that the body of either Λ or **fix** be a value. The syntax for patterns is to be explained in Section 2.4.

We use Θ for substitutions mapping type variables to types and $\mathbf{dom}(\Theta)$ for the domain of Θ. Note that $\Theta[\alpha \mapsto \tau]$, where we assume $\alpha \notin \mathbf{dom}(\Theta)$, extends Θ with a mapping from α to τ. Similar notations are also used for substitutions θ mapping variables xf to expressions. We write $\bullet[\Theta]$ ($\bullet[\theta]$) for the result of applying Θ (θ) to $\bullet$, where $\bullet$ can be a type, an expression, a type variable context, an expression variable context, etc.

We use Δ for type variable contexts in $\lambda_{2,G\mu}$, which require some explanation. As usual, we can declare a type variable α in a type variable context Δ. We use $\Delta \vdash \tau : *$ to mean that τ is a well-formed type in which every type variable is declared in Δ. All type formation rules are standard and thus omitted. We can also declare a type equality $\tau_1 \equiv \tau_2$ in Δ. Intuitively, when deciding type equality under Δ, we assume that the types τ_1 and τ_2 are equal if $\tau_1 \equiv \tau_2$ is declared in Δ.

Given two types τ_1 and τ_2, we write $\tau_1 = \tau_2$ to mean that τ_1 is α-equivalent to τ_2. The following rules are for deriving judgments of form $\vdash \Theta : \Delta$, which roughly means that Θ matches Δ.

$$\frac{}{\vdash [] : \cdot} \qquad \frac{\vdash \Theta : \Delta \quad \vdash \tau : *}{\vdash \Theta[\alpha \mapsto \tau] : \Delta, \alpha} \qquad \frac{\vdash \Theta : \Delta \quad \tau_1[\Theta] = \tau_2[\Theta]}{\vdash \Theta : \Delta, \tau_1 \equiv \tau_2}$$

[2]For a constructor taking no argument, we can treat it as a constructor taking the unit $\langle\rangle$ as its argument.

Pattern typing rules $\quad p \downarrow \tau \Rightarrow (\Delta; \Gamma)$

$$\frac{\Delta_0 \vdash \tau : *}{\Delta_0 \vdash x \downarrow \tau \Rightarrow \cdot; x:\tau}\ \textbf{(pat-var)}$$

$$\frac{}{\Delta_0 \vdash \langle\rangle \downarrow \mathbf{1} \Rightarrow \cdot; \cdot}\ \textbf{(pat-unit)}$$

$$\frac{\Delta_0 \vdash p_1 \downarrow \tau_1 \Rightarrow \Delta_1; \Gamma_1 \quad \Delta_0 \vdash p_2 \downarrow \tau_2 \Rightarrow \Delta_2; \Gamma_2}{\Delta_0 \vdash \langle p_1, p_2\rangle \downarrow \tau_1 * \tau_2 \Rightarrow \Delta_1, \Delta_2; \Gamma_1, \Gamma_2}\ \textbf{(pat-tup)}$$

$$\frac{\Sigma(c) = \forall\vec{\alpha}.\tau \to (\vec{\tau}_1)T \quad \Delta_0, \vec{\alpha}, \vec{\tau}_1 \equiv \vec{\tau}_2 \vdash p \downarrow \tau \Rightarrow \Delta; \Gamma}{\Delta_0 \vdash c[\vec{\alpha}](p) \downarrow (\vec{\tau}_2)T \Rightarrow \vec{\alpha}, \vec{\tau}_1 \equiv \vec{\tau}_2, \Delta; \Gamma}\ \textbf{(pat-cons)}$$

Clause typing rule $\quad \Delta; \Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2$

$$\frac{\Delta \vdash p \downarrow \tau_1 \Rightarrow (\Delta'; \Gamma') \quad \Delta, \Delta'; \Gamma, \Gamma' \vdash e : \tau_2}{\Delta; \Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2}$$

Clauses typing rule $\quad \Delta; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2$

$$\frac{\Delta; \Gamma \vdash p_i \Rightarrow e_i : \tau_1 \Rightarrow \tau_2 \ \text{ for } i = 1, \ldots, n}{\Delta; \Gamma \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau_1 \Rightarrow \tau_2}$$

Figure 5: Pattern typing rules

We use $\Delta \models \tau_1 \equiv \tau_2$ for a type constraint; this constraint is satisfied if we have $\vdash \tau_1[\Theta] \equiv \tau_2[\Theta]$ for every Θ such that $\vdash \Theta : \Delta$ is derivable. As can be expected, we have the following proposition.

PROPOSITION 2.1.

- *If $\Delta \vdash \tau : *$ is derivable, then $\Delta \models \tau \equiv \tau$ holds.*
- *If $\Delta \models \tau_1 \equiv \tau_2$ holds, then $\Delta \models \tau_2 \equiv \tau_1$ also holds.*
- *If $\Delta \models \tau_1 \equiv \tau_2$ and $\Delta \models \tau_2 \equiv \tau_3$ hold, then $\Delta \models \tau_1 \equiv \tau_3$ also holds.*

2.2 Solving Type Constraints

There is a need for solving type constraints of the form $\Delta \models \tau_1 \equiv \tau_2$ when we form typing rules for $\lambda_{2,G\mu}$. Fortunately, there is a decision procedure for doing this based on the set of rules in Figure 7. In these rules, we use T to range over all type constructors, either built-ins ($*$ and $\to$), user-defined g.r. datatype constructors, or skolemized constants.

THEOREM 2.2. *$\Delta \models \tau_1 \equiv \tau_2$ holds if and only if $\Delta \vdash \tau_1 \equiv \tau_2$ is derivable.*

Proof By induction on a derivation of $\Delta \vdash \tau_1 \equiv \tau_2$. ■

2.3 G.R. Datatype Constructors

We use $*$ as the kind for types and $(*, \ldots, *) \to *$ as the kind for type constructors of arity n, where n the number of $*$'s in $(*, \ldots, *)$. We use T for a recursive type constructor of arity n and associate with T a list of (value) constructors $c_1, \ldots, c_k$; for each $1 \le i \le k$, the type of c_i is of the form $\forall\vec{\alpha}_i.\tau_i \to (\vec{\tau}_i)T$, where $\vec{\tau}_i$ is for a sequence of types $\tau_1^i, \ldots, \tau_n^i$,

Expression typing rules $\quad \Delta; \Gamma \vdash e : \tau$

$$\frac{\Delta \models \tau_1 \equiv \tau_2 \quad \Delta; \Gamma \vdash e : \tau_1}{\Delta; \Gamma \vdash e : \tau_2} \textbf{ (ty-eq)}$$

$$\frac{\Gamma(xf) = \tau}{\Delta; \Gamma \vdash xf : \tau} \textbf{ (ty-var)}$$

$$\frac{\Sigma(c) = \forall \vec{\alpha}.\tau_1 \to \tau_2 \quad \Delta \vdash \vec{\tau} : \vec{*} \quad \Delta; \Gamma \vdash e : \tau_1[\vec{\alpha} \mapsto \vec{\tau}]}{\Delta; \Gamma \vdash c[\vec{\tau}](e) : \tau_2[\vec{\alpha} \mapsto \vec{\tau}]} \textbf{ (ty-cons)}$$

$$\frac{}{\Delta; \Gamma \vdash \langle\rangle : \mathbf{1}} \textbf{ (ty-unit)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2} \textbf{ (ty-tup)}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 * \tau_2}{\Delta; \Gamma \vdash \mathbf{fst}(e) : \tau_1} \textbf{ (ty-fst)}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 * \tau_2}{\Delta; \Gamma \vdash \mathbf{snd}(e) : \tau_2} \textbf{ (ty-snd)}$$

$$\frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2} \textbf{ (ty-lam)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1(e_2) : \tau_2} \textbf{ (ty-app)}$$

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \textbf{ (ty-tlam)}$$

$$\frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau \quad \Delta \vdash \tau_1 : *}{\Delta; \Gamma \vdash e[\tau_1] : \tau[\alpha \mapsto \tau_1]} \textbf{ (ty-tapp)}$$

$$\frac{\Delta; \Gamma, f : \tau \vdash e : \tau}{\Delta; \Gamma \vdash \mathbf{fix}\ f : \tau.e : \tau} \textbf{ (ty-fix)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2} \textbf{ (ty-let)}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \quad \Delta; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\Delta; \Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ ms : \tau_2} \textbf{ (ty-case)}$$

Figure 6: Typing rules for expressions

$$\frac{\vec{\alpha} \vdash \tau : *}{\vec{\alpha} \vdash \tau \equiv \tau} \qquad \frac{T \text{ is not } T'}{\vec{\alpha}, (\vec{\tau}_1)T \equiv (\vec{\tau}_2)T', \Delta \vdash \tau_1 \equiv \tau_2}$$

$$\frac{\vec{\alpha}, \Delta \vdash \tau_1 \equiv \tau_2}{\vec{\alpha}, \alpha \equiv \alpha, \Delta \vdash \tau_1 \equiv \tau_2}$$

$$\frac{\vec{\alpha}, \Delta[\alpha \mapsto \tau] \vdash \tau_1[\alpha \mapsto \tau] \equiv \tau_2[\alpha \mapsto \tau] \quad \alpha \text{ has no free occurrences in } \tau}{\vec{\alpha}, \alpha \equiv \tau, \Delta \vdash \tau_1 \equiv \tau_2}$$

$$\frac{\vec{\alpha}, \Delta[\alpha \mapsto \tau] \vdash \tau_1[\alpha \mapsto \tau] \equiv \tau_2[\alpha \mapsto \tau] \quad \alpha \text{ has no free occurrences in } \tau}{\vec{\alpha}, \tau \equiv \alpha, \Delta \vdash \tau_1 \equiv \tau_2}$$

$$\frac{\vec{\alpha}, \vec{\tau}_1 \equiv \vec{\tau}_2 \vdash \tau_1 \equiv \tau_2}{\vec{\alpha}, (\vec{\tau}_1)T \equiv (\vec{\tau}_2)T, \Delta \vdash \tau_1 \equiv \tau_2}$$

$$\frac{\vec{\alpha}, \tau_1'[\alpha_1 \mapsto (\vec{\alpha})A] \equiv \tau_2'[\alpha_2 \mapsto (\vec{\alpha})A] \vdash \tau_1 \equiv \tau_2 \quad A \text{ is a fresh skolemized constant}}{\vec{\alpha}, \forall\alpha_1.\tau_1' \equiv \forall\alpha_2.\tau_2' \vdash \tau_1 \equiv \tau_2}$$

Figure 7: Rules for solving type constraints

and $\forall\vec{\alpha}_i$ stands for a (possibly empty) sequence of quantifiers $\forall\alpha_1^i \ldots \forall\alpha_{n_i}^i$ (assuming $\vec{\alpha}_i = \alpha_1^i, \ldots, \alpha_{n_i}^i$). In our concrete syntax, T can be declared as follows.

$$\begin{array}{rcl} \mathit{typecon}\ (\mathit{type}, ..., \mathit{type})\ T & = & \{\vec{\alpha}_1\}.(\vec{\tau}_1)\ c_1 \text{ of } \tau_1 \\ & | & \{\vec{\alpha}_2\}.(\vec{\tau}_2)\ c_2 \text{ of } \tau_2 \\ & | & ... \\ & | & \{\vec{\alpha}_k\}.(\vec{\tau}_n)\ c_k \text{ of } \tau_k \end{array}$$

We present some simple examples of g.r. datatype constructors to facilitate the understanding of this concept.

Example 1 The following syntax

```
typecon TOP = Top of 'a
```

declares a value constructor *Top* of the type $\forall\alpha.\alpha \to \mathit{TOP}$; *TOP* is defined as $\mu t.\exists\{\alpha\}.\alpha$, which is equivalent to $\exists\alpha.\alpha$.

Example 2 The following syntax

```
typecon (type) list =
  ('a) nil | ('a) cons of 'a * 'a list
```

declares two constructors *nil* and *cons* of the types $\forall\alpha.\mathbf{1} \to (\alpha)\mathit{list}$ and $\forall\alpha.\alpha * (\alpha)\mathit{list} \to (\alpha)\mathit{list}$, respectively; the type constructor *list* is define as follows, which is essentially equivalent to the type constructor $\mu t.\lambda\alpha.\mathbf{1} + \alpha * (\alpha)t$.

$$\mu t.\lambda\alpha.\exists\{\alpha_1, \alpha_1 \equiv \alpha\}.\mathbf{1} + \exists\{\alpha_2, \alpha_2 \equiv \alpha\}.\alpha_2 * (\alpha_2)t$$

Note that the usual list type constructor in ML is defined as $\lambda\alpha.\mu t.\mathbf{1} + \alpha * t$.

2.4 Pattern Matching

We use p for patterns. As usual, either a type variable or a value variable may occur at most once in each pattern. We use a judgment of form $v \downarrow p \vdash (\Theta; \theta)$ to mean that matching a value v against a pattern p yields substitutions Θ and θ for the type and value variables in p. The rules for

deriving such judgments are listed as follows.

$$\frac{}{v \downarrow x \Rightarrow ([];[x \mapsto v])} \qquad \frac{}{\langle\rangle \downarrow \langle\rangle \Rightarrow ([];[])}$$

$$\frac{v_1 \downarrow p_1 \Rightarrow (\Theta_1;\theta_1) \quad v_2 \downarrow p_2 \Rightarrow (\Theta_2;\theta_2)}{\langle v_1, v_2\rangle \downarrow \langle p_1, p_2\rangle \Rightarrow (\Theta_1 \cup \Theta_2; \theta_1 \cup \theta_2)}$$

$$\frac{v \downarrow p \Rightarrow (\Theta;\theta)}{c[\vec{\tau}](v) \downarrow c[\vec{\alpha}](p) \Rightarrow ([\vec{\alpha} \mapsto \vec{\tau}] \cup \Theta; \theta)}$$

Given a type variable context Δ_0, a pattern p and a type τ, we can use the rules in Figure 5 to derive a judgment of the form $\Delta_0 \vdash p \downarrow \tau \Rightarrow (\Delta;\Gamma)$, whose meaning is formally captured by Lemma 2.4.

2.5 Static and Dynamic Semantics

We present the typing rules for $\lambda_{2,G\mu}$ in Figure 6. We assume the existence of a signature Σ in which the types of constructors are declared.

Most of the typing rules are standard. The type **(ty-eq)** indicates the type equality in $\lambda_{2,G\mu}$ is modulo type constraint solving. Please notice the great difference between the rules presented in Figure 5 for typing clauses and the "standard" ones in [15].

We form the dynamic semantics of $\lambda_{2,G\mu}$ through the use of evaluation contexts, which are defined below.

Evaluation context E ::=
$[] \mid \mathbf{fst}(E) \mid \mathbf{snd}(E) \mid \langle E, e\rangle \mid \langle v, E\rangle \mid E(e) \mid v(E) \mid E[\tau] \mid \mathbf{let}\ x = E\ \mathbf{in}\ e\ \mathbf{end} \mid \mathbf{case}\ E\ \mathbf{of}\ ms$

DEFINITION 2.3. *A redex is defined as follows.*

- $\mathbf{fst}(\langle v_1, v_2\rangle)$ *is a redex that reduces to* v_1.
- $\mathbf{snd}(\langle v_1, v_2\rangle)$ *is a redex that reduces to* v_2.
- $(\lambda x : \tau.e)(v)$ *is a redex that reduces to* $e[x \mapsto v]$.
- $(\Lambda\alpha.v)[\tau]$ *is a redex that reduces to* $v[\alpha \mapsto \tau]$.
- $\mathbf{let}\ x = v\ \mathbf{in}\ e\ \mathbf{end}$ *is a redex that reduces to* $e[x \mapsto v]$.
- $\mathbf{fix} f : \tau.v$ *is a redex that reduces to* $v[f \mapsto \mathbf{fix} f : \tau.v]$.
- $\mathbf{case}\ v\ \mathbf{of}\ ms$ *is a redex if* $v \downarrow p \Rightarrow (\Theta;\theta)$ *is derivable for some clause* $p \Rightarrow e$ *in* ms*, and the redex reduces to* $e[\Theta][\theta]$*. Note that there may be certain amount of nondeterminism in the reduction of* $\mathbf{case}\ v\ \mathbf{of}\ ms$ *as* v *may match the patterns in several clauses in* ms.

Given a redex e_1, we write $e_1 \hookrightarrow e_2$ if e_1 reduces to e_2. If $e_i' = E[e_i]$ for $i = 1, 2$ and e_1 is a redex reducing to e_2, then we write $e_1' \hookrightarrow e_2'$ and say that e_1' reduces to e_2' in one step. Let $\hookrightarrow^*$ be the reflexive and transitive closure of $\hookrightarrow$. We say that e_1 reduces to e_2 (in many steps) if $e_1 \hookrightarrow^* e_2$ holds.

Given a closed well-typed expression e in $\lambda_{2,G\mu}$, we use $|e|$ for the type erasure of e, that is, the expression obtained from erasing all types in e. We can then evaluate $|e|$ in a untyped λ-calculus extended with pattern matching. Clearly, $e \hookrightarrow^* e'$ holds if and only if $|e|$ evaluates to $|e'|$. In other words, $\lambda_{2,G\mu}$ supports type-erasure semantics.

2.6 Type Soundness

Given an expression variable context Γ such that $\Gamma(x)$ is a closed type for each $x \in \mathbf{dom}(\Gamma)$, we write $\theta : \Gamma$ if $\cdot;\cdot \vdash \theta(x) : \Gamma(x)$ is derivable for each $x \in \mathbf{dom}(\theta) = \mathbf{dom}(\Gamma)$. In general, we write $(\Theta;\theta) : (\Delta;\Gamma)$ to mean that $\vdash \Theta : \Delta$ is derivable and $\theta : \Gamma[\Theta]$ holds. The following lemma essentially verifies that the rules for deriving judgments of the form $p \downarrow \tau \Rightarrow (\Delta;\Gamma)$ are properly formed.

LEMMA 2.4. *Assume that* $\Delta_0 \vdash p \downarrow \tau \Rightarrow (\Delta;\Gamma)$ *is derivable and* $\Theta_0 : \Delta_0$ *holds. If* v *is a closed value of type* $\tau[\Theta_0]$*, that is,* $\cdot;\cdot \vdash v : \tau[\Theta_0]$ *is derivable, and we have* $v \downarrow p \Rightarrow (\Theta,\theta)$ *for some* Θ *and* θ*, then* $(\Theta;\theta) : (\Delta[\Theta_0];\Gamma[\Theta_0])$ *holds.*

Proof By structural induction on a derivation of $\Delta_0 \vdash p \downarrow \tau \Rightarrow (\Delta;\Gamma)$ ■

As usual, we need the following substitution lemma to establish the subject reduction theorem for $\lambda_{2,G\mu}$.

LEMMA 2.5. *Assume that* $\Delta;\Gamma \vdash e : \tau$ *is derivable. If* $\vdash (\Theta;\theta) : (\Delta;\Gamma)$ *holds, then* $\cdot;\cdot \vdash e[\Theta][\theta] : \tau[\Theta]$ *is derivable.*

Proof By structural induction on a derivation of $\Delta;\Gamma \vdash e : \tau$. ■

THEOREM 2.6. *(Subject Reduction) Assume that* $\cdot;\cdot \vdash e : \tau$ *is derivable. If* $e \hookrightarrow e'$ *holds, then* $\cdot;\cdot \vdash e' : \tau$ *is also derivable.*

Proof Assume that $e = E[e_1]$ and $e' = E[e_2]$ for some redex e_1 that reduces to e_2. The proof follows from structural induction on E. In the case where $E = []$, the proof proceeds by induction on the height of a derivation of $\cdot;\cdot \vdash e : \tau$, handling various cases through the use of Lemma 2.5. For handling the typing rule **(ty-case)**, Lemma 2.4 is needed. ■

However, we *cannot* prove that if e is a well-typed non-value expression then e must reduce to another well-typed expression. In the case where $e = E[e_1]$ for some $e_1 = \mathbf{case}\ v\ \mathbf{of}\ ms$ that is not a redex (because v does not match any pattern in ms), the evaluation of e becomes stuck. This is so far the only reason for the evaluation of an expression to become stuck.

3. IMPLEMENTING OBJECTS

In this section, we briefly outline an approach to implementing objects through the use of g.r. datatype constructors.

3.1 Classes

In Section 1, we have noticed a serious problem with the type *OBJ*, as it allows no differentiation of objects. We address this problem by providing the type constructor *MSG* with another parameter. Given a type τ and a class C, $(\tau)MSG(C)$ is a type; the intuition is that a message of type $(\tau)MSG(C)$ should only be sent to objects in the class C, to which we assign the type $OBJ(C)$ defined as follows:

$$OBJ(C) = \forall\alpha.(\alpha)MSG(C) \to \alpha$$

First and foremost, we emphasize that a class is *not* a type; it is really a tag used to differentiate messages. For instance, we may declare a class *IntPairClass* and associate with it the

```
fun newPair x y = let
  val xref = ref x
  val yref = ref y
  fun dispatch MSGgetfst = !xref
    | dispatch MSGgetsnd = !yref
    | dispatch (MSGsetfst x') = (xref := x')
    | dispatch (MSGsetsnd y') = (yref := y')
    | dispatch msg = UnknownMessageError (msg)
in dispatch end
withtype {'a,'b}. 'a -> 'b -> OBJ(PairClass('a,'b))
```

Figure 8: A constructor for pairs

following message constructors of the corresponding types:

$$\begin{array}{rcl} \mathit{MSGgetfst} & : & (\mathtt{int})\mathit{MSG}(\mathit{IntPairClass}) \\ \mathit{MSGgetsnd} & : & (\mathtt{int})\mathit{MSG}(\mathit{IntPairClass}) \\ \mathit{MSGsetfst} & : & \mathtt{int} \to (\mathbf{1})\mathit{MSG}(\mathit{IntPairClass}) \\ \mathit{MSGsetsnd} & : & \mathtt{int} \to (\mathbf{1})\mathit{MSG}(\mathit{IntPairClass}) \end{array}$$

The function *newIntPair* can now be given the type $\mathtt{int} \to \mathtt{int} \to \mathit{OBJ}(\mathit{IntPairClass})$. Since *anIntPair* has the type $\mathit{OBJ}(\mathit{IntPairClass})$, $\mathit{anIntPair}(\mathit{MSGfoo})$ becomes ill-typed if *MSGfoo* has a type $(\mathbf{1})\mathit{MSG}(C)$ for some class C that is not *IntPairClass*. Although classes can be treated as types *syntactically*, we feel it better to treat them as type index expressions. Following Dependent ML [20, 18], we use `class` as the sort for classes. In the following presentation, we assume the availability of g.r. datatype constructors in DML.

3.2 Parameterized Classes

There is an immediate need for class tags parameterizing over types. Suppose we are to generalize the monomorphic function *newIntPair* into a polymorphic function *newPair*, which can take arguments x and y of any types and then return an object representing the pair whose first and second components are x and y, respectively. We need a class constructor *PairClass* that takes two given types τ_1 and τ_2, and forms a class $(\tau_1, \tau_2)\mathit{PairClass}$. We may use some syntax to declare such a class constructor and associate with it the following polymorphic message constructors:

$$\begin{array}{rcl} \mathit{MSGgetfst} & : & \forall\alpha.\forall\beta.(\alpha)\mathit{MSG}((\alpha,\beta)\mathit{PairClass}) \\ \mathit{MSGgetsnd} & : & \forall\alpha.\forall\beta.(\beta)\mathit{MSG}((\alpha,\beta)\mathit{PairClass}) \\ \mathit{MSGsetfst} & : & \forall\alpha.\forall\beta.\alpha \to (\mathbf{1})\mathit{MSG}((\alpha,\beta)\mathit{PairClass}) \\ \mathit{MSGsetsnd} & : & \forall\alpha.\forall\beta.\beta \to (\mathbf{1})\mathit{MSG}((\alpha,\beta)\mathit{PairClass}) \end{array}$$

The function *newPair* for constructing pair objects is implemented in Figure 9.

3.3 Subclasses

Inheritance is a major issue in object-oriented programming as it can significantly facilitate code organization and reuse. We approach the issue of inheritance by introducing a predicate $\leq$ on the sort `class`; given two classes C_1 and C_2, $C_1 \leq C_2$ means that C_1 is a subclass of C_2. The type of a message constructor *mc* is now of the general form $\forall\vec{\alpha}.\Pi a \lhd C.(\tau)\mathit{MSG}(a)$ or $\forall\vec{\alpha}.\Pi a \lhd C.\tau_1 \to (\tau_2)\mathit{MSG}(a)$, where $a \lhd C$ means that a is of the subset sort $\{a : \mathtt{class} \mid a \leq C\}$, i.e., the sort for all subclasses of the class C; for a sequence of types $\vec{\tau}$ with the same length as $\vec{\alpha}$, $\mathit{mc}[\vec{\tau}]$ becomes a message constructor that is polymorphic on all subclasses of $C_0 = C[\vec{\alpha} \mapsto \vec{\tau}]$; therefore, *mc* can be used to

```
fun newPair x y = let
  val xref = ref x and yref = ref y
  fun dispatch MSGgetfst = !xref
    | dispatch MSGgetsnd = !yref
    | dispatch (MSGsetfst x') = (xref := x')
    | dispatch (MSGsetsnd y') = (yref := y')
    | dispatch msg = raise UnknownMessage
in dispatch end
withtype {'a,'b}. 'a -> 'b -> OBJ(('a,'b)PairClass)

fun newColoredPair c x y = let
  val cref = ref c
  and xref = ref x
  and yref = ref y
  fun dispatch MSGgetcolor = !cref
    | dispatch (MSGsetcolor c') = (cref := c')
    | dispatch MSGgetfst = !xref
    | dispatch MSGgetsnd = !yref
    | dispatch (MSGsetfst x') = (xref := x')
    | dispatch (MSGsetsnd y') = (yref := y')
    | dispatch msg = raise UnknownMessage
in dispatch end
withtype {'a,'b}. color -> 'a -> 'b ->
                  OBJ (('a,'b)ColoredPairClass)
```

Figure 9: Functions for constructing objects in the classes *PairClass* and *ColoredPairClass*

construct a message for any object tagged by a subclass of the class C_0. For instance, the message constructors associated with *PairClass* are now assigned the types in Figure 10. Suppose we introduce another class constructor *ColoredPairClass*, which takes two types to form a class. Also assume the following, i.e., $(\tau_1, \tau_2)\mathit{ColoredPairClass}$ is a subclass of $(\tau_1, \tau_2)\mathit{PairClass}$ for any types τ_1 and τ_2:

$$\forall\alpha\forall\beta.(\alpha,\beta)\mathit{ColoredPairClass} \leq (\alpha,\beta)\mathit{PairClass}$$

We then associate with *ColoredPairClass* the message constructors *MSGgetcolor* and *MSGsetcolor*, which are assigned the types in Figure 10. We can then implement the function *newColoredPair* in Figure 9 for constructing colored pairs. Clearly, the implementation of *newColoredPair* shares a lot of common code with that of *newPair*. We will provide proper syntax later so that the programmer can efficiently reuse the code in the implementation of *newPair* when implementing *newColoredPair*.

3.4 Binary Methods

Our approach to typed object-oriented programming offers a particularly clean solution to handling binary methods. For instance, we can declare a class *EqClass* and associate with it two message constructors *MSGeq* and *MSGneq* which are given the following types:

$$\begin{array}{rcl} \mathit{MSGeq} & : & \Pi a \lhd \mathit{EqClass}.\mathit{OBJ}(a) \to (\mathtt{bool})\mathit{MSG}(a) \\ \mathit{MSGneq} & : & \Pi a \lhd \mathit{EqClass}.\mathit{OBJ}(a) \to (\mathtt{bool})\mathit{MSG}(a) \end{array}$$

Suppose *self* is an object of type $\mathit{OBJ}(C)$ for some $C \leq \mathit{Eq}$. If we pass a message $\mathit{MSGeq}(\mathit{other})$ to *self*, *other* is required to have the type $\mathit{OBJ}(C)$ in order for $\mathit{self}(\mathit{MSGeq}(\mathit{other}))$ to be well-typed. Unfortunately, such a requirement cannot be enforced by the type system of Java; as a consequence,

$$
\begin{array}{rcl}
MSGgetfst & : & \forall\alpha.\forall\beta.\Pi a \lhd (\alpha,\beta)PairClass.(\alpha)MSG(a) \\
MSGgetsnd & : & \forall\alpha.\forall\beta.\Pi a \lhd (\alpha,\beta)PairClass.(\beta)MSG(a) \\
MSGsetfst & : & \forall\alpha.\forall\beta.\Pi a \lhd (\alpha,\beta)PairClass.\alpha \rightarrow (\mathbf{1})MSG(a) \\
MSGsetsnd & : & \forall\alpha.\forall\beta.\Pi a \lhd (\alpha,\beta)PairClass.\beta \rightarrow (\mathbf{1})MSG(a) \\
MSGgetcolor & : & \forall\alpha\forall\beta.\Pi a \lhd (\alpha,\beta)ColoredPairClass.(color)MSGgetcolor(a) \\
MSGsetcolor & : & \forall\alpha\forall\beta.\Pi a \lhd (\alpha,\beta)ColoredPairClass.color \rightarrow (\mathbf{1})MSGsetcolor(a)
\end{array}
$$

Figure 10: Some message constructors and their types

type downcasts are often needed for implementing and testing equality on objects.

3.5 The Self Type

Our approach also offers a particularly clean solution to handling the notion of *self type*, namely, the type of the receiver of a message. Suppose we want to support a message *MSGcopy* that can be sent to any object to obtain a copy of the object.[3]. We may assume *MSGcopy* is a message constructor associated with some class *ObjClass* and $C \leq ObjClass$ holds for any class C. We can assign *MSGcopy* the following type to indicate that the returned object is in the same class as the object to which the message is sent.

$$MSGcopy \quad : \quad \Pi a \lhd ObjClass.(OBJ(a))MSG(a)$$

If this is done in Java, all we can state in the type system of Java is that an object is to return another object after receiving the message *MSGcopy*. This is imprecise and is a rich source for the use of type downcasting.

3.6 Inheritance

Inheritance is done in a Smalltalk-like manner, but there is some significant difference. We now use a concrete example to illustrate how inheritance can be implemented. This is also a proper place for us to introduce some syntax that is designed to facilitate object-oriented programming. We use the following syntax to declare a class *ObjClass* and a message constructor *MSGcopy* of the type:

$$\Pi a \lhd ObjClass.(OBJ(a))MSG(a)$$

Note `selfType` is merely syntactic sugar here.

```
class ObjClass {
  MSGcopy: selfType => self;
}
```

In addition, the syntax also *automatically* induces the definition of a function *superObj*, which is written as follows in ML-like syntax.

```
(* self is just an ordinary variable *)
fun superObj self = let
  fun dispatch MSGcopy = self
    | dispatch msg = raise UnknownMessage
in dispatch end
withtype {a <: ObjClass} OBJ(a) -> OBJ(a)
```

The function *superObj* we present here is solely for explaining how inheritance can be implemented; such a function is not to occur in a source program. The type of the function $\Pi a \lhd ObjClass.OBJ(a) \rightarrow OBJ(a)$ indicates this is a function that takes an object tagged by a subclass C of *ObjClass* and returns an object tagged by the same class. In general, for each class C, a "super" function of type $\Pi a \lhd C.OBJ(a) \rightarrow OBJ(a)$ is associated with C. It should soon be clear that such a function holds the key to implementing inheritance. Now we use the following syntax to declare classes *Int1Class* and *ColoredInt1Class* as well as some message constructors associated with them.

```
class Int1Class inherits ObjClass {
  MSGget_x: int;
  MSGset_x (int): unit;
  MSGdouble: unit =>
    self(MSGset_x(2 * self(MSGget_x));
}

class ColoredInt1Class inherits Int1Class {
  (* color is just some already defined type *)
  MSGget_c: color;
  MSGset_c (color): unit;
}
```

The "super" functions associated with the classes *Int1Class* and *ColoredInt1Class* are automatically induced as follows.

```
fun superInt1 self = let
  fun dispatch MSGdouble =
      self(MSGset_x(2 * self(MSGget_x)))
    | dispatch msg = superObj self msg
in dispatch end
withtype {a <: Int1Class} OBJ(a) -> OBJ(a)

fun superColoredInt1 self = let
  fun dispatch msg = superInt1 self msg
in dispatch end
withtype {a <: ColoredInt1Class} OBJ(a) -> OBJ(a)
```

The functions for constructing objects in the classes *Int1Class* and *ColoredInt1Class* are implemented in Figure 11. There is something really interesting here. Suppose we use *newInt1* and *newColoredInt1* to construct objects o_1 and o_2 that are tagged with *Int1Class* and *ColoredInt1Class*, respectively. If we send the message *MSGcopy* to o_1, then a copy of o_1 (not o_1 itself) is returned. If we send *MSGdouble* to o_2, then the integer value of o_2 is doubled as it inherits the corresponding method from the class *Int1Class*. What is remarkable is that the object o_2 itself is returned if we send the message *MSGcopy* to o_2. The reason is that no copying method is defined for o_2; searching for a copying method, o_2 eventually finds the one defined in the class *ObjClass* (as there is no such a method defined in either the class *ColoredInt1Class* or the class *Int1Class*). This is a desirable consequence: if o_2 were treated as an object in the

[3]It is up to the actual implementation as to how such a copy can be constructed.

```
fun newInt1 (x0: int) = let
  val x = ref x0
  fun dispatch MSGget_x = !x
    | dispatch (MSGset_x x') = (x := x')
    | dispatch MSGcopy = newInt1 (!x)
    | dispatch msg = superInt1 dispatch msg
in dispatch end
withtype int -> OBJ(Int1Class)

fun newColoredInt1 (c0: color, x0: int) = let
  val c = ref c0 and x = ref x0
  fun dispatch MSGget_c = !c
    | dispatch (MSGset_c c') = (c := c')
    | dispatch MSGget_x = !x
    | dispatch (MSGset_x x') = (x := x')
    | dispatch msg = superColoredInt1 dispatch msg
in dispatch end
withtype int -> OBJ(ColoredInt1Class)
```

Figure 11: Functions for constructing objects in *Int1Class* and *ColoredInt1Class*

class *Int1Class* (through either F-bounded polymorphism or match-bounded polymorphism), the returned object would be in the class *Int1Class*, not in the class *ColoredInt1Class*, as it would be generated by $newInt1(o_2(MSGget\_x))$, making the type system unsound. We are currently not aware of any other approach to correctly typing this simple example. Note that the function *newInt* becomes ill-typed if we employ the notion *MyType* here.

3.7 Subtyping

There is not an explicit subtyping relation in our approach. Instead, we can use existentially quantified dependent types to simulate subtyping. For instance, given a class tag C, the type $OBJECT(C) = \Sigma a \lhd C.OBJ(a)$ is the sum of all types $OBJ(a)$ satisfying $a \leq C$. Hence, for each $C_1 \leq C$, $OBJ(C_1)$ can be regarded as a subtype of $OBJECT(C)$ as each value of the type $OBJ(C_1)$ can be coerced into a value of the type $OBJECT(C)$. As an example, the type

$$OBJ((OBJECT(EqClass), OBJECT(EqClass))PairClass)$$

is for pair objects whose both components support equality test.

4. RELATED WORK AND CONCLUSION

Our work is related to both intentional polymorphism and type classes.

There have already been a rich body of studies in the literature on passing types at run-time in a type-safe manner [11, 10, 17]. Many of such studies follow the framework in [13], which essentially provides a construct `typecase` at term level to perform type analysis and a primitive recursor `Typerec` over type names at type level to define new type constructors. The language λ_i^{ML} in [13] is subsequently extended to λ_R in [11] to support type-erasure semantics. The type constructor R in λ_R can be seen as a special g.r. datatype constructor.

The system of type classes in Haskell provides a programming methodology that is of great use in practice. A common approach to implementing type classes is through dictionary-passing, where a dictionary is essentially a record of the member functions for a particular instance of a type class [1]. We encountered the notion of g.r. datatype constructors when seeking an alternative implementation of type classes through intensional polymorphism. An approach to implementing type classes through the use of g.r. datatype constructors can be found at [19].

The dependent datatypes in DML [20, 18] also shed some light on g.r. datatype constructors. For instance, we can have the following dependent datatype declaration in DML.

```
datatype 'a list with nat =
  nil(0) | {n:nat} cons(n+1) of 'a * 'a list(n)
```

The syntax introduces a type constructor *list* that takes a type and a type index of sort *nat* to form a list type. The constructors *nil* and *cons* are assigned the following types.

$$\begin{array}{rcl} nil & : & \forall\alpha.(\alpha)list(0) \\ cons & : & \forall\alpha.\alpha * (\alpha)list(n) \rightarrow (\alpha)list(n+1) \end{array}$$

Given a type τ and natural number n, the type $(\tau)list(n)$ is for lists with length n in which each element has the type τ. Formally, the type constructor *list* can be defined as follows:

$$\lambda\alpha.\mu t.\lambda a : nat.\exists\{0 = a\}.\mathbf{1} + \exists\{a' : nat, a' + 1 = a\}.\alpha * t(a')$$

Clearly, this is also a form of guarded datatype constructor, where the guards are constraints on type index expressions (rather than on types).

Our notion of objects in this paper is largely taken from Smalltalk [12], for which a particularly clean and intuitive articulation can be found in [14]. The literature on types in object-oriented programming is simply too vast for us to give an even modestly comprehensive overview of the related work. Please see [5] for references. Instead, we focus on some closely related work that either directly influences or motivates our current work.

Bounded polymorphism [8, 6] essentially imposes subtyping restrictions on quantified type variables. For instance, suppose we want to implement a class for ordered sequences. In order to insert an element into a sequence, we must compare it with other elements in the sequence. Therefore, we should only insert elements of a class that provide the appropriate methods for comparison. This can be achieved through bounded polymorphism.

F-bounded polymorphism [7], which generalizes the simple bounded polymorphism, was introduced to handle some complex issues such as typing binary methods in object-oriented programming. It has since been adopted in the design of GJ [2], helping to significantly increase the expressiveness of the type system of Java. However, F-bounded polymorphism does not seem to interact well with the subclass relation (e.g., please see the example on page 59 [5]).

Matching-bounded polymorphism is similar to bounded polymorphism. The main difference is that matching constraints are imposed on quantified type variables instead of subtyping constraints. The notion of *MyType* [4] essentially refers to the type of the receiving object of a message. With match-bounded polymorphism, the notion of *MyType* can allow the possibility of dispensing with most of the uses of F-bounded polymorphism. The language in [4] is really a state-of-the-art object-oriented programming language (when static typing is concerned). This work is carried further in [3], where imperative features are introduced. The type system that we are to design shares many common features with the work in [4], though we employ a completely

different type-theoretical approach. In particular, we intend to not only simplify the notion of *MyType* but also make it more effective in capturing program invariants.

Currently, we are particularly interested in implementing a CLOS-like object system on the top of DML extended with g.r. datatype constructors, facilitating object-oriented programming styles in a typed functional programming setting.

5. REFERENCES

[1] Lennart Augustsson. Implementing Haskell overloading. In *Functional Programming Languages and Computer Architecture*, 93.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Objected-Oriented Programming Systems, Languages, Applications (OOPSLA)*, Vancouver, BC, 1998.

[3] Kim Bruce, A. Schuett, and R. van Gent. a type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming*, pages 27–51. Springer-Verlag LNCS 933, 1995.

[4] Kim B. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[5] Kim B. Bruce. *Foundations of Object-Oriented Languages*. The MIT Press, Cambridge, MA, 2002.

[6] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance and bounded quantification. In *Proceedings Third Annual Symposium on Logic in Computer Science*, pages 38–50, Edinburgh, Scotland, 1988.

[7] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Programming and Computer Architecture (FPCA)*, pages 273–280, 1989.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[9] Alonzo Church. A formulation of the simple type theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[10] Karl Crary and Stephanie Weirich. Flexible Type Analysis. In *Proceedings of International Conference on Functional Programming (ICFP '99)*, Paris, France, 1999.

[11] Karl Crary, Stephanie Weirich, and Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the International Conference on Functional Programming (ICFP '98)*, pages 301–312, Baltimore, MD, September 1998.

[12] A. Goldenberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, 1983.

[13] Robert W. Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, 1995.

[14] Chamond Liu. *Smalltalk, Objects, and Design*. Manning Publications Co., Greenwich, CT 06830, 1996.

[15] Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[16] Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2002.

[17] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully Reflexive Intensional Type Analysis. In *Proceedings of the International Conference on Functional Programming*, September 1999.

[18] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[19] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors, 2002. Available at `http://www.cs.bu.edu/~hwxi/GRecTypecon/`.

[20] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999.