Guarded Recursive Datatype Constructors

Hongwei Xi[*] Chiyan Chen[*] Gang Chen[†]
Boston University
{hwxi, chiyan, gangchen}@cs.bu.edu

ABSTRACT

We introduce a notion of guarded recursive (g.r.) datatype constructors, generalizing the notion of recursive datatypes in functional programming languages such as ML and Haskell. We address both theoretical and practical issues resulted from this generalization. On one hand, we design a type system to formalize the notion of g.r. datatype constructors and then prove the soundness of the type system. On the other hand, we present some significant applications (e.g., implementing objects, implementing staged computation, etc.) of g.r. datatype constructors, arguing that g.r. datatype constructors can have far-reaching consequences in programming. The main contribution of the paper lies in the recognition and then the formalization of a programming notion that is of both theoretical interest and practical use.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative Languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Datatypes and Structures*

General Terms

Languages, Theory

Keywords

Guarded Recursive Datatype Constructors

1. INTRODUCTION

Although we have since found a variety of applications of guarded recursive (g.r.) datatype constructors, we initially encountered this notion in a study on run-time type-passing. In parametric polymorphism, there is no facility for a polymorphic function to inspect its type arguments and such a function behaves uniformly on all possible type arguments. As a consequence, there is no need for passing types at run-time. However, there are numerous occasions in practice where we would want polymorphic functions to behave differently on different type arguments. This is often called ad-hoc polymorphism. For instance, we may want to implement a function *val2string* for converting run-time values into string representations. In order to construct such a function, we need to analyze the types of values at run-time. With run-time type analysis, we can also support various advanced implementation techniques such as flattened data representation [10], polymorphic marshaling [16], unboxed function arguments [12], tag-free garbage collection [20], etc.

A (conceptually) simple and direct approach to run-time type analysis is to pass types or, more precisely, terms representing types at run-time and then inspect the structure of such terms. In order to capture the relation between a type and its term representation, we can declare a g.r. datatype constructor *TY* as follows.

```
typecon (type) TY =
  (int) TYint
| {'a,'b}.('a * 'b) TYtup of 'a TY * 'b TY
| {'a,'b}.('a -> 'b) TYfun of 'a TY * 'b TY
| {'a}.('a TY) TYtyp of 'a TY
```

The syntax `typecon (type) TY` means that *TY* is a type constructor of the kind $* \rightarrow *$, that is, *TY* takes one type to form another type. There are four value constructors *TYint*, *TYtup*, *TYfun* and *TYtyp* associated with *TY*, which are assigned the following types.

$$\begin{array}{rcl} \mathit{TYint} & : & (\mathit{int})\,\mathit{TY} \\ \mathit{TYtup} & : & \forall\alpha\forall\beta.(\alpha)\,\mathit{TY} * (\beta)\,\mathit{TY} \rightarrow (\alpha * \beta)\,\mathit{TY} \\ \mathit{TYfun} & : & \forall\alpha\forall\beta.(\alpha)\,\mathit{TY} * (\beta)\,\mathit{TY} \rightarrow (\alpha \rightarrow \beta)\,\mathit{TY} \\ \mathit{TYtyp} & : & \forall\alpha\forall\beta.(\alpha)\,\mathit{TY} \rightarrow ((\alpha)\,\mathit{TY})\,\mathit{TY} \end{array}$$

For instance, the following value,

$$\mathit{TYfun}(\langle \mathit{TYtup}(\langle \mathit{TYint}, \mathit{TYint}\rangle), \mathit{TYint}\rangle)$$

which has the type $(\mathtt{int} * \mathtt{int} \rightarrow \mathtt{int})TY$, represents the type $\mathtt{int} * \mathtt{int} \rightarrow \mathtt{int}$. In general, a value of the type $(\tau)\,TY$ represents the type τ. Now we can implement the *val2string* function as follows.

```
fun val2string TYint x = int2string x
  | val2string (TYtup (pf1, pf2)) (x1, x2) =
    "(" ^ val2string pf1 x1
        ^ "," ^ val2string pf2 x2 ^ ")"
  | val2string (TYfun _) _  = "[a function value]"
  | val2string (TYtyp _) _  = "[a type value]"
withtype {'a}. 'a TY -> 'a -> string
```

[*]Partially supported by the NSF Grants No. 0224244 and No. CCR-0229480
[†]Partially supported by the NSF Grant No. 9988529.

The `withtype` clause in the definition is a type annotation, which assigns the type $\forall\alpha.(\alpha)\,TY \rightarrow \alpha \rightarrow \texttt{string}$ to the defined function *val2string*.

The above idea of using terms to represent types can already be found in [6, 22], where a typed calculus λ_R is introduced to facilitate type-passing. The type constructor R in λ_R corresponds to *TY*, and its associated value constructors R_{int}, $\mathrm{R}_{\times}$, $\mathrm{R}_{\rightarrow}$ and R_R correspond to *TYint*, *TYtup*, *TYfun* and *TYtyp*, respectively. However, with g.r. datatype constructors, we can also form interesting and useful types that cannot be handled in λ_R. For instance, we can declare a g.r. datatype constructors *HOAS* as follows.

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

This declaration indicates that *HOAS* is a unary type constructor, and the value constructors associated with *HOAS* are assigned the types in Figure 1. The type constructor *HOAS*, which is intended to construct types for a form of higher-order abstract syntax trees [4, 18], *cannot* be inductively defined because of the type of the value constructor *HOASfix*. Given a type τ, $(\tau)HOAS$ is the type for higher-order abstract syntax trees representing monomorphically typed expressions of the type τ. For instance, the following expression in ML, which has the type $(\texttt{int} \rightarrow \texttt{int}) \rightarrow \texttt{int} \rightarrow \texttt{int}$,

```
      fn (x:int -> int) => fn (y:int) => x(y)
```

can be represented as follows:

```
 HOASlam(fn (x:(int -> int) HOAS) =>
      HOASlam (fn (y:int HOAS) => HOASapp (x, y)))
```

Note that the type of the above expression is $((\texttt{int} \rightarrow \texttt{int}) \rightarrow \texttt{int} \rightarrow \texttt{int})HOAS$. Furthermore, we can implement the following function *eval* for evaluating higher-order abstract syntax trees.

```
fun eval (HOASlift v) = v
  | eval (HOAStup (e1, e2)) = (eval e1, eval e2)
  | eval (HOASlam f) = fn x => eval (f (HOASlift x))
  | eval (HOASapp (e1, e2)) = (eval e1) (eval e2)
  | eval (e as HOASfix f) = eval (f e)
withtype {'a}. 'a HOAS -> 'a
```

Note *eval* is assigned the type $\forall\alpha.(\alpha)HOAS \rightarrow \alpha$, indicating that the evaluation of higher-order abstract syntax trees is type-preserving. Later, we will show that the type constructor *HOAS* can play a key rôle in implementing staged computation [7, 19].

The introduction of g.r. datatype constructors raises a number of theoretical and practical issues. We briefly outline our results and design decisions.

The first and foremost issue that arises is the formalization of g.r. datatype constructors in type theory. We show how fixed-point operators over type constructors of higher kinds can be used to formally define g.r. datatype constructors. In particular, we show how the standard fold/unfold operations (for recursive types) and injection operations (for sum types) can be used to define the value constructors associated with a declared g.r. datatype constructor.

The second issue is the decidability and practicality of type-checking in the presence of g.r. datatype constructors. We address this question in two steps. We first present an explicitly typed (and overly verbose) internal language $\lambda_{2,G\mu}$ in which type-checking can be handled straightforwardly. We next present an external language $\mathrm{ML}_{2,G\mu}$, a slightly extended fragment of ML, and then mention an elaboration process from $\mathrm{ML}_{2,G\mu}$ to $\lambda_{2,G\mu}$ that preserves the standard operational semantics. When programming in $\mathrm{ML}_{2,G\mu}$, the programmer may omit writing types at various occasions (determined by the elaboration process).

The third issue is the usefulness of g.r. datatype constructors in practice. We present various examples to illustrate some interesting applications of g.r. datatype constructors in capturing program invariants. In particular, we show that g.r. datatype constructors can be combined with a restricted form of dependent types [25, 23] to type programming objects, overcoming some significant deficiencies in many existing type systems for object-oriented programming [2].

In summary, we present a generalization of the notion of recursive datatypes, allowing the programmer to form g.r. datatype constructors. The most significant contribution of the paper lies in the recognition of such a simple and natural generalization that can have far-reaching consequences in programming. We formalize the notion of g.r. datatype constructors in a type system and then establish the type soundness of the type system, which constitutes the main technical contribution of the paper. We also present various realistic and interesting examples in support of the introduction of g.r. datatype constructors into a programming language.

We organize the rest of the paper as follows. In Section 2, we introduce an internal language $\lambda_{2,G\mu}$ to formalize the notion of g.r. datatype constructors, presenting both static and dynamic semantics for $\lambda_{2,G\mu}$ and proving its type soundness. We next form an external language $\mathrm{ML}_{2,G\mu}$ in Section 3 and then mention an elaboration process from $\mathrm{ML}_{2,G\mu}$ to $\lambda_{2,G\mu}$ to support unobtrusive programming. We use some interesting examples in Section 4 to illustrate various applications of g.r. datatype constructors. Lastly, we mention some future research directions and related works.

There is a full version of the paper on-line [24], where various omitted details can be found.

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.

The language $\lambda_{2,G\mu}$ is designed to be an internal language. Later, we will also present an external language $\mathrm{ML}_{2,G\mu}$ and mention an elaboration process from $\mathrm{ML}_{2,G\mu}$ to $\lambda_{2,G\mu}$. When programming in $\mathrm{ML}_{2,G\mu}$, the programmer may omit writing certain types, which can be reconstructed through the elaboration process.

2.1 Syntax

We present the syntax for $\lambda_{2,G\mu}$ in Figure 2, 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.[1] Also, we require that the

[1]For a constructor taking no argument, we can treat it as a

$$\begin{array}{rcl}
\mathit{HOASlift} & : & \forall\alpha.\alpha \to (\alpha)\mathit{HOAS} \\
\mathit{HOAStup} & : & \forall\alpha\forall\beta.(\alpha)\mathit{HOAS} * (\beta)\mathit{HOAS} \to (\alpha * \beta)\mathit{HOAS} \\
\mathit{HOASlam} & : & \forall\alpha\forall\beta.((\alpha)\mathit{HOAS} \to (\beta)\mathit{HOAS}) \to (\alpha \to \beta)\mathit{HOAS} \\
\mathit{HOASapp} & : & \forall\alpha\forall\beta.(\alpha \to \beta)\mathit{HOAS} * (\alpha)\mathit{HOAS} \to (\beta)\mathit{HOAS} \\
\mathit{HOASfix} & : & \forall\alpha.((\alpha)\mathit{HOAS} \to (\alpha)\mathit{HOAS}) \to (\alpha)\mathit{HOAS}
\end{array}$$

Figure 1: The value constructors associated with *HOAS* and their types

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

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

body of either Λ or **fix** be a value. The syntax for patterns is to be explained in Section 2.3.

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 $\lambda_{2,G\mu}$. 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 the form $\vdash \Theta : \Delta$, which roughly means that Θ matches Δ.

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

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.*

constructor taking the unit $\langle\rangle$ as its argument.

$$\boxed{\text{Pattern typing rules} \quad \Delta_0 \vdash p \downarrow \tau \Rightarrow (\Delta; \Gamma)}$$

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

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

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

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

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

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

$$\boxed{\text{Clauses typing rule} \quad \Delta; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}$$

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

Figure 3: Pattern typing rules

$$\boxed{\text{Expression typing rules} \quad \Delta; \Gamma \vdash e : \tau}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4: Typing rules for expressions

At this moment, there is no need to be concerned with how type constraints can be solved; we may simply assume the existence of an oracle for doing this. In Section 3, we will present a complete procedure for solving type constraints.

2.2 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 the occurrences of $*$ 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 $1 \leq i \leq k$, 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 $\tau^i_1, \ldots, \tau^i_{n_i}$, and $\forall \vec{\alpha}_i$ stands for a (possibly empty) sequence of quantifiers $\forall \alpha^i_1 \ldots \forall \alpha^i_{m_i}$ (assuming $\vec{\alpha}_i = \alpha^i_1, \ldots, \alpha^i_{m_i}$). In our concrete syntax, T can be declared as follows.

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

We write $\exists\Delta.\tau$ for a guarded type, where Δ is a type variable context that may contain some type equalities. We use the name *guard* for such a type equality. 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 $\texttt{int}$ in order to satisfy the type equality $\alpha_1 * \alpha_2 \equiv \texttt{int} * \texttt{bool}$. 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 equality $\alpha_1 * \alpha_2 \equiv \texttt{int}$ cannot be satisfied. For $\Delta = (\alpha_1, \alpha_2, \alpha_1 * \alpha_2 \equiv \alpha)$, we observe that the type constructor $\lambda\alpha\exists\Delta.\tau$ has the following interesting feature: when applied to a type τ_0, the type constructor forms a type that is equivalent to $\tau_1 * \tau_1$ if τ_0 is of the form $\tau_1 * \tau_2$, or `void` otherwise.

We show that the type constructor T can be formally defined as $\mu t.\sigma$, where σ is the following sum of guarded types. Therefore, we call T a *guarded recursive datatype constructor.*

$$\mu t.\lambda\vec{\alpha}.(\exists\{\vec{\alpha}_1, \vec{\tau}_1 \equiv \vec{\alpha}\}.\tau_1 + \ldots + \exists\{\vec{\alpha}_k, \vec{\tau}_k \equiv \vec{\alpha}\}.\tau_k)$$

Note that for $i = 1, \ldots, k$, $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ and $\vec{\alpha}_i$ are assumed to share no common type variables, and the T's in $\vec{\tau}_i$ and τ_i have been replaced with t's.

Given a type variable context Δ, we define $|\Delta|$ as follows.

$$|\cdot| = \cdot \qquad |\Delta, \alpha| = |\Delta|, \alpha \qquad |\Delta, \tau_1 \equiv \tau_2| = |\Delta|$$

For a type substitution Θ and a sequence of type variables $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$, we write $\Theta(\vec{\alpha})$ for $\Theta(\alpha_1), \ldots, \Theta(\alpha_n)$. The introduction and elimination rules for guarded types can be formed as follows.

$$\frac{\Delta_1; \Gamma \vdash e : \tau[\Theta] \quad \Delta_1 \vdash \Theta : \Delta_2}{\Delta_1; \Gamma \vdash \langle \Theta(|\Delta_2|) \mid e \rangle : \exists\Delta_2.\tau}\ \textbf{($\exists$-intro)}$$

$$\frac{\Delta_1; \Gamma \vdash e_1 : \exists\Delta_2.\tau_1 \quad \Delta_1, \Delta_2; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta_1; \Gamma \vdash \texttt{open}\ e_1\ \texttt{as}\ \langle |\Delta_2| \mid x \rangle\ \texttt{in}\ e_2 : \tau_2}\ \textbf{($\exists$-elim)}$$

Note that a judgment of the form $\Delta_1 \vdash \Theta : \Delta$ means that $\mathbf{dom}(\Theta) = \mathbf{dom}(\Delta_2)$ and $\Delta_1 \models \tau_1[\Theta] \equiv \tau_2[\Theta]$ for each type equality $\tau_1 \equiv \tau_2$ in Δ_2. In the elimination rule, we require that τ_2 contain no type variable in $\mathbf{dom}(\Delta_2)$.

Given types $\tau^1, \ldots, \tau^k$ and $1 \leq i \leq k$, we use $\textit{inj}_i^{\tau^1 + \ldots + \tau^k}$ for the ith injection that injects values of the type τ^i into values of the type $\tau^1 + \ldots + \tau^k$. In addition, we use $\textit{unfold}((\vec{\tau})T)$ for the expansion of $(\vec{\tau})T$ to $\sigma[t \mapsto T][\vec{\alpha} \mapsto \vec{\tau}]$, where $\vec{\tau}$ is a

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vec{\alpha}_i; x : \tau_i \vdash x : \tau_i \quad \vec{\alpha}_i \vdash [\vec{\beta}_i \mapsto \vec{\alpha}_i] : \vec{\beta}_i, \vec{\tau}_i[\vec{\alpha}_i \mapsto \vec{\beta}_i] \equiv \vec{\tau}_i}{\vec{\alpha}_i; x : \tau_i \vdash \langle \vec{\alpha}_i \mid x \rangle : \exists\{\vec{\beta}_i, \vec{\tau}_i[\vec{\alpha}_i \mapsto \vec{\beta}_i] \equiv \vec{\tau}_i\}.\tau_i[\vec{\alpha}_i \mapsto \vec{\beta}_i]}\ (\exists\textbf{-intro})}{\vec{\alpha}_i; x : \tau_i \vdash \mathit{inj}_i^{\mathit{unfold}((\vec{\tau}_i)T)}(\langle \vec{\alpha}_i \mid x \rangle) : \mathit{unfold}((\vec{\tau}_i)T)}\ (+\textbf{-intro})}{\vec{\alpha}_i; x : \tau_i \vdash \mathit{inj}_i^{\mathit{unfold}((\vec{\tau}_i)T)}(\langle \vec{\alpha}_i \mid x \rangle) : (\vec{\tau}_i)T}\ \textbf{(fold)}}{\vec{\alpha}_i; \cdot \vdash \lambda x : \tau_i.\mathit{inj}_i^{\mathit{unfold}((\vec{\tau}_i)T)}(\langle \vec{\alpha}_i \mid x \rangle) : \tau_i \rightarrow (\vec{\tau}_i)T}\ (\rightarrow\textbf{-intro})}{\cdot; \cdot \vdash \Lambda\vec{\alpha}_i.\lambda x : \tau_i.\mathit{inj}_i^{\mathit{unfold}((\vec{\tau}_i)T)}(\langle \vec{\alpha}_i \mid x \rangle) : \forall\vec{\alpha}_i.\tau_i \rightarrow (\vec{\tau}_i)T}\ (\forall\textbf{-intro})$$

Figure 5: Defining value constructors associated with T

sequence of n types. Then we can construct a typing derivation in Figure 5, where all the applied rules are standard. By the derivation, we can define c_i as follows for $i = 1, \ldots, k$:

$$\Lambda\vec{\alpha}_i.\lambda x : \tau_i.\mathit{inj}_i^{\mathit{unfold}((\vec{\tau}_i)T)}\langle \vec{\alpha}_i \mid x \rangle$$

Therefore, we have justified the notion of g.r. datatype constructors in terms of standard type-theoretical concepts.

We now present some simple examples of g.r. datatype constructors so as 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 $\forall\alpha.\alpha \rightarrow \mathit{TOP}$; *TOP* is defined as $\mu t.\exists\{\alpha\}.\alpha$, which is equivalent to $\exists\alpha.\alpha$.

The type *TOP* is called an abstract datatype in [11]. In general, the the notion of abstract datatypes is subsumed by the notion of g.r. datatype constructors.

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 $\forall\alpha.\mathbf{1} \rightarrow (\alpha)\mathit{list}$ and $\forall\alpha.\alpha * (\alpha)\mathit{list} \rightarrow (\alpha)\mathit{list}$, 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 $\lambda\alpha.\mu t.\mathbf{1} + \alpha * t$.

In the rest of the paper, we are no longer in need of guarded types of the form $\exists\Delta.\tau$ directly. The typing rules **($\exists$-intro)** and **($\exists$-elim)** are to be absorbed into the typing rules **(ty-cons)** and **(ty-case)**, respectively. Similarly, we we will make no use of expressions of the form $\langle\vec{\tau} \mid e\rangle$ or the form `open` e_1 `as` $\langle\vec{\alpha} \mid x\rangle$ `in` e_2 directly.

2.3 Pattern Matching

We use p for patterns. As usual, a type (value) variable may occur at most once in each pattern. We use a judgment of the 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.

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

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

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

Given a type variable context Δ_0, a pattern p and a type τ, we can use the rules in Figure 3 to derive a judgment of the form $\Delta_0 \vdash p \downarrow \tau \Rightarrow (\Delta; \Gamma)$, whose meaning is formally captured by Lemma 2.3.

2.4 Static and Dynamic Semantics

We present the typing rules for $\lambda_{2,G\mu}$ in Figure 4. We assume the existence of a signature Σ in which the types of constructors are declared.

Most of the typing rules are standard. The rule **(ty-eq)** indicates that the type equality in $\lambda_{2,G\mu}$ is modulo type constraint solving. Please notice the great difference between the rules presented in Figure 3 for typing clauses and the "standard" ones in [14].

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 2.2. *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.

2.5 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 2.3. *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 the 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 2.4. *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 2.5. *(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 2.4. For handling the typing rule **(ty-case)**, Lemma 2.3 is also 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 $e_1 =$ **case** v **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. As is the case for the usual datatypes in ML, it can also be checked whether pattern matching is exhaustive with respect to types of the form $(\vec{\tau})T$ for g.r. datatype constructors T.

3. ELABORATION

We have presented an explicitly typed language $\lambda_{2,G\mu}$. Since a programmer may be quickly overwhelmed with the need for writing types in such a setting, it becomes apparent that we need to provide an external language $\mathrm{ML}_{2,G\mu}$ together with an elaboration process from $\mathrm{ML}_{2,G\mu}$ to $\lambda_{2,G\mu}$ that preserves dynamic semantics.

Some of the syntax for $\mathrm{ML}_{2,G\mu}$ is presented in Figure 6 and it should be straightforward to relate the concrete syntax in the examples we present to that of $\mathrm{ML}_{2,G\mu}$. The types in $\mathrm{ML}_{2,G\mu}$ are the same as in $\lambda_{2,G\mu}$. The syntax for type ascription is $(e : \tau)$, which basically means the expression e is required to be of the type τ. Also, the types for bound variables in $\mathrm{ML}_{2,G\mu}$ may be omitted. Note that the language $\mathrm{ML}_{2,G\mu}$ is not a conservative extension of ML as there are strictly more programs that are typable in $\mathrm{ML}_{2,G\mu}$ than in ML.

$$
\begin{array}{llcl}
\text{patterns} & p & ::= & x \mid \langle\rangle \mid \langle p_1, p_2\rangle \mid c(p) \\
\text{clauses} & ms & ::= & (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \\
\text{expressions} & e & ::= & x \mid f \mid c \mid \\
 & & & \langle\rangle \mid \langle e_1, e_2\rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid \\
 & & & \lambda x.e \mid \lambda x:\tau.e \mid e_1(e_2) \mid \\
 & & & \mathbf{fix}\ f.v \mid \mathbf{fix}\ f:\tau.v \mid \\
 & & & \mathbf{case}\ e\ \mathbf{of}\ ms \mid \\
 & & & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid (e : \tau) \\
\text{values} & v & ::= & x \mid c \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \\
 & & & \lambda x.e \mid \lambda x:\tau.e
\end{array}
$$

Figure 6: Syntax for the external language $\mathrm{ML}_{2,G\mu}$

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

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

$$
\frac{\tau \text{ contains a free occurrence of } \alpha \text{ but is not } \alpha}{\vec{\alpha}, \alpha \equiv \tau, \Delta \vdash \tau_1 \equiv \tau_2}
$$

$$
\frac{\tau \text{ contains a free occurrence of } \alpha \text{ but is not } \alpha}{\vec{\alpha}, \tau \equiv \alpha, \Delta \vdash \tau_1 \equiv \tau_2}
$$

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

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

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

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

Figure 7: The rules for solving type constraints

During the elaboration of a program, type constraints of the form $\Delta \vdash \tau_1 \equiv \tau_2$ are generated. We present a set of rules for solving such type constraints in Figure 7, where we use T to range over all type constructors, either built-ins ($*$ and $\rightarrow$), user-defined g.r. datatype constructors, or skolemized constants (introduced by applying the last rule in Figure 7)

DEFINITION 3.1. *Given a type τ, the size $[\tau]$ of τ is defined as follows.*

$$[\alpha] = 1 \qquad [(\tau_1, \ldots, \tau_n)T] = 1 + [\tau_1] + \ldots + [\tau_n]$$

Furthermore, given a type equality $\tau_1 \equiv \tau_2$, its size is defined as $[\tau_1] + [\tau_2]$; given a type variable context Δ, its size is the *sum of the sizes of the type equalities in Δ.*

Note that for each rule in Figure 7 of the following form:

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2}{\Delta' \vdash \tau_1' \equiv \tau_2'} \ ,$$

we have $[\Delta] < [\Delta']$. This observation is needed to justify the soundness and the completeness of the rules for solving type constraints.

THEOREM 3.2. *$\Delta \models \tau_1 \equiv \tau_2$ holds if and only if $\Delta \vdash \tau_1 \equiv \tau_2$ is derivable.*

Proof Assume that $\Delta = \vec{\alpha}, \Delta'$ for some Δ' that does not begin with a type variable. The proof follows from induction on the lexicographic ordering (n_1, n_2, n_3), where n_1 is the number of free type variables in Δ', n_2 is the number of occurrences of $\forall$ in Δ' and n_3 is the size of Δ'. ■

Therefore, the rules in Figure 7 for solving type constraints are both sound and complete.

The elaboration process for $\mathrm{ML}_{2,G\mu}$ is similar to the one for DML [25, 23], following essentially the same strategy. Given Δ, Γ and e, a synthesizing judgment $\Delta; \Gamma \vdash e \uparrow \tau \Rightarrow e^*$ means that e can be elaborated into e^* with type τ such that $\Delta; \Gamma \vdash e^* : \tau$ is derivable and $|e|$ and $|e^*|$ are operationally equivalent, where $|e|$ and $|e^*|$ are the erasures of e and e^*, respectively. Given Δ, Γ, e and τ, a checking judgment $\Delta; \Gamma \vdash e \downarrow \tau \Rightarrow e^*$ means that e can be elaborated into e^* such that $\Delta; \Gamma \vdash e^* : \tau$ is derivable and $|e|$ and $|e^*|$ are operationally equivalent. We have formulated a list of rules for deriving both synthesizing and checking judgments. Please refer to [24] for these rules plus other details on the issue of elaboration in $\mathrm{ML}_{2,G\mu}$.

4. APPLICATIONS

In this section, we show how g.r. datatype constructors can be used to handle some realistic and interesting examples that involve a variety of programming features. We have finished a prototype implementation that supports most of the main features in the core of ML (e.g., pattern matching, polymorphism, effects, etc.) plus g.r. datatype constructors. The implementation, written in Objective Caml, is available at [24], where we have also presented many other examples in addition to the following ones, including an implementation of queue in message-passing style, an implementation of polymorphic marshaling as is described in [16], etc.

4.1 Polymorphic Generic Functions

The notion of polymorphic generic functions is introduced in [8], which allows the programmer to define polymorphic functions that may behave differently on different type arguments. We use an example to show that generic functions can be readily handled through the use of g.r. datatype constructors.

In the C programming language, *sprintf* is a function that takes a format string and a list of arguments, and then returns a string representation of the arguments according to the format string. Let us declare the type *FORMAT* as follows.

```
typecon (type) FORMAT =
  (int -> 'a) I of 'a FORMAT
| (char -> 'a) C of 'a FORMAT
| ('a) S_ of string * 'a FORMAT
| (string) S0 of string
```

```
fun sprintf fmt = let
  fun aux pre (I fmt) =
      (fn i => aux (pre ^ int2string i) fmt)
    | aux pre (C fmt) =
      (fn c => aux (pre ^ char2string c) fmt)
    | aux pre (S_ (s, fmt)) = aux (pre ^ s) fmt
    | aux pre (S0 s) = pre ^ s
  withtype string -> 'a FORMAT -> 'a
in aux "" fmt end
withtype 'a FORMAT -> 'a
```

Figure 8: An implementation of the sprintf function

When applied to a format of the type $(\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \mathtt{string})$*FORMAT*, the *sprintf* function expects that the n arguments following the format have the types $\tau_1, \ldots, \tau_n$, respectively. We give an implementation of *sprintf* in Figure 8. We can make a format expression more readable by defining an infix operator *\$* for application and currying *S_* into *S*.

```
infixr $
fun f $ x = f (x)
fun S s fmt = S_ (s, fmt)
```

For instance, the following expression binds *fmt* with a format of the type $(\mathtt{int} \rightarrow \mathtt{char} \rightarrow \mathtt{string})$*FORMAT*.

```
val fmt = S "int i = " $ I $
          S " and char c = " $ C $ S0 ""
```

As can be expected, *sprintf*(*fmt*) returns a function of the type $\mathtt{int} \rightarrow \mathtt{char} \rightarrow \mathtt{string}$, which yields the following string

```
"int i = 1 and char c = a"
```

when applied to the integer 1 and the character a.

Certainly, a functions like *sprintf* can also be implemented through the use of type classes (or their variants). The above implementation of *sprintf* is actually adopted from [15]. In this case, the format argument of the *sprintf* function is most likely to be provided by the user instead of being automatically synthesized and the very issue of overloading addressed by type classes does not really seem to exist here. Therefore, we feel the above implementation of *sprintf* through g.r. datatype constructors is more natural and direct.

4.2 Implementing Staged Computation

We outline an implementation of staged computation [7, 19] through the use of higher-order abstract syntax. We first associate a few more value constructors with the g.r. datatype constructor *HOAS* defined in Section 1.

```
typecon (type) HOAS =
  ... ...
| {'a}. ('a) HOASvar of string
| {'a}. ('a) HOASif of
        bool HOAS * 'a HOAS * 'a HOAS
```

Obviously, *HOASif* is introduced for forming h.o.a.s trees to represent conditional expressions. In general, many more language constructs can be readily handled by properly associating some value constructors with *HOAS*. The only (ad hoc and ugly) use of *HOASvar* is shown in Figure 9, where a translation from higher-order abstract syntax (h.o.a.s.) to

```
typecon FOAS =
  FOASvar of string
| {'a}. FOASlift of 'a
| FOAStup of FOAS * FOAS
| FOASlam of string * FOAS
| FOASapp of FOAS * FOAS
| FOASfix of string * FOAS
| FOASif of FOAS * FOAS * FOAS

(* 'new_name()' produces a fresh name *)
val new_name: unit -> string

fun h2f (HOASvar name) = FOASvar name
  | h2f (HOASlift v) = FOASlift v
  | h2f (HOAStup (e1,e2)) = FOAStup(h2f e1,h2f e2)
  | h2f (HOASlam f) = let
      val name = new_name ()
    in
      FOASlam (name, h2f (f (HOASvar name)))
    end
  | h2f (HOASapp (e1,e2)) = FOASapp(h2f e1,h2f e2)
  | h2f (HOASfix f) = let
      val name = new_name ()
    in
      FOASfix (name, h2f (f (HOASvar name)))
    end
  | h2f (HOASif (e1, e2, e3)) =
    FOASif (h2f e1, h2f e2, h2f e3)
withtype {'a}. 'a HOAS -> FOAS
```

Figure 9: A translation from h.o.a.s to f.o.a.s

first-order abstract syntax (f.o.a.s.) is given. For instance, the translation of $\mathit{HOASlam}(\mathit{fn}\ x \Rightarrow x))$ is simply

$$\mathit{FOASlam}(\text{“}x\text{”}, \mathit{FOASvar}(\text{“}x\text{”}))$$

(assuming “x” is the freshly generated name for the bound variable x).

Also, we assume a built-in function *compile* of the type $\forall\alpha.(\alpha)\mathit{HOAS} \rightarrow \alpha$ that compiles h.o.a.s. trees. For instance, *compile* can be implemented in such a manner: Given a h.o.a.s tree e, we translate e into a f.o.a.s tree $\mathit{h2f}(e)$ and then compile the f.o.a.s. tree with a standard approach (as is done in Scheme). Note that the function *compile* corresponds to the function *run* in MetaML [19].

As an example, we stage the usual power function as follows, where the bloated syntax is soon to be replaced with some syntactic sugar.

```
fun power1 n =
  if n = 0 then HOASlam (fn x => HOASlift 1)
  else HOASlam (fn (x: int HOAS) =>
         HOASapp (HOASlift *,
           HOAStup (x, HOASapp (power1 (n-1), x))))
withtype int -> (int -> int) HOAS
```

Then we can define the square function as follows.

```
val square1: int -> int = compile (power1 2)
```

It can be readily verified that the f.o.a.s. tree translated from (*power1* 2) represents the following program:

```
(fn x2 => x2 * (fn x1 => x1 * (fn x0 => 1) x1) x2)
```

Now suppose we stage the power function as follows.

$$\begin{aligned}
\text{‘}(xf) &= \mathit{HOASlift}(xf)\\
\text{‘}(c) &= \mathit{HOASlift}(c)\\
\text{‘}(\langle\rangle) &= \mathit{HOASlift}(\langle\rangle)\\
\text{‘}(\langle e_1, e_2\rangle) &= \mathit{HOAStup}(\text{‘}(e_1), \text{‘}(e_2))\\
\text{‘}(\mathbf{if}(e_1, e_2, e_2)) &= \mathit{HOASif}(\text{‘}(e_1), \text{‘}(e_2), \text{‘}(e_3))\\
\text{‘}(\lambda x.e) &= \mathit{HOASlam}(\lambda x.\text{‘}(e[x \mapsto \hat{}(x)]))\\
\text{‘}(\lambda x : \tau.e) &= \mathit{HOASlam}(\lambda x : \langle\tau\rangle.\text{‘}(e[x \mapsto \hat{}(x)]))\\
\text{‘}(e_1(e_2)) &= \mathit{HOASapp}(\text{‘}(e_1), \text{‘}(e_2))\\
\text{‘}(\mathbf{fix}\ f.v) &= \mathit{HOASfix}(\lambda f.\text{‘}(v[f \mapsto \hat{}(f)]))\\
\text{‘}(\mathbf{fix}\ f : \tau.v) &= \mathit{HOASfix}(\lambda f : \langle\tau\rangle.\text{‘}(v[f \mapsto \hat{}(f)]))\\
\text{‘}(e : \tau) &= (\text{‘}(e) : \langle\tau\rangle)\\
\text{‘}(\hat{}(e)) &= e
\end{aligned}$$

Figure 10: Syntactic sugar for staged computation
.

```
fun power2 n x =
  if n = 0 then (HOASlift 1)
  else HOASapp (
       HOASlift *, HOAStup (x, power2 (n-1) x))
withtype int -> (int) HOAS -> (int) HOAS
```

Then the square function can be define as:

```
val square2: int -> int =
  compile(HOASlam(fn (x: int HOAS) => power2 2 x))
```

This time, the f.o.a.s. tree translated from the following h.o.a.s tree

```
    HOASlam (fn (x:  int HOAS) => power2 2 x)
```

is `(fn x => x * (x * 1))`.

In Figure 10, we introduce some syntactic sugar to facilitate staged computation. Note that we now write $\langle\tau\rangle$ for $(\tau)\mathit{HOAS}$. Essentially, ‘(e) corresponds to $\langle e\rangle$ in the syntax of MetaML, and ˆ(e) corresponds to ˜(e). For instance, in our concrete syntax, the code `‘(fn (x: int) => x * x)` expands into the following code, which has the type $\langle\texttt{int} \rightarrow \texttt{int}\rangle$.

```
HOASlam (fn (x: <int>) =>
         HOASapp (HOASlift *, HOAStup (x, x)))
```

We reject code in which some syntactic sugar cannot be removed. For instance, `^x + 1` is ill-formed since the symbol `^` can not be translated away. However, `‘(^x+1)` is well-formed, which translates into:

```
HOASapp(HOASlift +, HOAStup(x, HOASlift 1))
```

Therefore, ill-formedness is context-sensitive.

The above functions *power1*, *square1*, *power2* and *square2* can now be written as follows.

```
fun power1 n =
  if n = 0 then ‘(fn x => 1)
  else ‘(fn x => x * ^(power1 (n-1)) x)
withtype int -> <int -> int>

val square1 = compile ‘(fn x => ^(power1 2) x)

fun power2 n x =
  if n = 0 then ‘1 else ‘(^x * ^(power2 (n-1) x))

val square2 = compile ‘(fn x => ^(power2 2 ‘x))
```

Generally speaking, we expect the following. Assume that e is an expression of type τ in MetaML. Let e^* be the expression obtained from replacing each $\langle\cdot\rangle$ in e with ‘$(\cdot)$ and each ˜ with ˆ and then translating away the syntactic sugar. Then e^* should have the same type τ (if we identify the type constructor $\langle\cdot\rangle$ in MetaML with *HOAS*).

Unfortunately, the outlined implementation of staged computation contains an annoying problem, which is often called *open code extrusion*. For instance, the following program:

```
‘(fn x: int HOAS => ^(compile ‘(x)))
```

translates into the following h.o.a.s. tree t:

```
HOASlam (fn x:  (int HOAS) HOAS => compile (x))
```

Suppose we want to compile t; then we need to turn t into a f.o.a.s tree by applying the function *h2f* to t; clearly, evaluating $h2f(t)$ leads to a call of *compile* on $HOASvar("x")$ (assuming "x" is the freshly generated name for the bound variable x); but this call leads to a run-time error as there is no way to compile a variable.

It is of great difficulty to properly address the problem with open code extrusion. For an approach to implementing staged computation that can prevent open code extrusion, please refer to [3], where a g.r. datatype constructor is formed for representing typed code via deBruijn indices.

In [7], a language Mini-ML$_e^{\square}$ with a type system based on the modal logic S4 is presented for studying staged computation. There, the type constructor $\square$ is intended to capture the closedness of code. Since h.o.a.s. trees (with no use of *HOASvar*) can only represent closed expressions, we naturally expect a relation between $\square$ and *HOAS*. This is to be studied in future.

4.3 Implementing Programming Objects

We briefly outline an approach to implementing objects through the use of g.r. datatype constructors. When compared with various existing approaches in the literature, this approach addresses many difficult issues in objected-oriented programming (e.g., parametric polymorphism, binary methods, the self type, etc.) in a purely type-theoretical manner, which we feel is both natural and satisfactory. In the following presentation, we take a view of objects in the spirit of Smalltalk [9, 13], suggesting to conceptualize an object as a little intelligent being that is capable of performing actions according to the messages it receives.

We assume the existence of a guarded recursive datatype constructor *MSG* that takes a type τ and forms a message type $(\tau)MSG$.[2] Also, we require that *MSG* be extensible (like the exception type in ML). After receiving a message of type $(\tau)MSG$, an object is expected to return a value of type τ. Therefore, we assign an object the following type *OBJ*:

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

Suppose that we have declared through some syntax 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 11, 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*. We point out it is solely for illustration purpose that we use the prefix *MSG* in the name of each message constructor. Given integers x and y, we can form an integer pair *anIntPair* by calling $newIntPair(x)(y)$; we can then send the message *MSGgetfst* to the pair to obtain its first component: $anIntPair(MSGgetfst)$; we can also reset its first component to x' by sending the message $MSGsetfst(x')$ to the pair: $anIntPair(MSGsetfst(x'))$; operations on the second component of the pair can be performed similarly. Note that an exception is raised at run-time if *anIntPair* cannot interpret a message sent to it.

[2]In the following presentation, when the arguments of a constructor are types, we always write the arguments in front of the constructor.

```
fun newIntPair 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 int -> int -> OBJ
```

Figure 11: An implementation of integer pairs

Classes Obviously, there exist some serious problems 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 the type $(\mathbf{1})MSG$; then $anIntPair(MSGfoo)$ is well-typed, but its execution leads to an uncaught exception *UnknownMessage* at run-time. This is clearly undesirable: $anIntPair(MSGfoo)$ should be rejected at compile-time as an ill-typed expression. 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 the 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:

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

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 following message constructors of the corresponding types:

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

The function *newIntPair* can now be given the type $\texttt{int} \rightarrow \texttt{int} \rightarrow OBJ(IntPairClass)$; since *anIntPair* has the type $OBJ(IntPairClass)$, $anIntPair(MSGfoo)$ becomes ill-typed if *MSGfoo* has a type $(\mathbf{1})MSG(C)$ for some class C that is not *IntPairClass*. Following Dependent ML [25, 23], we use `class` as the sort for classes.

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, to form a class $(\tau_1, \tau_2)PairClass$. We may use some syntax to declare such a class construc-

```
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 12: Functions for constructing objects in the classes *PairClass* **and** *ColoredPairClass*

tor 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 12.

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, we write $C_1 \leq C_2$ to mean that C_1 is a subclass of C_2. The type of a message constructor *mc* is now of the general form $\forall\vec{\alpha}.\Pi a \lhd C.(\tau)MSG(a)$ or $\forall\vec{\alpha}.\Pi a \lhd C.\tau_1 \to (\tau_2)MSG(a)$, where $a \lhd C$ means that a is of the subset sort $\{a : \texttt{class} \mid a \leq C\}$, i.e., the sort for all subclasses of the class C; for a sequence of types $\vec{\tau}$ that is of the same length as $\vec{\alpha}$, $mc[\vec{\tau}]$ 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 construct a message for any object tagged by a subclass of C_0. For instance, the types of the message constructors associated with *PairClass* are now listed in Figure 13. Suppose we introduce another class constructor *ColoredPairClass*, which takes two types to form a class, and assume the following holds, 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*, whose types are given in Figure 13. Note that *color* is just some already defined type for colors.

We can then implement the function *newColoredPair* in Figure 12 for constructing colored pairs. Clearly, the implementation of *newColoredPair* shares a lot of common code with that of *newPair*. We plan to provide some syntactic support for the programmer to effectively reuse the code in the implementation of *newPair* when implementing *newColoredPair*.

Binary Methods Our approach to typed object-oriented programming offers a 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 assigned the following types:

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

Suppose *self* is an object of type $OBJ(C)$ for some $C \leq Eq$. If we pass a message $\mathit{MSGeq}(\mathit{other})$ to *self*, *other* is required to have the type $OBJ(C)$ in order for $\mathit{self}(\mathit{MSGeq}(\mathit{other}))$ to be well-typed. Therefore, *self* and *other* must be two objects belonging to the same class.

The Self Type Our approach also offers a clean solution to handling the notion of *self type*, namely, the type for 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 $C \leq \mathit{ObjClass}$ 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, since $\mathit{self}(\mathit{MSGcopy})$ has the type $OBJ(C)$ whenever *self* is an object of type $OBJ(C)$ for some class C.

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

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.

Inheritance Inheritance is handled in a Smalltalk-like manner, but there is some significant difference. For those who are familiar with exceptions in Standard ML, we point out that the way that method lookup is implemented resembles how exceptions are handled by exception handlers. 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 OOP. We use the following syntax:

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

to declare a class tag *ObjClass* and a message constructor *MSGcopy* of the type:

$$\Pi a \lhd \mathit{ObjClass}.(\mathit{OBJ}(a))\mathit{MSG}(a)$$

Note `selfType` is merely syntactic sugar here. 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)
```

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

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

Figure 13: Some message constructors and their types

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 $\Pi a \lhd \mathit{ObjClass}.\mathit{OBJ}(a) \to \mathit{OBJ}(a)$ 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 the type $\Pi a \lhd C.\mathit{OBJ}(a) \to \mathit{OBJ}(a)$ 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 14. 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 class *Int1Class*, the returned object would be in the class

```
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 14: Functions for constructing objects in *Int1Class* and *ColoredInt1Class*

Int1Class, not in the class *ColoredInt1Class*, as it would be generated by $\mathit{newInt1}(o_2(\mathit{MSGget\_x}))$, 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.

Subtypes 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 $\mathit{OBJECT}(C) = \Sigma a \lhd C.\mathit{OBJ}(a)$ is the sum of all types $\mathit{OBJ}(a)$ satisfying $a \leq C$. Hence, for each $C_1 \leq C$, $\mathit{OBJ}(C_1)$ can be regarded as a subtype of $\mathit{OBJECT}(C)$ as each value of the type $\mathit{OBJ}(C_1)$ can be coerced into a value of the type $\mathit{OBJECT}(C)$. As an example, the type

$$\mathit{OBJ}((\mathit{OBJECT}(\mathit{Int1Class}), \mathit{OBJECT}(\mathit{Int1Class}))\mathit{PairClass})$$

is for pair objects whose both components are objects in some subclasses of *Int1Class*.

5. 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 [6, 5, 21]. Many of such studies follow the framework in [10], 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. However, in the presence of Typerec, it becomes rather difficult to define a proper equality on types. For instance, the type equality defined in [10] is not closed under substitution.

The language λ_i^{ML} in [10] is subsequently extended to λ_R in [6] 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 [24].

The dependent datatypes in DML [25, 23] 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).

Although we initially met the notion of g.r. datatype constructors during a study on run-time type-passing, we have since found a variety applications of this notion beyond type-passing (e.g., implementing staged computation and implementing programming objects). 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.

6. REFERENCES

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

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

[3] C. Chen and H. Xi. Implementing typed meta-programming. Available at http://www.cs.bu.edu/~hwxi/academic/papers/TMP.ps, November 2002.

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

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

[6] K. Crary, S. Weirich, and G. 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.

[7] R. Davies and F. Pfenning. A Modal Analysis of Staged Computation. *Journal of ACM*, 2002.

[8] C. Dubois, F. Rouaix, and P. Weis. Generic Polymorphism. In *Proceeding of the 22th ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 118–129, London, UK, January 1995.

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

[10] R. W. Harper and G. 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.

[11] K. Läufer and M. Odersky. Ploymorphic Type Inference and Abstract Data Types. *ACM Transactions of Programming Languages and Systems (TOPLAS)*, 16(5):1411–1430, September 1994.

[12] X. Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, January 1992.

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

[14] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[15] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A Functional Notation for Functional Dependencies. In *Proceedings of 2001 Haskell Workshop*, pages 101–120, Florence, Italy, September 2001.

[16] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Conference Record of the Twentieth Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 99–112, Charleston, SC, January 1993.

[17] S. Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available at http://www.haskell.org/onlinereport/, Feb. 1999.

[18] F. Pfenning. *Computation and Deduction*. Cambridge University Press, 2002.

[19] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[20] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of ACM Conference on LISP and Functional Programming*, pages 1–11, Orlando, FL, June 1994.

[21] V. Trifonov, B. Saha, and Z. Shao. Fully Reflexive Intensional Type Analysis. In *Proceedings of the International Conference on Functional Programming*, September 1999.

[22] S. Weirich. Encoding intensional type analysis. In D. Sands, editor, *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2001.

[23] H. 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.

[24] H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors, 2002. Available at http://www.cs.bu.edu/~hwxi/GRecTypecon/.

[25] H. Xi and F. 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.