NPSCS-94-006

NAVAL POSTGRADUATE SCHOOL
Monterey, California

A CRITIQUE OF TYPE SYSTEMS
FOR GLOBAL OVERLOADING

by Dennis M. Volpano

October 1993

Prepared for:

Naval Postgraduate School
Monterey, California 93943

NAVAL POSTGRADUATE SCHOOL

Monterey, California

REAR ADMIRAL T. A. MERCER
Superintendent

HARRISON SHULL
Provost

This report was prepared with research funded by the Naval Research Laboratory under the Reimbursable Funding.

This report was prepared by:

REPORT DOCUMENTATION PAGE

EPORT SECURITY CLASSIFICATION JNCLASSIFIED	1b. RESTRICTIVE MARKINGS
ECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
ECLASSIFICATION/DOWNGRADING SCHEDULE	
RFORMING ORGANIZATION REPORT NUMBER(S) PSCS-94-006	5. MONITORING ORGANIZATION REPORT NUMBER(S)
AME OF PERFORMING ORGANIZATION omputer Science Dept. aval Postgraduate School 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
DDRESS (City, State, and ZIP Code) Ionterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943
AME OF FUNDING/SPONSORING RGANIZATION aval Postgraduate School 8b. OFFICE SYMBOL (if applicable) NPS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN Direct Funding
DDRESS (City, State, and ZIP Code) Ionterey, CA 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

TLE (Include Security Classification)
A Critique of Type Systems for Global Overloading

ERSONAL AUTHOR(S)
Dennis M. Volpano

TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
inal	FROM 10/91 TO 9/92	July1993	13

UPPLEMENTARY NOTATION

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
LD	GROUP	SUB-GROUP	type systems, global overloading

BSTRACT (Continue on reverse if necessary and identify by block number)

roposed extensions of the ML type system to incorporate global overloading include the systems of [Kae88, D091, Smi91, Kae92, Jon92] and those related to the design of the functional programming language Haskell VaB89, CH092, niP93]. These systems have in common the notion of a constrained type scheme which in some realized by type kinds and in others as explicit predicates. An analysis of these type systems reveals that some e unsound with regard to a suitable criterion for typability and some adopt a notion of type generality that is in-onsistent with that of system ML [DaM82].

STRIBUTION/AVAILABILITY OF ABSTRACT INCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
AME OF RESPONSIBLE INDIVIDUAL Dennis M. Volpano	22b. TELEPHONE (Include Area Code) (408) 656-3091 22c. OFFICE SYMBOL CSVo

A Critique of Type Systems for Global Overloading

Dennis M. Volpano
Department of Computer Science
Naval Postgraduate School
Monterey, CA, USA

July 15, 1993

Abstract

Proposed extensions of the *ML* type system to incorporate global overloading include the systems of [Kae88, CDO91, Smi91, Kae92, Jon92] and those related to the design of the functional programming language Haskell [WaB89, CHO92, NiP93]. These systems have in common the notion of a *constrained type scheme* which in some is realized by *type kinds* and in others as explicit *predicates*. An analysis of these type systems reveals that some are unsound with regard to a suitable criterion for typability and some adopt a notion of type generality that is inconsistent with that of system *ML* [DaM82].

1 Introduction

Much of the research in type theory is devoted to finding new type systems that offer programmers more flexibility without compromising the advantages of a strongly-typed language. One of the simplest type systems is Λ_0, the implicit, finitely-typed lambda calculus [Tiu90, BaH90]. It assigns finite types to type-free λ terms. For example, in Λ_0 one can derive the typing $\vdash \lambda x.\, x : \alpha \rightarrow \alpha$. In fact, one can derive $\vdash \lambda x.\, x : \tau \rightarrow \tau$ for any finite type τ. Consequently, $\lambda x.\, x$ can be assigned multiple types within a single term as is necessary in deriving a type for $(\lambda x.\, x)\, \lambda x.x$ in Λ_0.

Although Λ_0 allows a term denoting a polymorphic value to be assigned multiple types, it does not allow a *free identifier* that denotes such a term to be given multiple types within a single λ term. For example, the term $(\lambda y.\, y\, y)\, \lambda x.\, x$ is untypable in Λ_0 even though y, which is free in $y\, y$, denotes $\lambda x.x$. So although Λ_0 has desirable properties like a decidable typability problem and principal types for all typable terms, it is for the most part unsuitable for direct use in a programming language.

This limitation was overcome in the *ML* type system [Mil78, DaM82] which is an extension of Λ_0 with *type schemes*. A type scheme represents a very simple kind of polymorphism called *parametric polymorphism*. A free identifier may be asserted to have infinitely many types via a type scheme. However, in the interest of retaining principal types, lambda abstraction in system *ML* is monomorphic as in Λ_0. Thus $(\lambda y.\, y\, y)\, \lambda x.\, x$ is untypable in system *ML* as well. How then does system *ML* allow free identifiers denoting polymorphic values to be assigned multiple types? Only through its **let** construct. So for example, $(\lambda y.\, y\, y)\, \lambda x.\, x$ is written **let** $y = \lambda x.\, x$ **in** $y\, y$.

Like Λ_0, system *ML* has a decidable typability problem and typable terms have principal types. Further, it seems to offer an acceptable level of flexibility as evidenced by its incorporation in languages like Standard ML [HMM86] and Miranda [Tur86]. Yet it remains limited in some ways that appear to be solely of theoretical interest, and in others that are very obvious in practice.

One practical limitation is that it prohibits *overloading* by restricting to at most one the number of assumptions per identifier in a type assumption set, a limitation noted by Milner himself in [Mil78]. Suppose we wish to assert that a free identifier, say $+$, has precisely finite types $int \to int \to int$ and $real \to real \to real$. Any assumption set in which $+$ has one of the two desired finite types precludes a derivation that it has the other. On the other hand, any assumption set that assigns type scheme $\forall\alpha.\alpha \to \alpha \to \alpha$ to $+$ is one from which too many types can be derived for $+$. Thus there is no assumption set in system *ML* from which we can derive precisely the desired finite types for $+$. Even system *ML* with subtypes is inadequate. From

$$A = \{int \subseteq real,\ \ + : real \to real \to real\}$$

one could derive $A \vdash + : int \to int \to real$ but not $A \vdash + : int \to int \to int$.

Many extensions of system *ML* have been proposed to incorporate overloading. There are the systems of [Kae88, CDO91, Smi91, Kae92, Jon92] and those related to the design of the functional programming language Haskell

[WaB89, CHO92, NiP93]. This paper reviews these systems. As we shall see, some are unsound under a proper notion of typability and some even adopt a notion of type generality that is inconsistent with the original notion proposed in [DaM82].

2 Constrained Type Schemes

The terms of the type systems we consider are typical. Every variable x is a term, and if M and N are terms then so are $\lambda x.M$, $(M\,N)$, and **let** $x = M$ **in** N. There is no expression that overloads identifiers locally like the **over** expression proposed in [WaB89]. The only overloading the systems attempt to address is that which comes from an initial basis or sequence of top-level definitions, which we call *global* overloading.

Each system has the same set of finite types defined in the usual way. Every type variable α is a finite type, and if $\tau_1, \ldots, \tau_n$ are finite types then so are $\tau_1 \to \tau_2$ and $\chi(\tau_1, \ldots, \tau_n)$ where χ is a type constructor of arity n.

The systems have in common the notion of a *constrained type scheme* which in some is realized by a type of types called a *kind* and in others as an explicit *predicate*. A kind is a universe of types over which a type variable may be quantified. Kinds appear in [Kae88, CDO91, CHO92, NiP93] where the general form of a constrained type scheme is $\forall\alpha : \rho.\,\sigma$ for some kind ρ. Explicit predicates appear in the systems of [WaB89, Smi91, Jon92, Kae92] where the general form of a constrained type scheme is

$$\forall\alpha_1, \ldots, \alpha_n \textbf{ with } x_1 : \tau_1, \ldots, x_m : \tau_m \,.\, \tau$$

for predicates $x_1 : \tau_1, \ldots, x_m : \tau_m$. Predicates also appeared earlier in the work of Holmstrom [Hol83].

Although syntactically different, constrained type schemes of all these systems may be interpreted as Horn clauses. For example, suppose multiplication $(*)$ and the multiplicative identity (1) are overloaded in a type basis. Then if both were free in a definition of *expon* that raises a value to some integer power then *expon* should have as types $\tau \to \mathit{int} \to \tau$ for any finite type τ for which predicates $1 : \tau$ and $* : \tau \to \tau \to \tau$ are true. This may be expressed by a constrained type scheme using predicates as

$$\mathit{expon} : \forall\alpha \textbf{ with } 1 : \alpha,\ * : \alpha \to \alpha \to \alpha \,.\, \alpha \to \mathit{int} \to \alpha\,.$$

If $T_x(\tau)$ means $x : \tau$ then this is interpreted as the Horn clause

$$T_{expon}(\alpha \to int \to \alpha) \leftarrow T_1(\alpha) \wedge T_*(\alpha \to \alpha \to \alpha).$$

It is natural to consider Horn clauses with other kinds of predicates as well expressing, for example, subtype inclusions [Smi91, Jon92, Kae92] and even unification among finite types [Kae92]. The latter kind of predicate shows that typability in system *ML* is actually an instance of typability in a predicate-based system. Of course in system *ML*, the absence of a unifier implies the given term is untypable and if unification succeeds, then a unifying substitution can be reflected in an unconstrained type scheme, so no explicit predicates or kinds are needed.

3 A Criterion for Typability

When is a term typable with respect to a set of type assumptions? One criterion for typability often used for system *ML* is called *strong type inference* [Tiu90]. Under it, a term M is typable with respect to an assumption set B if there is an assumption set A and type σ such that $B \subseteq A$ and $A \vdash M : \sigma$ is derivable. Thus in a formulation of the typability problem, M and B are considered input. Yet the problem only requires finding an A that contains B so there is considerable latitude in the choice of A. In fact, within the context of overloading, there is too much. Consider the term **true** + **true**, where **true** denotes a truth value and + integer addition as conveyed by the assumption set

$$B = \{\mathbf{true} : bool,\ \ + : int \to int \to int\}.$$

Then the term could be typable under B in a sound type system since with

$$A = B \cup \{+ : bool \to bool \to bool\}$$

we can derive $A \vdash \mathbf{true} + \mathbf{true} : bool$. But $B \vdash \mathbf{true} + \mathbf{true} : bool$ is not derivable, since + does not have an instance for type *bool* within B. Therefore the term should not be typable under B.

Thus a suitable criterion for typability is strong type inference with the requirement that $B \vdash M : \sigma$ be derivable. However σ is constrained so unless a type system prevents quantification over an empty type universe, it may be *unsound* in that terms with domain incompatibilities are typable.

4 Soundness of the Proposed Systems

The early work of Kaes [Kae88] led to an extension of system *ML* based on type kinds. It permits a restricted form of overloading called *parametric overloading* which, when extended to allow overloading of constants, corresponds to the kind allowed in Haskell. Kaes gives a type inference algorithm where unification is done with variables having kinds. This was later discovered to be an application of order-sorted unification in [NiS91] where type inference in Haskell, which permits overloading via classes, is implemented in the context of order-sorted algebras.

Kaes's extension has two kinds of assumption sets, one for ordinary type assumptions and the other for overloading assumptions. For example,

$$A = \left\{ \begin{array}{l} f : \forall\alpha_{\{f\}}.\, \alpha_{\{f\}} \rightarrow \alpha_{\{f\}}, \\ g : \forall\alpha_{\{g\}}.\, \alpha_{\{g\}} \rightarrow \alpha_{\{g\}} \end{array} \right\}$$

is a type assumption set and

$$O = \left\{ \begin{array}{l} f : \langle \omega \rightarrow \omega,\ \{int, real\} \rangle, \\ g : \langle \omega \rightarrow \omega,\ \{bool, char\} \rangle \end{array} \right\}$$

is an overloading assumption set. Set O specifies the different finite types for the overloaded identifiers f and g. The types of f, for example, are formed by replacing ω in $\omega \rightarrow \omega$ by *int* and *real*.

A type variable may be annotated with a set of identifiers that effectively specifies its kind. The variable ranges only over those finite types for which all functions in the set are simultaneously defined. So α_f, for example, ranges over types *int* and *real*, but not *bool*, with respect to O.

Under the strong type inference criterion, a term M would be typable with respect to A and O in this system if

$$A, O \vdash M : \sigma \quad \text{for some type } \sigma.$$

But the system is unsound under this criterion, for even though O specifies that f and g have no finite types in common, we can derive

$$A, O \vdash \lambda x.\, g(f\ x) : \forall\alpha_{\{f,g\}}.\, \alpha_{\{f,g\}} \rightarrow \alpha_{\{f,g\}}$$

in the system:

$$\begin{array}{ll}
\alpha_{\{f,g\}} \to \alpha_{\{f,g\}} <_o \forall\alpha_{\{f\}}.\,\alpha_{\{f\}} \to \alpha_{\{f\}} & (<_o) \\
A \cup \{x : \alpha_{\{f,g\}}\}, O \vdash f : \alpha_{\{f,g\}} \to \alpha_{\{f,g\}} & (\text{var}) \\
\alpha_{\{f,g\}} \to \alpha_{\{f,g\}} <_o \forall\alpha_{\{g\}}.\,\alpha_{\{g\}} \to \alpha_{\{g\}} & (<_o) \\
A \cup \{x : \alpha_{\{f,g\}}\}, O \vdash g : \alpha_{\{f,g\}} \to \alpha_{\{f,g\}} & (\text{var}) \\
A \cup \{x : \alpha_{\{f,g\}}\}, O \vdash g(f\ x) : \alpha_{\{f,g\}} & (\text{app})^2 \\
A, O \vdash \lambda x.\, g(f\ x) : \alpha_{\{f,g\}} \to \alpha_{\{f,g\}} & (\text{abs}) \\
A, O \vdash \lambda x.\, g(f\ x) : \forall\alpha_{\{f,g\}}.\,\alpha_{\{f,g\}} \to \alpha_{\{f,g\}} & (\text{closure})
\end{array}$$

In essence we can hypothesize overlap in the types of f and g through type $\alpha_{\{f,g\}}$ for the sake of showing $\lambda x.\, g(f\ x)$ is typable, yet we are not required to show that any overlap exists. Thus a type can be derived for the term under A and O even though the term has a type incompatibility. A sound variant of this system is given in [Rou90, Kae92].

Parametric overloading prevents certain kinds of overloadings like

$$\left\{ \begin{array}{l} + : \mathit{int} \to \mathit{real} \to \mathit{real}, \\ + : \mathit{real} \to \mathit{complex} \to \mathit{complex} \end{array} \right\}$$

which is useful for describing mixed-mode arithmetic. In an attempt to permit more general kinds of overloadings, Wadler and Blott give a predicate-based extension of system *ML* [WaB89]. But like [Kae88], it too is unsound under strong type inference. Further, unlike Kaes's system which restricts overloading, typability in their system under a revised criterion is undecidable [VoS91]. A restriction that allows overloadings for mixed-mode arithmetic and for which typability is decidable is given in [VoS91].

The type system ML_o described in [Smi91] is sound. It replaces the type generalization and instantiation rules of *ML* by the following rules.

(∀-intro) $$\frac{A \cup C \vdash M : \tau',\ \ A \vdash C[\bar{\alpha} := \bar{\tau}]}{A \vdash M : \forall\bar{\alpha}\ \textbf{with}\ C\,.\,\tau'} \quad (\bar{\alpha} \text{ not free in } A)$$

(∀-elim) $$\frac{A \vdash M : \forall\bar{\alpha}\ \textbf{with}\ C\,.\,\tau',\ \ A \vdash C[\bar{\alpha} := \bar{\tau}]}{A \vdash M : \tau'[\bar{\alpha} := \bar{\tau}]}$$

To illustrate these rules, consider a term $M = \lambda x.\,(x + x)$, with free identifier $+$, and a type assumption set B defined as follows.

$$B = \left\{ \begin{array}{l} + : \mathit{int} \to \mathit{int} \to \mathit{int}, \\ + : \mathit{real} \to \mathit{real} \to \mathit{real}, \\ \leq : \mathit{char} \to \mathit{char} \to \mathit{bool}, \\ \leq : \forall\alpha \textbf{ with } \leq : \alpha \to \alpha \to \mathit{bool}\,.\,\mathit{seq}(\alpha) \to \mathit{seq}(\alpha) \to \mathit{bool} \end{array} \right\}.$$

We show a derivation of $B \vdash M : \forall\alpha \textbf{ with } + : \alpha \to \alpha \to \alpha\,.\,\alpha \to \alpha$ in ML_o.

$$\begin{array}{ll} B \cup \{+ : \alpha \to \alpha \to \alpha\} \cup \{x : \alpha\} \vdash x : \alpha & \text{(hypoth)} \\ B \cup \{+ : \alpha \to \alpha \to \alpha\} \cup \{x : \alpha\} \vdash + : \alpha \to \alpha \to \alpha & \text{(hypoth)} \\ B \cup \{+ : \alpha \to \alpha \to \alpha\} \cup \{x : \alpha\} \vdash (x + x) : \alpha & (\to\text{-elim})^2 \\ B \cup \{+ : \alpha \to \alpha \to \alpha\} \vdash \lambda x.\,(x + x) : \alpha \to \alpha & (\to\text{-intro}) \\ B \vdash \{+ : \alpha \to \alpha \to \alpha\}\,[\alpha := \mathit{real}] & \text{(hypoth)} \\ B \vdash \lambda x.\,(x + x) : \forall\alpha \textbf{ with } + : \alpha \to \alpha \to \alpha\,.\,\alpha \to \alpha & (\forall\text{-intro}) \end{array}$$

The last step of the derivation discharges assumption $+ : \alpha \to \alpha \to \alpha$ which was used to derive a type for M, making a type derivation possible from the initial set B. Before this could be done however, we needed to establish that the assumption is derivable with α instantiated to some finite type (*real* was chosen in our example although *int* would have sufficed). For some terms and type assumptions, this is impossible. The term $M = \lambda x.\,\lambda y.\,y \leq (x + x)$ together with B is an example. We can hypothesize a type assumption set C in which $+$ and $\leq$ overlap on some finite type, say α, by defining C as

$$C = \{+ : \alpha \to \alpha \to \alpha,\ \leq : \alpha \to \alpha \to \mathit{bool}\}$$

and then derive $B \cup C \vdash M : \alpha \to \alpha \to \mathit{bool}$. But the derivation must end here because there is no finite type for α that allows us to derive C from B, so M is untypable.

The work of [CDO91] is an effort to relax the restrictions of a parametric overloading. They give a kind-based extension of system *ML* that restricts recursive overloadings slightly less than parametricity and for which typability is decidable. Yet assumptions for mixed-mode arithmetic are still prohibited.

The system however is sound under strong type inference. It is interesting to see how the system prevents a type from being derived for term $\lambda x.\,g(f\,x)$ when free identifiers f and g have no finite types in common.

Derivations are made with respect to a traditional type environment A and a kind environment Γ that maps type variables to kinds. There are two operators on kinds, **fix** and a union operator $\sqcup$. Kinds are partially ordered by a containment relation $\leq$ such that $\bot \leq \rho$ for any kind ρ.

The type instantiation and generalization rules of system *ML* are modified to accomodate constrained type schemes with kinds:

$$\text{(inst)} \quad \frac{\Gamma;\ A \vdash M : \forall\alpha : \rho.\, \sigma}{\Gamma;\ A \vdash M : \sigma[\alpha := \tau]} \quad (\Gamma(\tau) \leq \rho)$$

$$\text{(gen)} \quad \frac{\Gamma, \alpha : \rho;\ A \vdash M : \sigma}{\Gamma;\ A \vdash M : \forall\alpha : \rho.\, \sigma} \quad (\alpha \text{ not free in } A \wedge \rho \neq \bot)$$

Here $\Gamma(\tau)$ is finite type τ with all free type variables replaced by the kinds prescribed for them by Γ.

Suppose

$$A = \left\{ \begin{array}{l} f : \forall\alpha : \mathit{int} \sqcup \mathit{real}.\ \alpha \to \alpha, \\ g : \forall\alpha : \mathit{bool} \sqcup \mathit{char}.\ \alpha \to \alpha \end{array} \right\}.$$

The only way we can hypothesize any overlap between f and g is through use of kind $\bot$ since $\bot \leq \rho$ for any kind ρ. Proceeding in this way yields

$\{\beta : \bot\};\ A \cup \{x : \beta\} \vdash x : \beta$	(var)
$\Gamma(\beta) = \bot \leq \mathit{int} \sqcup \mathit{real}$	($\leq$)
$\{\beta : \bot\};\ A \cup \{x : \beta\} \vdash f : \forall\alpha : \mathit{int} \sqcup \mathit{real}.\ \alpha \to \alpha$	(var)
$\{\beta : \bot\};\ A \cup \{x : \beta\} \vdash f : \beta \to \beta$	(inst)
$\Gamma(\beta) = \bot \leq \mathit{bool} \sqcup \mathit{char}$	($\leq$)
$\{\beta : \bot\};\ A \cup \{x : \beta\} \vdash g : \forall\alpha : \mathit{bool} \sqcup \mathit{char}.\ \alpha \to \alpha$	(var)
$\{\beta : \bot\};\ A \cup \{x : \beta\} \vdash g : \beta \to \beta$	(inst)
$\{\beta : \bot\};\ A \cup \{x : \beta\} \vdash g(f\ x) : \beta$	(app)[2]
$\{\beta : \bot\};\ A \vdash \lambda x.\, g(f\ x) : \beta \to \beta$	(abs)

The generalization rule cannot be applied at this point to discharge the remaining kind assumption $\{\beta : \bot\}$ so $\lambda x.\, g(f\ x)$ is untypable under A.

The type system of [CHO92] is another kind-based extension of system *ML*, that unlike [CDO91], is unsound under strong type inference. This work

is not an attempt to relax the restrictions of parametric overloading but rather to formalize a type system for Haskell with parametric type classes.

Derivations in [CHO92] are also made with respect to a traditional type environment A and a kind environment, which they call a context C, mapping type variables to kinds. Kinds here are taken to be sets of class names Γ with each class having an optional finite type as parameter. So for example, if F and G are class names, then $\{\alpha :: \{F, G\}\}$ is a kind environment that asserts α is a *class instance* of F and G, meaning that α is a type over which the operations of both classes F and G are defined.

The type instantiation and generalization rules of system ML are replaced by rules ($\forall$-elim) and ($\forall$-intro):

$$(\forall\text{-elim}) \qquad \frac{A,\ C \vdash M : \forall\alpha{::}\Gamma.\ \sigma}{A,\ C \vdash M : \sigma[\alpha := \tau]} \quad (C \Vdash \tau :: \Gamma)$$

$$(\forall\text{-intro}) \qquad \frac{A,\ C, \alpha :: \Gamma \vdash M : \sigma}{A,\ C \vdash M : \forall\alpha{::}\Gamma.\ \sigma} \quad (\alpha \text{ not free in } A \text{ or } C)$$

where $C \Vdash \tau :: \Gamma$ is derived using a set of *entailment* inference rules with class instance declarations D as axioms. For example, if $D = \{int :: F\}$ then $\emptyset \Vdash int :: F$.

Suppose that

$$D = \{int :: F,\ real :: F,\ bool :: G,\ char :: G\}$$

which asserts that class F has as instances finite types *int* and *real*, and class G, types *bool* and *char*. So with respect to D, there is no finite type over which the operations of F and G are defined. But with

$$A = \left\{ \begin{array}{l} f : \forall\alpha{::}\{F\}.\ \alpha \rightarrow \alpha, \\ g : \forall\alpha{::}\{G\}.\ \alpha \rightarrow \alpha \end{array} \right\}$$

we can derive

$$A \vdash \lambda x.\, g(f\ x) : \forall\beta{::}\{F, G\}.\ \beta \rightarrow \beta$$

in the system, thus establishing that $\lambda x.\, g(f\ x)$ is typable with respect to A under the strong type inference criterion even though the derived type scheme

has no finite types as instances with respect to D. Using kind $\{F, G\}$, we hypothesize overlap between f and g yielding the derivation

$$
\begin{array}{ll}
A \cup \{x : \beta\},\ \{\beta :: \{F, G\}\} \vdash x : \beta & \text{(var)} \\
\{\beta :: \{F, G\}\} \Vdash \beta :: F & F \in \{F, G\} \\
\{\beta :: \{F, G\}\} \Vdash \beta :: \{F\} & (\Vdash) \\
A \cup \{x : \beta\},\ \{\beta :: \{F, G\}\} \vdash f : \forall\alpha{::}\{F\}.\ \alpha \to \alpha & \text{(var)} \\
A \cup \{x : \beta\},\ \{\beta :: \{F, G\}\} \vdash f : \beta \to \beta & (\forall\text{-elim}) \\
\{\beta :: \{F, G\}\} \Vdash \beta :: G & G \in \{F, G\} \\
\{\beta :: \{F, G\}\} \Vdash \beta :: \{G\} & (\Vdash) \\
A \cup \{x : \beta\},\ \{\beta :: \{F, G\}\} \vdash g : \forall\alpha{::}\{G\}.\ \alpha \to \alpha & \text{(var)} \\
A \cup \{x : \beta\},\ \{\beta :: \{F, G\}\} \vdash g : \beta \to \beta & (\forall\text{-elim}) \\
A \cup \{x : \beta\},\ \{\beta :: \{F, G\}\} \vdash g(f\ x) : \beta & (\lambda\text{-elim})^2 \\
A,\ \{\beta :: \{F, G\}\} \vdash \lambda x.\, g(f\ x) : \beta \to \beta & (\lambda\text{-intro}) \\
A \vdash \lambda x.\, g(f\ x) : \forall\beta{::}\{F, G\}.\ \beta \to \beta & (\forall\text{-intro})
\end{array}
$$

In the final step, the entire context is discharged even though there is no finite type that satisfies it with respect to D. An almost identical derivation can be constructed in the system of [NiP93] and in the predicate-based extension of system *ML* given in [Jon92]. With a minor reformulation of A and D requiring kinds to be interpreted as predicate sets so that for example D is

$$\{F\ \mathit{int},\ F\ \mathit{real},\ G\ \mathit{bool},\ G\ \mathit{char}\}$$

viewing F and G as predicates, one can derive in the system of [Jon92]

$$D \,|\, A \vdash \lambda x.\, g(f\ x) : \forall\alpha.\{F\,\alpha,\ G\,\alpha\} \Rightarrow \alpha \to \alpha\,.$$

Although D specifies no overlap for f and g, we can prove the judgement.

5 Type Generality

In order to define in the systems considered here a notion of principal type among derivable types, a *generic instance* relation is needed that tells us whether one type is more general than another [DaM82]. In the context of constrained type schemes, this relation becomes a preorder rather than a

partial order since two distinct types can each be a generic instance of the other.

Some of the proposed type systems require that in order for a constrained type scheme σ' to be a generic instance of another such scheme σ, it must be possible to replace the bound variables of σ by finite types so as to make the bodies of σ and σ' syntactically equal [WaB89, CHO92, Kae92, Jon92]. This is an artifact of the original relation of [DaM82], which for system *ML*, gives a way to check that σ has as instances all finite types that are instances of σ', or in other words, that σ is *more general* than σ'. However, the relation is also typically defined relative to some form of assumptions from which an attempt can be made to show that a finite type is an element of a kind or that a predicate is true. Now it is no longer appropriate to demand syntactic equality of type bodies in order to establish that one constrained type scheme is more general than another.

For example, in the system of [CHO92], constrained type scheme

$$\forall\alpha::\{F\}.\forall\beta::\{F\}.\,\alpha \to \beta$$

is not a generic instance of

$$\forall\gamma::\{F\}.\,\gamma \to \gamma$$

with respect to any set of class instance declarations since there is no substitution on γ that when applied to $\gamma \to \gamma$ gives $\alpha \to \beta$. Yet in the context of a single declaration, say $\{int :: F\}$, the latter type does indeed have as instances all finite types that are instances of the former (only $int \to int$) and is therefore more general in this context. Similar examples can be given in the systems of [WaB89, Kae92, Jon92].

References

[BaH90] Barendregt, H. and Hemerik, K.: Types in Lambda Calculi and Programming Languages, *Proc. 3rd European Symposium on Programming*, *LNCS* **432**, Springer-Verlag, pp. 1–35, 1990.

[CHO92] Chen, K., Hudak, P. and Odersky, M.: Parametric Type Classes, *Proc. 7th ACM Conf. on Lisp and Functional Programming*, pp. 170–181, 1992.

[CDO91] Cormack, G. Duggan, D. and Ophel, J.: Decidable Type Reconstruction with Recursive Overloading (Extended Abstract), Department of Computer Science, University of Waterloo, 1991.

[DaM82] Damas, L. and Milner, R.: Principal Type Schemes for Functional Programs, *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[HMM86] Harper, R., MacQueen, D. and Milner, R.: Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[Hol83] Holmström S.: Polymorphic Type Systems and Concurrent Computation in Functional Languages, Ph.D. Thesis, Department of Computer Science, Chalmers University of Technology, 1983.

[Jon92] Jones, M.: A theory of qualified types, *Proc. 4th European Symposium on Programming*, *LNCS* **582**, Springer-Verlag, pp. 287–306, 1992.

[Kae88] Kaes, S.: Parametric Overloading in Polymorphic Programming Languages, *Proc. 2nd European Symposium on Programming*, *LNCS* **300**, Springer-Verlag, pp. 131–144, 1988.

[Kae92] Kaes, S.: Type Inference in the Presence of Overloading, Subtyping, and Recursive Types, *Proc. 7th ACM Conf. on Lisp and Functional Programming*, pp. 193–204, 1992.

[Mil78] Milner, R.: A Theory of Type Polymorphism in Programming, *J. of Computer and System Sciences*, **17**, pp. 348–375, 1978.

[NiP93] Nipkow, T. and Prehofer, C.: Type Checking Type Classes, *Proc. 20th ACM Symposium on Principles of Programming Languages*, pp. 409–418, 1993.

[NiS91] Nipkow, T. and Snelting, G.: Type Classes and Overloading Resolution via Order-Sorted Unification, *Proc. 5th Conf. on Functional Programming Languages and Computer Architecture*, *LNCS* **523**, Springer-Verlag, pp. 1–14, 1991.

[Rou90] Rouaix, F.: Safe Run-time Overloading *Proc. 17th ACM Symposium on Principles of Programming Languages*, pp. 355–366, 1990.

[Smi91] Smith, G.S.: Polymorphic Type Inference for Languages with Overloading and Subtyping, Ph.D. Thesis, Department of Computer Science, Cornell University, Technical Report 91-1230, 1991.

[Tiu90] Tiuryn, J.: Type Inference Problems: A Survey, *Proc. Mathematical Foundations of Computer Science*, *LNCS* **452**, Springer-Verlag, pp. 105–120, 1990.

[Tur86] Turner, D.: An Overview of Miranda, *ACM SIGPLAN Notices*, **21** pp. 156–166, 1986.

[VoS91] Volpano, D.M. and Smith, G.S.: On the Complexity of *ML* Typability with Overloading, *Proc. 5th Conf. on Functional Programming Languages and Computer Architecture*, *LNCS* **523**, Springer-Verlag, pp. 15–28, 1991.

[WaB89] Wadler, P. and Blott, S.: How to make *ad-hoc* polymorphism less *ad-hoc*, *Proc. 16th ACM Symposium on Principles of Programming Languages*, pp. 60–76, 1989.

Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Library, Code 52 Naval Postgraduate School Monterey, CA 93943	2
Director of Research Administration Code 08 Naval Postgraduate School Monterey, CA 93943	1
Dr. Neil C. Rowe, Code CSRp Naval Postgraduate School Computer Science Department Monterey, CA 93943-5118	1
Prof. Robert B. McGhee, Code CSMz Naval Postgraduate School Computer Science Department Monterey, CA 93943-5118	1
Dr. Ralph Wachter Software Program Office of Naval Research 800 N. Quincy St. Arlington VA 22217-5000	2
Dr. Dennis Volpano, Code CSVo Naval Postgraduate School Computer Science Dept. Monterey, CA 93943-5118	20