JFP **19** (2): 157–172, 2009. © 2008 Cambridge University Press
doi:10.1017/S0956796808006886 First published online 30 June 2008 Printed in the United Kingdom

THEORETICAL PEARL

Coherence of subsumption for monadic types

JAN SCHWINGHAMMER

Programming Systems Lab, Saarland University, 66041 Saarbrücken, Germany
(*e-mail:* jan@ps.uni-sb.de)

Abstract

One approach to give semantics to languages with subtypes is by translation to target languages without subtyping: subtypings $A \leqslant B$ are interpreted via conversion functions $A \rightarrow B$. This paper shows how to extend the method to languages with computational effects, using Moggi's computational metalanguage.

1 Introduction

Subtyping is a binary relation $\leqslant$ on types, where $A \leqslant B$ states that expressions of type A may be used in contexts expecting values of type B. The metatheory is well developed, covering systems of simple, higher-order and dependent types with subtyping (e.g. Cardelli 1988; Curien & Ghelli 1992; Pierce & Steffen 1997; Zwanenburg 1999; Aspinall & Compagnoni 2001). When it comes to the semantics of subtyping, a flexible and robust method is to interpret $A \leqslant B$ as a *conversion* $c : A \rightarrow B$ from type A to type B (Reynolds 1980; Breazu-Tannen *et al.* 1991; Mitchell 1996). Conversions give rise to a translation into a target language without subtyping, thereby enabling the reuse of existing models. The key step is the elimination of the subsumption rule that allows to infer the type B for a term e from the assumption that e has type A and $A \leqslant B$. In the target language, subsumption is replaced by an application $c(e)$ of the conversion function corresponding to $A \leqslant B$.

Such a conversion interpretation is defined recursively, following the structure of the subtyping derivations for $A \leqslant B$ and typing derivations $\Gamma \triangleright e : A$, respectively, in the source language. But in the presence of subtyping, type derivations are no longer uniquely determined by Γ, e, and A alone, and, *a priori*, the translation may differ on different type derivations of the same judgment. Breazu-Tannen *et al.* (1991) address the problem by proving *coherence*, in the sense that the translation is independent of the chosen derivation, up to provable equality in the target language. Coherence results have been obtained for a variety of typed lambda calculi, including polymorphic recursive and sum types (Breazu-Tannen *et al.* 1991), intersection types (Reynolds 1991), and system $F_{\leqslant}$ (Curien & Ghelli 1992).

Subtyping is also an important ingredient of *imperative* programming, in particular object-oriented languages. In fact, the motivation for this work stems from an

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

attempt to reason about Abadi and Cardelli's imperative object calculus. It is surprising, therefore, that no corresponding coherence results for languages that combine subtyping and computational effects (notably state) can be found in the literature. The aim of this short note is to fill this gap, by considering subtypes in the context of Moggi's computational metalanguage.

Moggi's calculus extends the simply typed lambda calculus by *monadic types* TA (Moggi 1991). Monads provide for a distinction between "pure" values and "effectful" computations, where TA is the type of computations over A. Every monad T comes equipped with an operation map_T that lifts a function $f : A \to B$ to $map_T\, f : TA \to TB$. For instance, to accommodate recursion, a type A might denote a complete partial order, TA the lifted partial order $A_\perp$, and $map_T f$ the strict extension of a continuous map $f : A \to B$. In the case of the list monad, $map_T f$ is the function that maps a list $[x_1, \ldots, x_n]$ to $[f x_1, \ldots, f x_n]$, well known to functional programmers. Monads have proved a useful tool, both in programming theory and practice. Benton *et al.* (2002) give a very accessible introduction to their many applications.

Looking at the instances of computational monads from Moggi (1990, 1991), it appears sensible to postulate $TA \leqslant TB$ whenever $A \leqslant B$. Indeed, the conversion interpretation of subtyping extends to Moggi's calculus in a generic, monad-independent way: given a conversion $c : A \to B$ corresponding to $A \leqslant B$, $map_T\, c$ provides a conversion from TA to TB. (Readers with a background in type theory will recognize that this construction is quite standard, using functoriality of the type constructor T to define the conversions.) There exist translations of call-by-value and call-by-name lambda calculi into Moggi's language where function spaces are decomposed as $A \to_{cbv} B = A \to TB$ and $A \to_{cbn} B = TA \to TB$, respectively. Because of the (covariant) monadic subtyping sketched above, these translations now extend to lambda calculi with subtyping.

It is worth pointing out that, while the coherence result as such is to be expected, the fact that we can give an elementary proof is perhaps less so. For instance, Breazu-Tannen *et al.* (1991) axiomatized a separate type of coercion functions, because of problems arising from the interaction of fixed points and the eta law for sum types. In Schwinghammer (2005), where I add subtyping on top of a semantic model that combines nontermination and dynamically allocated state, coherence is proved by a semantic construction due to Reynolds (2003). The method is elegant but does not easily generalize to other effects as it relies on the existence of suitable reflexive objects, i.e., appropriate untyped models. In contrast, the strict separation between pure and effectful computations in the monadic calculus allows for a pleasingly straightforward extension of previous work (Curien & Ghelli 1992): the coherence proof proceeds by transforming type derivations to a unique normal form. The equational theory of the monadic metalanguage suffices to show that the transformations preserve the semantics.

The next section recalls the computational metalanguage of Moggi (1991), including a notion of subtyping. Section 3 develops the conversion semantics. The coherence theorem is proved in Section 4, and Section 5 discusses some extensions to the basic setting. In the choice of notation we keep close to Mitchell (1996).

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

Table 1. *Subtypes and typing*

$$\frac{}{\Sigma \vdash A \leqslant A}\,(\mathit{ref}) \qquad \frac{\Sigma \vdash A \leqslant B \quad \Sigma \vdash B \leqslant C}{\Sigma \vdash A \leqslant C}\,(\mathit{trans}) \qquad \frac{\Sigma \vdash B_1 \leqslant A_1 \quad \Sigma \vdash A_2 \leqslant B_2}{\Sigma \vdash A_1 \to A_2 \leqslant B_1 \to B_2}\,(\mathit{arrow})$$

$$\frac{b_1 {\leqslant} b_2 \in S_\Sigma}{\Sigma \vdash b_1 \leqslant b_2}\,(\mathit{ax}) \qquad \frac{\Sigma \vdash A \leqslant B}{\Sigma \vdash TA \leqslant TB}\,(\mathit{monad}) \qquad \frac{\Gamma \rhd e : A \qquad \Sigma \vdash A \leqslant B}{\Gamma \rhd e : B}\,(\mathit{sub})$$

$$\frac{x{:}A \in \Gamma}{\Gamma \rhd x : A}\,(\mathit{var}) \qquad \frac{\Gamma, x{:}A \rhd e : B}{\Gamma \rhd \lambda x{:}A.e : A \to B}\,(\mathit{abs}) \qquad \frac{\Gamma \rhd e_1 : A \to B \qquad \Gamma \rhd e_2 : A}{\Gamma \rhd e_1\, e_2 : B}\,(\mathit{app})$$

$$\frac{c : \mathit{typeOf}(c) \in C_\Sigma}{\Gamma \rhd c : \mathit{typeOf}(c)}\,(\mathit{const}) \qquad \frac{\Gamma \rhd e : A}{\Gamma \rhd [e] : TA}\,(\mathit{unit}) \qquad \frac{\Gamma \rhd e_1 : TA \qquad \Gamma, x{:}A \rhd e_2 : TB}{\Gamma \rhd \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 : TB}\,(\mathit{bind})$$

2 A monadic metalanguage with subtyping

Let x, y, z range over a countably infinite set of variables. Let $\Sigma = \langle B_\Sigma, C_\Sigma, S_\Sigma \rangle$ be a signature, consisting of a set B_Σ of *type constants* ranged over by b, a set C_Σ of *term constants* ranged over by c, and a set S_Σ of basic *subtyping assertions* $b \leqslant b'$ between type constants. We assume that each constant $c \in C_\Sigma$ has a specified type $\mathit{typeOf}(c)$ built from T, →, and the type constants. The types and terms of the computational metalanguage $ML_T(\Sigma)$ are defined by the following grammar:

$$\begin{aligned} A, B \in \mathit{Type} &\ ::=\ b \mid A \to B \mid TA \\ e \in \mathit{Exp} &\ ::=\ c \mid x \mid \lambda x{:}A.e \mid e_1\, e_2 \mid [e] \mid \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 \end{aligned}$$

As mentioned above, $ML_T(\Sigma)$ is an extension of the simply typed lambda calculus over signature Σ by monadic types. The terms $[e]$ and $\mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2$ are the inclusion of values and composition of computations, respectively. We let Γ range over type contexts $x_1{:}A_1, \ldots, x_n{:}A_n$, where all x_i are distinct.

Table 1 defines the subtype relation and typing rules, parameterized by the signature Σ. With the exception of rule (*monad*) this is entirely standard. The equational theory of $ML_T(\Sigma)$ consists of the β- and η-equalities of simply typed lambda calculus, together with three additional equalities that axiomatize the monadic computations. Only equations $\Gamma \rhd e_1{=}e_2 : A$ between well-typed terms of the same type will be considered; if the type and context are clear from context we may simply write $e_1 = e_2$. This is summarized in Table 2.

Types A and B *match* if they have the same shape. Formally, matching is the least relation between types such that b_1 matches b_2, for any type constants $b_1, b_2 \in B_\Sigma$, and if A matches B and A' matches B' then $A \to A'$ matches $B \to B'$ and TA matches TB. The following observation is an immediate consequence of restricting basic subtypings S_Σ to type constants:

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

Table 2. *ML_T equations*

$(\to.\beta)$	$\Gamma \rhd (\lambda x.e_1)\, e_2 = e_1[e_2/x] : A$
$(\to.\eta)$	$\Gamma \rhd \lambda x.ex = e : A \to B \qquad \text{provided } x \notin fv(e)$
$(T.\beta)$	$\Gamma \rhd \mathsf{let}\ x \Leftarrow [e_2]\ \mathsf{in}\ e_1 = e_1[e_2/x] : TA$
$(T.\eta)$	$\Gamma \rhd \mathsf{let}\ x \Leftarrow e\ \mathsf{in}\ [x] = e : TA$
$(T.ass)$	$\Gamma \rhd \mathsf{let}\ x_2 \Leftarrow (\mathsf{let}\ x_1 \Leftarrow e_1\ \mathsf{in}\ e_2)\ \mathsf{in}\ e_3 = \mathsf{let}\ x_1 \Leftarrow e_1\ \mathsf{in}\ (\mathsf{let}\ x_2 \Leftarrow e_2\ \mathsf{in}\ e_3) : TA$

Lemma 2.1 (*Structural subtyping*)
If $\Sigma \vdash A \leqslant B$ then A and B match.

We introduce some useful notation. Let $id_A = \lambda x{:}A.x$ and write $e_1;e_2 = \lambda x{:}A.e_2(e_1\,x)$ for the composition of $e_1 : A \to B$ and $e_2 : B \to C$ in diagrammatic order. Using $(\to.\beta)$ and $(\to.\eta)$ it is easily verified that composition is associative and has id as left and right unit: for all $f : A \to B$, $g : B \to C$, and $h : C \to D$,

$$f;(g;h) = (f;g);h \quad \text{and} \quad id_A;f = f = f;id_B$$

Let $map_T : (A \to B) \to TA \to TB$ be defined by

$$map_T = \lambda f{:}A \to B.\lambda x{:}TA.\,\mathsf{let}\ y \Leftarrow x\ \mathsf{in}\ [fy]$$

Strictly speaking, map_T should also be indexed by the types A and B, but in the following these can usually be reconstructed from context. Compatibility of map_T with identities and composition follows from the equational axioms of $ML_T(\Sigma)$

$$\begin{aligned} map_T(id_A) &= \lambda x{:}TA.\,\mathsf{let}\ y \Leftarrow x\ \mathsf{in}\ [id_A y] && \text{by } (\to.\beta)\\ &= \lambda x{:}TA.\,\mathsf{let}\ y \Leftarrow x\ \mathsf{in}\ [y] && \text{by } (\to.\beta)\\ &= id_{TA} && \text{by } (T.\eta) \end{aligned}$$

and

$$\begin{aligned} map_T(f;g) &= \lambda x{:}TA.\,\mathsf{let}\ z \Leftarrow x\ \mathsf{in}\ [g(fz)] && \text{by } (\to.\beta)\\ &= \lambda x{:}TA.\,\mathsf{let}\ z \Leftarrow x\ \mathsf{in}\ \mathsf{let}\ y \Leftarrow [fz]\ \mathsf{in}\ [gy] && \text{by } (T.\beta)\\ &= \lambda x{:}TA.\,\mathsf{let}\ y \Leftarrow \mathsf{let}\ z \Leftarrow x\ \mathsf{in}\ [fz]\ \mathsf{in}\ [gy] && \text{by } (T.ass)\\ &= \lambda x{:}TA.\,\mathsf{let}\ y \Leftarrow map_T(f)(x)\ \mathsf{in}\ [gy] && \text{by } (\to.\beta)\\ &= map_T(f);map_T(g) && \text{by } (\to.\beta) \end{aligned}$$

These identities show that the monad T is a functor in the category theoretic sense, with the action on morphisms given by map_T, and relate to an alternative axiomatization of monads (Moggi 1991). Corresponding to the functorial action of function types we let $map_\to : (A_2{\to}A_1) \to (B_1{\to}B_2) \to (A_1{\to}B_1) \to A_2 \to B_2$ be

$$map_\to = \lambda f{:}A_2 \to A_1 \lambda g{:}B_1 \to B_2 \lambda h{:}A_1 \to B_1.\,f;h;g$$

The equations $map_\to(id_A)(id_B) = id_{A\to B}$ and

$$(map_\to\, f_1\, g_1);(map_\to\, f_2\, g_2) = map_\to\,(f_2;f_1)\,(g_1;g_2)$$

are direct consequences of $(\to.\beta)$ and $(\to.\eta)$. Note the contravariance of $map_\to$ in its first argument.

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

Many notions of computation fit the monadic framework. The following examples, taken from Moggi (1991), illustrate this:

Exceptions where $TA = A + E$ adjoins a set of exceptions E to A, and

$$[a] = inl(a)$$
$$\mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 = case\ e_1\ of\ inl(x) \Rightarrow e_2 \mid inr(x) \Rightarrow inr(x)$$

Thus, omitting the injections, the definition of map_T yields $map_T\, f\, x = x$ if $x \in E$ and $map_T\, f\, x = f\, x$ otherwise.

Nondeterminism where $TA = \wp(A)$ is the powerset on A, and

$$[a] = \{a\}$$
$$\mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 = \bigcup\{e_2 \mid x \in e_1\}$$

Then $map_T\, f\, x = \{f\, y \mid y \in x\}$.

Global state where $TA = (A \times S)^S$ for a set of states S, and

$$[a] = \lambda s.\,(a, s)$$
$$\mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 = \lambda s.let\ (x, s') = e_1\, s\ in\ e_2\, s'$$

Then $map_T\, f\, x\, s = (f\, y, s')$ where $x\, s = (y, s')$

Continuations where $TA = R^{(R^A)}$ for some fixed set R of "results", and

$$[a] = \lambda k.k\, a$$
$$\mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 = \lambda k.e_1\,(\lambda x.e_2\, k)$$

In this case, $map_T\, f\, x\, k = x\,(k \circ f)$

3 Conversion semantics

We follow Pierce (2002) and write $\mathscr{C} :: \Sigma \vdash A \leqslant B$ to distinguish the derivation $\mathscr{C}$ of a subtype judgment from the judgment itself. Similarly, we write $\mathscr{D} :: \Gamma \rhd e : A$ for the derivation of a typing judgment $\Gamma \rhd e : A$.

To obtain a conversion semantics, a conversion function $c_b^{b'} : b \to b'$ must be assumed for every basic subtyping $b \leqslant b'$. More precisely, let $\Sigma_{sub} = \langle B_\Sigma, C_{sub}\rangle$ be the signature with the same type constants as Σ, no basic subtypings, and where C_{sub} extends C_Σ by new constants $c_b^{b'}$ with $typeOf(c_b^{b'}) = b \to b'$, for every $b \leqslant b' \in S_\Sigma$. These basic conversions are required to commute: if $a, b \in B$ and $a_1, \ldots, a_m, b_1, \ldots, b_n \in B$ are such that both $a = a_1 \leqslant a_2 \leqslant \cdots \leqslant a_m = b$ and $a = b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n = b$ in S_Σ, then

$$\rhd\ c_{a_1}^{a_2}; \ldots; c_{a_{m-1}}^{a_m} = c_{b_1}^{b_2}; \ldots; c_{b_{n-1}}^{b_n} : a \to b \tag{1}$$

where associativity of composition allows us to omit parentheses. Let $\mathscr{E}_\Sigma$ be the set consisting of all equations of this form. Note that for any c_b^b, Equation (1) implies $c_b^b = id_b$.

Every subtype derivation $\mathscr{C} :: \Sigma \vdash A \leqslant B$ gives rise to a conversion $[\![\mathscr{C}]\!]$ which is a term over signature Σ_{sub}. This conversion function is defined by induction on $\mathscr{C}$.

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

- If $\mathscr{C} = \overline{\Sigma \vdash A \leqslant A}$ then $[\![\mathscr{C}]\!] = id_A$.
- If $\mathscr{C} = \dfrac{\mathscr{C}_1 :: \Sigma \vdash A \leqslant B \qquad \mathscr{C}_2 :: \Sigma \vdash B \leqslant C}{\Sigma \vdash A \leqslant C}$ then $[\![\mathscr{C}]\!] = [\![\mathscr{C}_1]\!] \,;[\![\mathscr{C}_2]\!]$.
- If $\mathscr{C} = \dfrac{b_1 \leqslant b_2 \in \Sigma}{\Sigma \vdash b_1 \leqslant b_2}$ then $[\![\mathscr{C}]\!] = c_{b_1}^{b_2}$.
- If $\mathscr{C} = \dfrac{\mathscr{C}_1 :: \Sigma \vdash B_1 \leqslant A_1 \qquad \mathscr{C}_2 :: \Sigma \vdash A_2 \leqslant B_2}{\Sigma \vdash A_1 \to A_2 \leqslant B_1 \to B_2}$ then $[\![\mathscr{C}]\!] = map_{\to} [\![\mathscr{C}_1]\!]\ [\![\mathscr{C}_2]\!]$.
- If $\mathscr{C} = \dfrac{\mathscr{C}' :: \Sigma \vdash A \leqslant B}{\Sigma \vdash TA \leqslant TB}$ then $[\![\mathscr{C}]\!] = map_T\ [\![\mathscr{C}']\!]$.

It is easy to verify that, for every $\mathscr{C} :: \Sigma \vdash A \leqslant B$, the conversion $[\![\mathscr{C}]\!]$ is a closed, well-typed term of type $A \to B$ over signature Σ_{sub}.

Intuitively at least, choosing $map_T\ [\![\mathscr{C}']\!]$ as conversion function between monadic types is sensible. A brief glance at the examples given earlier confirms this.

Exceptions. For the exception monad, the conversion is applied to proper values but exceptions are passed on.

Nondeterminism. For nondeterministic computations, the conversion is applied pointwise to coerce every possible result value to the supertype.

Global state. For stateful computations, the conversion is applied to the result value but the store remains unaffected.

Continuations. For computations in continuation passing style, the continuation is coerced by precomposition with the conversion.

The translation of terms proceeds by replacing instances of rule (*sub*) by applications of the correspondingly derived conversions. Formally, this is defined by induction on the derivation $\mathscr{D} :: \Gamma \triangleright e : A$.

- If $\mathscr{D} = \dfrac{\mathscr{D}' :: \Gamma \triangleright e : A \qquad \mathscr{C} :: \Sigma \vdash A \leqslant B}{\Gamma \triangleright e : B}$ then $[\![\mathscr{D}]\!] = [\![\mathscr{C}]\!]\,([\![\mathscr{D}']\!])$.
- In all other cases, the translation is trivial.

Note that $\Gamma \triangleright [\![\mathscr{D}]\!] : A$ is a well-typed term over signature Σ_{sub}. In fact, this may be derived without use of (*sub*).

4 Coherence

In this section, we establish coherence of the conversion semantics: the translations of any two derivations of the same judgment are provably equal. As in previous work, the proof is by a sequence of transformations of derivations, employing the rules of Table 3.

We say that two derivations $\mathscr{C}_1, \mathscr{C}_2 :: \Sigma \vdash A \leqslant B$ are *equivalent* if $[\![\mathscr{C}_1]\!] = [\![\mathscr{C}_2]\!]$ is provable in $ML_T(\Sigma_{sub})$ from the equations in $\mathscr{E}_\Sigma$, and analogously for derivations $\mathscr{D}_1, \mathscr{D}_2$ of the same typing judgment $\Gamma \triangleright e : A$. Note that the conversion semantics is compositional: if $\mathscr{C}_1$ is equivalent to $\mathscr{C}_2$ and $\mathscr{D}_1$ is equivalent to $\mathscr{D}_2$ then

$$[\![\mathscr{C}[\mathscr{C}_1]]\!] = [\![\mathscr{C}[\mathscr{C}_2]]\!] \quad \text{and} \quad [\![\mathscr{D}[\mathscr{D}_1]]\!] = [\![\mathscr{D}[\mathscr{D}_2]]\!]$$

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

Table 3. *Proof transformations: subtyping derivations*

T-ArrowRef

$$\frac{}{\Sigma \vdash A \to B \leqslant A \to B} \quad \Longrightarrow \quad \frac{\frac{}{\Sigma \vdash A \leqslant A} \qquad \frac{}{\Sigma \vdash B \leqslant B}}{\Sigma \vdash A \to B \leqslant A \to B}$$

T-MonadRef

$$\frac{}{\Sigma \vdash TA \leqslant TA} \quad \Longrightarrow \quad \frac{\frac{}{\Sigma \vdash A \leqslant A}}{\Sigma \vdash TA \leqslant TA}$$

T-Arrow

$$\frac{\frac{\begin{array}{c}\mathscr{C}_1 :: \Sigma \vdash C_1 \leqslant A_1 \\ \mathscr{C}_2 :: \Sigma \vdash A_2 \leqslant C_2\end{array}}{\Sigma \vdash A_1 \to A_2 \leqslant C_1 \to C_2} \qquad \frac{\begin{array}{c}\mathscr{C}_3 :: \Sigma \vdash B_1 \leqslant C_1 \\ \mathscr{C}_4 :: \Sigma \vdash C_2 \leqslant B_2\end{array}}{\Sigma \vdash C_1 \to C_2 \leqslant B_1 \to B_2}}{\Sigma \vdash A_1 \to A_2 \leqslant B_1 \to B_2} \quad \Longrightarrow \quad \frac{\frac{\mathscr{C}_3 \qquad \mathscr{C}_1}{\Sigma \vdash B_1 \leqslant A_1} \qquad \frac{\mathscr{C}_2 \qquad \mathscr{C}_4}{\Sigma \vdash A_2 \leqslant B_2}}{\Sigma \vdash A_1 \to A_2 \leqslant B_1 \to B_2}$$

T-Monad

$$\frac{\frac{\mathscr{C}_1 :: \Sigma \vdash A \leqslant C}{\Sigma \vdash TA \leqslant TC} \qquad \frac{\mathscr{C}_2 :: \Sigma \vdash C \leqslant B}{\Sigma \vdash TC \leqslant TB}}{\Sigma \vdash TA \leqslant TB} \quad \Longrightarrow \quad \frac{\frac{\mathscr{C}_1 \qquad \mathscr{C}_2}{\Sigma \vdash A \leqslant B}}{\Sigma \vdash TA \leqslant TB}$$

where $\mathscr{C}[\mathscr{C}']$ denotes (one or more) occurrences of a subderivation $\mathscr{C}'$ in $\mathscr{C}$, and similarly for $\mathscr{D}[\mathscr{D}']$. This observation is used to simplify the arguments given below.

Lemma 4.1

For any derivation $\mathscr{C}$, the application of a transformation rule from Table 3 yields an equivalent derivation $\mathscr{C}'$.

Proof

It is easy to see that each rule transforms a derivation $\mathscr{C}$ into a derivation $\mathscr{C}'$ of the same judgment. Moreover, the conversions obtained by translating the left-hand and right-hand sides, respectively, agree.

- Case T-ArrowRef. Equivalence follows from $id_{A\to B} = map_{\to}(id_A)(id_B)$.
- Case T-MonadRef. Similarly, since $id_{TA} = map_T(id_A)$.
- Case T-Arrow. By $(map_{\to} f_1\, f_2); (map_{\to} f_3\, f_4) = map_{\to}(f_3 ; f_1)\,(f_2 ; f_4)$.
- Case T-Monad. Similarly, by $map_T f_1 ; map_T f_2 = map_T (f_1 ; f_2)$.

The statement now follows by compositionality of the conversion semantics. □

By repeated use of T-ArrowRef and T-MonadRef, derivations of $\Sigma \vdash A \leqslant A$ may be expanded so that they reflect the structure of A. The rules T-Arrow and T-Monad can be used to push instances of the transitivity rule to the leaves of a derivation tree. The following lemmas make this procedure precise:

Lemma 4.2

For any subtype derivation $\mathscr{C}$ there is an equivalent derivation where (*ref*) is used only on type constants.

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

Proof

Suppose there is a subderivation of a judgment $\Sigma \vdash A \to B \leqslant A \to B$ by (*ref*). It may be replaced by the right-hand side of T-ArrowRef to obtain an equivalent derivation that uses (*ref*) only on strictly smaller types. Similarly, any subderivation of $\Sigma \vdash TA \leqslant TA$ by (*ref*) may be replaced by the right-hand side of T-MonadRef, using (*ref*) on strictly smaller types. Repeating this transformation process exhaustively must therefore terminate, with a derivation equivalent to $\mathscr{C}$ where (*ref*) is used only on type constants. □

Lemma 4.3

For any subtype derivation $\mathscr{C}$ there is an equivalent derivation $\mathscr{C}'$ where (*ref*) and (*trans*) are used only on type constants, but not for functional or monadic types.

Proof

Transforming derivations by T-Arrow and T-Monad reduces the total number of "→" and "T" symbols, respectively, occurring in the derivation. Consequently the process of exhaustively transforming a derivation must terminate. Since neither transformation introduces new instances of (*ref*), by Lemma 4.2 it is clear that we may restrict attention to derivations where inference rule (*ref*) is used only on base types.

We argue that a derivation $\mathscr{C}$ is already of the required form if no transformation applies. Suppose that $\mathscr{C}$ contains an inference

$$\frac{\mathscr{C}_1 :: \Sigma \vdash TA \leqslant C' \qquad \mathscr{C}_2 :: \Sigma \vdash C' \leqslant TB}{\Sigma \vdash TA \leqslant TB}$$

using (*trans*) for monadic types. Amongst all such inferences, consider one that does not use (*trans*) on function or monadic types in the derivation of its hypotheses. By Lemma 2.1, C' must be of the form TC for some C. By our earlier assumption, neither hypothesis is a direct inference by (*ref*), so both $\mathscr{C}_1$ and $\mathscr{C}_2$ must end with an application of (*monad*) which is the only rule besides (*ref*) and (*trans*) with conclusions of the form $\Sigma \vdash TA \leqslant TC$ and $\Sigma \vdash TC \leqslant TB$, respectively. Clearly this entails that a further transformation by T-Monad is possible.

The case where $\mathscr{C}$ contains an inference by (*trans*) with conclusion of the form $\Sigma \vdash A_1 \to A_2 \leqslant B_1 \to B_2$ is similar, and may be found in Mitchell (1996). □

Lemma 4.4 (*Uniqueness of conversions*)

If $\mathscr{C}_1, \mathscr{C}_2 :: \Sigma \vdash A \leqslant B$ then the equation $\triangleright [\![\mathscr{C}_1]\!] = [\![\mathscr{C}_2]\!] : A \to B$ is provable in $ML_T(\Sigma)$ from $\mathscr{E}_\Sigma$.

Proof

By Lemma 2.1, types A and B match. By Lemma 4.3, we may assume that (*trans*) and (*ref*) are only used on type constants. This entails that $\mathscr{C}_1$ and $\mathscr{C}_2$ have the same structure, with the possible exception of the way that a subtyping $b \leqslant b'$ between type constants $b, b' \in B_\Sigma$ is proved by (*trans*) and (*ref*) from basic subtyping assertions in S_Σ. The equivalence of all such proofs is guaranteed, however, by the identities (1) contained in $\mathscr{E}_\Sigma$. □

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

Table 4. *Proof transformations: typing derivations*

T-Sub

$$\dfrac{\dfrac{\begin{array}{c}\mathscr{D} :: \Gamma \rhd e : A \\ \mathscr{C}_1 :: \Sigma \vdash A \leqslant B\end{array}}{\Gamma \rhd e : B} \qquad \mathscr{C}_2 :: \Sigma \vdash B \leqslant C}{\Gamma \rhd e : C} \quad \Longrightarrow \quad \dfrac{\mathscr{D} \qquad \dfrac{\mathscr{C}_1 \qquad \mathscr{C}_2}{\Sigma \vdash A \leqslant C}}{\Gamma \rhd e : C}$$

T-Abs

$$\dfrac{\dfrac{\mathscr{D} :: \Gamma, x{:}A \rhd e : C \qquad \mathscr{C} :: \Sigma \vdash C \leqslant B}{\Gamma, x{:}A \rhd e : B}}{\Gamma \rhd \lambda x{:}A.e : A \to B} \quad \Longrightarrow \quad \dfrac{\dfrac{\mathscr{D}}{\Gamma \rhd \lambda x{:}A.e : A{\to}C} \quad \dfrac{\dfrac{}{\Sigma \vdash A \leqslant A} \qquad \mathscr{C}}{\Sigma \vdash A{\to}C \leqslant A{\to}B}}{\lambda x{:}A.e : A \to B}$$

T-App

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\mathscr{C}_1 :: \Sigma \vdash A \leqslant A' \\ \mathscr{C}_2 :: \Sigma \vdash B' \leqslant B\end{array}}{\Sigma \vdash A'{\to}B' \leqslant A{\to}B} \\ \mathscr{D}_1 :: \Gamma \rhd e_1 : A'{\to}B'\end{array}}{\Gamma \rhd e_1 : A \to B} \qquad \mathscr{D}_2 :: \Gamma \rhd e_2 : A}{\Gamma \rhd e_1\, e_2 : B} \quad \Longrightarrow \quad \dfrac{\dfrac{\mathscr{D}_1 \qquad \dfrac{\mathscr{D}_2 \qquad \mathscr{C}_1}{\Gamma \rhd e_2 : A'}}{\Gamma \rhd e_1\, e_2 : B'} \qquad \mathscr{C}_2}{\Gamma \rhd e_1\, e_2 : B}$$

T-Unit

$$\dfrac{\dfrac{\mathscr{D} :: \Gamma \rhd e : A \qquad \mathscr{C} :: \Sigma \vdash A \leqslant B}{\Gamma \rhd e : B}}{\Gamma \rhd [e] : TB} \quad \Longrightarrow \quad \dfrac{\dfrac{\mathscr{D}}{\Gamma \rhd [e] : TA} \qquad \dfrac{\mathscr{C}}{\Sigma \vdash TA \leqslant TB}}{\Gamma \rhd [e] : TB}$$

T-BindL

$$\dfrac{\dfrac{\mathscr{D}_1 :: \Gamma \rhd e_1 : TA \qquad \dfrac{\mathscr{C} :: \Sigma \vdash A \leqslant A'}{\Sigma \vdash TA \leqslant TA'}}{\Gamma \rhd e_1 : TA'} \qquad \mathscr{D}_2 :: \Gamma, x{:}A' \rhd e_2 : TB}{\Gamma \rhd \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 : TB}$$

$$\Longrightarrow \quad \dfrac{\mathscr{D}_1 \qquad \mathscr{D}_2[x{:}A, \mathscr{C}]}{\Gamma \rhd \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 : TB}$$

T-BindR

$$\dfrac{\mathscr{D}_1 :: \Gamma \rhd e_1 : TA \qquad \dfrac{\begin{array}{c}\mathscr{D}_2 :: \Gamma, x{:}A \rhd e_2 : TB' \\ \mathscr{C} :: \Sigma \vdash TB' \leqslant TB\end{array}}{\Gamma, x{:}A \rhd e_2 : TB}}{\Gamma \rhd \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 : TB} \quad \Longrightarrow \quad \dfrac{\dfrac{\mathscr{D}_1 \qquad \mathscr{D}_2}{\Gamma \rhd \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 : TB'} \qquad \mathscr{C}}{\Gamma \rhd \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 : TB}$$

Table 4 presents several transformations on typing derivations, generally moving subsumption "down" to the conclusion (T-BindL is an exception). Repeating these transformations results in a derivation of the *minimum typing* of a term, where the use of (*sub*) is restricted to the arguments of function applications.

T-BindL transforms derivations in a nonlocal way, and relies on the following weakening property. Suppose $\mathscr{C} :: \Sigma \vdash A \leqslant A'$ and $\mathscr{D} :: \Gamma, x{:}A' \rhd e : B$. We let $\mathscr{D}[x{:}A,\mathscr{C}]$ be the derivation obtained from $\mathscr{D}$ by Equation (1) replacing the assumption $x{:}A'$ in the type contexts by $x{:}A$, and Equation (2) replacing every inference of $\Gamma, x{:}A', \Gamma' \rhd x : A'$ by (var) with an inference $\Gamma, x{:}A, \Gamma' \rhd x : A$ followed by (sub) with hypothesis $\mathscr{C}$. That $\mathscr{D}[x{:}A,\mathscr{C}]$ is a derivation of $\Gamma, x{:}A \rhd e : B$ and

$$\Gamma, x{:}A \rhd [\![\mathscr{D}[x{:}A,\mathscr{C}]]\!] = [\![\mathscr{D}]\!]\ [([\![\mathscr{C}]\!]\ x)/x] : B \tag{2}$$

follows by an induction on $\mathscr{D}$.

Lemma 4.5
For any derivation $\mathscr{D}$, transformation by a rule from Table 4 yields an equivalent derivation $\mathscr{D}'$.

Proof
Clearly the transformation provides a derivation of the same judgment (the case of T-BindL is an immediate consequence of the considerations above). The translation of the left-hand and right-hand sides agree in all cases.

- Case T-Sub. Immediate, by $f_2(f_1\, e) = (f_1\,; f_2)\, e$.
- Case T-Abs. By $map_{\to}(id_A)(f)(\lambda x.e) = (\lambda x.e); f = \lambda x.(f\, e)$ and Lemma 4.4.
- Case T-App. Since $(map_{\to} f_1\, f_2\, e_1)\, e_2 = (f_1\,; e_1\,; f_2)\, e_2 = f_2(e_1(f_1\, e_2))$.
- Case T-Unit. By $(\to.\beta)$ and $(T.\beta)$, $map_T f\ [e] = \mathsf{let}\ x \Leftarrow [e]\ \mathsf{in}\ [f x] = [f e]$.
- Case T-BindR. Since $map_T f\ (\mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2) = \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ map_T f\ e_2$, by $(\to.\beta)$ and $(T.ass)$.
- Case T-BindL. We have $\mathsf{let}\ x \Leftarrow map_T f\ e_1\ \mathsf{in}\ e_2 = \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2[(f x)/x]$, by $(\to.\beta)$ and $(T.ass)$. The required equality follows by Equation (2).

Compositionality of the conversion semantics implies the lemma. □

Lemma 4.6
For any typing derivation $\mathscr{D}$ there is an equivalent derivation $\mathscr{D}'$ that uses (sub) only for the arguments of function applications and (possibly) the final inference.

Proof
Establishing termination for the transformations from Table 4 is tricky, because T-BindL introduces (many) new instances of the subsumption rule. Suppose w_Γ maps each x in Γ to a natural number $w_\Gamma(x) \geqslant 0$; we extend this to associate a measure $w_\Gamma(\mathscr{D}) \in \mathbb{N}$ with each derivation $\mathscr{D}$ of a judgment $\Gamma \rhd e : A$ as follows:

- $w_\Gamma(\mathscr{D}) = 0$ if $\mathscr{D} :: \Gamma \rhd c : A$ is an instance of $(const)$.
- $w_\Gamma(\mathscr{D}) = w_\Gamma(x)$ if $\mathscr{D} :: \Gamma \rhd x : A$ is an instance of (ax).
- $w_\Gamma(\mathscr{D}) = 1 + w_\Gamma(\mathscr{D}')$ if $\mathscr{D}$ ends in (sub) applied to $\mathscr{D}'$ and some $\mathscr{C}$.
- $w_\Gamma(\mathscr{D}) = 3 \cdot w_\Gamma(\mathscr{D}_1) + w_\Gamma(\mathscr{D}_2)$, if $\mathscr{D} :: \Gamma \rhd e_1\, e_2 : A$ ends in (app) applied to $\mathscr{D}_1 :: \Gamma \rhd e_1 : B \to A$ and $\mathscr{D}_2 :: \Gamma \rhd e_2 : B$.
- $w_\Gamma(\mathscr{D}) = 2 \cdot w_{\Gamma,x:B}(\mathscr{D}')$ if $\mathscr{D} :: \Gamma \rhd \lambda x{:}B.e : A$ ends in (abs) applied to $\mathscr{D}' :: \Gamma, x{:}B \rhd e : B'$, where $w_{\Gamma,x:B}$ maps x to 0 and otherwise agrees with w_Γ.
- $w_\Gamma(\mathscr{D}) = 2 \cdot w_\Gamma(\mathscr{D}')$ if $\mathscr{D} :: \Gamma \rhd [e] : A$ ends in $(unit)$ applied to $\mathscr{D}' :: \Gamma \rhd e : B$.
- $w_\Gamma(\mathscr{D}) = w_\Gamma(\mathscr{D}_1) + 2 \cdot w_{\Gamma,x:B}(\mathscr{D}_2)$, if $\mathscr{D} :: \Gamma \rhd \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 : A$ ends in $(bind)$ applied to $\mathscr{D}_1 :: \Gamma \rhd e_1 : TB$ and $\mathscr{D}_2 :: \Gamma, x{:}B \rhd e_2 : A$, where $w_{\Gamma,x:B}$ maps x to $w_\Gamma(\mathscr{D}_1)$ and otherwise agrees with w_Γ.

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

Table 5. *Equations for products, sums, polymorphism, and recursion*

$(\times.\beta)$	$\Gamma \rhd \mathsf{fst}\,(e_1,e_2) = e_1 : A \qquad \Gamma \rhd \mathsf{snd}\,(e_1,e_2) = e_2 : A'$
$(\times.\eta)$	$\Gamma \rhd (\mathsf{fst}\,e,\mathsf{snd}\,e) = e : A \times A'$
$(+.\beta)$	$\Gamma \rhd \mathsf{case}\ \mathsf{in}_i(e)\ \mathsf{of}\ \mathsf{in}_1\,y \Rightarrow e_1 \mid \mathsf{in}_2\,y \Rightarrow e_2 = e_i[e/x] : A$
$(+.\eta)$	$\Gamma \rhd f(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\,y \Rightarrow e_1 \mid \mathsf{in}_2\,y \Rightarrow e_2) = \mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\,y \Rightarrow f(e_1) \mid \mathsf{in}_2\,y \Rightarrow f(e_2) : A$
$(\forall.\beta)$	$\Gamma \rhd (\Lambda\alpha.e)_A = e[A/\alpha] : B[A/\alpha]$
$(\forall.\eta)$	$\Gamma \rhd \Lambda\alpha.e_\alpha = e : \forall\alpha.A \qquad \text{provided } \alpha \notin fv(e)$
$(\mu.\beta)$	$\Gamma \rhd \mathsf{unfold}(\mathsf{fold}_{\mu X.A}\,e) = e : A[\mu X.A/X]$
$(\mu.\eta)$	$\Gamma \rhd \mathsf{fold}_{\mu X.A}(\mathsf{unfold}\,e) = e : \mu X.A$
(fix)	$\Gamma \rhd \mathsf{fix}_B\,f = f(\mathsf{fix}_B\,f) : B \qquad B \text{ pointed}$

Inspection of the transformations now shows that each application strictly decreases this measure. Since the measure is independent of the subtyping derivations appearing in $\mathscr{D}$, it is invariant under application of transformations from Table 3. Consequently, the union of both systems is terminating.

Applying the transformations exhaustively results in an equivalent derivation, by Lemmas 4.1 and 4.5. By Lemma 4.3, this derivation uses (*trans*) and (*ref*) only on type constants. But then this derivation must already have the required shape, for otherwise one of the transformations in Table 4 applies. □

Now suppose $\Gamma \rhd e : A$ is derivable. An induction on e shows that any typing derivation that uses (*sub*) only for the arguments of applications and a final step is uniquely determined by Γ, e, and A, except for the derivation of subtyping judgments. In combination with Lemmas 4.4 and 4.6 this proves the following:

Theorem 4.1 (*Coherence*)
If $\mathscr{D}_1, \mathscr{D}_2 :: \Gamma \rhd e : A$ then $\Gamma \rhd [\![\mathscr{D}_1]\!] = [\![\mathscr{D}_2]\!] : A$ is provable from $\mathscr{E}_\Sigma$.

5 Extensions

This section briefly discusses some extensions to the basic setting, as needed for more realistic applications. Most of them have appeared in the literature before, and the main point is that they combine well with the monadic types.

Products. The addition of product types follows the earlier procedure. Subtyping is covariant, i.e., $A \times A' \leqslant B \times B'$ whenever $A \leqslant B$ and $A' \leqslant B'$, and we omit the (standard) typing rules. For $f : A \to B$ and $g : A' \to B'$ the function

$$map_\times\, f\, g = \lambda x{:}A{\times}A'.(f(\mathsf{fst}\,x), g(\mathsf{snd}\,x))$$

is used to define the conversion from $A \times A'$ to $B \times B'$. For the coherence proof, the β- and η-equalities for products (Table 5) establish that $map_\times$ preserves identities and function composition. This then allows us to replace all uses of (*ref*) and (*trans*) on product types, by semantics-preserving proof transformations analogous to those of Table 3. Typing derivations ending in the introduction of pairs or a projection are transformed by pushing the subsumption rule from antecedent to conclusion, analogous to Table 4, with an appropriate adaptation of the subtype derivation.

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

A generalization is possible to records, i.e., labeled n-ary products, with additional width-subtyping. The nullary case then gives a terminal type, unit. In turn, this can be used to interpret a greatest type $\top$ with respect to the subtype ordering.

Sums. For sum types, with covariant subtyping, the function

$$map_+ \, f \, g = \lambda x{:}A{+}A'.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\, y \Rightarrow \mathsf{in}_1(f\, y) \mid \mathsf{in}_2\, y \Rightarrow \mathsf{in}_2(g\, y)$$

is used to define the conversion $A + A' \to B + B'$ from $f : A \to B$ and $g : A' \to B'$. From the β- and η-equalities for sums (Table 5) it follows that map_+ preserves identities and function composition, and we can use semantics-preserving proof transformations to eliminate uses of (*ref*) and (*trans*) on sum types as in Table 3.

For the transformation of typing derivations, subsumption can simply be pushed down through the introduction rules. Now consider the elimination construct:

$$\frac{\mathscr{D} :: \Gamma \triangleright e : A_1 + A_2 \qquad \mathscr{D}_1 :: \Gamma, y{:}A_1 \triangleright e_1 : B \qquad \mathscr{D}_2 :: \Gamma, y{:}A_2 \triangleright e_2 : B}{\Gamma \triangleright \mathsf{case}\ e\ \mathsf{of}\ \mathsf{in}_1\, y \Rightarrow e_1 \mid \mathsf{in}_2\, y \Rightarrow e_2 : B}$$

The case where $\mathscr{D}$ ends with (*sub*) is handled by a transformation similar to T-BindL. The case where one or both of $\mathscr{D}_1$ and $\mathscr{D}_2$ end with (*sub*) is more complicated: in general, e_1 could be coerced from B_1 to B while e_2 is coerced from some *different* B_2. Therefore, to achieve the uniqueness of "normalized" derivations, it is necessary that all (bounded) joins exist in the type system. With this proviso, a transformation similar to T-BindR from Table 4 works, by factoring the conversions $B_i \to B$ through $B_1 \vee B_2$, and pushing the coercion $B_1 \vee B_2 \to B$ to the conclusion.

Polymorphism. As shown by Breazu-Tannen *et al.* (1991), subtyping of (bounded) polymorphic types can be interpreted by conversion functions expressible in a polymorphically typed lambda calculus: a bounded quantification $\forall\alpha \leqslant A.B$ is viewed as depending on a witness conversion, and becomes $(\forall\alpha \leqslant A.B)^* = \forall\alpha.(\alpha \to A^*) \to B^*$. Accordingly, (the derivation of) a subtype judgment $\Sigma; \vec{\alpha} \leqslant \vec{A} \vdash B \leqslant B'$ with free type variables $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ determines a term e, with free variables $\vec{x} = x_1, \ldots, x_n$ that correspond to coercions from α_i to A_i. Consider the inference rule for subtyping quantified types,

$$\frac{\Sigma; \vec{\alpha} \leqslant \vec{A}, \alpha \leqslant A \vdash B \leqslant B'}{\Sigma; \vec{\alpha} \leqslant \vec{A} \vdash \forall\alpha{\leqslant}A.B \leqslant \forall\alpha{\leqslant}A.B'}$$

Corresponding to the type constructor $A, B \mapsto \forall\alpha{\leqslant}A.B$, with α possibly free in B, the conversion map determined by this inference rule is derived from the following function:

$$map_\forall\, f = \lambda z{:}(\forall\alpha{\leqslant}A.B)^*.\Lambda\alpha.\lambda x{:}\alpha{\to}A^*.f_\alpha\, x\, (z_\alpha\, x)$$

where $f : \forall\alpha.(\alpha \to A^*) \to B^* \to B'^*$. Thus, assuming $g : \forall\alpha.(\alpha \to A^*) \to B'^* \to B''^*$, the equations $map_\forall\, (\Lambda\alpha.\lambda x{:}\alpha{\to}A.id_B) = id_{\forall\alpha.(\alpha\to A)\to B}$ and

$$(map_\forall\, f); (map_\forall\, g) = map_\forall\, (\Lambda\alpha.\lambda x{:}\alpha{\to}A.(f_\alpha x; g_\alpha x))$$

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

follow from the various β- and η-equations. They serve to justify proof transformations to eliminate (*ref*) and (*trans*) from derivations, analogous to the ones of Table 3. That (*sub*) and type abstraction can be permuted is a consequence of

$$map_\forall\, f\,(\Lambda\alpha\lambda x{:}\alpha{\to}A.e) = \Lambda\alpha\lambda x{:}\alpha{\to}A.\, f_\alpha\, x\, e$$

and that (*sub*) can be pushed through type application is an immediate consequence of the definition of $map_\forall$.

Breazu-Tannen *et al.* (1991) also discuss a stronger rule for subtyping-bounded universals which permits (contravariant) subtyping of the bounds. However, this more general rule is incompatible with the existence of joins which are needed for sums, as described above.

Type recursion. Polymorphism is also useful to give an account of type recursion. Cardelli's Amber rule (Pierce 2002) gives a form of subtyping between recursive types under a set $\vec{\alpha} \leqslant \vec{\alpha}'$ of subtype assumptions for type variables:

$$\frac{\Sigma;\vec{\alpha} \leqslant \vec{\alpha}', \alpha \leqslant \alpha' \vdash B \leqslant B'}{\Sigma;\vec{\alpha} \leqslant \vec{\alpha}' \vdash \mu\alpha.B \leqslant \mu\alpha'.B'}$$

For instance, it lets us infer that the type of integer lists, $B = \mu\alpha.\mathsf{unit} + \mathsf{int}{\times}\alpha$, is a subtype of lists with real number entries, $B' = \mu\alpha.\mathsf{unit}{+}\mathsf{float}{\times}\alpha$. Intuitively, it is clear how to coerce an integer list to a real number list: the conversion $c : \mathsf{int} \to \mathsf{float}$ is applied to every element of the list, i.e., using the recursively defined function

$$\begin{aligned} coerce(x) = \mathsf{case}\ (\mathsf{unfold}\, x)\ \mathsf{of}\ \mathsf{in}_1\, y &\Rightarrow \mathsf{fold}_{B'}(\mathsf{in}_1\, y) \\ \mid \mathsf{in}_2\, y &\Rightarrow \mathsf{fold}_{B'}(\mathsf{in}_2(c(\mathsf{fst}\, y), coerce(\mathsf{snd}\, y))) \end{aligned}$$

In fact, analogously defined conversions work for arbitrary recursive types; for simplicity, let us consider the case without nested recursion. Then, each subtype derivation of $\Sigma;\alpha \leqslant \alpha' \vdash B \leqslant B'$, with α possibly free in B and α' possibly free in B', respectively, yields a polymorphic conversion k with type $\forall\alpha\alpha'.(\alpha \to \alpha') \to B \to B'$. The conversion from $A = \mu\alpha.B$ to $A' = \mu\alpha'.B'$ is determined by

$$\mathsf{fix}_{A\to A'}(\lambda f{:}A{\to}A'.\lambda x{:}A.\,\mathsf{fold}_{A'}(k_{AA'}\, f\,(\mathsf{unfold}\, x)))$$

Technically, given k and k' corresponding to $\Sigma;\alpha \leqslant \alpha' \vdash B \leqslant B'$ and $\Sigma;\alpha' \leqslant \alpha'' \vdash B' \leqslant B''$, one has for all $g;h = g';h' : A \to A'$ (via C and C', respectively) that the equation $(k_{AC}g);(k'_{CA'}h) = (k_{AC'}g');(k'_{C'A'}h')$ holds. From this, by the axioms, it is possible to show that the transitivity rule can be removed from subtype derivations. (The proof relies on fixed point induction with fix denoting the least fixed point operator, as in the "usual" complete partial order (CPO) models of recursive types, or on an operationally based unwinding property of recursive functions.)

The introduction of recursively defined types requires some care because the combination of fixed points and the eta axiom for sum types is inconsistent. Deviating from the approach of (Tannen *et al.* 1989; Breazu-Tannen *et al.* 1991) we can take advantage of the monadic types: we restrict recursion to *pointed* types,

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

i.e., those of the form TA, $A \to B$, $B \times B'$ and $\mu\alpha.B$ where B is pointed, and only consider recursive types $\mu\alpha.B$ well formed if B is pointed.

Monadic operations. For the particular operations associated with each notion of computation, one usually introduces additional constructs. This can be tricky.

Exceptions where one may consider expressions $\Gamma \triangleright \mathsf{raise}_e : TA$ (for arbitrary A) to throw an exception $e \in E$. Semantically, raise_e is $inr(e) \in TA = A + E$. Note that $\Gamma \triangleright map_T f\,(\mathsf{raise}_e) = \mathsf{raise}_e : TB$ for all $f : A \to B$, so one expects that this extension does preserve coherence of the interpretation: removing all uses of subsumption following the introduction rule for raise_e leads to the uniqueness of derivations that is exploited in the coherence theorem.

Nondeterminism where one may consider a binary choice $\Gamma \triangleright e_1 \oplus e_2 : TA$, assuming that $\Gamma \triangleright e_i : TA$ for $i = 1, 2$. It is interpreted as set union on $TA = \wp(A)$. Note that $map_T f\,(e_1 \oplus e_2) = (map_T f\, e_1) \oplus (map_T f\, e_2)$ holds under this interpretation, for all $f : A \to B$. This allows us to extend the coherence proof by pushing uses of subsumption on the components e_1 and e_2 from the premise to the conclusion of the typing rule for choice. Similar to the case construct for sum types, however, this relies on the existence of joins in the type system.

State where one may consider constructs to update and dereference memory locations. Due to the invariance of the reference type constructor, there are no coherence conditions required. However, adding an operation $\Gamma \triangleright \mathsf{new}\, e : T(Ref\, A)$, assuming that $\Gamma \triangleright e : A$, to dynamically allocate new memory is problematic: the invariance of *Ref* prevents us from moving possible uses of subsumption from the antecedent to the conclusion of the typing rule for new. Technically, this rule breaks the minimal type property, and the coherence proof does not extend.

An alternative to extending the expression syntax is to add the required operations as (polymorphic) constants. For instance, choice and allocation take the form $\oplus : \forall\alpha.T\alpha \times T\alpha \to T\alpha$ and $\mathsf{new} : \forall\alpha.\alpha \to T(Ref\,\alpha)$. Since type instantiation is reflected in the term syntax, any application of (*sub*) on the arguments of such constants becomes explicit. This case is already covered by the given coherence theorem.

Subtyping monadic types. It is also interesting to consider several monadic types, and subtyping between them. This situation arises naturally in work on effect analysis and in security type systems (Wadler & Thiemann 2003; Crary *et al.* 2005). For instance, a computation without side-effects can be viewed as having only trivial effects, corresponding to a conversion from the identity monad to the state monad. Similarly, a low-security computation (e.g., one that does not read high-security data) can be run in a high-security context (e.g., one that permits reading of high-security data), corresponding to a conversion between different state monads.

To make the elimination of reflexivity and transitivity from subtyping derivations work in this case, the conversion from T_1 to T_2 must be a function $c : \forall\alpha.T_1\alpha \to T_2\alpha$ such that for all $f : A \to B$, $c_A ; map_{T_2} f = map_{T_1} f ; c_B$.

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

6 Concluding remarks

The structure of our coherence proof is similar to the classic ones of Breazu-Tannen *et al.* (1991) and Curien and Ghelli (1992); the rewriting aspect of coherence proofs has been emphasized in Curien and Ghelli's work. Using a monadically typed language gives a somewhat dual approach to that of Breazu-Tannen *et al.* Rather than introducing a separate type of conversions (corresponding to "pure" functions), all potentially "impure" computations are encapsulated in the monad. Later work already conjectured that a simplification could be achieved with a computational metalanguage (Breazu-Tannen *et al.* 1990). As argued in Section 5 this is indeed the case, provided that fixed points are restricted to monadic types.

The coherence theorem provides an alternative (and considerably more elementary) proof to Schwinghammer (2005), where subtyping was considered for an ML-like language with general references: a model for this language (Levy 2004) may be presented as an instance of Moggi's calculus, where the monad combines nontermination, side-effects, and dynamic allocation.

Acknowledgments

I would like to thank Marco Kuhlmann and Bernhard Reus for their helpful comments on an earlier draft.

References

Aspinall, David & Compagnoni, Adriana B. (2001) Subtyping dependent types. *Theor. Comput. Sci.* **266**(1–2), 273–309.

Benton, Nick, Hughes, John & Moggi, Eugenio. (2002) Monads and effects. In *Advanced Lectures from International Summer School on Applied Aemantics, APPSEM 2000*, Barthe, Gilles, Dybjer, Peter, Pinto, Luís & Saraiva, João (eds), Lecture Notes in Computer Science, vol. 2395. Springer, Heidelberg, pp. 42–122.

Breazu-Tannen, Val, Coquand, Thierry, Gunter, Carl & Scedrov, Andre. (1991) Inheritance as implicit coercion. *Inf. Comput.* **93**(1), 172–221. Reprinted in Gunter and Mitchell (1994).

Breazu-Tannen, Val, Gunter, Carl A. & Scedrov, Andre. (1990) Computing with coercions. In *Proceedings of the ACM Conference on LISP and Functional Programming*, Kahn, Gilles (ed). ACM Press, New York, pp. 44–60.

Cardelli, Luca. (1988) A semantics of multiple inheritance. *Inf. Comput.* **76**(2/3), 138–164.

Crary, Karl, Kliger, Aleksey & Pfenning, Frank. (2005) A monadic analysis of information flow security with mutable state. *J. Funct. Program.* **15**(2), 249–291.

Curien, Pierre-Louis & Ghelli, Giorgio. (1992) Coherence of subsumption, minimum subtyping and type-checking in $F_{\leqslant}$. *Math. Struct. Comput. Sci.* **2**, 55–91. Reprinted in Gunter and Mitchell (1994).

Gunter, Carl A. & Mitchell, John C. (eds). (1994) *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, Cambridge, MA.

Levy, Paul Blain. (2004) *Call-by-Push-Value. A Functional/Imperative Synthesis*. Semantic Structures in Computation, vol. 2. Springer, New York.

Mitchell, John C. (1996) *Foundations for Programming Languages*. MIT Press, Cambridge, MA.

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press

Moggi, Eugenio. (1990) *An Abstract View of Programming Languages*. Tech. rept. ECS-LFCS-90-113. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

Moggi, Eugenio. (1991) Notions of computation and monads. *Inf. Comput.* **93**, 55–92.

Pierce, Benjamin C. (2002) *Types and Programming Languages*. MIT Press, Cambridge, MA.

Pierce, Benjamin C. & Steffen, Martin. (1997) Higher-order subtyping. *Theor. Comput. Sci.* **176**(1–2), 235–282.

Reynolds, John C. (1980) Using category theory to design implicit conversions and generic operators. In *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, Jones, Neil D. (ed). Lecture Notes in Computer Science, no. 94. Springer, Heidelberg. Reprinted in Gunter and Mitchell (1994).

Reynolds, John C. (1991) The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software (TACS'91)*, Ito, Takayasu & Meyer, Albert R. (eds). Lecture Notes in Computer Science, no. 526. Springer, Heidelberg, pp. 675–700.

Reynolds, John C. (2003) What do types mean?—From intrinsic to extrinsic semantics. In *Programming Methodology*, McIver, Annabelle & Morgan, Carroll (eds). Monographs in Computer Science. Springer, New York.

Schwinghammer, Jan. (2005) A typed semantics of higher-order store and subtyping. In *Proceedings Ninth Italian Conference on Theoretical Computer Science (ICTCS'05)*, Coppo, Mario, Lodi, Elena & Pinna, G. Michele (eds). Lecture Notes in Computer Science, vol. 3701. Springer, Heidelberg,pp. 390–405.

Tannen, Val, Gunter, Carl A. & Scedrov, Andre. (1989) *Denotational Semantics for Subtyping Between Recursive Types*. Research Report MS-CIS-89-63/Logic & Computation 12. Department of Computer and Information Science, University of Pennsylvania.

Wadler, Philip & Thiemann, Peter. (2003) The marriage of effects and monads. *ACM Trans. Comput. Logic* **4**(1), 1–32.

Zwanenburg, Jan. (1999) Pure type systems with subtyping. In *International Conference on Typed Lambda Calculi and Applications (TLCA'99)*. Lecture Notes in Computer Science, vol. 1581. Springer, Heidelberg, pp. 381–396.

https://doi.org/10.1017/S0956796808006886 Published online by Cambridge University Press