November 1989

Denotational Semantics for Subtyping Between Recursive Types

Val Tannen
University of Pennsylvania

Carl A. Gunter
University of Pennsylvania

Andre Scedrov
Stanford University

Denotational Semantics for Subtyping Between Recursive Types

Abstract

Inheritance in the form of subtyping is considered in the framework of a polymorphic type discipline with records, variants, and recursive types. We give a denotational semantics based on the paradigm that interprets subtyping as explicit coercion. The main technical result gives a coherent interpretation for a strong rule for deriving inheritances between recursive types.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-63.

Denotational Semantics For Subtyping Between Recursive Types

MS-CIS-89-63
LOGIC & COMPUTATION 12

V. Breazu-Tannen
C. A. Gunter
A. Scedrov

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

November 1989

DENOTATIONAL SEMANTICS FOR SUBTYPING BETWEEN RECURSIVE TYPES

(Extended Abstract)

V. Breazu-Tannen *C. A. Gunter* *A. Scedrov*[2]

University of Pennsylvania
December 4, 1989

Abstract. Inheritance in the form of subtyping is considered in the framework of a polymorphic type discipline with records, variants, and recursive types. We give a denotational semantics based on the paradigm that interprets subtyping as explicit coercion. The main technical result gives a coherent interpretation for a strong rule for deriving inheritances between recursive types.

1 Introduction

There have been several efforts recently to integrate the flexibility of object-oriented programming [GR83] with rich type disciplines such as the polymorphic lambda calculus. Such reseach has dealt with semantics, language design and type inference algorithms (see, for example, [Car88, CW85, Car, CM88, Wan87, JM88, OB88, Rem89, Wan89]). In [BCGS89] we proposed a semantic paradigm for typed languages which feature inheritance in the form of subtyping. This paradigm sees the subtyping relation as indicating the presence of a coercion function; we therefore refer to this as the "inheritance-as-implicit-coercion" paradigm. We showed how these coercions could be compiled by making them explicit using definable terms of a typed calculus *without* subtyping. For example, if a subtyping relation $s < t$ is made explicit as $P : s \to t$, then an instance of the inheritance rule

$$\frac{\vdash e : s \qquad \vdash s < t}{\vdash e : t}$$

is interpreted as

$$\frac{\vdash e : s \qquad \vdash P : s \to t}{\vdash P(e) : t}$$

In this approach it must be shown that the obtained explicit coercions do not depend on the way in which inheritance between complex type expressions is derived from the basic inheritance relations on records and variants. It must also be shown that the typing information obtained by using the inheritance rule in different ways at various stages of type-checking is *coherent.* In other words, the meaning of typing information $a : r$ in a context should not depend on the way in which the typing is derived. These coherence properties were established in [BCGS89] for a calculus in which interaction between inheritance and recursive types is (basically) trivial. Using this paradigm, we showed how various models of parametric polymorphism and recursive types could be seen as models for calculi with inheritance in

[2]Author's addresses. Breazu-Tannen and Gunter: Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia PA 19104, USA. Scedrov: Department of Mathematics, University of Pennsylvania.

the form of subtyping, even though these models have no relevant concept of type incusion. However, the problem of subtyping between recursive types was left open.

The work which we discuss in this abstract extends the inheritance-as-implicit-coercion paradigm and the related coherence properties to a system which has a strong form of subtyping between recursive types. In [BCGS89] we suggested the rule

$$\frac{A, a \vdash s < t}{A \vdash \mu a.s < \mu a.t}$$

where A is an inheritance context and the variable a has only *positive* occurrences in the types s and t. This restriction rules out unwanted inheritance judgements which have no clear computational or mathematical interpretation. For example, if this restriction were relaxed, we could derive such inheritances as $\mu a.a \to s < \mu a.a \to t$ for types s and t which satisfy $s < t$. We see no reasonable coercion between these two types. On the other hand, there are instances in which this rule seems too restrictive. Consider, for example, the following recursive ML datatype declarations

```
datatype small = Int2 of int | Rec2 of {l:small};
```

and the declaration

```
datatype large = Int1 of int | Rec1 of {l:large, m:large->large};
```

There *is* an evident coercion from `large` into `small`. Indeed, this coercion is definable as a (recursive) ML program:

```
fun f (Int1 n) = Int2 (n)
  | f (Rec1 {l=y, m=_}) = Rec2 {l=f(y)};
```

which has type `large -> small`.

In this paper we consider the following rule

$$\frac{a < b \vdash s < t}{\vdash \mu a.s < \mu b.t}$$

which is strong enough to derive inheritances such as the one between the ML recursive type examples in the paragraph above.[3]

Once the inheritance-as-implicit-coercion paradigm is adopted, it is straightforward to see what the explicit coercions corresponding to such inheritances ought to be:

$$\frac{f : a \to b \vdash P : s \to t}{\vdash (\mu f : (\mu a.s \to \mu b.t).\ \lambda x : (\mu a.s).\ \mathsf{intro}[\mu b.t](P(\mathsf{elim}[\mu a.s](x)))) : \mu a.s \to \mu b.t}$$

(The precise syntax is given in section 2; in particular one cannot quite use P in the coercion between the recursive types because the type of f is not right.)

The problem, as in general with this paradigm, is to show that such an interpretation is *coherent*. The need for coherence arises because interpretations are given to typing derivations, rather than typed terms. Since we consider the *typed terms* as the syntactic entities to which meaning is to be assigned, we need

[3]We thank Jim O'Toole for bringing this rule to our attention. Subsequently, we learned that this rule was already included in Cardelli's Amber [Car86].

to show that the interpretation of different typing derivations of the same typed term is actually the same. This coherence result is the main technical point of the paper.

In section 2, we begin by describing the syntax of a polymorphic lambda calculus with records, variants, recursive types and subtyping. In particular, we give give type-checking rules in the style of [CW85] for a language we call F++. We indicate how to translate typing derivations of F++ into terms of F+, an extension of the polymorphic lambda calculus with variants, records, and recursive types, and we prove the semantic coherence of this translation. In section 3 we present some directions for further research.

2 Technical results

We describe our calculus F++ which features inheritance in the form of subtyping, records, variants, recursive types, and parametric polymorphism. The type expressions of the language are given by the following abstract syntax:

$$s ::= a \mid \mu a.s \mid s \rightarrow s \mid \{L : S\} \mid [L : S] \mid \forall a.s$$

The inheritance relation. In order to give the rules for deriving inheritance between types in our calculus, we utilize a meta-linguistic notational convention indended to minimize the use of subscripts. Lower case letters a, b, a_1, *etc* from the beginning of the alphabet are used for type variables. An upper case letter A, B, A', *etc* from the beginning of the alphabet will denote a *list* of distinct type variables $a_1, \ldots, a_n$. Appending of lists is denoted by juxtaposition. For example, if A is the list $a_1, \ldots, a_n$ and B is the list $b_1, \ldots, b_m$, then AB is the list $a_1, \ldots, a_n, b_1, \ldots, b_m$. Moreover, we denote the list $a_1, \ldots, a_n, a$ by Aa. If $n = m$ for the lists A and B, we will write $A < B$ for the *set* of subtypings $\{a_1 < b_1, \ldots, a_n < b_n\}$. In particular, $Aa < Bb$ represents the set $\{a_1 < b_1, \ldots, a_n < b_n, a < b\}$. We use s, t, s', *etc* for type expressions and S, T, S', *etc* for lists of type expressions and follow conventions such as those given above for lists of variables.

An *inheritance judgement* is a sequent of the form $A < B \vdash s < t$, where A and B are disjoint. (As it will be apparent from the subtyping rules, the assumptions $a < b$ are used for deriving subtypings between recursive types.) Given lists $S = s_1, \ldots, s_n$ and $T = t_1, \ldots, t_n$, a sequent of the form $A < B \vdash S < T$ abbreviates the set of sequents $A < B \vdash s_i < t_i$ where $1 \leq i \leq n$. The axioms and rules for inferring inheritance judgements are given as follows:

$$A < B \vdash c < c$$

where c does not occur in AB

$$Aa < Bb \vdash a < b$$

$$\frac{Aa < Bb \vdash s < t}{A < B \vdash \mu a.s < \mu b.t}$$

provided a is not free in t and b is not free in s

$$\frac{A < B \vdash s' < s \qquad A < B \vdash t < t'}{A < B \vdash s \rightarrow t < s' \rightarrow t'}$$

$$\frac{A < B \vdash S < T}{A < B \vdash \{LM : SU\} < \{L : T\}}$$

$$\frac{A < B \vdash S < T}{A < B \vdash [L : S] < [LM : TU]}$$

$$\frac{A < B \vdash s < t}{A < B \vdash \forall c.s < \forall c.t}$$

where c does not occur in AB.

Proposition 1 *The transitivity rule*

$$\frac{A < B \vdash r < s \qquad B < C \vdash s < t}{A < C \vdash r < t}$$

is derivable.

Terms Variables will be denoted by lower case letters x, y, x_1, f, g, *etc* and lists of variables by upper case letters X, Y, X', F, *etc.* Given a list $X = x_1, \ldots, x_n$ of variables and a list $S = s_1, \ldots, s_n$ of types, we write $X : S$ for the *set* of typings $\{x_1 : s_1, \ldots, x_n : s_n\}$. Terms are denoted by e, e', *etc* and lists of terms by E, E', *etc.* A *typing judgement* is a sequent $\Gamma \vdash e : s$ where Γ is a pairs of sets $A < B,\ X : S$. A sequent $\Gamma \vdash E : T$ abbreviates a set of such judgements. Terms are given by the following abstract syntax:

$$\begin{array}{rcl} e & ::= & x : s \mid \mathsf{intro}[\mu a.s]e \mid \mathsf{elim}[\mu a.t]e \mid \\ & & \lambda x : s.\ e \mid e(e') \mid \\ & & \{L = E\} \mid e{\downarrow}l \mid \\ & & [l = e] \mid \mathsf{case}\ e\ \mathsf{of}\ L \Rightarrow F \mid \\ & & \Lambda a.\ e \mid e(t) \end{array}$$

The axiom and rules for inferring typing judgements are given as follows:

$$A < B,\ Xx : Ss \vdash x : s$$

$$\frac{A < B,\ X : S \vdash e : s \qquad A < B \vdash s < t}{A < B,\ X : S \vdash e : t}$$

$$\frac{\Gamma \vdash e : [\mu a.s/a]s}{\Gamma \vdash \mathsf{intro}[\mu a.s](e) : \mu a.s}$$

$$\frac{\Gamma \vdash e : \mu a.s}{\Gamma \vdash \mathsf{elim}[\mu a.s](e) : [\mu a.s/a]s}$$

$$\frac{A < B,\ Xx : Ss \vdash e : t}{A < B,\ X : S \vdash \lambda x : s.e : s \to t}$$

$$\frac{\Gamma \vdash e : s \to t \qquad \Gamma \vdash e' : s}{\Gamma \vdash e(e') : t}$$

$$\frac{\Gamma \vdash E : S}{\Gamma \vdash \{L = E\} : \{L : S\}}$$

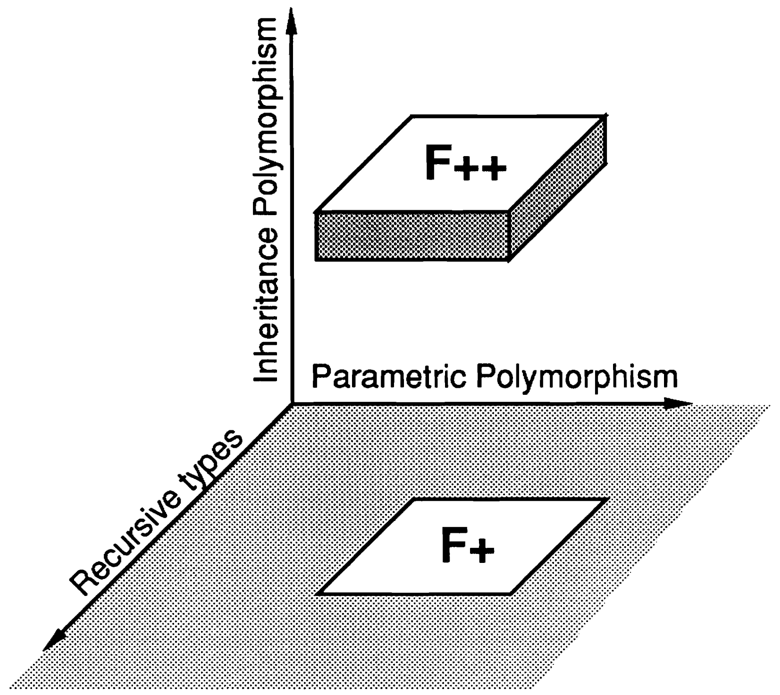

Figure 1: **F++** is projected onto the calculus **F+**.

$$\frac{\Gamma \vdash e : \{Ll : Ss\}}{\Gamma \vdash e{\downarrow}l : s}$$

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash [l = e] : [Ll : Ss]}$$

$$\frac{\Gamma \vdash e : [L : S] \qquad \Gamma \vdash F : S \rightarrow s}{\Gamma \vdash \textsf{case}\ e\ \textsf{of}\ L \Rightarrow F : s}$$

where $F : S \rightarrow s$ abbreviates a set of typings $f_1 : s_1 \rightarrow s,\ \ldots,\ f_n : s_n \rightarrow s$

$$\frac{A < B,\ X : S \vdash e : t}{A < B,\ X : S \vdash \Lambda c.e : \forall c.t}$$

where c does not occur free in AB or S

$$\frac{\Gamma \vdash e : \forall a.s}{\Gamma \vdash e(t) : [t/a]s}$$

The translation. Our paradigm offers an interpretation for F++ via a translation that eliminates the subtyping in favor of definable explicit coercion terms. For the purposes of this abstract, the target of this translation is simply the syntax of the Girard-Reynolds polymorphic lambda calculus, extended with records, variants, and recursive types. We will call this target syntax F+. Figure 1 might be helpful in visualizing the basic idea behind this interpretation.

It would take too much space to write out the full translation here. We will give details in the full paper. [BCGS89] already discusses many cases not mentioned here. We focus only on the subtyping rule for recursive types. As we mentioned in the introduction, this represents the main new feature that we treat here:

$$\frac{Aa < Bb \vdash s < t}{A < B \vdash \mu a.s < \mu b.t}$$

is translated using a *recursively defined* coercion:

$$\frac{Ff : Aa \to Bb \vdash P : s \to t}{F : A \to B \vdash (\mu g : (\mu a.s \to \mu b.t).\ \lambda x : (\mu a.s).\ \mathsf{intro}[\mu b.t](Q(\mathsf{elim}[\mu a.s](x)))) : \mu a.s \to \mu b.t}$$

where Q is the term $([\mu a.s/a,\ \mu b.t/b](\lambda f : a \to b.P))(g)$ For a concrete example, the reader can check that this describes exactly the coercion derived for the ML types discussed in section 1.

We can now state precisely what our coherence result says: for each of a number of domain-theoretic models of F+, the translations of any two F++ typing derivations that yield the same judgement have, as F+ terms, the same denotation. To be specific, we state the theorem for one such model, namely Scott domains and continuous maps [CGW].

Recall that a continuous map is *strict* iff it preserves bottom. Because variants are interpreted as separated sums, and because the coherence property requires certain identities [BCGS89] about these sums which hold only for strict maps, we need to make sure that coercions are always interpreted as strict maps:

Lemma 2 *Let $F : A \to B \vdash P : s \to t$ be the translation in F+ of an F++ inheritance judgement. Then, whenever F is interpreted by strict maps, the meaning of P is also a strict map.*

Theorem 3 (Coherence for Typing Judgements) *The translations of any two F++ typing derivations that yield the same judgement have, as F+ terms, the same denotation.*

Almost any of the known interpretations of the polymorphic lambda calculus with recursive types will also suffice. For example, algebraic lattices with continuous functions and the usual domain-theoretic interpretation of recursive types have the desired property [TT87, HP87]. Universal domains [CGW, ABL86] and approaches using stable functions [Gir87, CGW87] also validate the theorem (in the latter case, coercions are linear maps). It would take too much space to state abstractly the precise properties required of such models in general. We will postpone this to a fuller version of this paper. However, we must mention that the key point is to use continuous (respectively stable) interpretations of function types and to interpret recursive definitions as least fixed points.

3 Directions for Further Investigations

So far, both in [BCGS89] and in the present paper, the technical results have been concerned only with denotational aspects. We believe, however, that our inheritance-as-implicit-coercion paradigm is a full-fledged semantic paradigm. We intend to substantiate this claim through an analysis of the connection between our interpretation and operational semantics.

Moreover, we hope that this paradigm will also yield in a natural fashion proof principles for verifying program equivalences for the kind of languages we interpret.

The fact that in [BCGS89] and in the present paper we were able to apply the inheritance-as-implicit-coercion paradigm to a variety of type constructs is strong evidence for its robustness. Therefore, we expect that these ideas will naturally extend to other language design efforts in progress, such as [CM88].

Acknowledgements.

We would like to acknowledge many useful discussions with Thierry Coquand.

Breazu-Tannen's research was partially supported by ONR Grant NOOO14-88-K-0634. Gunter's research was supported by ONR Grant NOOO14-88-K-0557. Scedrov's research was partially supported by NSF Grant CCR-87-05596, by ONR Grant NOOO14-88-K-0635, and by the 1987 Young Scientist Award from the Natural Sciences Association of the University of Pennsylvania.

References

[ABL86] R. Amadio, K. B. Bruce, and G. Longo. The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In A. Meyer, editor, *Logic in Computer Science*, pages 122–130, IEEE Computer Society Press, 1986.

[BCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explict coercion (preliminary report). In R. Parikh, editor, *Logic in Computer Science*, IEEE Computer Society, June 1989.

[Car] L. Cardelli. A Quest preview. Preprint.

[Car86] L. Cardelli. Amber. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47, *Lecture Notes in Computer Science*, Vol. 242, Springer-Verlag, 1986.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[CGW] T. Coquand, C. A. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation.* To appear.

[CGW87] T. Coquand, C. A. Gunter, and Glynn Winskel. Di-domains as a model of polymorphism. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, pages 344–363, *Lecture Notes in Computer Science vol. 298,* Springer, April 1987.

[CM88] L. Cardelli and J. C. Mitchell. Semantic methods for object-oriented languages. Unpublished lecture notes for a tutorial given at the Conference on Object-Oriented Programming Systems, Languages, and Applications. September 1988.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17, 1985.

[Gir87] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, Reading, MA, 1983.

[HP87] J. M . E. Hyland and A. Pitts. The theory of constructions: categorical semantics and topos-theoretic models. In *Proceedings of the AMS Conference on Categories in Computer Science and Logic, Boulder, June 1987*, 1987. To appear in Contemporary Mathematics.

[JM88] L. Jategaonkar and J. C. Mitchell. Ml with extended pattern matching and subtypes. In R. Cartwright, editor, *Symposium on LISP and Functional Programming*, pages 198–211, ACM, 1988.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In R. Cartwright, editor, *Symposium on LISP and Functional Programming*, pages 174–183, ACM, 1988.

[Rem89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Symposium on Principles of Programming Languages*, pages 77–88, ACM, 1989.

[TT87] T. Coquand and T. Ehrhard. An equational presentation of higher-order logic. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, pages 40–56, *Lecture Notes in Computer Science vol. 283,* Springer, 1987.

[Wan87] M. Wand. Complete type inference for simple objects. In D. Gries, editor, *Symposium on Logic in Computer Science*, pages 37–46, IEEE Computer Society Press, Ithaca, New York, June 1987.

[Wan89] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the Symposium on Logic in Computer Science*, IEEE, June 1989. To appear.