January 1995

Proof Theoretic Concepts for the Semantics of Types and Concurrency

Carl A. Gunter
University of Pennsylvania

Val Tannen
University of Pennsylvania, val@cis.upenn.edu

Thierry Coquand
INRIA

Andre Scedrov
University of Pennsylvania, scedrov@math.upenn.edu

Recommended Citation

Carl A. Gunter, Val Tannen, Thierry Coquand, and Andre Scedrov, "Proof Theoretic Concepts for the Semantics of Types and Concurrency", . January 1995.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-19.

Proof Theoretic Concepts for the Semantics of Types and Concurrency

Abstract

We present a method for providing semantic interpretations for languages with a type system featuring *inheritance* polymorphism. Our approach is illustrated on an extension of the language Fun of Cardelli and Wegner, which we interpret via a translation into an extended polymorphic lambda calculus. Our goal is to interpret inheritances in Fun via *coercion functions* which are definable in the target of the translation. Existing techniques in the theory of semantic domains can be then used to interpret the extended polymorphic lambda calculus, thus providing many models for the original language. This technique makes it possible to model a rich type discipline which includes parametric polymorphism and recursive types as well as inheritance.

A central difficulty in providing interpretations for explicit type disciplines featuring inheritance in the sense discussed in this paper arises from the fact that programs can type-check in more than one way. Since interpretations follow the type-checking derivations, *coherence* theorems are required: that is, one must prove that the meaning of a program does not depend on the way it was type-checked. The proof of such theorems for our proposed interpretation are the basic technical results of this paper. Interestingly, proving coherence in the presence of recursive types, variants, and abstract types forced us to reexamine fundamental equational properties that arise in proof theory (in the form of commutative reductions) and domain theory (in the form of strict *vs.* non-strict functions).

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-19.

Proof Theoretic Concepts for the Semantics of Types and Concurrency

MS-CIS-95-19
LOGIC & COMPUTATION LAB 90

Edited by Carl Gunter

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

April 1995

INHERITANCE AS IMPLICIT COERCION[1]

Val Breazu-Tannen *Thierry Coquand* *Carl A. Gunter* *Andre Scedrov*[2]

Abstract. We present a method for providing semantic interpretations for languages with a type system featuring *inheritance* polymorphism. Our approach is illustrated on an extension of the language Fun of Cardelli and Wegner, which we interpret via a translation into an extended polymorphic lambda calculus. Our goal is to interpret inheritances in Fun via *coercion functions* which are definable in the target of the translation. Existing techniques in the theory of semantic domains can be then used to interpret the extended polymorphic lambda calculus, thus providing many models for the original language. This technique makes it possible to model a rich type discipline which includes parametric polymorphism and recursive types as well as inheritance.

A central difficulty in providing interpretations for explicit type disciplines featuring inheritance in the sense discussed in this paper arises from the fact that programs can type-check in more than one way. Since interpretations follow the type-checking derivations, *coherence* theorems are required: that is, one must prove that the meaning of a program does not depend on the way it was type-checked. The proof of such theorems for our proposed interpretation are the basic technical results of this paper. Interestingly, proving coherence in the presence of recursive types, variants, and abstract types forced us to reexamine fundamental equational properties that arise in proof theory (in the form of commutative reductions) and domain theory (in the form of strict *vs.* non-strict functions).

1 Introduction

In this paper we will discuss an approach to the semantics of a particular form of inheritance which has been promoted by John Reynolds and Luca Cardelli. This inheritance system is based on the idea that one may axiomatize a relation $\leq$ between type expressions in such a way that whenever the *inheritance judgement* $s \leq t$ is provable for type expressions s and t, then an expression of type s can be "considered as" an expression of type t. This property is expressed by the *inheritance* rule (sometimes also called the *subsumption* rule), which states that if an expression e is of type s and $s \leq t$, then e also has type t. The consequences from a semantic point of view of the inclusion of this form of typing rule are significant. It is our goal in this paper to look carefully at what we consider to be a robust and intuitive approach to systems which have this form of inheritance and examine in some detail the semantic implications of the inclusion of inheritance judgements and the inheritance rule in a type discipline.

Several attempts have been made recently to express some of the distinctive features of object-oriented programming, principally *inheritance*, in the framework of a rich type discipline which can accommodate strong static type-checking. This endeavor searches for a language that offers some of the flexibility of object-oriented programming [GR83] while maintaining the reliability, and sometimes increased efficiency of programs which type-check at compile-time (see [BBG88] for a related comparison).

[1] Appears in **Information and Computation** vol. 93 (1991), pp. 172–221.

[2] Author's addresses. Breazu-Tannen and Gunter: Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia PA 19104, USA. Coquand: INRIA, Domaine de Voluceau, 78150 Rocquencourt, France. Scedrov: Department of Mathematics, University of Pennsylvania, Philadelphia PA 19104, USA.

of the translation. We hope that the results in this simpler setting will help the reader get an idea of what our program is before we proceed to a more interesting calculus in the remainder of the paper. The fourth section is devoted to developing a translation for an expanded calculus which adds variants. Fundamental equational properties of variants lead us to develop a target language which has a *type of coercions.* The fifth section, which contains the difficult technical results of the paper, shows that our translation is coherent. In the sixth section we discuss mathematical models for the full calculus. Since most of the work has already been done, we are able to produce many models using standard domain-theoretic techniques. The concluding section makes some remarks about what we feel has been achieved and what new challenges still need to be confronted.

2 Inheritance as implicit coercion.

A simple analogy will help explain our translation-based technique. Consider how the ordinary *untyped* λ-calculus is interpreted semantically in such sources as [Sco80, Mey82, Koy82, Bar84]. One begins by postulating the existence of a semantic domain D and a pair of arrows $\Phi\colon D \to (D \to D)$ and $\Psi\colon (D \to D) \to D$ such that $\Phi \circ \Psi$ is the identity on $D \to D$. Certain conditions are required of $D \to D$ to insure that "enough" functions are present. To interpret an untyped λ-term, one defines a translation $M \mapsto M^*$ on terms which takes an untyped term M and creates a typed term M^*. This operation is defined by induction:

- for a variable, $x^* \equiv x\colon D$,
- for an application, $M(N)^* \equiv \Phi(M^*)(N^*)$ and,
- for an abstraction, $(\lambda x.\ M)^* \equiv \Psi(\lambda x\colon D.\ M^*)$

(where we use $\equiv$ for syntactic equality of expressions). For example, the familiar term

$$\lambda f.\ (\lambda x.\ f(xx))(\lambda x.\ f(xx))$$

translates to

$$\Psi(\lambda f\colon D.\ \Phi(\Psi(\lambda x\colon D.\ \Phi(f)(\Phi(x)(x))))(\Psi(\lambda x\colon D.\ \Phi(f)(\Phi(x)(x))))).$$

The fact that the latter term is unreadable is perhaps an indication of why we use the former term *in which the semantic coercions are implicit.* Nevertheless, this translation provides us with the desired semantics for the untyped term since we have converted that term into a term in a calculus which we know how to interpret. Of course, this assumes that we really do know how to provide a semantics for the typed calculus supplemented with triples such as D, Φ, Ψ. Moreover, there are some equations we must check to show that the translation is sound. But, at the end of the day, we have a simple, intuitive explanation of the interpretation of untyped λ-terms based on our understanding of a certain simply typed λ-theory. In this paper we show how a similar technique may be used to provide an intuitive interpretation for inheritance, even in the presence of parametric polymorphism and type recursion. As mentioned earlier, our interpretation is carried out by translating the full calculus into a calculus without inheritance (the *target* calculus) whose semantics we already understand. However, our idea differs significantly from the interpretation of the untyped λ-calculus as described above in at least one important respect: typically, the coercions (such as Φ and Ψ above) which we introduce will be *definable* in the target calculus. Hence our target calculus needs to be an extension of the ordinary polymorphic λ-calculus with records, variants, abstract types, and recursive types. But it need not have any inheritance.

Although we believe that the translation just illustrated is intuitive, we need to show that it is *coherent.* In other words, we must show that the semantic function is well defined. The need for coherence comes from the fact that a typing judgement may have many different derivations. In general, it is customary to present the semantics of typed lambda calculi as a map defined inductively on type-checking derivations. Such a method would therefore assign a meaning to each derivation tree. We do believe though, that the *language* consists of the derivable typing judgements, rather than of the derivation trees. For many calculi, such as the simply typed or the polymorphic lambda calculus, there is at most one derivation for any typing judgement. Therefore, in such calculi, giving meaning to derivations is the same as giving meaning to derivable judgements. But for other calculi, such as Martin-Löf's Intuitionistic Type Theory (ITT) [Mar84] (see [Sal88]), and the Calculus of Constructions [CH88] (see [Str88]), and—of immediate concern to us—Cardelli and Wegner's Fun, this is not so, and one must prove that derivations yielding the same judgement are given the same meaning. This idea has also appeared in the context of category theory and our use of the term "coherence" is partially inspired by its use there, where it means the uniqueness of certain canonical morphisms (see *e.g.* [KL71] and [LP85]). Although we have not attempted a rigorous connection in this paper, the possibility of unifying coherence results for a variety of different calculi offers an interesting direction of investigation. In the case of Fun, we show the coherence of our semantic approach by proving that *translations of any two derivations of the same typing judgement are equated in the target calculus.*

Hence, the coherence of a given translation is a property of the equational theory of the target calculus. When the target calculus is the polymorphic lambda calculus extended with records and recursive types, the standard axiomatization of its equational theory is sufficient for the coherence theorem. But when we add variants, the standard axiomatization of these features, while sufficient for coherence, clashes with the standard axiomatization of recursive types, yielding an inconsistent theory (see [Law69, HP89a] for variants, that is, coproducts). The solution lies in two observations: (1) the (too) strong axioms are only needed for "coercion terms", and (2) in the various models we examined these coercion terms have special interpretations (such as *strict*, or *linear* maps), so special in fact, that they satisfy the corresponding restrictions of the strong axioms! Correspondingly, one has to restrict the domains over which "coercion variables" can range, which leads naturally to the type of coercions mentioned above.

3 Translation for a fragment of the calculus

For pedagogical reasons, we begin by considering a language whose type structure features function spaces (exponentials), record types, bounded generic types (an inheritance-generalized form of universal polymorphism), recursive types, and, of course, inheritance. In the next section we will enrich this calculus by the addition of variants. As we have mentioned before, this leads to some (interesting) complications which we avoid by restricting ourselves to the simpler calculus of this section. Since the calculus in the next section is stronger, we omit details for the proofs of results in this section. They resemble the proofs for the calculus with variants, but the calculations are simpler. Rather than generate four different names for the calculi which we shall consider in this section and the next we simply refer to the calculus with inheritance as **SOURCE** and the inheritance-free calculus into which it is translated as **TARGET**. The fragment of the calculus which we consider in this section is fully described in the appendices to the paper.

We provide semantics to **SOURCE** via a *translation* into a language for which several well-understood semantics already exist. This "target" language, which we shall call **TARGET**, is an extension with record and recursive types of the Girard-Reynolds polymorphic lambda calculus

Appendix A under the heading **Fragment.**

Among these proof rules, the following two illustrate the effect of inheritance on type-checking:

[INH] $$\frac{\Gamma \vdash e : s \qquad \widehat{\Gamma} \vdash s \leq t}{\Gamma \vdash e : t}$$

[B-SPEC] $$\frac{\Gamma \vdash e : \forall a \leq s.\, t \qquad \widehat{\Gamma} \vdash r \leq s}{\Gamma \vdash e(r) : [r/a]t}$$

They make use of *inheritance judgements* which have the form $C \vdash s \leq t$ where C is an inheritance context. *Inheritance contexts* are contexts in which only declarations of the form $a \leq t$ appear. If Γ is a context, we denote-by $\widehat{\Gamma}$ teh inheritance context obtained from Γ by erasing the declarations of the form $x : t$. The proof system for deriving inheritance judgments is, with the exception of one rule, the same as the relevant fragment of the corresponding proof system for Fun (see [CW85], on page 519). In this paper we do not attempt to enrich it with any rule deriving inheritances *between* recursive types. A discussion of this issue appears in our conclusions. The Appendix contains a complete list of these proof rules too.

In comparison with Fun, we would like to strengthen the rule deriving inheritances between bounded generics, and we are able to do so for some of our results. Where Fun had just

(W-FORALL) $$\frac{C, a \leq t \vdash u \leq v}{C \vdash \forall a \leq t.\, u \leq \forall a \leq t.\, v}$$

we will consider

(FORALL) $$\frac{C \vdash s \leq t \qquad C, a \leq s \vdash u \leq v}{C \vdash \forall a \leq t.\, u \leq \forall a \leq s.\, v}$$

This makes the system strictly stronger, allowing more inheritances to be derived, and thus more terms to type-check.

Originally, we believed that coherence could be proved for a system that includes variants and the stronger rule (FORALL) [BCGS89]. In dealing with the *case* construct for variant types, however, our coherence proof uses an order-theoretic property (see Lemma 11) which fails for the stronger system for deriving inheritances that uses (FORALL) (for a counterexample, see Giorgio Gelli's dissertation [Ghe90]). Thus, we prove the coherence of the translation of variants (Theorem 13) only for the weaker system with (W-FORALL). Note, however, that we prove coherence in the presence of (FORALL) for the system without variants (Theorem 4) and for the system for deriving inheritances between types, including variant types (Lemma 9).

Remark. Decidability of type-checking in the stronger system is a non-trivial question. The question whether an algorithm of Luca Cardelli will decide the provability of judgements in this calculus has only recently been settled by Ghelli [Ghe90].

The salient feature of bringing inheritance into a type system is that (in given contexts) terms will *not* have a unique type any more. For example, due to the rule

(TOP) $$C \vdash t \leq Top$$

where the free variables of t are declared in C, by [INH], all terms that type-check with some type will also type-check with type *Top*. This makes it possible to define ordinary generics as syntactic sugar: $\forall a.\, t \overset{\text{def}}{=} \forall a \leq Top.\, t$.

where $n \geq 1$, and

{RECD-ETA}. $$\{l_1 = M.l_1, \ldots, l_n = M.l_n\} \; = \; M$$

where $M \, : \, \{l_1 \colon s_1, \ldots, l_n \colon s_n\}$ .The last rule gives, for $n = 0$, the equation $\{\} \; = \; M$ which makes 1 into a terminator. Under our interpretation, the type *Top* will be nothing like a "universal domain" which can be used to interpret *Type:Type* [CGW89, GJ90]. On the contrary, it will be interpreted as a one point domain in the models we list below!

The translation. For any **SOURCE** `item` we will denote by `item`* its translation into **TARGET**. We begin with the types. Note the translation of bounded generics and of *Top*.

$$\begin{array}{rclrcl} a^* & \overset{\text{def}}{=} & a & \{l_1 \colon s_1, \ldots, l_n \colon s_n\}^* & \overset{\text{def}}{=} & \{l_1 \colon s_1^*, \ldots, l_n \colon s_n^*\} \\ Top^* & \overset{\text{def}}{=} & \mathbf{1} & (\forall a \leq s.\, t)^* & \overset{\text{def}}{=} & \forall a.\, (a \to s^*) \to t^* \\ (s \to t)^* & \overset{\text{def}}{=} & s^* \to t^* & (\mu a.\, t)^* & \overset{\text{def}}{=} & \mu a.\, t^* \end{array}$$

One shows immediately that $([s/a]t)^* \equiv [s^*/a]t^*$. We extend this to contexts and inheritance contexts, which translate into just typing contexts in **TARGET**.

$$\begin{array}{rclrcl} \emptyset^* & \overset{\text{def}}{=} & \emptyset & \emptyset^* & \overset{\text{def}}{=} & \emptyset \\ (\Gamma,\, a \leq t)^* & \overset{\text{def}}{=} & \Gamma^*,\, a,\, f \colon a \to t^* & (C,\, a \leq t)^* & \overset{\text{def}}{=} & C^*,\, a,\, f \colon a \to t^* \\ & & & (\Gamma,\, x \colon t)^* & \overset{\text{def}}{=} & \Gamma^*,\, x \colon t^* \end{array}$$

where f is a *fresh* variable for each a.

Next we will describe how we translate the derivations of judgments of **SOURCE**. The translation is defined by recursion on the structure of the derivation trees. Since these are freely generated by the derivation rules, it is sufficient to provide for each derivation rule of **SOURCE** a corresponding rule on trees of **TARGET** judgments. It will be a lemma (Lemma 2 to be precise) that these corresponding rules are *directly derivable* in **TARGET**, therefore the translation takes derivations in **SOURCE** into derivations in **TARGET**.

A **SOURCE** derivation yielding an inheritance judgment $C \, \vdash \, s \, \leq \, t$ is translated as a tree of **TARGET** judgments yielding $C^* \, \vdash \, P \, : \, s^* \to t^*$. We present three of the rules here; the full list for the fragment appears in Appendix C. The coercion into *Top* is simply the constant map:

(TOP)* $$C^* \, \vdash \, \lambda x \colon t^*.\, \{\} \, : \, t^* \to \mathbf{1}$$

To see how coercion works on types, assume that we are given a coercion $P \colon s \to t$ from s into t and a coercion $Q \colon u \to v$ from u into v. Then it is possible to coerce a function $f \colon t \to u$ into a function from s to v as follows. Given an argument of type s, coerce it (using P) into an argument of type t. Apply the function f to get a value of type u. Now coerce this value in u into a value in v by applying Q. This describes a function of the desired type. More formally, we translate the (ARROW) rule by

(ARROW)* $$\frac{C^* \, \vdash \, P \, : \, s^* \to t^* \qquad C^* \, \vdash \, Q \, : \, u^* \to v^*}{C^* \, \vdash \, R \, : \, (t^* \to u^*) \to (s^* \to v^*)}$$

where $R \overset{\text{def}}{=} \lambda z \colon t^* \to u^*.\, P; z; Q$. (We use ; as shorthand for *composition*. For example, $P; z; Q$ above stands for $\lambda x \colon s^*.\, Q(z(P(x)))$ where x is fresh.) Now, to translate the rule (FORALL)

4 Between incoherence and inconsistency: adding variants

The calculus described so far does not deal with a crucial type constructor: variants. In particular, it is very useful to have a combination of variant types with recursive types. On the other hand, the combination of these operators in the same calculus is also problematic, especially for the equational theory. The situation is familiar from both domain theory and proof theory. In this section we propose an approach which will suffice to prove the coherence theorem which we need to show that our semantic function is well-defined.

We extend the type formation rules of **SOURCE** by adding *variant* type expressions: $[l_1\colon t_1, \ldots, l_n\colon t_n]$ where $n \geq 1$. We also extend the term formation rule by the formation of variant terms $[l_1\colon t_1, \ldots, l_i = e, \ldots, l_n\colon t_n]$ and the *case statement:*

$$\textsf{case } e \textsf{ of } l_1 \Rightarrow f_1, \ldots, l_n \Rightarrow f_n$$

The inheritance judgement derivation rules are extended correspondingly with the rule:

(VART) $$\frac{C \vdash s_1 \leq t_1 \quad \cdots \quad C \vdash s_p \leq t_p}{C \vdash [l_1\colon s_1, \ldots, l_p\colon s_p] \leq [l_1\colon t_1, \ldots, l_p\colon t_p, \ldots, l_q\colon t_q]}$$

Note the "duality" between this rule and the inheritance rule (RECD) for records (see Appendix A). While a record subtype has more fields, a variant subtype has fewer variations (summands).

Like before, we intend to translate this calculus into a calculus without inheritance and, naturally, we extend **TARGET** with variants (see Appendix B). Note how the syntax of variant injections differs from [CW85]. This is in order for the resulting system to enjoy the property of having unique type derivations: the proof of Proposition 1 extends immediately to the variant constructs. Most importantly, we must extend the equational theory of **TARGET** in a manner that insures the coherence of our translation. It is here that we encounter an interesting problem which readers who know domain theory will find familiar. The following two axioms hold in a variety of models:

{VART-BETA} $$\textsf{case } \textsf{inj}_{l_i}(M_i) \textsf{ of } l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n \;=\; F_i(M_i)$$

where $F_1 : t_1 \to t, \ldots, F_n : t_n \to t$, $M_i : t_i$ and $\textsf{inj}_{l_i}$ is shorthand for $\lambda x\colon t_i.\, [l_1\colon t_1, \ldots, l_i = x, \ldots, l_n\colon t_n]$.

{VART-ETA} $$\textsf{case } M \textsf{ of } l_1 \Rightarrow \textsf{inj}_{l_1}, \ldots, l_n \Rightarrow \textsf{inj}_{l_n} \;=\; M$$

where $M : [l_1\colon t_1, \ldots, l_n\colon t_n]$. Unfortunately, these two axioms do not suffice to prove all the identifications required by the coherence of our translation!

To see the problem, we start with an example. In **SOURCE**, suppose that $t \leq s$ is derivable in the context $\widehat{\Gamma}$, and that we have a derivation Δ of $\Gamma \vdash e : [l_1\colon t_1, l_2\colon t_2]$ and derivations Δ_i of $\Gamma \vdash f_i : t_i \to t$, $i = 1, 2$. Consider then the following two **SOURCE** derivations of the typing judgement $\Gamma \vdash \textsf{case } e \textsf{ of } l_1 \Rightarrow f_1, l_2 \Rightarrow f_2 : s$.

1. by Δ, Δ_1, Δ_2 and the rule [CASE], one deduces $\Gamma \vdash \textsf{case } e \textsf{ of } l_1 \Rightarrow f_1, l_2 \Rightarrow f_2 : t$. Since $\widehat{\Gamma} \vdash t \leq s$ by hypothesis, one infers by inheritance $\Gamma \vdash \textsf{case } e \textsf{ of } l_1 \Rightarrow f_1, l_2 \Rightarrow f_2 : s$.

2. from $\widehat{\Gamma} \vdash t \leq s$ we can deduce $\widehat{\Gamma} \vdash (t_i \to t) \leq (t_i \to s)$. Hence, by inheritance from Δ_i, one deduces $\Gamma \vdash f_i : t_i \to s$. Then, from Δ and by the rule [CASE], one deduces $\Gamma \vdash \textsf{case } e \textsf{ of } l_1 \Rightarrow f_1, l_2 \Rightarrow f_2 : s$.

translation. Thus, the previous discussion of variants leads us to introduce a new type constructor $s \circ\!\!\to t$, the type of "coercions" from s to t. Consequently, the coercion assumptions $a \leq t$ that occur in inheritance contexts must translate to variables ranging over types of coercions $f\colon a \circ\!\!\to t^*$. As a consequence, the translation of bounded quantification must change:

$$(\forall a \leq s.\, t)^* \overset{\text{def}}{=} \forall a.\, ((a \circ\!\!\to s^*) \to t^*)$$

In order to express the correct versions of {VART-CRN}, we introduce a family of constants in **TARGET**

$$\iota_{s,t} : (s \circ\!\!\to t) \to (s \to t)$$

called *coercion-coercion combinators.* With this, we have

$$\{\text{VART-CRN}\} \quad \iota(P)(\mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) \;=\; \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1; \iota(P), \ldots, l_n \Rightarrow F_n; \iota(P)$$

where $M\colon [l_1\colon t_1, \ldots, l_n\colon t_n],\ F_1\colon t_1 \to t, \ldots, F_n\colon t_n \to t,\ P\colon t \circ\!\!\to s$.

(the complete list is in Appendix B).

In order to translate all inheritance judgements into coercion terms, we add a special set of constants (coercion combinators) that "compute" the translations of the rules for deriving inheritance judgements. To prove coherence, we axiomatize the behavior of the ι-images of these combinators. For example, the coercion combinator for the rule (ARROW) takes a pair of coercions as arguments and yields a new coercion as value:

$$\mathsf{arrow}[s,t,u,v] : (s \circ\!\!\to t) \to (u \circ\!\!\to v) \to ((t \to u) \circ\!\!\to (s \to v))$$

Since (ARROW) is a rule *scheme*, we naturally have a *family* of such combinators, indexed by types. To simplify the notation, these types will be omitted whenever possible. The equational property of the arrow combinator is given in terms of the coercion coercer:

$$\iota(\mathsf{arrow}(P)(Q)) \;=\; \lambda z\colon t \to u.\, (\iota(P)); z; (\iota(Q))$$

where $P\colon s \circ\!\!\to t,\ Q\colon u \circ\!\!\to v$. For the rule (TRANS), we introduce

$$\mathsf{trans}[r,s,t] : (r \circ\!\!\to s) \to (s \circ\!\!\to t) \to (r \circ\!\!\to t)$$

which, of course, behaves like composition, modulo the coercion coercer:

$$\iota(\mathsf{trans}(P)(Q)) \;=\; \iota(P); \iota(Q)$$

where $P\colon r \circ\!\!\to s,\ Q\colon s \circ\!\!\to t$. The combinator for the rule (FORALL) is the most involved:

$$\mathsf{forall}[s,t,a,u,v] : (s \circ\!\!\to t) \to \forall a.\, ((a \circ\!\!\to s) \to (u \circ\!\!\to v)) \to (\forall a.\, ((a \circ\!\!\to t) \to u) \circ\!\!\to \forall a.\, ((a \circ\!\!\to s) \to v))$$

with the equational axiomatization

$$\iota(\mathsf{forall}(P)(W)) \;=\; \lambda z\colon (\forall a.\, (a \circ\!\!\to t) \to u).\, \Lambda a.\, \lambda f\colon a \circ\!\!\to s.\, \iota(W(a)(f))(z(a)(\mathsf{trans}(f)(P)))$$

where $P\colon s \circ\!\!\to t,\ W\colon \forall a.\, (a \circ\!\!\to s) \to (u \circ\!\!\to v)$. Of course, we have gone to the extra inconvenience of introducing the type of coercions in order to provide a satisfactory account of variants. These require a scheme of combinators having the types:

$$\mathsf{vart}[s_1, \ldots, s_p, t_1, \ldots, t_q] : (s_1 \circ\!\!\to t_1) \to \cdots \to (s_p \circ\!\!\to t_p) \to ([l_1\colon s_1, \ldots, l_p\colon s_p] \circ\!\!\to [l_1\colon t_1, \ldots, l_p\colon t_p, \ldots, l_q\colon t_q])$$

5 Coherence of the translation for the full calculus

In this section we prove first the coherence of the translation of inheritance judgements. This result is then used to show the coherence of the translation of typing judgements.

The main cause for having distinct derivations of the same inheritance judgements is the rule (TRANS). Our strategy is to show that the usage of (TRANS) can be coherently postponed to the end of derivations (Lemma 6), and then to prove the coherence of the translation of (TRANS)-postponed derivations (Lemma 8).

We introduce some convenient notations for the rest of this section. For any derivation Δ in **SOURCE**, let Δ^* be the **TARGET** derivation into which it is translated. We will write $C \vdash r_0 \leq \cdots \leq r_n$ instead of $C \vdash r_0 \leq r_1 ,\ldots, C \vdash r_{n-1} \leq r_n$. The composition of coercions given by trans occurs so often that we will write $P \odot Q$ instead of $\mathsf{trans}(P)(Q)$. It is easy to see, making essential use of the rule {IOTA-INJ}, that $\odot$ is provably *associative.* We will take advantage of this to unclutter the notation. We will also write I instead of refl . Again it is easy to see that I is provably an identity for $\odot$, that is, $I \odot M = M \odot I = M$ is provable in **TARGET**.

Lemma 6 *For any* **SOURCE** *derivation* Δ *yielding the inheritance judgement* $C \vdash s \leq t$ *, there exist types* $r_0, \ldots, r_n$ *such that* $s \equiv r_0$ *,* $r_n \equiv t$*, and (TRANS)-free derivations* $\Delta_1, \ldots, \Delta_n$ *yielding respectively*

$$C \vdash r_0 \leq \cdots \leq r_n$$

Moreover, if the translations $\Delta^*, \Delta_1^*, \ldots, \Delta_n^*$ *yield respectively the (coercion) terms* $C^* \vdash P : s^* \multimap t^*$ *,* $C^* \vdash P_1 : r_0^* \multimap r_1^* ,\ldots,$ $C^* \vdash P_n : r_{n-1}^* \multimap r_n^*$ *then*

$$C^* \vdash P = P_1 \odot \cdots \odot P_n$$

is provable in **TARGET**.

Proof: By induction on the height of the derivation Δ. The base is trivial since derivations consisting of instances of (TOP), (VAR), or (REFL) are already (TRANS)-free. We present the more interesting cases of the induction step.

Suppose Δ ends with an application of (ARROW). By induction hypothesis there are (TRANS)-free derivations for

$$s \equiv r_0 \leq \cdots \leq r_m \equiv t \quad \text{and} \quad u \equiv w_0 \leq \cdots \leq w_n \equiv v$$

(for simplicity, we omit the context). From these, using (REFL) and (ARROW) we get (TRANS)-free derivations for

$$t \to u \equiv r_m \to u \leq \cdots \leq r_0 \to u \equiv s \to w_0 \leq \cdots \leq s \to w_n \equiv s \to v \ .$$

(This is not most economical: one can get a derivation requiring only $\max(m,n)$, rather than $m+n$, steps of (TRANS) at the end.) Proving the equality of the corresponding translations uses the associativity of $\odot$ and the fact that I acts like an identity, as well as

$$(1) \qquad \mathsf{arrow}(P)(Q) \odot \mathsf{arrow}(R)(S) = \mathsf{arrow}(R \odot P)(Q \odot S)$$

which can be verified, in view of {IOTA-INJ}, by applying ι to both sides, resulting in a simple {BETA}-conversion.

Proof: By induction on the height of Θ. ▮

Lemma 8 *Let $\Delta_1, \ldots, \Delta_m$ be (TRANS)-free derivations in* **SOURCE** *yielding respectively* $C \vdash s_0 \le \cdots \le s_m$ *and $\Theta_1, \ldots, \Theta_n$ be (TRANS)-free derivations yielding respectively* $C \vdash t_0 \le \cdots \le t_n$. *Let the translations $\Delta_1^*, \ldots, \Delta_m^*, \Theta_1^*, \ldots, \Theta_n^*$ yield respectively the (coercion) terms*

$$C^* \vdash P_1 : s_0^* \multimap s_1^* , \ldots, C^* \vdash P_m : s_{m-1}^* \multimap s_m^* , C^* \vdash Q_1 : t_0^* \multimap t_1^* , \ldots, C^* \vdash Q_n : t_{n-1}^* \multimap t_n^* .$$

If $s_0 \equiv t_0$ and $s_m \equiv t_n$ then

$$C^* \vdash P_1 \odot \cdots \odot P_m = Q_1 \odot \cdots \odot Q_n$$

is provable in **TARGET**.

Proof: We begin with the following remarks:

- If one of $s_0, \ldots, s_m, t_0, \ldots, t_n$ is *Top* then the desired equality holds. Indeed, then $s_m \equiv Top \equiv t_n$ and the equality follows from the identity

  $$(4) \qquad P \le \mathsf{top}$$

  which is verified by applying ι to both sides (recall that **1** is a terminator).

- Those derivations among $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$ which consist entirely of one application of (REFL) can be eliminated without loss of generality. Indeed, the corresponding coercion term is I which acts as an identity for $\odot$.

- If none of the derivations among $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$ consists of just (TOP), then those derivations which consist of just (VAR) can also be eliminated without loss of generality. Indeed, once we have eliminated the (REFL)'s, the (VAR)'s must form an initial segment of both $\Delta_1, \ldots, \Delta_m$ and $\Theta_1, \ldots, \Theta_n$ because whenever $s \le a$ is derivable, s must also be a type variable. Let's say that $s_0 \equiv a_0, \ldots, s_p \equiv a_{p-1}$, $(p \le m)$, where $\Delta_1, \ldots, \Delta_p$ are *all* the derivations consisting of just (VAR), and also that $t_0 \equiv b_0, \ldots, t_q \equiv b_{q-1}$, $(q \le n)$, where $\Theta_1, \ldots, \Theta_q$ are all the derivations consisting of just of (VAR). Then, $a_0 \le a_1, \ldots, a_{p-1} \le s_p$ as well as $b_0 \le b_1, \ldots, b_{q-1} \le t_q$ must all occur in C. But $a_0 \equiv s_0 \equiv t_0 \equiv b_0$ so by the uniqueness of declarations in contexts, $a_1 \equiv b_1, \ldots$, *etc.* Suppose $p < q$. Then, $s_p \equiv b_p$ is a variable. Since Δ_{p+1} can't be just a (REFL) or a (TOP) is must be a (VAR) contradicting the maximality of p. Thus $p = q$ and $s_p \equiv t_q$ and the (VAR)'s can be eliminated.

We proceed to prove the lemma by induction on the maximum of the heights of the derivations $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$. The basis of the induction is an immediate consequence of the remarks above.

For the induction step, in the view of the remarks above, we can assume without loss of generality that none of the derivations is just a (TOP), (VAR), or (REFL). Consequently, $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$ must all end with the same rule, depending on the type construction used in $s_0 \equiv t_0$.

If all derivations end in (ARROW), the desired equality follows from the induction hypothesis, the associativity of $\odot$ and the equation (1). Similarly for (VART) using the equation (3). The desired equality in the case (FORALL) follows from the induction hypothesis using Lemma 7, from the associativity of $\odot$ and from the equation (2). The remaining cases are straight-forward. ▮

- $C \vdash t_1 \sqcap t_2 \leq t_i$, $(i = 1,2)$ *and*
- *for any* s *such that* $C \vdash s \leq t_i$, $(i = 1,2)$ *we have* $C \vdash s \leq t_1 \sqcap t_2$. ▮

2. There is a type $t_1 \sqcup t_2$ *such that*

- $C \vdash t_i \leq t_1 \sqcup t_2$, $(i = 1,2)$ *and*
- *for any* s *such that* $C \vdash t_i \leq s$, $(i = 1,2)$ *we have* $C \vdash t_1 \sqcup t_2 \leq s$. ▮

Proof: Because of the contravariance property of the first argument of the function space operator manifest in the rule (ARROW), we will prove items 1 and 2 simultaneously. In view of Lemma 6, it is sufficient to work with proofs where all instances of (TRANS) appear at the end. Since moreover any two types have a common upper bound, *Top*, the statement of the lemma is equivalent to the following formulation:

For any $\Delta_1, \ldots, \Delta_m$*, (TRANS)-free derivations in* **SOURCE** *yielding respectively* $C \vdash u_0 \leq \cdots \leq u_m$ *and any* $\Theta_1, \ldots, \Theta_n$*, (TRANS)-free derivations yielding respectively* $C \vdash v_0 \leq \cdots \leq v_n$ *,*

1. *if* $u_0 \equiv v_0$*, and let* $t_1 \equiv u_m$ *and* $t_2 \equiv v_n$*, then there is a type* $t_1 \sqcap t_2$ *having the properties in item 1 of the lemma;*
2. *if* $u_m \equiv v_n$*, and let* $t_1 \equiv u_0$ *and* $t_2 \equiv v_0$*, then there is a type* $t_1 \sqcup t_2$ *having the properties in item 2 of the lemma.*

This is shown by induction on the maximum of m, n and of the heights of $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$. To be able to apply the induction hypothesis, a case analysis is performed, depending on the structure of t_1 and t_2. We will only look at a few illustrative cases. The facts listed in Remark 10 and the reasoning that produced these facts as well as the remarks opening the proof of Lemma 8 are used throughout.

For example, if t_1 is a type variable in item 1, then u_i is also a type variable for each i, and $u_{i-1} \leq u_i \in C$, $i = 1, \ldots, n$. Then, one of $C \vdash u_0 \leq \cdots \leq u_m$ or $C \vdash v_0 \leq \cdots \leq v_n$, must be an initial segment of the other, so t_1 and t_2 are comparable and $t_1 \sqcap t_2$ can be taken as the smaller among them. For item 2, if t_1 is a type variable, then $u_0 \leq u_1 \in C$ and, by induction hypothesis (m decreases), $t_1 \sqcup t_2$ can be taken to be $u_1 \sqcup t_2$.

As another example, suppose that in item 1 t_1 has the form $\forall a \leq s.\, r_1$. If $u_0 \equiv v_0$ is a type variable, then $u_0 \leq u_1 \in C$ and $v_0 \leq v_1 \in C$ hence $u_1 \equiv v_1$ and we can apply the induction hypothesis by eliminating Δ_1, Θ_1. Assume that $u_0 \equiv v_0$ is not a type variable. By Remark 10 (simplified to take into account the weakening of (FORALL)), it must have the form $\forall a \leq s.\, r$. Again by Remark 10 t_2 is either *Top* or has the form $\forall a \leq s.\, r_2$. If $t_2 \equiv Top$ then $t_1 \sqcap t_2$ can be taken to be t_1. Otherwise, there are (TRANS)-free derivations $\Delta'_1, \ldots, \Delta'_m$ yielding $C, a \leq s \vdash u'_0 \leq \cdots \leq u'_m$ and $\Theta'_1, \ldots, \Theta'_n$ yielding respectively $C, a \leq u \vdash v'_0 \leq \cdots \leq v'_n$ where $u'_0 \equiv v'_0$ and $u'_m \equiv r_1$ and $v'_n \equiv r_2$, and where each of these derivations has strictly smaller height than the corresponding one among $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$. By induction hypothesis we get a type $r_1 \sqcap r_2$, and we can then take $t_1 \sqcap t_2$ to be $\forall a \leq s.\, r_1 \sqcap r_2$. This calculation makes clear where our proof breaks down if we were to use the more general rule (FORALL) instead of (W-FORALL). Indeed, if the bounds on the type variables were allowed to differ, as in the more general case, we would be unable to apply the induction hypothesis since the two contexts would differ between the Θ's and the Δ's.

We omit the remaining cases, which use similar ideas. ▮

This implies that the statement of the lemma holds for Δ_1, Δ_2, with common type $s \to r$, with $\Sigma \equiv [\text{ABS}]\langle \Sigma' \rangle$, and with $\Theta_i \equiv (\text{ARROW})\langle (\text{REFL}), \Theta_i' \rangle$, $(i = 1,2)$. The congruence claim follows from

$$\lambda x{:}\, s.\, \iota(P)(M) \;=\; \iota(\mathsf{arrow}(I)(P)(\lambda x{:}\, s.\, M)$$

which is readily verified.

Rule[B-SPEC]. To simplify the notation, we omit the contexts. Suppose that $\Delta_i \equiv [\text{B}-\text{SPEC}]\langle \Delta_i', \Xi_i \rangle$ and that Δ_i yields $e(r)$: $[r/a]t_i$ (r is the same since it appears in the term and we can take the bound variable to be the same without loss of generality), thus Δ_i' yields e : $\forall a \le s_i.\, t_i$ and Ξ_i yields $r \le s_i$, $(i = 1,2)$. Apply the induction hypothesis to Δ_1', Δ_2' obtaining $w, \Sigma', \Theta_1', \Theta_2'$. Also by induction hypothesis,

$$(5) \qquad \Delta_i \cong [\text{B}-\text{SPEC}]\langle [\text{INH}]\langle \Sigma', \Theta_i' \rangle, \Xi_i \rangle, \quad (i = 1,2).$$

Since $w \le \forall a \le s_i.\, t_i$, $(i = 1,2)$ it follows from Remark 10 (simplified to take into account the weakening of (FORALL)) that there must exist types u, v such that $s_i \equiv u$, $a \le s_i \vdash v \le t_i$, $(i = 1,2)$ and $w \le \forall a \le u.\, v$ are derivable. It follows that $r \le u$, and, by Lemma 7, that $a \le r \vdash v \le t_i$, $(i = 1,2)$ are derivable. Next, we will use the following sublemma:

> **Sublemma** For any derivation Δ yielding $C, a \le r \vdash s \le t$ there exists a derivation Σ yielding $C \vdash [r/a]s \le [r/a]t$ such that, if the translations Δ^*, Σ^* yield respectively
>
> $$C^*,\, a,\, f{:}\, a \multimap r^* \;\vdash\; P : s^* \multimap t^* \;,\; C^* \;\vdash\; Q : [r^*/a]s^* \multimap [r^*/a]t^*$$
>
> then
>
> $$C^* \;\vdash\; Q \;=\; (\Lambda a.\, \lambda f{:}\, a \multimap r^*.\, P)(r^*)(I)$$
>
> is provable in **TARGET**. ▮

The sublemma is proved by induction on the height of Δ and is omitted. The sublemma allows us to obtain $[r/a]v \le [r/a]t_i$ from $a \le r \vdash v \le t_i$, $(i = 1,2)$. Let Θ_i be some derivation of $[r/a]v \le [r/a]t_i$, $(i = 1,2)$. Let Ξ be some derivation of $r \le u$. Let Ω be some derivation of $w \le \forall a \le u.\, v$. One can readily verify that the right hand side of (5) is congruent to

$$[\text{INH}]\langle [\text{B}-\text{SPEC}]\langle [\text{INH}]\langle \Sigma', \Omega \rangle, \Xi \rangle, \Theta_i \rangle$$

This implies that the statement of the lemma holds for Δ_1, Δ_2, with common type $[r/a]v$, with $\Sigma \equiv [\text{B}-\text{SPEC}]\langle [\text{INH}]\langle \Sigma', \Omega \rangle, \Xi \rangle$, and with Θ_i being just Θ_i, $(i = 1,2)$. (**Note.** There is no difficulty in dealing with (FORALL) instead of (W-FORALL) here: $s_i \equiv u$ would be simply replaced by $s_i \le u$.)

Rule[R-ELIM]. Suppose that $\Delta_i \equiv [\text{R}-\text{ELIM}]\langle \Delta_i' \rangle$ and that Δ_i yields $\Gamma \vdash$ elim e : $[\mu a_i.\, t_i/a_i]t_i$, thus Δ_i' yields $\Gamma \vdash e : \mu a_i.\, t_i$, $(i = 1,2)$;. Apply the induction hypothesis to Δ_1', Δ_2' obtaining $s', \Sigma', \Theta_1', \Theta_2'$. Also by induction hypothesis,

$$\Delta_i \cong [\text{R}-\text{ELIM}]\langle [\text{INH}]\langle \Sigma', \Theta_i' \rangle \rangle, \quad (i = 1,2).$$

Since $s' \le \mu a_i.\, t_i$, $(i = 1,2)$ are derivable, it follows from Remark 10 that there must exist a, t such that $\mu a_i.\, t_i \equiv \mu a.\, t$, $(i = 1,2)$ and $s' \le \mu a.\, t$ are derivable. Let Θ' be any derivation of $s' \le \mu a.\, t$. Since by Lemma 9, $\Theta_1' \cong \Theta_2' \cong \Theta'$, the statement of the lemma holds with common type $[\mu a.\, t/a]t$, with $\Sigma \equiv [\text{R}-\text{ELIM}]\langle [\text{INH}]\langle \Sigma', \Theta' \rangle \rangle$, and with $\Theta_i \equiv (\text{REFL})$, $(i = 1,2)$.

Theorem 13 (Coherence) *Replace (FORALL) with (W-FORALL). If Δ_1 and Δ_2 are two* **SOURCE** *derivations yielding the same typing judgement then $\Delta_1 \cong \Delta_2$ (their translations yield provably equal terms in* **TARGET***).*

Proof: Take $t_1 \equiv t_2$ in Lemma 12. By Lemma 9, $\Theta_1 \cong \Theta_2$. It follows that $\Delta_1 \cong \Delta_2$. ▮

6 Models

So far we have not actually given a model for the language **SOURCE**. In this section we correct this omission. However, it is a central point of this paper that there is *basically nothing new that we need to do in this section,* since calculi satisfying the equational theory of **TARGET** have been thoroughly studied in the literature on the semantics of type systems. Domain-theoretic semantics suggests natural candidates for a special class of maps with the properties needed to interpret the operators $\rightarrow$ and $\circ\!\!\rightarrow$. Here we present list some of these semantic solutions; all of which apply to abstract types as well as to variants. A syntactic version could also be given by a syntactic translation into an extension of the target calculus of section 2, which expresses the properties mentioned above and the consistency of which is ensured by our semantic considerations.

The domain-theoretic interpretations that we have examined so far are summarized in the following table. The necessary properties for all but the last row can be found in [TT87, HP89b], [CGW89],[ABL86], [CGW87], and [Gir87] respectively. The properties needed for the last row can be checked in a manner similar to [Gir87].

TYPES	TERMS	COERCIONS	VARIANTS
Algebraic lattices	continuous maps	bistrict maps	sep sum of lattices
Scott domains		strict maps	separated sums
Finitary projections			
dI domains	stable maps	strict stable maps	
coherent spaces		linear maps	$!A \oplus !B$
dI domains			

By a bistrict map of lattices we mean a continuous map which preserves both bottom and top elements. A separated sum of lattices L and M is the disjoint sum of L and M together with new top and bottom elements. Note that the category of Scott domains (finitary projections, respectively) and strict maps does have finite coproducts, given by coalesced sums of domains, and this implies that the required equation

$$\{\text{VART-CRN?}\} \qquad P(\textsf{case}\ M\ \textsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) \ = \ \textsf{case}\ M\ \textsf{of}\ l_1 \Rightarrow F_1; P, \ldots, l_n \Rightarrow F_n; P$$

holds if P is a strict map (in fact, a separated sum of domains A and B is just the coalesced sum of the lifted domains $A_\perp$ and $B_\perp$). Furthermore, it may be checked that strictness is preserved by the formation of coercion maps from given ones according to the coercion rules given in section 3 and at the beginning of this section. This model satisfies also {VART-BETA}+{VART-ETA}. An important property used in the case of Scott domains (finitary projections, respectively) is that the continuous maps from C to D are in one-to-one correspondence with the strict maps from $C_\perp$ to D. Analogous remarks hold for stable maps and linear maps, with $!C$ instead of $C_\perp$ (see [Gir89], Chapter 8).

From a category-theoretic point of view, the main point is that we are dealing with *two categories,* one a reflective subcategory of the other, i.e. the inclusion functor has a left adjoint. The where s, t, u, v are type expressions and $\leq$ is the relation of inheritance (reading $s \leq t$ as "s inherits from t"). Note, in particular, the *contra*variance in the first argument of the → operator. In contrast, semantic domains which solve recursive domain equations such as $D = D \rightarrow D$ are generally constructed using a technique—adjoint pairs to be precise—which make it possible to "order" types using a concept of approximation based on the rule

$$\frac{\phi\colon s \rightarrow u \qquad \psi\colon t \rightarrow v}{\phi \rightarrow \psi\colon (s \rightarrow t) \rightarrow (u \rightarrow v)}$$

where $\phi = \langle \phi^L, \phi^R \rangle$ and $\psi = \langle \psi^L, \psi^R \rangle$ are adjoint pairs and $\phi \rightarrow \psi$ is the adjoint pair $\langle \lambda f.\ \psi^L \circ f \circ \phi^R,\ \lambda f.\ \psi^R \circ f \circ \phi^L \rangle$. Note, for this case, the *co*variance in the first argument of the → operator. Because of this difference, models such as the PER interpretation of Bruce and Longo [BL88], which provides a semantics for inheritance and parametric polymorphism, do not evidently extend to a semantics for recursive types. To provide for recursive types under this interpretation M. Coppo and M. Zacchi [Cop85, CZ86] utilize an appeal to the structure of the underlying universal domain, which is itself an inverse limit which solves a recursive equation. R. Amadio [Ama89, Ama90] and F. Cardone [Car89b] have explored this approach in considerable detail. There has also been progress on understanding the solution of recursive equations over domains internally to the PER model which should provide further insights [FMRS89, Fre89]. On the other hand, models such as those of Girard [Gir86] and Coquand, Gunter and Winskel [CGW87, CGW89], which handle parametric polymorphism and recursive types, do not provide an evident interpretation for inheritance. It has been the purpose of this paper to resolve this problem by an appeal to the paradigm of "inheritance and implicit coercion". However, this leaves open the question of how recursive types can be treated with this technique if one is to include a more powerful set of rules for deriving inheritance judgements between recursive types.

One complicating problem is to decide exactly what form of inheritance between recursive types is desired. For example, it seems very reasonable that if s is a subtype of t then the type of lists of s's should be a subtype of lists of t's. This is not actually derivable in the inheritance system described in this paper since there are no rules for inheritance between recursive types. But care must be taken: if s is a subtype if t then is the solution of the equations $a = a \rightarrow s$ be a subtype of the solution of $a = a \rightarrow t$? There are several possible approaches to answering this question. The PER interpretation provides a good guide: we can ask whether the solutions of these two equations have the desired relation in the PER model. Concerning the coercions approach we are forced to ask whether there is any intuitive coercion between these two types. If there is, we have not seen it! It is reasonable to conjecture that inheritance relations derived using the following rule will be acceptable:

$$\text{(REC)} \qquad \frac{C, a \leq Top \ \vdash\ s \leq t}{C \ \vdash\ \mu a.\, s \leq \mu a.\, t}$$

where types s and t have only *positive* occurrences of the variable a. Unfortunately, this misses many interesting inheritance relations that one would like to settle. Discussions of this problem will appear in several future publications on this subject. A rather satisfactory treatment using coercions has been described in [BGS89] by using the "Amber rule" of Cardelli [Car86].

Operational semantics. Despite its importance there is virtually no literature on theoretical issues concerning the operational semantics of languages with inheritance polymorphism. In particular, at the time we are writing there are no published discussions of the relationship (if any!) of the denotational models which have been studied to the intended operational semantics of a programming language based on the models. In fact, the operational semantics of no existing "practical"

e with a field l of type s, we would like to modify or update the l field of e by replacing $e.l$ by $f(e.l)$ *without losing or modifying any of the other fields of* e. The development of calculi which can deal with this form of polymorphism and the ways in which Fun and related languages can be used to represent similar techniques are an object of considerable current investigation. One recent effort in this direction is [CM89] but several other efforts are under way. Despite its importance we have not explored this issue in this paper since the discussion about it is very unsettled and it will merit independent treatment at a later date.

We believe that the "inheritance as implicit coercion" method is quite robust. For example, it easily extends to accommodate "constant" inheritances between base types, such as $int \leq real$, as long as coherence conditions similar to the ones arising in the proofs of the relevant lemmas in this paper hold between the the constant coercions which interpret these inheritances. Moreover, we expect that our methods will extend to the functional part of Quest [Car89a] and to the language described in [CM89], using the techniques of Coquand [Coq88] and Lamarche [Lam88]. Current work on inheritance and subtyping such as [CHC90] and [Mit90] will provide new challenges. We *do not claim* that every interesting aspect of inheritance can necessarily be handled in this way. However, our treatment, by showing that inheritance can be uniformly eliminated in favor of definable coercion, provides a challenge to formalisms which purport to introduce inheritance as a fundamentally new concept. Moreover, our basic approach to the semantics of inheritance should provide a useful contrast with other approaches.

8 Acknowledgements.

Breazu-Tannen's research was partially supported by ARO Grant DAAG29-84-K-0061 and ONR Grant NOOO14-88-K-0634. Many of the results of this paper were obtained during Coquand's visit to the University of Pennsylvania, partially sponsored by the Natural Sciences Association. Gunter's research was partially supported by ARO Grant DAAG29-84-K-0061 and 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.

(FORALL)
$$\frac{C \vdash s \leq t \qquad C, a{\leq}s \vdash u \leq v}{C \vdash \forall a{\leq}t.\, u \leq \forall a{\leq}s.\, v}$$

For Lemmas 11 and 12, and for Theorem 13 this is replaced with the weaker

(W-FORALL)
$$\frac{C, a{\leq}t \vdash u \leq v}{C \vdash \forall a{\leq}t.\, u \leq \forall a{\leq}t.\, v}$$

(REFL)
$$C \vdash t \leq t$$

where the free variables of t are declared in C

(TRANS)
$$\frac{C \vdash r \leq s \qquad C \vdash s \leq t}{C \vdash r \leq t}$$

Variants:

(VART)
$$\frac{C \vdash s_1 \leq t_1 \quad \cdots \quad C \vdash s_p \leq t_p}{C \vdash [l_1{:}\, s_1, \ldots, l_p{:}\, s_p] \leq [l_1{:}\, t_1, \ldots, l_p{:}\, t_p, \ldots, l_q{:}\, t_q]}$$

Rules for deriving typing judgements:

Fragment:

[VAR]
$$\Gamma_1, x{:}t, \Gamma_2 \vdash x : t$$

[ABS]
$$\frac{\Gamma, x{:}s \vdash e : t}{\Gamma \vdash \lambda x{:}s.\, e : s \to t}$$

[APPL]
$$\frac{\Gamma \vdash d : s \to t \qquad \Gamma \vdash e : s}{\Gamma \vdash d(e) : t}$$

[RECD]
$$\frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash \{l_1 = e_1, \ldots, l_m = e_m\} : \{l_1{:}\, t_1, \ldots, l_m{:}\, t_m\}}$$

Appendix B: The language TARGET

Type expressions:
Fragment: $a \mid s \to t \mid \{l_1 \colon s_1, \ldots, l_m \colon s_m\} \mid \forall a.\, t \mid \mu a.\, t$
Variants: $\mid [l_1 \colon t_1, \ldots, l_n \colon t_n]$
Coercion space: $\mid s \circ\!\!\rightarrow t$
where a ranges over type variables and $n \geq 1$. For $m = 0$ we get the *empty record type* $\mathbf{1} \overset{\text{def}}{=} \{\}$.

Raw terms:
Fragment:

$$x \mid M(N) \mid \lambda x \colon t.\, M \mid \{l_1 = M_1, \ldots, l_m = M_m\} \mid M.l \mid \Lambda a.\, M \mid M(t) \mid \mathsf{intro}[\mu a.\, t]M \mid \mathsf{elim}\ M$$

Variants:

$$\mid [l_1 \colon t_1, \ldots, l_i = M, \ldots, l_n \colon t_n] \mid \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n$$

Coercion-coercion combinator:

$$\mid \iota_{s,t}$$

Coercion combinators:

$$\mid \mathsf{top}[t] \mid \mathsf{arrow}[s, t, u, v] \mid \mathsf{recd}[s_1, \ldots, s_q, t_1, \ldots, t_p] \mid \mathsf{forall}[s, t, a, u, v] \mid$$

$$\mathsf{vart}[s_1, \ldots, s_p, t_1, \ldots, t_q] \mid \mathsf{refl}[t] \mid \mathsf{trans}[r, s, t]$$

where x ranges over (term) variables and $n \geq 1$. For $m = 0$ we get the *empty record*, for which we will keep the notation $\{\}$. We will usually omit the cumbersome type tags on the coercion(-coercion) combinators. We use $[N/x]M$ for substitution.

Typing judgements, have the form $\Upsilon \vdash M : t$, where Υ is a typing context. *Typing contexts* are defined recursively as follows: $\emptyset$ is a context; if Υ is a context which does not declare a, then Υ, a is a typing context; if Υ is a context which does not declare x, and the free variables of t are declared in Υ, then $\Upsilon, x \colon t$ is a typing context.

Rules for deriving typing judgements:

Fragment:
Same as in Appendix A: [VAR] , [ABS] , [APPL] , [RECD] (in particular, for $n = 0$, $\Upsilon \vdash \{\} : \mathbf{1}$) , [SEL].

[GEN]
$$\frac{\Upsilon, a \vdash M : t}{\Upsilon \vdash \Lambda a.\, M : \forall a.\, t}$$

[SPEC]
$$\frac{\Upsilon \vdash M : \forall a.\, t}{\Upsilon \vdash M(s) : [s/a]t}$$

Same as in Appendix A: [R-INTRO] , [R-ELIM].

{BETA} $(\lambda x{:}\, s.\, M)(N) \; = \; [N/x]M$

where $N : s$.

{ETA} $\lambda x{:}\, s.\, M(x) \; = \; M$

where $M : s \to t$ and x not free in M.

{RECD-BETA} $\{l_1 = M_1, \ldots, l_m = M_m\}.l_i \; = \; M_i$

where $m \geq 1$, $M_1{:}\, t_1, \ldots, M_m{:}\, t_m$.

{RECD-ETA} $\{l_1 = M.l_1, \ldots, l_m = M.l_m\} \; = \; M$

where $M : \{l_1{:}\, t_1, \ldots, l_m{:}\, t_m\}$. For $m = 0$, this rule gives $\{\} \; = \; M$ which makes **1** into a terminator.

{FORALL-BETA} $(\Lambda a.\, M)(r) \; = \; [r/a]M$

{FORALL-ETA} $\Lambda a.\, M(a) \; = \; M$

where $M : \forall a.\, t$ and a not free in M.

{R-BETA} $\mathsf{elim}\,(\mathsf{intro}[\mu a.\, t]M) \; = \; M$

where $M : \mu a.\, t$.

{R-ETA} $\mathsf{intro}[\mu a.\, t](\mathsf{elim}\; M) \; = \; M$

where $M : [\mu a.\, t/a]t$.

Variants:

We omit the simple rules for congruence with respect to variant formation, and case analysis.

Appendix C: The translation

We present first the remaining of the translation of the fragment discussed in section 3.

(VAR)*

$$C_1^*,\, a,\, f\colon a \to t^*,\, C_2^* \;\vdash\; f : a \to t^*$$

(RECD)*

$$\frac{C^* \;\vdash\; P_1 : s_1^* \to t_1^* \quad \cdots \quad C^* \;\vdash\; P_p : s_p^* \to t_p^*}{C^* \;\vdash\; R : \to \{l_1\colon s_1^*, \ldots, l_p\colon s_p^*, \ldots, l_q\colon s_q^*\}\{l_1\colon t_1^*, \ldots, l_p\colon t_p^*\}}$$

where $R \overset{\text{def}}{=} \lambda w\colon \{l_1\colon s_1^*, \ldots, l_p\colon s_p^*, \ldots, l_q\colon s_q^*\}.\, \{l_1\colon P_1(w.l_1), \ldots, l_p\colon P_p(w.l_p)\}$

(REFL)*

$$C^* \;\vdash\; \lambda x\colon t^*.\, x : t^* \to t^*$$

where the free variables of t^* are declared in C^*

(TRANS)*

$$\frac{C^* \;\vdash\; P : r^* \to s^* \qquad C^* \;\vdash\; Q : s^* \to t^*}{C^* \;\vdash\; P; Q : r^* \to t^*}$$

The rules [VAR] , [ABS] , [APPL] , [RECD] , [SEL] , [R-INTRO] , [R-ELIM] are translated straightforwardly, see below. Here is the translation of the only other rule left (the translations of the other rules appears in section 3).

[B-GEN]

$$\frac{\Gamma^*,\, a,\, f\colon a \to s^* \;\vdash\; M : t^*}{\Gamma^* \;\vdash\; \Lambda a.\, \lambda f\colon a \to s^*.\, M : \forall a.\, ((a \to s^*) \to t^*)}$$

In the following, we present the translation for the full calculus. As before, for any **SOURCE** `item` we will denote by `item*` its translation into **TARGET** . We begin with the types. Note the translation of bounded generics and of *Top*.

$$\begin{array}{rcl@{\qquad}rcl}
a^* & \overset{\text{def}}{=} & a & (\forall a \leq s.\, t)^* & \overset{\text{def}}{=} & \forall a.\, ((a \multimap s^*) \to t^*) \\
Top^* & \overset{\text{def}}{=} & 1 & (\mu a.\, t)^* & \overset{\text{def}}{=} & \mu a.\, t^* \\
(s \to t)^* & \overset{\text{def}}{=} & s^* \to t^* & [l_1\colon s_1, \ldots, l_n\colon s_n]^* & \overset{\text{def}}{=} & [l_1\colon s_1^*, \ldots, l_n\colon s_n^*] \\
\{l_1\colon s_1, \ldots, l_m\colon s_m\}^* & \overset{\text{def}}{=} & \{l_1\colon s_1^*, \ldots, l_m\colon s_m^*\} & & &
\end{array}$$

where $s \times t \overset{\text{def}}{=} \{\mathit{left}\colon s, \mathit{right}\colon t\}$.

One shows immediately that $([s/a]t)^* \equiv [s^*/a]t^*$. We extend this to contexts and inheritance contexts, which translate into just typing contexts in **TARGET** .

$$\begin{array}{rcl@{\qquad}rcl}
\emptyset^* & \overset{\text{def}}{=} & \emptyset & \emptyset^* & \overset{\text{def}}{=} & \emptyset \\
(\Gamma, a \leq t)^* & \overset{\text{def}}{=} & \Gamma^*,\, a,\, f\colon a \multimap t^* & (C, a \leq t)^* & \overset{\text{def}}{=} & C^*,\, a,\, f\colon a \multimap t^* \\
(\Gamma, x\colon t)^* & \overset{\text{def}}{=} & \Gamma^*,\, x\colon t^* & & &
\end{array}$$

A **SOURCE** derivation yielding an typing judgment $\Gamma \ \vdash \ e : t$ is translated as a tree of **TARGET** judgments yielding $\Gamma^* \ \vdash \ M : t^*$. Here are the **TARGET** rules that correspond to the rules for deriving typing judgements in **SOURCE**.

The rules [VAR] , [ABS] , [APPL] , [RECD] , [SEL] , [R-INTRO] , [R-ELIM] , [VART] , [CASE] all have direct correspondents in **TARGET** so their translation is straightforward. We ilustrate it with two examples.

[VAR]*

$$\Gamma_1^*, x{:}\,t^*, \Gamma_2^* \ \vdash \ x : t^*$$

[ABS]*

$$\frac{\Gamma^*, x{:}\,s^* \ \vdash \ M : t^*}{\Gamma^* \ \vdash \ \lambda x{:}\,s^*.\, M : s^* \to t^*}$$

Here is the translation of the other three rules.

[B-GEN]

$$\frac{\Gamma^*, a, f{:}\,a \mathrel{\circ\!\!\rightarrow} s^* \ \vdash \ M : t^*}{\Gamma^* \ \vdash \ \Lambda a.\, \lambda f{:}\,a \mathrel{\circ\!\!\rightarrow} s^*.\, M : \forall a.\, ((a \mathrel{\circ\!\!\rightarrow} s^*) \to t^*)}$$

[B-SPEC]*

$$\frac{\Gamma^* \ \vdash \ M : \forall a.\, ((a \mathrel{\circ\!\!\rightarrow} s^*) \to t^*) \qquad \widehat{\Gamma}^* \ \vdash \ P : r^* \mathrel{\circ\!\!\rightarrow} s^*}{\Gamma^* \ \vdash \ M(r^*)(P) : [r^*/a]t^*}$$

[INH]*

$$\frac{\Gamma^* \ \vdash \ M : s^* \qquad \widehat{\Gamma}^* \ \vdash \ P : s^* \mathrel{\circ\!\!\rightarrow} t^*}{\Gamma^* \ \vdash \ \iota(P)(M) : t^*}$$

Lemma 14 *The rules* (TOP)* – (TRANS)* *and* [VAR]* – [INH]* *are directly derivable in* **TARGET** . ▮

[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, 1986.

[Car88a] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car88b] L. Cardelli. Structural subtyping and the notion of power type. In J. Ferrante and P. Mager, editors, *Symposium on Principles of Programming Languages*, pages 70–79, ACM, 1988.

[Car89a] L. Cardelli. *Typeful programming.* Research Report 45, DEC Systems, Palo Alto, May 1989.

[Car89b] F. Cardone. Relational semantics for recursive types and bounded quantification. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *International Colloquium on Automata, Languages and Programs*, pages 164–178, *Lecture Notes in Computer Science vol. 372,* Springer, July 1989.

[CG90] P.-L. Curien and G. Ghelli. Coherence of subsumption. In *Proceedings CAAP'90, LNCS 431*, 1990. Full version to appear in *Mathematical Structures in Computer Science.*

[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.

[CGW89] T. Coquand, C. A. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation.*, 81:123–167, 1989.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CHC90] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In P. Hudak, editor, *Principles of Programming Languages*, pages 125–135, ACM, 1990.

[CM89] L. Cardelli and J. Mitchell. Operations on records. In M. Mislove, editor, *Mathematical Foundations of Programming Semantics*, *Lecture Notes in Computer Science,* Springer, March 1989.

[Cop85] M. Coppo. A completeness theorem for recursively defined types. In W. Brauer, editor, *International Colloquium on Automata, Languages and Programs*, pages 120–129, *Lecture Notes in Computer Science vol. 194,* Springer, 1985.

[Coq88] T. Coquand. Categories of embeddings. In Y. Gurevich, editor, *Logic in Computer Science*, pages 256–263, IEEE Computer Society, July 1988.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[CZ86] M. Coppo and M. Zacchi. Type inference and logical relations. In A. Meyer, editor, *Symposium on Logic in Computer Science*, pages 218–226, ACM, 1986.

[Law69] F. W. Lawvere. Diagonal arguments and cartesian closed categories. In *Category theory, homology theory, and their applications II*, pages 134–145, *Lecture Notes in Mathematics*, Vol. 92, Springer-Verlag, 1969.

[LP85] S. Mac Lane and R. Pare. Coherence for bicategories and indexed categories. *Journal of Pure and Appled Algebra*, 37:59–80, 1985.

[Mar84] P. Martin-Löf. *Intutionistic Type Theory. Studies in Proof Theory*, Bibliopolis, 1984.

[Mar88] S. Martini. Bounded quantifiers have interval models. In R. Cartwright, editor, *Symposium on LISP and Functional Programming*, pages 164–173, ACM, 1988.

[Mey82] A. R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

[Mit90] J. Mitchell. Toward a typed foundation for method specialization and inheritance. In P. Hudak, editor, *Principles of Programming Languages*, pages 109–124, ACM, 1990.

[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, New York, 1988.

[Rey80] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 211–258, *Lecture Notes in Computer Science vol. 94,* Springer, 1980.

[Sal88] A. Salvesen. Polymorphism and monomorphism in Martin-Löf's Type Theory. In *Logic Colloquium'88*, 1988.

[Sco80] D. S. Scott. Relating theories of the lambda calculus. In J. R. Hindley, editor, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 403–450, Academic Press, 1980.

[Sta88] R. Stansifer. Type inference with subtypes. In J. Ferrante and P. Mager, editors, *Symposium on Principles of Programming Languages*, pages 88–97, ACM, 1988.

[Str88] T. Streicher. *Correctness and completeness of a categorical semantics of the Calculus of Constructions.* PhD thesis, Passau University, 1988.

[Tro73] A. S. Troelstra. *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics vol. 344*, Springer, 1973.

[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.

COMPUTING WITH COERCIONS[1]

(Extended Abstract)

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

University of Pennsylvania

Abstract. This paper relates two views of the operational semantics of a language with multiple inheritance. It is shown that the introduction of explicit coercions as an interpretation for the implicit coercion of inheritance does not affect the evaluation of a program in an essential way. The result is proved by semantic means using a denotational model and a computational adequacy result to relate the operational and denotational semantics.

1 Introduction

There have been a number of efforts to understand the denotational semantics of inheritance polymorphism and a variety of mathematical models for languages with subtle semantic features have been discovered. However, as far as the authors of this paper know, no one has attempted to discuss what, if anything, these denotational models have to do with the intended execution of programs in the languages they model. For example, *all* of the published denotational models of the language Fun of Cardelli Wegner [CW85] (including the work of authors of this paper) model this language in way that corresponds to no reasonable interpretation of its operational semantics! No functional programming language in common use diverges when evaluating the program $\lambda x.\ e$, even when the expression e may diverge. Yet the models for Fun which have been studied identify the abstraction $\lambda x.\ \bot$ with the divergent program $\bot$. Besides this problem, all existing models satisfy the unrestricted β rule, which fails to be a legitimate transformation in call-by-value languages. Since call-by-value is the most common form of evaluation, one is led to ask whether this commitment to β was an important feature of the models concerned. In short, very little has been done to close the gap between denotational and operational theories of inheritance. We see two basic things as missing from the current theories: (1) a careful discussion of the structional operational semantics of languages with inheritance type systems and (2) any account of the relationship between the suggested models and a reasonable account of operational semantics.

Our goal in this paper is to attempt an account of problem (1) guided by an approach to (2). We carry out this study in a simple, familiar context by using an extension of Plotkin's illustrative language PCF [Plo77]. We develop a simple structural operational semantics for this language in the spirit of the evaluation mechanisms of languages such as LISP and ML in which functions call their parameters by value. Our extension, which we call PCF+, is obtained by adding record and variant types. This language is extended to a new language, PCF++, by permitting the use of a form of inheritance which allows more programs to be viewed as type correct. We then study the question of the proper operational interpretation of PCF++. One possible approach is simple to understand: after a PCF++ program is shown to be type correct, the type information in the term is erased and the resulting term (which lives in an extended untyped lambda-calculus) is

[1] Appears in **Conference on Lisp and Functional Programming**, edited by M. Wand, Nice, France, July 1990, pp. 44–60.

before executing it, it is reasonable to ask whether translation would affect the evaluation. Since coercions remove the "junk" in a term, they may play a useful role in efficient implementation. However, our primary interest is in the abstract specification of the language and not the details of its efficient implementation.

Our *main result* relates the direct execution of a PCF++ program phrase e to the execution of *any* of its PCF+ translations, e^*. We prove that

$$e \text{ terminates if and only if } e^* \text{ terminates.}$$

If both e and e^* terminate, what can we say about the relationship between the results of the two computations? Of course, we are able to show that if the type of e is *ground*, (integer or boolean) then the results are the exactly the same. In this language we are also interested in computing with more complex objects, such as records/variants of records/variants of ground data (this is particularly consistent with the way things are viewed in object-oriented database programming applications [OBB89] for example). We call the types of such data *observable types*. Now, the philosophy of PCF++ is that the type of program phrases is part of them, i.e., user-supplied in some sense. (This is in contrast with the approaches based on type inference; see for example [Wan89].) At observable types, we show that the results of the two computations have the same *components* in those record fields which appear in the prescribed type of the program phrase. This is the best we can hope for, since the introduction of coercions yields computations which may remove "junk" fields, namely the fields not occurring in the prescribed type. Moral: if you specify a type for your program, don't expect to observe more than what the type allows. Anyway, our conclusion is that coercions *make no essential difference* to the computation.

While this result only relates our translation to the operational semantics, it can be used for *transfer of computational adequacy.* Consider a denotational semantics $\mathcal{D}^+$ of PCF+ for which our translation is coherent. This yields a denotational semantics $\mathcal{D}^{++}$ for PCF++ where a term is interpreted by first translating it into PCF+ and then taking the $\mathcal{D}^+$-meaning of the translation. Under some reasonable assumptions about $\mathcal{D}^+$, our main result implies that if $\mathcal{D}^+$ is computationally adequate (*i.e.* the meaning of a term e is non-bottom iff the evaluation of e terminates) for the operational semantics of PCF+ then $\mathcal{D}^{++}$ is computationally adequate for the operational semantics of PCF++.

An interesting methodological twist is that our proof of the main result actually uses a specific denotational semantics $[\![\cdot]\!]^+$ which is computationally adequate for PCF+ and for which this transfer can be done! As it is, we show directly that $[\![\cdot]\!]^{++}$ is computationally adequate for PCF++ and we derive our main result from this. We regard this as a nice example of the use of a domain-theoretic semantics for obtaining an essentially syntactic result.

Another comment on methodology. We have chosen to focus on call-by-value operational semantics since this is the most common style of implementation for the languages we are studying and because it offers a change of pace from our earlier results [BCGS89] where we focused on models in which the unrestricted β axiom holds. We expect that results such as the ones we are proving in this paper could be formulated for a call-by-name operational semantics, although this would call for some changes in our concept of observability.

In section 2 we begin by introducing the syntax of PCF++ as an extension of PCF+. Then we describe the translation back, from PCF++ to PCF+. Finally we give the call-by-value operational semantics and state our main theorem. In section 3 we give a domain-theoretic denotational semantics of PCF+ for which our translation is coherent and for which the operational semantics of PCF+ is sound and computationally adequate. We prove that the operational semantics of PCF++ is sound and computationally adequate for the induced denotational semantics and then

$$\mathbf{0} : \mathbf{num} \qquad \mathbf{false} : \mathbf{bool} \qquad \mathbf{true} : \mathbf{bool}$$

$$\frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{Pred}(e) : \mathbf{num}} \qquad \frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{Succ}(e) : \mathbf{num}} \qquad \frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{IsZero}(e) : \mathbf{bool}}$$

$$H,\ x : s,\ H' \vdash x : s \qquad \frac{H,\ x : s \vdash e : t}{H \vdash \lambda x : s.\ e : t} \qquad \frac{H \vdash e : s \to t \qquad H \vdash e' : s}{H \vdash e(e') : t}$$

$$\frac{H,\ x : s \vdash e : s}{H \vdash \mu x : s.\ e : s} \qquad \frac{H \vdash e : \mathbf{bool} \qquad H \vdash e' : s \qquad H \vdash e'' : s}{H \vdash \mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' : s}$$

$$\frac{H \vdash e_1 : s_1 \quad \cdots \quad H \vdash e_n : s_n}{H \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : s_1, \ldots, l_n = s_n\}} \qquad \frac{H \vdash e : \{l_1 : s_1, \ldots, l_n : s_n\}}{H \vdash e.l_i : s_i}$$

$$\frac{H \vdash e_i : s_i}{H \vdash [l_i = e_i] : [l_1 : s_1, \ldots, l_n = s_n]}$$

$$\frac{H \vdash e : [l_1 : s_1, \ldots, l_n = s_n] \qquad H \vdash f_1 : s_1 \to s \quad \cdots \quad H \vdash e_n : s_n \to s}{H \vdash \mathbf{case}\ e\ \mathbf{of}\ l_1 \Rightarrow f_1 \cdots l_n \Rightarrow f_n : s}$$

Table 1: Typing rules for PCF+.

$$\begin{array}{l} \mathbf{num} < \mathbf{num} \\ \mathbf{bool} < \mathbf{bool} \end{array} \qquad \frac{s' < s \qquad t < t'}{s \to t < s' \to t'}$$

$$\frac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{\{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\} < \{l_1 : t_1, \ldots, l_n : t_n\}}$$

$$\frac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{[l_1 : s_1, \ldots, l_n : s_n] < [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]}$$

Table 2: Inheritance rules.

$$\mathbf{0} \Downarrow \mathbf{0} \qquad \mathbf{true} \Downarrow \mathbf{true} \qquad \mathbf{false} \Downarrow \mathbf{false}$$

$$\frac{e \Downarrow \mathbf{Succ}(c)}{\mathbf{Pred}(e) \Downarrow c} \qquad \frac{e \Downarrow c}{\mathbf{Succ}(e) \Downarrow \mathbf{Succ}(c)} \qquad \frac{e \Downarrow \mathbf{0}}{\mathbf{IsZero}(e) \Downarrow \mathbf{true}} \qquad \frac{e \Downarrow \mathbf{Succ}(c)}{\mathbf{IsZero}(e) \Downarrow \mathbf{false}}$$

$$\lambda x : s.\ e \Downarrow \lambda x : s.\ e \qquad \frac{e \Downarrow \lambda x : s.\ e'' \quad e' \Downarrow c' \quad [c'/x]e'' \Downarrow c}{e(e') \Downarrow c}$$

$$\frac{e_1 \Downarrow \mathbf{true} \quad e_2 \Downarrow c}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow c} \qquad \frac{e_1 \Downarrow \mathbf{false} \quad e_3 \Downarrow c}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow c}$$

$$\frac{e_1 \Downarrow c_1 \quad \cdots \quad e_n \Downarrow c_n}{\{l_1 = e_1, \ldots, l_n = e_n\} \Downarrow \{l_1 = c_1, \ldots, l_n = c_n\}} \qquad \frac{e \Downarrow \{l_1 = c_1, \ldots, l_n = c_n\}}{e.l_i \Downarrow c_i}$$

$$\frac{e \Downarrow c}{[l = e] \Downarrow [l = c]} \qquad \frac{e \Downarrow [l_i = c'] \quad f_i(c') \Downarrow c}{\mathbf{case}\ e\ \mathbf{of}\ l_1 \Rightarrow f_1, \ldots, l_i \Rightarrow f_i, \ldots, l_n \Rightarrow f_n \Downarrow c}$$

$$\frac{[\mu x.\ e/x]e \Downarrow c}{\mu x.\ e \Downarrow c}$$

Table 3: Call-by-value evaluation.

For raw terms e and e' we write $[e'/x]e$ for the result of substituting e' for x in e. We demand all of the usual assumptions about the renaming of bound variables in e to avoid capturing free variables of e'. We assume that the substitution operation associates to the right and we may write $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$ for the simultaneous substitution of $e_1, \ldots, e_n$ for $x_1, \ldots, x_n$ respectively in e. In the event that the terms e_i are closed, note that this is the same as $[e_1/x_1] \cdots [e_n/x_n]e$ and, indeed, the order of the substitutions does not matter.

It is not hard to see that if e is a closed raw term such that $e \Downarrow c$, then c is uniquely determined. This can be proved by showing that, for a given term e, there is at most one axiom or rule from Table 3 which applies to it. Hence the rules define a deterministic evaluation strategy. The evaluation of function application is call-by-value, since the argument to the application is evaluated before being substituted into the body of the applied procedure. There is no evaluation under a lambda-abstraction, but note that records are eagerly evaluated. For example, the evaluation of an expression $\{l = e, l' = e'\}.l$ will result in the evaluation of e' as well as e even though e' is "not needed" in the result. Putting aside efficiency issues, this is only significant if e' diverges since, in that case, the evaluation of $\{l = e, l' = e'\}.l$ will also diverge. Since evaluation is deterministic, we may define a partial function $\mathcal{E}$ on raw terms as follows

$$\mathcal{E}(e) \simeq \begin{cases} c & e \Downarrow c \text{ if there is such a } c \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

- Let $s = [l_1 : s_1, \ldots, l_n : s_n]$, then $[l_i = c_i] =_s [l_j = c'_j]$ iff $c_i =_{s_i} c'_j$. ▮

If E and E' are expressions that may be undefined, write $E \simeq_s E'$ to mean that if one expression exists, then so does the other and $E =_s E'$. We may now express the desired result:

> **Main Theorem:** *Suppose $\vdash e : s$ is derivable in PCF++ and e^* is any PCF+ term which translates this sequent, then $e \Downarrow$ iff $e^* \Downarrow$. Moreover, if s is observable, then $\mathcal{E}(e) \simeq_s \mathcal{E}(e')$.* ▮

It seems difficult to prove this result directly because of the recursion case. This problem is resolved by appealing to denotational models for PCF+ and PCF++ which we now describe.

3 A computationally adequate denotational semantics.

For technical reasons we have found that it is useful to appeal to some results relating PCF+ and PCF++ to a specific denotational model which we will describe in this section. Although our goal is to prove a purely syntactic result (the Main Theorem at the end of the previous section), the semantic results which we will now establish are of independent interest.

We describe a domain-theoretic model for PCF+. The interpretation of types is as follows:

- $[\![\mathbf{bool}]\!]$ is the flat domain with three distinct elements *tt*, *ff* and least element $\perp$.
- $[\![\mathbf{num}]\!]$ is the flat domain consisting of the numbers $0, 1, 2, \ldots$ together with a least element $\perp$.
- $[\![s \to t]\!] = (s \circ\!\!\to t)_\perp$, the lifted domain of strict (*i.e.* $\perp$-preserving) functions from $[\![s]\!]$ into $[\![t]\!]$.
- $[\![\ \{l_1 : s_1, \ldots, l_n : s_n\}\ ]\!]$ consists of a bottom element $\perp$, together with the set of tuples $\{l_1 = d_1, \ldots, l_n = d_n\}$ where each d_i is a non-bottom element of $[\![s_i]\!]$. The ordering is defined by
$$\{l_1 = d_1, \ldots, l_n = d_n\} \sqsubseteq \{l_1 = d'_1, \ldots, l_n = d'_n\}$$
iff $d_i \sqsubseteq d'_i$ for each $i = 1, \ldots, n$ and $\perp \sqsubseteq d$ for each record d.
- $[\![\ [l_1 : s_1, \ldots, l_n : s_n]\ ]\!]$ consists of a bottom element $\perp$, together with the set of pairs $[l_i = d_i]$ such that d_i is a non-bottom element of $[\![s_i]\!]$. For two such pairs, $[l_i = d] \sqsubseteq [l_j = d']$ iff $i = j$ and $d \sqsubseteq d'$.

Suppose $H = x_1 : s_1, \ldots x_n : s_n$ is a type context. An *H-environment* is a function which assigns to each variable x_i an element $\rho(x_i)$ of the domain $[\![s_i]\!]$. The PCF+ interpretation of a sequent $H \vdash e : s$ is a function which assigns to each H-environment ρ a value $[\![H \vdash e : s]\!]^{++}\rho$ in $[\![s]\!]$.

We will refrain from writing out all of the semantic equations for the sequents of PCF+. The rules for the introduction and elimination operators for the record and variant types are straightforward, holding in mind that the interpretation of a record with a field which is $\perp$ is itself equal to $\perp$. Recursion is defined in the usual way using least fixed points. The function space requires some explanation which we now provide.

The *lift* $D_\perp$ of a domain D is obtained by adding a new bottom element. There is a continuous function $\mathsf{up} : D \to D_\perp$ which sends elements of D to their images in the lifted domain. This function is not strict, since it sends the bottom of D to an element of $D_\perp$ which dominates the "new" bottom

Lemma 6 *1. If $r < s < t$, then $[\![\mathsf{coerce}[r < t] : r \to t]\!] = [\![\mathsf{coerce}[s < t] : s \to t]\!] \circ [\![\mathsf{coerce}[r < s] : r \to s]\!]$*

2. If $s < t$, then $[\![\mathsf{coerce}[s < t] : s \to t]\!](d) = \bot$ iff $d = \bot$. ▮

Lemma 7 *If $\vdash c : s$ is a derivable judgement of PCF++ and c is a canonical form, then $[\![c : s]\!]^{++} \neq \bot$.* ▮

Most of the rest of this section is devoted to a proof of a kind of converse to the Soundness Theorem which we will call *computational adequacy* (the term is suggested by Albert Meyer [Mey88], although his definition includes soundness). For PCF++, it can be stated as follows:

Theorem 8 *(Computational Adequacy.) Suppose $e : s$ is derivable in PCF++. If $[\![e : s]\!]^{++} \neq \bot$ then $e \Downarrow c$ for some canonical form c.*

We focus on explaining how the methods that one uses for results such as those above are applied to a calculus with multiple inheritance. We will look at the proof of adequacy in some detail. The proof requires a relation between program meanings and programs sometimes called an "inclusive predicate". We define this relationship as follows:

Definition: Define a relation $\precsim_s$ between elements of $[\![s]\!]$ on the left and closed raw terms of type s on the right as follows. $d \precsim_s e$ if $d = \bot$ or $e \Downarrow c$ for some c and $d \precsim_s c$ where

- $f \precsim_{s \to t} \lambda x : r.\ e$ iff for each $d \in [\![s]\!]$ and term c, $d \precsim_s c$ implies $\mathsf{down}(f)(d) \precsim_t [c/x]e$.
- $\{l_1 = d_1, \ldots, l_n = d_n\} \precsim_{\{l_1 : s_1, \ldots, l_n : s_n\}} \{l_1 = e_1, \ldots, l_m = e_m\}$ iff $m \geq n$ and $d_i \precsim_{s_i} c_i$ for $i = 1, \ldots n$.
- $[l_i = d] \precsim_{[l_1 : s_1, \ldots, l_n : s_n]} [l_j = c]$ iff $i = j$ and $d \precsim_{s_i} c$.
- $tt \precsim_{\mathbf{bool}}$ **true** and $f\!f \precsim_{\mathbf{bool}}$ **false**.
- $0 \precsim_{\mathbf{num}} \mathbf{0}$ and if $n \precsim_{\mathbf{num}} c$ for a number n, then $n + 1 \precsim_{\mathbf{num}} \mathbf{Succ}(c)$. ▮

Some of the essential semantic properties of $\precsim$ are given in the following:

Lemma 9 *1. If $a \sqsubseteq b \precsim_s e$, then $a \precsim_s e$.*

2. If $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots$ is an ascending chain and $a_n \precsim_s e$ for each n, then $\bigsqcup_{n=0}^{\infty} a_n \precsim_s e$. ▮

We are now ready to sketch the proof of the primary technical lemma which is needed for the proof of PCF++ adequacy.

Lemma 10 *Suppose $H = x_1 : s_1^\dagger \ldots x_n : s_n^\dagger$ and $H \vdash e^\dagger : s^\dagger$ is derivable. If $d_i \in [\![s_i^\dagger]\!]$ and $d_i \precsim_{s_i^\dagger} e_i^\dagger$ for $i = 1, \ldots, k$, then $[\![H \vdash e^\dagger : s^\dagger]\!]^{++}[d_1, \ldots, d_k/x_1, \ldots, x_k] \precsim_{s^\dagger} [e_1^\dagger, \ldots, e_k^\dagger/x_1, \ldots, x_k]e^\dagger$.*

Proof: Let ρ be the environment $[d_1, \ldots, d_n/x_1, \ldots x_n]$ and σ be the substitution $[e_1^\dagger, \ldots, e_n^\dagger/x_1, \ldots, x_n]$. Let Δ be a PCF++ derivation of the typing judgement $H \vdash e^\dagger : s^\dagger$. We prove that $[\![H \vdash e^\dagger : s^\dagger]\!]^{++}\rho \precsim_{s^\dagger} \sigma e^\dagger$ by an induction on Δ. Assume that the Theorem is known for proofs of lesser height. There are eleven possibilities for the last step of Δ. Some of the more interesting cases (subsumption in particular) are written out fully below.

- *Subsumption rule:* $\dfrac{H \vdash e : s \qquad s < t}{H \vdash e : t}$.

  The proof for this case is by induction on the height of the proof that $s < t$. Assume that we know that the theorem holds for $H \vdash e : s$ and let $H \vdash e^* : s$ be any translation of this sequent to PCF+. There are four subcases:

  - *Base types:* These are both obvious since the coercion is the identity map.
  - *Functions:* $\dfrac{u' < u \qquad v < v'}{u \to v < u' \to v'}$.

    Suppose $s \equiv u \to v$ and $t \equiv u' \to v'$. Let $\xi_1 = \mathsf{down}[\![\mathsf{coerce}[u' < u]]\!]$ and $\xi_2 = \mathsf{down}[\![\mathsf{coerce}[v < v']]\!]$. Then $\xi = \mathsf{down}[\![\mathsf{coerce}[u \to v < u' \to v']]\!]$ satisfies $\xi(f) = \xi_2 \circ f \circ \xi_1$ for $f : [\![u]\!] \circ\!\!\to [\![v]\!]$. Set $f = \mathsf{down}[\![H \vdash e : s]\!]^{++}\rho$. If $d \precsim_{u'} c$, then $\xi_2(d) \precsim_u c$ by induction hypothesis on $u' < u$. Thus $f(\xi_2(d)) \precsim_v (\sigma e)(c)$ by induction hypothesis on $H \vdash e : s$. We may now apply the induction hypothesis on $v < v'$ to conclude that $\xi(f) = \xi_1(f(\xi_2(d))) \precsim_{v'} (\sigma e)(c)$. Since $\xi(f) = [\![H \vdash e : t]\!]^{++}\rho$ we conclude that $[\![H \vdash e : t]\!]^{++}\rho \precsim_t \sigma e$.
  - *Records:* $\dfrac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{\{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\} < \{l_1 : t_1, \ldots, l_n : t_n\}}$.

    Let $\xi_i = \mathsf{down}[\![\mathsf{coerce}[s_i < t_i]]\!]$ for $i = 1, \ldots n$ and let $\xi = \mathsf{down}[\![\mathsf{coerce}[s < t]]\!]$. By induction hypothesis, we have $d = [\![H \vdash e : s]\!]^{++}\rho \precsim_s \sigma e$. If $d = \bot$, then $\xi(d) = [\![H \vdash e : t]\!]^{++}\rho = \bot$ and we are done. If $d \neq \bot$, then $d = \{l_1 = d_1, \ldots, l_m = d_m\}$ where $d_1, \ldots, d_m \neq \bot$ and $\sigma e \Downarrow c$ for some canonical c of the form $c \equiv \{l_1 = c_1, \ldots, l_j = c_j\}$ such that $j \geq m$ and $d_i \precsim_{s_i} c_i$ for $i = 1, \ldots m$. By the induction hypothesis on inheritance judgements, we must therefore have $\xi_i(d_i) \precsim_{t_i} c_i$ for each $i = 1, \ldots, n$. Hence $\xi(d) = \{l_1 = \xi_1(d_1), \ldots, l_n = \xi_n(d_n)\} \precsim_t \{l_1 = c_1, \ldots, l_j = c_j\}$ by the definition of $\precsim_t$ and we are done.
  - *Variants:* $\dfrac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{[l_1 : s_1, \ldots, l_n : s_n] < [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]}$.

    Let $\xi_i = \mathsf{down}[\![\mathsf{coerce}[s_i < t_i]]\!]$ for $i = 1, \ldots n$ and let $\xi = \mathsf{down}[\![\mathsf{coerce}[s < t]]\!]$. By induction hypothesis, we have $d = [\![H \vdash e : s]\!]^{++}\rho \precsim_s \sigma e$. If $d = \bot$, then $\xi(d) = [\![H \vdash e : t]\!]^{++}\rho = \bot$ and we are done. If $d \neq \bot$, then $d = [l_i = d_i]$ where $d_i \neq \bot$ and $\sigma e \Downarrow c$ where $\sigma e \Downarrow c$ and $d \precsim_s c$. By the definition of $\precsim_s$, the term c has the form $[l_i = c_i]$ and $d_i \precsim_{s_i} c_i$. By induction hypothesis on $s_i < t_i$, we know that $\xi_i(d_i) \precsim_{t_i} c_i$ so $\xi(d) = [l_i = \xi_i(d)] \precsim_t [l_i = c_i]$. ▮

We may now express the desired proof of Computational Adequacy for PCF++.

Proof: (of Theorem 8) By Lemma 10 we know that $[\![e : s]\!]^{++} \precsim_s e$. Since the value on the left is assumed to differ from $\bot$, the Theorem follows immediately from the definition of $\precsim_s$. ▮

The following theorem follows immediately from Soundness and Computational Adequacy for PCF++ together with Corollary 4 of the Semantic Coherence Theorem for PCF++.

Theorem 11 *(Soundness and Adequacy for PCF+) If* $\vdash e : s$ *is derivable in PCF+, then*

1. *(Soundness)* $e \Downarrow c$ *implies* $[\![e : s]\!] = [\![c : s]\!]$.
2. *(Computational Adequacy)* $[\![e : s]\!]^{+} \neq \bot$ *implies* $e \Downarrow c$ *for some canonical form* c. ▮

The following lemma is needed for the proof of the Main Theorem:

that our translation is denotationally coherent with respect to $\mathcal{D}^+$, which induces a model $\mathcal{D}^{++}$ of PCF++, and that the operational semantics of PCF++ terms is sound in $\mathcal{D}^{++}$. Of course, by the main theorem of this paper, we can also get transfer of computational adequacy. Therefore, we would be able to neatly concentrate in the axiomatization of $\mathcal{T}$ all the conditions needed by a "good" model of PCF+ in order to become a model of PCF++ in accordance to our paradigm.

An intriguing question is whether $c^* = c'$ will turn out to be more than an r.e. statement, whether it is actually decidable? In other words, is full PCF+ computation required in order to systematically disentangle the coercions we introduce?

Finally, we should restate that we expect that the results of this paper generalize to more complicated type disciplines (Fun, Quest, *etc.*) and that analogs can be shown for call-by-name operational semantics.

5 Acknowledgements.

We thank Albert Meyer for inspiring us to examine operational issues surrounding calculi with multiple inheritance. We have had helpful discussions about our draft version of this paper with Hans Boehm, Luca Cardelli, Pierre-Louis Curien, Matthias Felleisen, Dave MacQueen, Jose Meseguer, John Mitchell and Mitch Wand.

Breazu-Tannen's research was partially supported by ONR Grant NOOO14-88-K-0634. Gunter's research was partially supported by ONR Grant NOOO14-88-K-0557. Breazu-Tannen and Gunter were also partially supported by ARO Grant DAAG29-84-K-0061. 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

[BCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explicit coercion (preliminary report). In R. Parikh, editor, *Logic in Computer Science*, pages 112–134, IEEE Computer Society, June 1989.

[BCGS90] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. University of Pennsylvania, Department of Computer and Information Science technical report number MS-CIS-89-01. Journal version of [BCGS89] submitted to *Information and Computation.*

[Car89] L. Cardelli. *Typeful programming.* Research Report 45, DEC Systems, Palo Alto, May 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Mey88] A. R. Meyer. Semantical paradigms: notes for an invited lecture. In Y. Gurevich, editor, *Logic in Computer Science*, pages 236–253, IEEE Computer Society, July 1988.

[Mog88] E. Moggi. *The Partial Lambda-Calculus.* PhD thesis, University of Edinburgh, 1988.

NETS AS TENSOR THEORIES*

Carl Gunter *Vijay Gehlot* [†]

University of Pennsylvania
Department of Computer and Information Sciences
Philadelphia, PA 19104 U.S.A.

October 1989

Abstract

This report is intended to describe and motivate a relationship between a class of nets and the fragment of linear logic built from the tensor connective. In this fragment of linear logic a net may be represented as a *theory* and a computation on a net as a *proof.* A rigorous translation is described and a soundness and completeness theorem is stated. The translation suggests connections between concepts from concurrency such as causal dependency and concepts from proof theory such as cut elimination. The main result of this report is a "cut reduction" theorem which establishes that any proof of a sequent can be transformed into another proof of the same sequent with the property that all cuts are "essential". A net-theoretic reading of this result tells that unnecessary dependencies from a computation can be eliminated resulting in a maximally concurrent computation. We note that it is possible to interpret proofs as arrows in the strictly symmetric strict monoidal category freely generated by a net and establish soundness of our proof reduction rules under this interpretation. Finally, we discuss how other linear connectives may be related to the concepts of internal and external choice.

1 Introduction

In this paper we explore the idea of describing the operational semantics of a net (the so-called "token game") in proof-theoretic terms. Under our approach, a net will correspond to a logical theory, and the token games on the net will be represented as proof trees in the "logic" of the net. This correspondence reveals an interesting relationship between concepts of proof theory (such as cut elimination) and fundamental concepts in concurrency (such as causal dependency) as they are

[*] This is an extended and revised version of the preliminary report that appeared in: **Application and Theory of Petri Nets,** edited by G. De Michelis, June 1989, pp. 174-191.

[†] Research of both authors is supported by Office of Naval Research Grant N00014-88-K-0557. Electronic mail addresses for the authors are gunter@linc.cis.upenn.edu and gehlot@linc.cis.upenn.edu

Structural Rules

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,(\text{Exchange}) \qquad \frac{}{A \vdash A}\,(\text{Identity}) \qquad \frac{\Gamma \vdash A \qquad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\,(\text{Cut})$$

Logical Rules

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,(\otimes\text{R}) \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\,(\otimes\text{L})$$

Figure 1: Structural and logical rules for a fragment of linear logic.

$\Gamma \vdash A$ is *provable in* T if $\Gamma \vdash A$ is in $\mathsf{Th}(T)$. We say that $\Gamma \vdash A$ is provable if it is in $\mathsf{Th}(\emptyset)$. Let us say that a pair $A \vdash\dashv B$ is provable if $A \vdash B$ and $B \vdash A$ are both provable. It is not hard to see from these axioms that the tensor connective is associative and commutative:

Proposition 1 *For any A, B, C, the sequents $A \otimes B \vdash\dashv B \otimes A$ and $(A \otimes B) \otimes C \vdash\dashv A \otimes (B \otimes C)$ are provable.* ▮

However, the tensor connective is *not* absorptive; for example, the sequent $A \otimes A \vdash A$ is not provable. It is therefore possible to think of a tensor formula as a *multi-set* (or "bag") of propositional atoms. Given a tensor formula A, let $\mathsf{M}(A)$ be the multi-set of propositional atoms determined by A. It follows from the proposition that tensor formulae A and B such that $\mathsf{M}(A) = \mathsf{M}(B)$ are equivalent, *i.e.* $A \vdash\dashv B$. Moreover, sequents $\Gamma \vdash A$ and $\Delta \vdash A$ are equivalent in the sense that each can be derived from the other if the lists Γ and Δ determine the same multi-set of propositions. For this reason, we will treat sequents as pairs $\Gamma \vdash A$ where Γ is a multi-set.

For the purposes of this paper, a *net* N is a set S_N of *places* together with a set T_N of pairs of multi-sets over S_N. A pair $t = ({}^{\cdot}t, t^{\cdot}) \in N$ is called a *transition* of the net with *pre-condition* ${}^{\cdot}t$ and *post-condition* $t^{\cdot}$. Of course, this is only one of the many flavors of nets that have been studied in the rich literature on such structures. Nets, as defined here, are similar to place/transition-systems as defined, for example, in [15]. However, our notion of net has less structure since there are no capacities and a transition is uniquely determined by its pre and post conditions. Moreover, a net in our sense does not have a specified initial marking. One of the appealing characteristics of nets is the way they lend themselves to pictorial representation. For example, the net N_0 consisting of the pairs $(\{A\}, \{B, B, C\})$ and $(\{B\}, \{A\})$ is pictured as a labelled graph in Figure 2.

Before we offer a technical definition of just how a net determines a theory, we will attempt to motivate the basic idea by means of examples. Consider the net N_1 pictured in Figure 3. In this net, if we are given a token on the condition A, then it is possible to fire the event r. Firing this event, exhausts the token on A but provides a token on B. Logically, let us read the event r as an axiom $A \vdash B$ meaning "from A it is possible to obtain B." Similar ideas apply to the events s, t and u which we may read as $B \vdash D$ and $A \vdash C$ and $C \vdash E$ respectively. Now, event v requires a

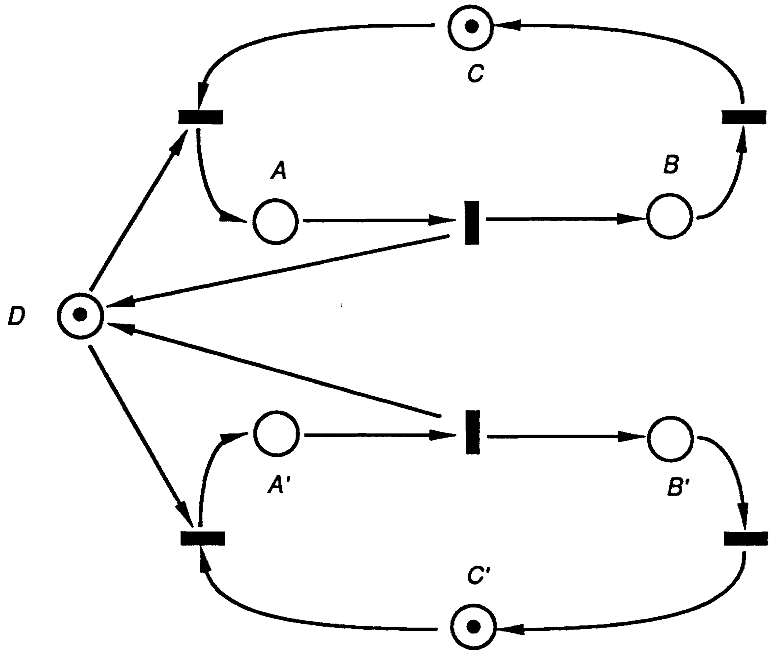

Figure 4: A net N_2 with a critical region.

pepsi and \$1 to buy a coke, then I can't expect to use \$1 to buy both a pepsi *and* a coke. Of course, one can also write the conjunction rule as

$$\frac{A \vdash D \qquad A \vdash E}{A, A \vdash D \wedge E}$$

but this only begs the issue, since some instance of the thinning rule:

$$\frac{\Gamma, X, X \vdash Y}{\Gamma, X \vdash Y}$$

would be used at a later step in the proof to remove the second copy of A and this rule is just as suspect as the earlier version of the conjunction rule. To deal with this problem, one needs a logic in which the thinning rule is omitted and the second of the conjunction rules is used for the "and" connective that we have in mind.

The proper rules are those given in Figure 1 for the linear logic tensor connective $\otimes$. These rules keep track of the resources as needed. In linear logic, the sequent $A \vdash F$ is *not* provable in T_1. However, it is possible to check that $A, A \vdash F$ *is* provable in T_1, as we expect it should be. There are, in fact, several proofs of $A, A \vdash F$ in T_1; three of these are listed in Figure 5 (on page 7). We will come back to these proofs later to discuss how they relate to the net token games that move a token from the marking A, A to the marking F.

To give a slightly larger example, which we hope will suffice in giving the reader the general idea, consider the net N_2 in Figure 4. This net corresponds to the tensor theory T_2 with the following logic involves expanding our discussion to a larger fragment of the calculus. Since rules from $\mathcal{L}(N)$ may be used arbitrarily often, they must be represented as linear logic propositions using the "of course" operator, written $!A$. (Given a linear proposition A, the proposition $!A$ represents the "pure propositional content" of A. In the current context we may think of it as an unlimited resource of A's.) Linear propositional logic with the ! operator is not known to be decidable. The result above suggests that the decision procedure for this calculus, if it exists, will not be easy to find.

Proof 1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A\vdash A \quad A\vdash B}{A,A\vdash A\otimes B}{\scriptstyle\otimes\mathrm{R}} \quad \dfrac{\dfrac{A\vdash C \quad B\vdash B}{A,B\vdash C\otimes B}{\scriptstyle\otimes\mathrm{R}}}{A\otimes B\vdash C\otimes B}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash C\otimes B}{\scriptstyle\mathit{Cut}} \quad \dfrac{\dfrac{B\vdash D \quad C\vdash C}{B,C\vdash D\otimes C}{\scriptstyle\otimes\mathrm{R}}}{C\otimes B\vdash D\otimes C}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes C}{\scriptstyle\mathit{Cut}} \quad \dfrac{\dfrac{D\vdash D \quad C\vdash E}{D,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{D\otimes C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes E}{\scriptstyle\mathit{Cut}} \quad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle\mathit{Cut}}$$

Proof 2.

$$\dfrac{\dfrac{\dfrac{A\vdash B \quad A\vdash C}{A,A\vdash B\otimes C}{\scriptstyle\otimes\mathrm{R}} \quad \dfrac{\dfrac{B\vdash D \quad C\vdash E}{B,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{B\otimes C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes E}{\scriptstyle\mathit{Cut}} \quad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle\mathit{Cut}}$$

Proof 3.

$$\dfrac{\dfrac{\dfrac{A\vdash B \quad B\vdash D}{A\vdash D}{\scriptstyle\mathit{Cut}} \quad \dfrac{A\vdash C \quad C\vdash E}{A\vdash E}{\scriptstyle\mathit{Cut}}}{A,A\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}} \quad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle\mathit{Cut}}$$

Figure 5: Three proofs that $A, A \vdash F$.

3 Proofs as Computations

Let us return now to our discussion of the net N_1 in Figure 3 (on page 4). This net displays some of the intuitive representations of concepts which have made nets an appealing model for both theoreticians and practitioners. The events r and t "compete" for the resource A and the events s and u are capable of running concurrently if they have the necessary resources B and C. There is a causal dependency between r and s: if r fires then s will be enabled. A similar dependency holds between t and u. If there is a line of computation which passes through r, s and another which equational axioms can be eliminated. However, the "maximally concurrent" proof we desire cannot be obtained by a straight-forward translation of these ideas. Instead, it is necessary to rely on other intuitions about the correct forms.

4 Cut reduction.

In this section, we formalize the concepts intuitively discussed in the previous section. Our goal is to demonstrate a set of rewrite rules for transforming a given proof into a "maximally concurrent" proof of the same sequent. We begin by defining essential cuts and then state and prove the cut reduction theorem. The proof is based on giving a finite set of proof reduction rules which is shown to be strongly normalizing.

Definition 1 An instance of the cut rule in a proof is *trivial* if at least one of the premisses is an axiom of the form $A \vdash A$.

Definition 2 An instance of a cut rule in a proof is called *essential* if it is non-trivial and has the form

$$\frac{\Gamma \vdash A \qquad A \vdash B}{\Gamma \vdash B}\,{\scriptstyle Cut}$$

where A is a netformula.

Theorem 4 (Cut-Reduction) *Given a net N and its associated deductive system $\mathcal{L}(N)$. If a sequent $\Gamma \vdash A$ is provable in $\mathcal{L}(N)$, then there is a proof of this sequent in $\mathcal{L}(N)$ such that all cuts are essential.*

Intuitively, essential cuts seem to capture dependencies exactly as dictated by the underlying net. A proof is *cut-reduced* if all instances of cuts in it are essential.

We will give a collection of rewrite rules for proofs and show the existence of a normalizing sequence. We will then strengthen this result by establishing that the set of reduction rules is strongly normalizing . The theorem above will immediately follow from the proposition that every normal proof is cut-reduced.

Remark: Prawitz [14] distinguishes "normal form theorem", "normalization theorem", and "strong normalization theorem". In his terminology then, our cut-reduction theorem is a normal form theorem, the second theorem will be a normalization theorem, and the last one will be a strong normalization theorem.

We begin by enumerating transformations on proofs. Assume that a proof $\mathcal{P}$ ends with an inessential cut, *i.e.* it has the following form:

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta,C,D,A\vdash B}{\Delta,C\otimes D,A\vdash A}\,{\scriptstyle \otimes\mathrm{L}}}{\Gamma,\Delta,C\otimes D\vdash B}\,{\scriptstyle \mathit{Cut}{:}A} \quad\Rightarrow\quad \dfrac{\dfrac{\Gamma\vdash A \qquad \Delta,C,D,A\vdash B}{\Gamma,\Delta,C,D\vdash B}\,{\scriptstyle \mathit{Cut}{:}A}}{\Gamma,\Delta,C\otimes D\vdash B}\,{\scriptstyle \otimes\mathrm{L}}$$

2.2.2 The last rule is a ⊗R. We distinguish following two cases.

2.2.2.1 Cut formula A in upper left sequent of the last rule of $\mathcal{P}''$.

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta',A\vdash B \qquad \Delta''\vdash C}{\Delta',\Delta'',A\vdash B\otimes C}\,{\scriptstyle \otimes\mathrm{R}}}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,{\scriptstyle \mathit{Cut}{:}A} \quad\Rightarrow\quad \dfrac{\dfrac{\Gamma\vdash A \qquad \Delta',A\vdash B}{\Gamma,\Delta'\vdash B}\,{\scriptstyle \mathit{Cut}{:}A} \qquad \Delta''\vdash C}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,{\scriptstyle \otimes\mathrm{R}}$$

2.2.2.2 Cut formula A in upper right sequent of the last rule of $\mathcal{P}''$.

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta'\vdash B \qquad \Delta'',A\vdash C}{\Delta',\Delta'',A\vdash B\otimes C}\,{\scriptstyle \otimes\mathrm{R}}}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,{\scriptstyle \mathit{Cut}{:}A} \quad\Rightarrow\quad \dfrac{\Delta'\vdash B \qquad \dfrac{\Gamma\vdash A \qquad \Delta'',A\vdash C}{\Gamma,\Delta''\vdash C}\,{\scriptstyle \mathit{Cut}{:}A}}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,{\scriptstyle \otimes\mathrm{R}}$$

2.2.3 The last rule of $\mathcal{P}''$ is an essential cut. In this case, the cut formula cannot come from the upper right sequent of the essential cut above. Thus we have only one case to consider.

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta',A\vdash B \qquad B\vdash C}{\Delta',A\vdash C}\,{\scriptstyle \mathit{Cut}{:}B}}{\Gamma,\Delta'\vdash C}\,{\scriptstyle \mathit{Cut}{:}A}$$

$$\Rightarrow\qquad \dfrac{\dfrac{\Gamma\vdash A \qquad \Delta',A\vdash B}{\Gamma,\Delta'\vdash B}\,{\scriptstyle \mathit{Cut}{:}A} \qquad B\vdash C}{\Gamma,\Delta'\vdash C}\,{\scriptstyle \mathit{Cut}{:}B}$$

Note once again that B belongs to some netaxiom in the two cases above.

3. *Logical.* This is the case where the cut formula is the main formula of a logical rule in both $\mathcal{P}'$ and $\mathcal{P}''$ and is introduced only by this instance of the rule. The transformation in this case depends on the outermost logical symbol of the cut formula and since we only have one logical connective, there is only one case to consider here.

$$\dfrac{\dfrac{\Gamma'\vdash A_1 \qquad \Gamma''\vdash A_2}{\Gamma',\Gamma''\vdash A_1\otimes A_2}\,{\scriptstyle \otimes\mathrm{R}} \qquad \dfrac{A_1,A_2,\Delta\vdash B}{A_1\otimes A_2,\Delta\vdash B}\,{\scriptstyle \otimes\mathrm{L}}}{\Gamma',\Gamma'',\Delta\vdash B}\,{\scriptstyle \mathit{Cut}{:}A_1\otimes A_2}$$

$$\Rightarrow\qquad \dfrac{\Gamma''\vdash A_2 \qquad \dfrac{\Gamma'\vdash A_1 \qquad A_1,A_2,\Delta\vdash B}{\Gamma',A_2,\Delta\vdash B}\,{\scriptstyle \mathit{Cut}{:}A_1}}{\Gamma'',\Gamma',\Delta\vdash B}\,{\scriptstyle \mathit{Cut}{:}A_2}$$

The proof of the theorem then immediately follows from the above lemma by an easy induction on the number of inessential cuts appearing in a proof. In any proof consider an inessential cut above whose lower sequent no inessential cuts appear; thus satisfying the condition of the lemma. According to the lemma this (sub) proof can be transformed into another (equivalent) proof which does not contain this cut. In doing so, rest of the proof remains unchanged. We get a cut-reduced (equivalent) proof by repeating this process until all the inessential cuts have been eliminated.
Proof: (of the main lemma) Easy induction on the number of nodes in a proof satisfying the condition of the lemma. ▌

The following is now immediate.

Theorem 8 *Let $\mathcal{P}$ be a proof. Then there exists a sequence of reductions such that $\mathcal{P} \Rightarrow^* \mathcal{P}'$, and $\mathcal{P}'$ is in normal form.* ▌

The following definition will be used in the proof of our next theorem.

Definition 4 The *grade g* of a formula A is the number of $\otimes$ contained in A. The grade of an inessential cut is the grade of its cut formula.

Thus, by the definition above, grade of an essential cut is 0.

Theorem 9 (Strong Normalization) *There is no infinite reduction sequence beginning with any proof $\mathcal{P}'$.*

Proof: We define a measure on proofs and show that each one step transformation reduces this measure.

Let the *complexity* of a proof be a pair (a, b), where

- $a =$ sum of the grade g of cut formulas of all inessential cuts in the proof.
- $b =$ sum of the nodes above all inessential cuts (including the premisses and conclusion of the cut).

Clearly, a cut-reduced proof has complexity (0,0).

Now consider the three (main) classes of the transformations above. It is easy to see that application of these transformations in each case to a proof reduces its complexity.

Axiom: Both a and b are reduced.

Permutation: b is reduced keeping a the same.

Logical: a is reduced.

Thus, all reduction sequences terminate. ▌

In Appendix A we have written out how the rewriting works on Proof 1 and 2 in Figure 5.

where $f : \Gamma \rightarrow A = \mathbf{I}(\overset{\Pi_1}{\Gamma \vdash A})$ and $g : A \otimes \Delta \rightarrow B = \mathbf{I}(\overset{\Pi_2}{A, \Delta \vdash B})$.

Proposition 10 *The proof reduction rules are sound with respect to the interpretation above.*

Proof: We just consider an illustrative case here. Consider the reduction rule 2.2.2.1. The function **I** yields an arrow corresponding to the left hand side as follows. Let $f : \Gamma \rightarrow A$, $g : \Delta' \otimes A \rightarrow B$, and $h : \Delta'' \rightarrow C$. Then we have:

$$\begin{aligned}
&(f \otimes i_{\Delta' \otimes \Delta''}) \circ (g \otimes h) : \Gamma \otimes (\Delta' \otimes \Delta'') \rightarrow B \otimes C \\
&= (f \otimes (i_{\Delta'} \otimes i_{\Delta''})) \circ (g \otimes h) \\
&= ((f \otimes i_{\Delta'}) \otimes i_{\Delta''}) \circ (g \otimes h) \\
&= ((f \otimes i_{\Delta'}) \circ g) \otimes (i_{\Delta''} \circ h) \\
&= ((f \otimes i_{\Delta'}) \circ g) \otimes h \\
&= \mathbf{I}(\text{right hand side}) \ \blacksquare
\end{aligned}$$

In view of the above proposition and the strong normalization theorem, the following is immediate.

Corollary 11 *Every proof reduces to a unique normal form modulo the interpretation.* ▮

It has long been argued by proof theorists that a notion of equivalence of proofs based on mere provability is too extensional and inadequate. But the question of the right notion of equivalence of proofs still remains open. Prawitz [14], for the system of Natural Deduction and his set of reduction rules, conjectured that two derivations represent the same proof if and only if they reduce to the same normal form. Now in view of corollary 11 above we may say something similar for the identification of the derivations in a tensor theory. However, it seems that such an identification does not quite capture the intuitive sense of equivalence (based on processes) that we have in mind for net computations and is still too extensional. For example, proof 2 and proof 3 of section 3 would be identified as the corresponding arrows are equal because $\_\otimes\_$ is a bifunctor. However, the process interpretation that we have in mind should not result in such an identification. Thus the sense in which proof 3 is not equivalent to proof 2 (and in fact better) is lost in the denotational view that we have presented in this section. We are currently looking at how to attach such intensional interpretations to proofs in our setting. We have made some partial progress towards this, though mostly via some *ad hoc* means.

6 Choice Situations

We have so far restricted attention to a rather small fragment of linear logic because this fragment is already sufficient to illustrate several important concepts that suggest interesting relationships

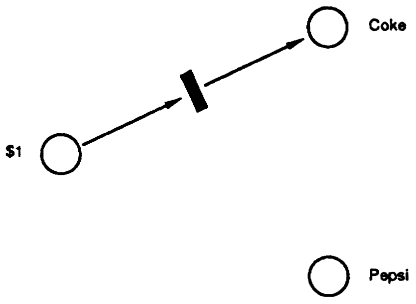

Figure 7: Coca Cola implements $\$1 \vdash \text{coke} \oplus \text{pepsi}$.

affairs with the formula $(D \otimes C)\,\&(A \otimes E)$. Here is a proof of the proper statment:

$$\dfrac{\dfrac{\dfrac{A \otimes B \vdash D \qquad C \vdash C}{A \otimes B, C \vdash D \otimes C}\,{\otimes}\mathrm{R}}{A \otimes B \otimes C \vdash D \otimes C}\,{\otimes}\mathrm{L} \qquad \dfrac{\dfrac{A \vdash A \qquad B \otimes C \vdash E}{A, B \otimes C \vdash A \otimes E}\,{\otimes}\mathrm{R}}{A \otimes B \otimes C \vdash A \otimes E}\,{\otimes}\mathrm{L}}{A \otimes B \otimes C \vdash (D \otimes C)\,\&(A \otimes E)}\,\&\,\mathrm{R}$$

It is our feeling that the *direct product* operator captures a form of *external* choice. On the other hand, the *linear disjunction* captures a concept of *internal* choice. Given two linear propositions A and B, one proves the linear disjunction $A \oplus B$ of A and B from hypotheses Γ by using one of the following rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}$$

In other words, the resource $A \oplus B$ can be obtained from Γ just in case *either* A *or* B can be. On the other hand, if one wishes to obtain C from Γ and resource $A \oplus B$, then it must be shown that C can be obtained from *both* A *and* B. The rule is

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}$$

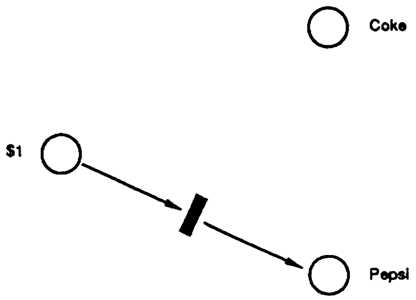

Figure 8: Pepsi Cola implements $\$1 \vdash \text{coke} \oplus \text{pepsi}$.

The internal/external distinction can be illustrated by a simple example which takes linear propositions as a specification language. Let us assume that we wish to contract a vendor to build earlier the unary operator ! which represents an unlimited resource. This operator plays a subtle role in the theory we have exposited; work of Carolyn Brown [4] provides helpful insight. All of the linear logic connectives seem to have their own significance in terms of computation on nets. (We have included a list of some of the rules of linear logic in Appendix B.) Work on the exploitation of these ideas is likely to be a profitable for the study of both concurrency and proof theory.

References

[1] A. Asperti. A logic for concurrency (extended abstract). Unpublished manuscript.

[2] A. Asperti, G. L. Ferrari, and R. Gorrieri. Implicative formulae in the "proofs as computations" analogy. To appear P. Hudak, editor, *Principles of Programming Languages*, pages ??–??, ACM, 1990.

[3] A. Avron. The semantics and proof theory of linear logic. *Theoretical Computer Science*, 57:161–184, 1988.

[4] C. Brown. Relating Petri nets to formulae of linear logic. Unpublished manuscript.

[5] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing net computations and processes. In R. Parikh, editor, *Logic in Computer Science*, pages 175–185, IEEE Computer Society, 1989.

[6] H. J. Genrich and E. Stankiewicz-Wiechno. A dictionary of some basic notions of net theory. In W. Brauer, editor, *Net Theory and Applications*, pages 519–535, *Lecture Notes in Computer Science vol. 84*, Springer-Verlag, 1980.

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

[8] J. Y. Girard. *Proofs and Types.* Cambridge University Press, 1989.

[9] J. Hopcroft and J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8:135–159, 1979.

[10] S. R. Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proc. 14th Annual ACM Symp. on Theory of Computing*, pages 267–281, 1982.

[11] N. Martí-Oliet and J. Meseguer. *From Petri nets to linear logic.* Research Report SRI-CSL-89-4, SRI International, Menlo Park, March 1989.

[12] E. W. Mayr. An algorithm for general Petri net reachability problem. In *Proc. 13th Annual ACM Symp. on Theory of Computing*, pages 238–246, 1981.

A Sample Proof Transformation

We give some examples of cut-reduction below. At each step of the reduction, the inessential cut to which a reduction rule is applied is denoted $\boxed{Cut}$. Other choices of inessential cuts, if any, at a step to which a rule could have been applied are denoted $\overline{Cut}$. Remaining inessential cuts are denoted $\underline{Cut}$.

Example 1

$$\dfrac{\dfrac{\dfrac{A\vdash B \qquad A\vdash C}{A,A\vdash B\otimes C}\otimes\mathrm{R} \qquad \dfrac{\dfrac{B\vdash D \qquad C\vdash E}{B,C\vdash D\otimes E}\otimes\mathrm{R}}{B\otimes C\vdash D\otimes E}\otimes\mathrm{L}}{A,A\vdash D\otimes E}\boxed{Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}Cut$$

$$\xRightarrow{3}\quad \dfrac{\dfrac{A\vdash C \qquad \dfrac{A\vdash B \qquad \dfrac{B\vdash D \qquad C\vdash E}{B,C\vdash D\otimes E}\otimes\mathrm{R}}{A,C\vdash D\otimes E}\boxed{Cut}}{A,A\vdash D\otimes E}\underline{Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}Cut$$

$$\xRightarrow{2.2.2.1}\quad \dfrac{\dfrac{A\vdash C \qquad \dfrac{\dfrac{A\vdash B \qquad B\vdash D}{A\vdash D}Cut \qquad C\vdash E}{A,C\vdash D\otimes E}\otimes\mathrm{R}}{A,A\vdash D\otimes E}\boxed{Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}Cut$$

$$\xRightarrow{2.2.2.2}\quad \dfrac{\dfrac{\dfrac{A\vdash B \qquad B\vdash D}{A\vdash D}Cut \qquad \dfrac{A\vdash C \qquad C\vdash E}{A\vdash E}Cut}{A,A\vdash D\otimes E}\otimes\mathrm{R} \qquad D\otimes E\vdash F}{A,A\vdash F}Cut$$

Example 2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A\vdash A \qquad A\vdash B}{A,A\vdash A\otimes B}\otimes\mathrm{R} \qquad \dfrac{\dfrac{A\vdash C \qquad B\vdash B}{A,B\vdash C\otimes B}\otimes\mathrm{R}}{A\otimes B\vdash C\otimes B}\otimes\mathrm{L}}{A,A\vdash C\otimes B}\boxed{Cut} \qquad \dfrac{\dfrac{B\vdash D \qquad C\vdash C}{B,C\vdash D\otimes C}\otimes\mathrm{R}}{C\otimes B\vdash D\otimes C}\otimes\mathrm{L}}{A,A\vdash D\otimes C}\underline{Cut} \qquad \dfrac{\dfrac{D\vdash D \qquad C\vdash E}{D,C\vdash D\otimes E}\otimes\mathrm{R}}{D\otimes C\vdash D\otimes E}\otimes\mathrm{L}}{A,A\vdash D\otimes E}\underline{Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}Cut$$

$$\xRightarrow{3}\quad \dfrac{\dfrac{\dfrac{\dfrac{A\vdash B \qquad \dfrac{A\vdash A \qquad \dfrac{A\vdash C \qquad B\vdash B}{A,B\vdash C\otimes B}\otimes\mathrm{R}}{A,B\vdash C\otimes B}\boxed{Cut}}{A,A\vdash C\otimes B}\underline{Cut} \qquad \dfrac{\dfrac{B\vdash D \qquad C\vdash C}{B,C\vdash D\otimes C}\otimes\mathrm{R}}{C\otimes B\vdash D\otimes C}\otimes\mathrm{L}}{A,A\vdash D\otimes C}\underline{Cut} \qquad \dfrac{\dfrac{D\vdash D \qquad C\vdash E}{D,C\vdash D\otimes E}\otimes\mathrm{R}}{D\otimes C\vdash D\otimes E}\otimes\mathrm{L}}{A,A\vdash D\otimes E}\underline{Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}Cut$$

$$\xRightarrow{3}\quad \dfrac{\dfrac{\dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}\,\mathit{Cut} \qquad \dfrac{A \vdash C \qquad \dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\,{\scriptstyle\otimes\mathrm{R}}}{A, D \vdash D \otimes E}\,\boxed{\mathit{Cut}}}{A, A \vdash D \otimes E}\,\mathit{Cut} \qquad D \otimes E \vdash F}{A, A \vdash F}\,\mathit{Cut}$$

$$\xRightarrow{2.2.2.2}\quad \dfrac{\dfrac{\dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}\,\mathit{Cut} \qquad \dfrac{D \vdash D \qquad \dfrac{A \vdash C \qquad C \vdash E}{A \vdash E}\,\mathit{Cut}}{A, D \vdash D \otimes E}\,{\scriptstyle\otimes\mathrm{R}}}{A, A \vdash D \otimes E}\,\boxed{\mathit{Cut}} \qquad D \otimes E \vdash F}{A, A \vdash F}\,\mathit{Cut}$$

$$\xRightarrow{2.2.2.2}\quad \dfrac{\dfrac{\dfrac{\dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}\,\mathit{Cut} \qquad D \vdash D}{A \vdash D}\,\boxed{\mathit{Cut}} \qquad \dfrac{A \vdash C \qquad C \vdash E}{A \vdash E}\,\mathit{Cut}}{A, A \vdash D \otimes E}\,{\scriptstyle\otimes\mathrm{R}} \qquad D \otimes E \vdash F}{A, A \vdash F}\,\mathit{Cut}$$

$$\xRightarrow{1.2}\quad \dfrac{\dfrac{\dfrac{A \vdash C \qquad C \vdash E}{A \vdash E}\,\mathit{Cut} \qquad \dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}\,\mathit{Cut}}{A, A \vdash D \otimes E}\,{\scriptstyle\otimes\mathrm{R}} \qquad D \otimes E \vdash F}{A, A \vdash F}\,\mathit{Cut}$$

NORMAL PROCESS REPRESENTATIVES

Vijay Gehlot *Carl Gunter* *

University of Pennsylvania
Department of Computer and Information Sciences
Philadelphia, PA 19104 U.S.A.

October 1989

Abstract

This paper discusses the relevance of a form of cut elimination theorem for linear logic tensor theories to the concept of a process on a Petri net. We base our discussion on two definitions of processes given by Best and Devillers. Their notions of process correspond to equivalence relations on linear logic proofs. It is noted that the cut reduced proofs form a process under the finer of these definitions. Using a strongly normalizing rewrite system and a weak Church-Rosser theorem, we show that each class of the coarser process definition contains exactly one of these finer classes which can therefore be viewed as a canonical or *normal* process representative. We also discuss the relevance of our rewrite rules to the categorical approach of Degano, Meseguer, and Montanari.

1 Introduction

It has often been useful to take ideas from proof theory and look at their computational significance. One very fruitful line of investigation has been the use of the Curry-Howard correspondence—the "propositions as types" idea—as a way of seeing proofs as programs and types as specifications. This correspondence reveals an analogy between cut elimination in systems of natural deduction and the reduction of lambda-terms, thus strongly connecting the study of a central proof-theoretic idea (with a history dating back at least to the 1930's) with a central computational concept in sequential functional programmming.

Another, more recent, line of investigation with a kinship to this sequential theory concerns the relationship between certain kinds of proofs and concepts in *concurrency.* A number of authors have discussed the idea of relating concurrent computations as represented by *Petri nets* to proofs in *linear logic* [7]. One line of research seeks to use the fact that nets give rise to a monoid structure and can therefore be used to model linear logic through the use of a phase semantics [6]. In this way a net can be viewed as a model of the linear connectives in which there is a correspondence between the *truth* of a linear sequent in the model and the *reachability* relation on the net. However, most of the research [8, 9, 1, 4] has focused on the idea that a net may be viewed as a *theory* in a fragment of linear logic (the tensor theory to be precise). In particular, when things are viewed in this way, there is a precise correspondence between *concurrent computations* on a Petri net and linear logic

*Research of both authors is supported by the Office of Naval Research. Electronic mail addresses for the authors are gunter@cis.upenn.edu and gehlot@cis.upenn.edu

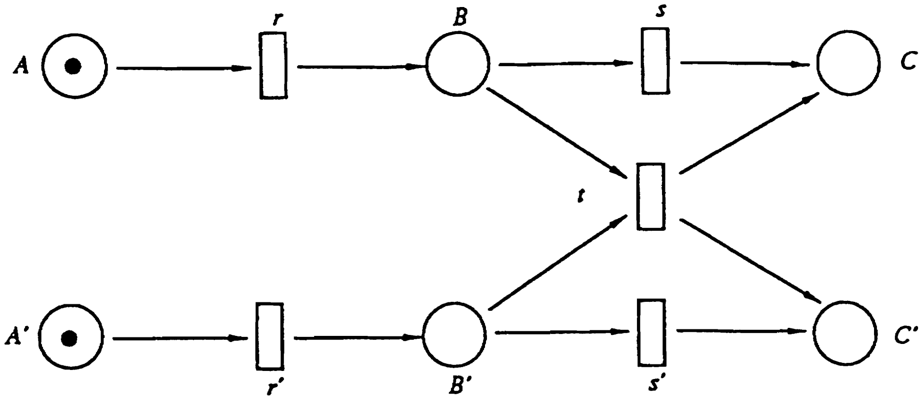

Figure 1: Net N with two processes of type $A \otimes A' \to C \otimes C'$.

respectively. Dynamically, computations proceed by the *firing* of transitions. If transition r fires, for example, then the token is removed from A and placed on its *postcondition* B; the transition r is now disabled since its precondition A is no longer filled. We may also speak of the *concurrent* firing of r and r' in the starting configuration of Figure 1 since there is no dependency between their pre-conditions.

For formal definitions we refer the reader to recent publications in LICS [10, 5]. For this paper we will take it as a working definition that a net (or, to be more precise, a place/transition net) is a set of pairs of multisets over a set S of places. This is really a special case of the definition of a net in [10, 5] where distinct transitions with the same pre and post conditions are permitted, but the restriction simplifies our notation since it avoids the need to label the linear sequents to preserve a precise correspondence between nets and linear tensor theories. For this preliminary discussion, it will be convenient to utilize their categorical treatment of nets and write transitions as arrows in a category with a binary operator $\otimes$ on its objects. In this notation, the transitions in the figure may be viewed as arrows:

$$\begin{array}{ll} r : A \to B & s : B \to C \\ & t : B \otimes B' \to C \otimes C' \\ r' : A \to B & s' : B \to C \end{array}$$

There are two operations on arrows. If $f : X \to Y$ and $g : Y \to Z$, then $f;g : X \to Z$ is the composition of f and g. If $f : X \to X'$ and $g : Y \to Y'$, then $f \otimes g : X \otimes Y \to Y \otimes Y'$ is the *tensor product* of f and g. Starting with the basic transitions, these operations generate a language of computations on the net. Intuitively we read $f;g$ as the *sequentialization* of f and g: "first do f and then do g". We read $f \otimes g$ as the *concurrent* performance of operations f and g: "do f and g at the same time".

Looking again at Figure 1, here are four sample computations of type $A \otimes A' \to C \otimes C'$ on the net N:

$$\begin{array}{rcl} f & = & (r \otimes A');(s \otimes A');(C \otimes r');(C \otimes s') \\ f' & = & (r \otimes r');(s \otimes s') \\ g & = & (r \otimes A');(B \otimes r');t \\ g' & = & (r \otimes r');t \end{array}$$

where the idle transition (identity map) on a place X is written simply as X. Much of the research on nets (and, indeed, concurrency as a whole) has focused on the question of when two computations such as the ones above are "essentially the same". In the case of the computations above, one may well expect to distinguish between processes f and g, for example, since one of these computations formulas of the theory T. An instance of a cut rule is said to be *essential* (in a proof in the theory T) if it is non-trivial and has the form

$$\dfrac{\Gamma\vdash A \qquad A\vdash B}{\Gamma\vdash B}\,\mathit{Cut}$$

where A is a netformula. A proof is said to be in *normal* form if all of its cuts are essential. We will discuss normalization of proofs in the next section.

Definition 1 Let N be a net. The equivalence relation $\mathcal{S}(N)$ on proofs is defined as the smallest equivalence relation satisfying the following equations between proof trees.

$$(1)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C\vdash D} \qquad \overset{\Pi'}{\Delta\vdash E}}{\Gamma,B,C,\Delta\vdash D\otimes E}\,\otimes\mathrm{R}}{\Gamma,B\otimes C,\Delta\vdash D\otimes E}\,\otimes\mathrm{L} \quad=\quad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C\vdash D}}{\Gamma,B\otimes C\vdash D}\,\otimes\mathrm{L} \qquad \overset{\Pi'}{\Delta\vdash E}}{\Gamma,B\otimes C,\Delta\vdash D\otimes E}\,\otimes\mathrm{R}$$

$$(2)\qquad \dfrac{\overset{\Pi}{\Gamma\vdash A} \qquad \overset{\Pi'}{\Delta\vdash B}}{\Gamma,\Delta\vdash A\otimes B}\,\otimes\mathrm{R} \quad=\quad \dfrac{\overset{\Pi'}{\Delta\vdash B} \qquad \overset{\Pi}{\Gamma\vdash A}}{\Delta,\Gamma\vdash B\otimes A}\,\otimes\mathrm{R}$$

$$(3)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C,\Delta,D,E\vdash F}}{\Gamma,B\otimes C,\Delta,D,E\vdash F}\,\otimes\mathrm{L}}{\Gamma,B\otimes C,\Delta,D\otimes E\vdash F}\,\otimes\mathrm{L} \quad=\quad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C,\Delta,D,E\vdash F}}{\Gamma,B,C,\Delta,D\otimes E\vdash F}\,\otimes\mathrm{L}}{\Gamma,B\otimes C,\Delta,D\otimes E\vdash F}\,\otimes\mathrm{L}$$

$$(4)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma\vdash A} \qquad \overset{\Pi'}{\Delta\vdash C}}{\Gamma,\Delta\vdash A\otimes B}\,\otimes\mathrm{R} \qquad \overset{\Pi''}{\Lambda\vdash C}}{\Gamma,\Delta,\Lambda\vdash A\otimes B\otimes C}\,\otimes\mathrm{R} \quad=\quad \dfrac{\overset{\Pi}{\Gamma\vdash A} \qquad \dfrac{\overset{\Pi'}{\Delta\vdash C} \qquad \overset{\Pi''}{\Lambda\vdash C}}{\Delta,\Lambda\vdash B\otimes C}\,\otimes\mathrm{R}}{\Gamma,\Delta,\Lambda\vdash A\otimes B\otimes C}\,\otimes\mathrm{R}$$

$$(5)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma\vdash A} \qquad \overset{\Pi'}{\Delta,A\vdash B}}{\Gamma,\Delta\vdash B}\,\mathit{Cut} \qquad \overset{\Pi''}{\Lambda,B\vdash C}}{\Gamma,\Delta,\Lambda\vdash C}\,\mathit{Cut} \quad=\quad \dfrac{\overset{\Pi}{\Gamma\vdash A} \qquad \dfrac{\overset{\Pi'}{\Delta,A\vdash B} \qquad \overset{\Pi''}{\Lambda,B\vdash C}}{\Delta,A,\Lambda\vdash C}\,\mathit{Cut}}{\Gamma,\Delta,\Lambda\vdash C}\,\mathit{Cut}$$

$$(6)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C\vdash D} \qquad \overset{\Pi'}{D\vdash E}}{\Gamma,B,C\vdash E}\,\text{e-cut}}{\Gamma,B\otimes C\vdash E}\,\otimes\mathrm{L} \quad=\quad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C\vdash D}}{\Gamma,B\otimes C\vdash D}\,\otimes\mathrm{L} \qquad \overset{\Pi'}{D\vdash E}}{\Gamma,B\otimes C\vdash E}\,\text{e-cut}$$

depends on the outermost logical symbol of the cut formula and since we only have one logical connective, there is only one case to consider here.

$$\dfrac{\dfrac{\Gamma' \vdash A_1 \qquad \Gamma'' \vdash A_2}{\Gamma', \Gamma'' \vdash A_1 \otimes A_2}\otimes\mathrm{R} \qquad \dfrac{A_1, A_2, \Delta \vdash B}{A_1 \otimes A_2, \Delta \vdash B}\otimes\mathrm{L}}{\Gamma', \Gamma'', \Delta \vdash B}\, \mathit{Cut}{:}A_1{\otimes}A_2$$

$$\Rightarrow \qquad \dfrac{\Gamma'' \vdash A_2 \qquad \dfrac{\Gamma' \vdash A_1 \qquad A_1, A_2, \Delta \vdash B}{\Gamma', A_2, \Delta \vdash B}\, \mathit{Cut}{:}A_1}{\Gamma'', \Gamma', \Delta \vdash B}\, \mathit{Cut}{:}A_2$$

The following property of the rewrite rules is not difficult to check:

Proposition 1 (Soundness of Rewrite Rules) *The above rewrite rules preserve the $\mathcal{T}$-equivalence of proofs.* ▌

We now show that the these reduction rules are strongly normalizing. We will need the following definition in the proof of the strong normalization theorem.

Definition 2 The *grade g* of a formula A is the number of occurrences of $\otimes$ contained in A. The grade of an inessential cut is the grade of its cut formula.

Thus, by the definition above, grade of an essential cut is 0.

Theorem 2 (Strong Normalization) *There is no infinite reduction sequence beginning with any proof $\mathcal{P}$.*

Proof: Let the *complexity* of a proof be a pair (a, b), where

- $a =$ sum of the grade g of cut formulas of all inessential cuts in the proof.
- $b =$ sum of the nodes above all inessential cuts (including the premisses and conclusion of the cut).

Clearly, a cut-reduced proof has complexity (0,0). We now show that each step of reduction on a proof reduces its complexity. Consider the three classes of the transformations above. It is easy to see that application of these transformations in each case to a proof reduces its complexity.

Axiom: Both a and b are reduced.

Permutation: b is reduced keeping a the same.

Logical: a is reduced.

Thus, all reduction sequences terminate. ▌

In the following section we show that the induced reduction relation on the equivalence classes modulo the relation $\mathcal{S}(N)$ on proofs enjoys the Church-Rosser property. We will then show that every $\mathcal{T}$- equivalence class has a unique normal process representative by showing that the equivalence defined by the reduction relation on $\mathcal{S}$-equivalence classes coincides with the relation $\mathcal{T}$. The $\mathcal{S}$-equivalence class of normal forms will then be the unique process representative of a $\mathcal{T}$-equivalence class.

Now Π' can be reduced to

$$\Sigma' = \dfrac{\dfrac{\dfrac{\overset{\Pi_1}{\Gamma, B, C \vdash A} \qquad \overset{\Pi_2}{\Delta', A \vdash D}}{\Gamma, B, C, \Delta' \vdash D}\,{\scriptstyle Cut} \qquad \overset{\Pi_3}{\Delta'' \vdash E}}{\Gamma, B, C, \Delta', \Delta'' \vdash D \otimes E}\,\otimes\mathrm{R}}{\Gamma, B \otimes C, \Delta', \Delta'' \vdash D \otimes E}\,\otimes\mathrm{L}$$

by another application of a permutation rule and similarly Π'' can be reduced to

$$\Sigma'' = \dfrac{\dfrac{\dfrac{\overset{\Pi_1}{\Gamma, B, C \vdash A} \qquad \overset{\Pi_2}{\Delta', A \vdash D}}{\Gamma, B, C, \Delta' \vdash D}\,{\scriptstyle Cut}}{\Gamma, B \otimes C, \Delta' \vdash D}\,\otimes\mathrm{L} \qquad \overset{\Pi_3}{\Delta'' \vdash E}}{\Gamma, B \otimes C, \Delta', \Delta'' \vdash D \otimes E}\,\otimes\mathrm{R}$$

It is easy to see that $\Sigma' \mathcal{S} \Sigma''$ and thus the required existence of Σ has been shown. ▌

Since the reduction rules are strongly normalizing by Theorem 2, we use the Newman's Lemma (see [2] on page 58) which says that WCR and SN implies CR to conclude that $\Rightarrow^*_{\mathcal{S}}$ satisfies the following diamond property which will be used in the proof of Theorem 4 below.

$$\begin{tikzcd}
& \Pi \arrow[dl, "\Rightarrow^*_{\mathcal{S}}"'] \arrow[dr, "\Rightarrow^*_{\mathcal{S}}"] & \\
\Pi' \arrow[dr, "\Rightarrow^*_{\mathcal{S}}"'] & & \Pi'' \arrow[dl, "\Rightarrow^*_{\mathcal{S}}"] \\
& \Sigma &
\end{tikzcd}$$

Definition 3 A *normal process representative* is an $\mathcal{S}$-equivalence class of normal forms.

Theorem 4 (Unique Process Representative) *Let N be a net. In every $\mathcal{T}$-equivalence class, there is a unique normal process representative.*

Proof: Let Π and Π' be two $\mathcal{S}$-equivalent classes. Define $\Pi \Downarrow \Pi'$ if they both reduce to same normal form modulo the equivalence $\mathcal{S}$. To prove the theorem, we only have to show that $\Pi \Downarrow \Pi'$ iff $\Pi\ \mathcal{T}\Pi'$,*i.e.* the two equivalences coincide. Since the only if part follows from the soundness of the rewrite rules, we are only left with the if part. To prove the if part, we show that if two proofs are equivalent by virtue of equation (7) in section 2, then there is a sequence of reduction $\Rightarrow^*_{\mathcal{S}}$ from one to another. We thus rewrite the left-hand side of the equation (7) to a form which is $\mathcal{S}$-equivalent to the right-hand side of the equation.

$$\dfrac{\dfrac{\overset{\Pi}{\Gamma \vdash A} \qquad \overset{\Pi'}{\Delta \vdash B}}{\Gamma, \Delta \vdash A \otimes B}\,\otimes\mathrm{R} \qquad \dfrac{\dfrac{\overset{\Pi''}{A \vdash C} \qquad \overset{\Pi'''}{B \vdash D}}{A, B \vdash C \otimes D}\,\otimes\mathrm{R}}{A \otimes B \vdash C \otimes D}\,\otimes\mathrm{L}}{\Gamma, \Delta \vdash C \otimes D}\,{\scriptstyle Cut}$$

In other words, this rewrite rule will give a unique "normal" $\mathcal{S}[N]$ arrow for each $\mathcal{T}[N]$ equivalent class of arrows of [5]. In the rewrite above, the left-hand side is always defined whenever the right-hand side is defined but not vice-versa. In particular, subject reduction fails drastically, so the rewrite system must maintain the types of the terms.

References

[1] A. Asperti, G. L. Ferrari, and R. Gorrieri. Implicative formulae in the "proofs as computations" analogy. In P. Hudak, editor, *Principles of Programming Languages*, pages ??–??, ACM, 1990.

[2] H. Barendregt. *The Lambda Calculus: Its syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, revised edition, 1984.

[3] E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55(1):87–136, November 1987.

[4] C. Brown. *Relating Petri nets to formulae of linear logic.* Technical Report ECS-LFCS-89-87, University of Edinburgh, 1989.

[5] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing net computations and processes. In R. Parikh, editor, *Logic in Computer Science*, pages 175–185, IEEE, IEEE Computer Society, June 1989.

[6] U. Engberg and G. Winskel. On linear logic and Petri nets. Unpublished manuscript.

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

[8] C. A. Gunter and V. Gehlot. Nets as tensor theories. (preliminary report). In G. De Michelis, editor, *Applications of Petri Nets*, pages 174–191, 1989. Also University of Pennsylvania, Logic and Computation Report Number 17.

[9] N. Martí-Oliet and J. Meseguer. *From Petri nets to linear logic.* Research Report SRI-CSL-89-4, SRI International, Menlo Park, March 1989.

[10] J. Meseguer and U. Montanari. *Petri Nets Are Monoids.* Research Report SRI-CSL-88-3, SRI International, Menlo Park, January 1988.

Reference Counting as a Computational Interpretation of Linear Logic

(To appear in: **Journal of Functional Programming.**)

Jawahar Chirimar
Department of CIS
University of Pennsylvania
Philadelphia, PA 19104

Carl A. Gunter
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Jon G. Riecke
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

April 1995

Abstract

We develop formal methods for reasoning about memory usage at a level of abstraction suitable for establishing or refuting claims about the potential applications of linear logic for static analysis. In particular, we demonstrate a precise relationship between type correctness for a language based on linear logic and the correctness of a reference-counting interpretation of the primitives that the language draws from the rules for the 'of course' operation. Our semantics is 'low-level' enough to express sharing and copying while still being 'high-level' enough to abstract away from details of memory layout. This enables the formulation and proof of a result describing the possible run-time reference counts of values of linear type.

Contents

Table 1: Translating to a Linear-Logic-Based Language.

```
let fun add x y =
      if x = 0
      then y
      else add (x-1) (y+1)
in add 2 1
end
```

```
let fun add x y =
      share w,z as x in
        if fetch w = 0
        then dispose z, add before y
        else (fetch add)
               (store ((fetch z)-1))
               (y+1)
in add (store 2) 1
end
```

to the variable z. These two variables *share* the value to which x is bound. The dispose primitive indicates that one of these sharing variables, z, is not used in the first branch of the conditional. The primitive store creates a sharable value and fetch obtains a shared value. In our interpretation, the LL-specific operations share and dispose explicitly manage reference counts of the share'able and dispose'able objects that are created and consulted by being store'd and fetch'ed. For the example in Table 1, the occurence of share indicates that two pointers are needed for the value associated with x (so the reference count of the associated value is incremented), but in the then branch of the conditional, one of the pointers is no longer needed (so the reference count of the associated value is decremented).

Analogs to the store and fetch operations are the *delay* and *force* operations that appear in many functional programming languages. In such languages, the delay primitive postpones the evaluation of a term until it is supplied to the force primitive as an argument. When this happens, the value of the delayed term is computed, returned, and memoized for any other applications of force. Abramsky [Abr] has argued that this is a natural way to view the operational semantics of the store and fetch operations of LL; we will follow this approach as well. The dispose primitive has an analog (and namesake) in several programming languages. Typically, an object is disposed by being deallocated; this operation is unsafe because it can lead to dangling pointers. In our LL language the primitive dispose will only deallocate memory if this is safe since its semantics will be to decrement a reference count; deallocation only happens when this count falls to zero. The share command is unique to LL, and its name accurately reflects the way in which it will be interpreted.

One of our primary goals in this paper is to offer an approach for rigorously expressing and proving optimizations obtained by analyzing an LL-based language. In particular, there is an adage that 'linear values have only one pointer to them' or 'linear values can be updated in place'. Wadler [Wad90] has informally observed that these claims must be stated with some care: a reference count of one can be maintained by copying, but this would negate the advantage of in-place updating. Our operational semantics allows us to check the claim rigorously: in particular, we show that linear variables may *fail* to have a count of one in our reference-counting operational semantics, which uses sharing heavily; when this is the case, a linear variable does not have a unique pointer to it and cannot safely be updated in place. The problem arises when a linear variable falls within the scope of an abstraction over a non-linear variable. We express a theorem asserting precisely when the value of a linear variable does indeed maintain a reference count of at

2 Operational Semantics with Memory

Here we give a preview of the operational semantics of the LL-based language by describing the familiar operational semantics of a simple functional language with `store` (delay) and `fetch` (force) operations. We base this preliminary discussion on a language with the grammar

$$\begin{array}{rcl} M & ::= & x \mid (\lambda x.\, M) \mid (M\ M) \mid \\ & & n \mid \mathsf{true} \mid \mathsf{false} \mid (\mathsf{succ}\ M) \mid (\mathsf{pred}\ M) \mid (\mathsf{zero?}\ M) \mid \\ & & (\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M) \mid (\mathsf{fix}\ M) \mid \\ & & (\mathsf{store}\ M) \mid (\mathsf{fetch}\ M) \end{array}$$

where x and n are from primitive syntax classes of variables and numerals respectively. This is a variant of PCF [Sco, Plo77, BGS90] augmented by primitive operations for forcing and delaying evaluations. The expression $(\mathsf{fix}\ M)$ is used for recursive definitions.

The key to providing a semantics for this language is to represent the memoization used in computing the `fetch` primitive so that certain recomputation is avioded. We aim to provide a semantics at a fairly high level of abstraction using what is sometimes known as a *natural semantics* [Des86, Kah87]. Such a semantics has been described in [PS91] using explicit substitution and in [Lau93] through the use of an intermediate representation in which all function applications have variables as arguments. Both of these approaches are appealingly simple but slightly more abstract than we would like for our purposes in this paper. Our own approach, first described in [CGR92], is based on a distinction between an *environment* which is an association of variables with locations and a *store* which is an association of values with locations. Sharing of computation results is achieved through creating multiple references to a location that holds a delayed computation called a *thunk*. When the value delayed in the thunk is needed, it is calculated and memoized for future reference. To define this precisely we must begin with some notation and basic operations for environments, stores, and memory allocation.

Fix an infinite set of locations Loc, with the letter l denoting elements of this set. Let us say that a partial function is finite just in case its domain of definition is finite.

- An **environment** is a finite partial function from variables to locations; ρ denotes an environment, and Env denotes the set of all environments. The notation $\rho(x)$ returns the location associated with variable x in ρ, and to update an environment, we use the notation

  $$(\rho[x \mapsto l])(y) = \begin{cases} l & \text{if } x = y \\ \rho(y) & \text{otherwise.} \end{cases}$$

  The symbol $\emptyset$ denotes the empty environment; we also use $[x \mapsto l]$ as shorthand for $\emptyset[x \mapsto l]$.

- A **value** is a
  - numeral k,
  - boolean b,
  - pointer $\mathsf{susp}(l)$ or $\mathsf{rec}(l, f)$, or
  - closure $\mathsf{closure}(\lambda x.\, M, \rho)$ or $\mathsf{recclosure}(\lambda x.\, M, \rho)$.

  The letter V denotes a value, and Value denotes the set of values.

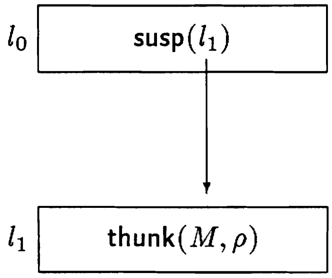

Figure 1: Structure generated by (store M)

```
meminterp((fetch $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp($M$, $\rho$, $\sigma$)
  in  case $\sigma_0(l_0)$ of susp($l_1$) =>
        case $\sigma_0(l_1)$
          of thunk($N$, $\rho'$) =>
               let ($l_2$, $\sigma_1$) = meminterp($N$, $\rho'$, $\sigma_0$)
               in  ($l_2$, $\sigma_1[l_0 \mapsto$ susp($l_2$)])
           | _ => ($l_1$, $\sigma_0$)
```

Note that there is no clause for the case when $\sigma_0(l_0)$ is *not* a suspension. In this case, we assume that the behavior of the interpreter on (fetch M) is undefined. This assumption simplifies the rules, and allows us to ignore what are, in effect, run-time type errors. Our other rules will also ignore run-time type errors.

There is another approach we might have taken to modelling memoization. The interpretation of (store M) allocates a location l_0 that holds a thunk, and returns a location l_1 that holds a pointer susp(l_0) to this location. Could we instead have returned l_0 as the value? That is, the rule could read

```
meminterp'((store $M$), $\rho$, $\sigma$) =  new(thunk($M$, $\rho$), $\sigma$)
```

The answer to this question is instructive, since it relates to the way in which we will represent the distinction between copying and sharing in our model. If we choose to return the location holding the thunk as the value of the store (as opposed to returning a location holding the pointer to this thunk), then this would require a change in the fetch command. In particular, when the location l_2 is obtained there, it would be essential to put the value $\sigma(l_2)$ in the location where the value of the thunk may be sought later:

```
meminterp'((fetch $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp'($M$, $\rho_1$, $\sigma$)
  in  case $\sigma_0(l_0)$
        of thunk($N$, $\rho'$) =>
             let ($l_2$, $\sigma_1$) = meminterp'($N$, $\rho'$, $\sigma_0$)
             in  ($l_0$, $\sigma_1[l_0 \mapsto \sigma_1(l_2)]$)
         | _ => ($l_0$, $\sigma_0$)
```

Note that in the second line from the bottom of the program, the values of l_0 and l_2 in the store are the same and we will say that the value of the thunk has been *copied* from location l_2 to l_0. In the case that $\sigma_1(l_2)$ is a 'small' value, like an integer that occupies only a word of storage, there is

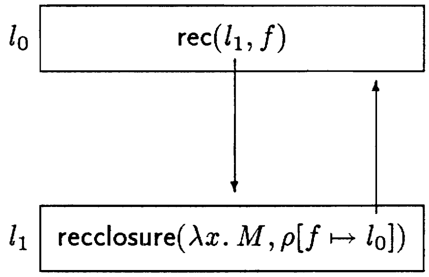

Figure 2: Structure generated by $(\mathsf{fix}\ \lambda f.\, \lambda x.\, M)$

which creates the circular structure in Figure 2. For this language we could create a single cell holding the **recclosure** that looped back to itself; we use two cells, though, since the additional cell holding **rec** will be used in the semantics of the LL-based language to facilitate connections with the type system. We also need here to change the semantics of applications so that if the operator evaluates to a **rec**, the pointer is traced to a **recclosure**; in turn, if the operator evaluates to a **recclosure**, the operator is used in the same way as a **closure**.

In the implementation of actual functional programming languages, a single recursion such as the one above would probably make its recursive calls through a jump instruction. This would be quite difficult to formalize with the source-code-based approach we are using to describe the interpreter. The important thing, for our purposes, is that recursive calls to f do not allocate further memory for the recursive closure. This means that, as far as memory is concerned, there is little difference between implementing the recursion with the jump and implementing it with a circular structure. The cycle created in this way introduces extra complexity into the structure of memory, of course, but the cycles introduced in this way must have precisely the form pictured in Figure 2.

It is easy to provide a clean type system for the language described above. One technical convenience is to tag certain bindings with types (such as the binding occurence in an abstraction $\lambda x : s.\, M$) to ensure that a given program has a unique type derivation. When it is not important for the discussion at hand, we will often drop the tags on bound variables to reduce clutter. The types for the language include ground types **Nat** and **Bool** for numbers and booleans respectively, higher types $(s \to t)$ for functions between s and t, and a unary operation $!s$ for the delayed programs of type s. The typing rules for **store** and **fetch** are introduction and elimination operations respectively:

$$\frac{M : s}{(\mathsf{store}\ M) : !s} \qquad \frac{M : !s}{(\mathsf{fetch}\ M) : s}.$$

These operations will also be found in our LL-based language with essentially the same types.

Table 2: Natural deduction rules and term assignment for linear logic.

$$x : s \vdash x : s$$

$$\frac{\Gamma, x : s \vdash M : t}{\Gamma \vdash (\lambda x : s.\, M) : (s \multimap t)} \qquad \frac{\Gamma \vdash M : (s \multimap t) \qquad \Delta \vdash N : s}{\Gamma, \Delta \vdash (M\ N) : t}$$

$$\frac{\Gamma \vdash M : !s \qquad \Delta \vdash N : t}{\Gamma, \Delta \vdash (\mathsf{dispose}\ M\ \mathsf{before}\ N) : t} \qquad \frac{\Gamma \vdash M : !s \qquad \Delta, x : !s, y : !s \vdash N : t}{\Gamma, \Delta \vdash (\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ N) : t}$$

$$\frac{\Gamma_1 \vdash M_1 : !s_1 \quad \ldots \quad \Gamma_n \vdash M_n : !s_n \qquad x_1 : !s_1, \ldots, x_n : !s_n \vdash N : t}{\Gamma_1, \ldots, \Gamma_n \vdash (\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n) : !t}$$

$$\frac{\Gamma \vdash M : !s}{\Gamma \vdash (\mathsf{fetch}\ M) : s}$$

binds the variables x and y in N. The notation for store can be somewhat unwieldy when writing programs, but most programs involving store bind the variables in the where clause to other variables. Thus, if the free variables of M are $x_1, \ldots, x_n$, then (store M) is shorthand for the expression (store M where $x_1 = x_1, \ldots, x_n = x_n$).

The typing rules for the language appear in Table 2, where the symbols Γ and Δ denote **type assignments**, which are lists of pairs $x_1 : s_1, \ldots, x_n : s_n$, where each x_i is a distinct variable and each s_i is a type. Each of the rules is built on the assumption that all left-hand sides of the $\vdash$ symbol are legal type assignments, *e.g.*, in the rule for typing applications, the type assignments Γ and Δ, which appear concatenated together in the conclusion of the rule, must have disjoint variables. Each type-checking rule corresponds to a proof rule in the ND presentation of linear logic. For instance, the rules for share and dispose essentially correspond to the proof rules generally called *contraction* and *weakening* respectively, while those for store and fetch correspond to the LL rules called *promotion* and *dereliction.* Due to the presence of explicit rules for weakening and contraction—the rules for type-checking dispose and share—one can easily see that the free variables of a well-typed term are *exactly* those contained in the type assignment. A particular note should be taken of the form of the rule for store; this operation puts the value of its body with bindings for its free variables in a location that can be shared by different terms during reduction—the type changes correspondingly from t to $!t$. The construct (fetch M) corresponds to reading the stored value—the type changes from $!t$ to t.

There may be other ND presentations of LL on which one could base a type system. It is our belief that results in this paper are robust with respect to the exact choice of term assignment and type-checking rules. All of the results in this paper—including negative results that say that values of linear type may have more than one pointer to them—hold in the system described in [CGR92], and we expect that they are true for the languages described in [LM92] and [Mac91].

The typing rules for our language are given by combining Tables 2 and 3. Two of these rules deserve special explanation. First, the rule for checking the expression if L then M else N checks both branches in the same type assignment, *i.e.*, the terms M and N must contain the same free variables. This is the only type-checking rule that allows variables to *appear* multiple times; it does not, however, violate the intuition that variables are *used* once, since only one branch will be taken during the execution of the program. Second, the slightly mysterious form of the typing rule for recursions is related to the idea that the formal parameter of a recursive definition must be `share`'d and `dispose`'d if there is to be anything interesting about it. Consider, for example, the rendering of the program of Table 1 into our language:

```
(fix (store λ add : !(!Nat⊸Nat⊸Nat). λx : !Nat. λy : Nat.
   share w, z as x in
      if zero? (fetch w)
      then dispose z before dispose add before y
      else (fetch add) (store (pred (fetch z))) (succ y)))
(store 2) 1
```

(where some liberties have been taken in dropping a few of the parentheses to improve readability). The recursive function *add* being defined gets used only in one of the branches; thus, the recursive call must have a non-linear type.

The definition of the addition function is a prototypical example of how one programs recursive functions in this language. In fact, both the high-level and low-level semantics will only interpret recursions (fix M) where M has the form

$$(\mathsf{store}\ (\lambda f : !s \multimap t.\ \lambda x : s.\ M)\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n).$$

This restriction is closely connected to the restriction on interpreting recursion mentioned in the previous section; the only difference here is the occurrence of the store. As before, this restriction is not essential, but it does simplify the semantic clause for the recursion somewhat without compromising the way programs are generally written.

Natural semantics.

Tables 4 and 5 give a high-level description of an interpreter for our language, written using natural semantics. A natural semantics describes a partial function $\Downarrow$ via proof trees. The notation $M \Downarrow c$, read 'the term M halts at the final result c', is used when there is a proof from the rules with the conclusion being $M \Downarrow c$. The terms at which the interpreter function halts are called **canonical forms**; it is easy to see from the form of the rules that the canonical forms are n, true, false, $(\lambda x.\, M)$, and (store M).

The natural semantics in Tables 4 and 5 describes a *call-by-value* evaluation strategy. That is, operands in applications are evaluated to canonical form before the substitution takes place. A basic property of the semantics is that types are preserved under evaluation:

Theorem 2 *Suppose* $\vdash M : s$ *and* $M \Downarrow c$*, then* $\vdash c : s$.

The proof can be carried out by an easy induction on the height of the proof tree of $M \Downarrow c$.

4 Semantics

The high-level natural semantics is useful as a specification for an intepreter for our language, and for proving facts like Theorem 2. One would not want to implement the semantics directly, however: explicit substitution into terms can be expensive, and one would therefore use some standard representation of terms like closures or graphs in order to perform substitution more efficiently. But there is another problem with the high-level semantics: it does not go very far in providing a computational intuition for the LL primitives in the language. For example, the dispose operation is treated essentially as 'no-op'. As such, there is no apparent relationship between these connectives and memory; indeed, the semantics entirely suppresses the concept of memory.

In order to understand what the constructs of linear logic have to do with memory, we construct a semantics that relates the LL primitives to reference counting. In this semantics, the linear logic primitives dispose and share maintain reference counts. The basic structure of the reference-counting interpreter is the same as the one outlined in Section 3. Environments, values, and storable objects have the same definition as before. Because we now want to maintain reference counts, however, the definition of stores must change. A **store** is now a function

$$\sigma : \mathsf{Loc} \to (\mathrm{N} \times \mathsf{Storable}),$$

where the left part of the returned pair denotes a reference count. Abusing notation, we use $\sigma(l)$ to denote the storable object associated with location l, and $\sigma[l \mapsto S]$ to denote a new store which is the same as σ except at location l, which now holds the storable object S with the reference count of l left unaffected. The reference count of a cell is denoted by $\mathsf{refcount}(l, \sigma)$. The **domain** of a store σ is the set

$$\mathsf{dom}(\sigma) = \{l \in \mathsf{Loc} : \mathsf{refcount}(l, \sigma) \geq 1\}.$$

The change in the definition of 'store' forces an adjustment in the definition of 'allocation relation'. A subset R of the product $(\mathsf{Storable} \times \mathsf{Store}) \times (\mathsf{Loc} \times \mathsf{Store})$ is an **allocation relation** if, for any store σ and storable object S, there is an l' and σ' where $(S, \sigma)\ R\ (l', \sigma')$ and

- $l' \notin \mathsf{dom}(\sigma)$ and $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \cup \{l'\}$;
- for all locations $l \in \mathsf{dom}(\sigma)$, $\sigma(l) = \sigma'(l)$ and $\mathsf{refcount}(l, \sigma) = \mathsf{refcount}(l, \sigma')$; and
- $\sigma'(l') = S$ and $\mathsf{refcount}(l', \sigma') = 1$.

The basic structure underlying a store may be captured abstractly by a graph. Formally, a **graph** is a tuple (V, E, s, t) where V and E are sets of **vertices** and **edges** respectively and s, t are functions from E to V called the **source** and **target** functions respectively. (Note that there may be more than one edge with the same source and target; such 'multiple edge' graphs are sometimes called *multigraphs*.) Given $v \in V$, the **in-degree** of v is the number of elements $e \in E$ such that $t(e) = v$. A vertex v is **reachable** from a vertex v' if $v = v'$ or there is a path between them, that is, there is a list of edges $e_1, \ldots, e_n$ such that $v = s(e_1)$, $v' = t(e_n)$ and $t(e_i) = s(e_{i+1})$.

A **memory graph** $\mathcal{G}$ is a tuple $(V, E, s, t, [\rho_1, \ldots, \rho_n])$ where (V, E, s, t) is a graph together with a list of functions ρ_i such that each ρ_i is a function with a finite domain and with V as its codomain. The functions ρ_i are called the **root set** of the memory graph. Given $v \in V$ and ρ_i,

Definition 5 A state $S = (\bar{l}, \bar{\rho}, \sigma)$ is called **regular**, written $\Re(S)$, provided the following conditions hold:

$\Re$1 S is count-correct.

$\Re$2 $\mathsf{dom}(\sigma)$ is finite.

$\Re$3 For each $l \in \mathsf{dom}(\sigma)$, if $\sigma(l) = \mathsf{thunk}(M, \rho)$, then $\mathsf{refcount}(l, \sigma) = 1$.

$\Re$4 A cycle in the memory graph induced by S arises only in the form of a rec and $\mathsf{recclosure}$ as in Figure 2: that is, it has two nodes l_0 and l_1 such that $\sigma(l_0) = \mathsf{rec}(l_1, f)$ and $\sigma(l_1) = \mathsf{recclosure}(\lambda x.\, M, \rho[f \mapsto l_0])$ for some f, x, M, and ρ.

$\Re$5 For each $l \in \mathsf{dom}(\sigma)$, if $\sigma(l) = \mathsf{thunk}(M, \rho)$, then the domain of ρ is the set of free variables of M, and M is typeable. Similarly, if $\sigma(l) = \mathsf{closure}(\lambda x.\, M, \rho)$ or $\mathsf{recclosure}(\lambda x.\, M, \rho)$, then the domain of ρ is the set of free variables of $\lambda x.\, M$, and $\lambda x.\, M$ is typeable.

Here, a term M is said to be **typeable** if there is some type context Γ and type t such that $\Gamma \vdash M : t$.

It is convenient to abuse notation slightly in denoting states by writing locations, environments, and store without grouping them as in the official definition. For example, $(l_1, l_2, \rho, \sigma, \bar{l}, \bar{\rho})$ should be read as $(l_1 :: l_2 :: \bar{l}, \rho :: \bar{\rho}, \sigma)$ (where :: is the 'cons' operation that puts a datum at the head of a list). There is no chance of confusion so long as the lexical conventions distinguish the parts of the tuple, and the locations and environments are properly ordered from left to right. However, the order of these lists is irrelevant for regularity: if $\Re(\bar{l}, \bar{\rho}, \sigma)$ and $\bar{l}', \bar{\rho}'$ are permutations of $\bar{l}$ and $\bar{\rho}$ respectively, then $\Re(\bar{l}', \bar{\rho}', \sigma)$. We will use this fact without explicit mention.

Basic reference-counting operations.

Our interpreter will need four auxiliary functions to manipulate reference counts. Two of these functions, inc and dec, increment and decrement reference counts. More formally, $\mathsf{inc}(l, \sigma)$ increments the reference count of l and returns the resultant store, while $\mathsf{dec}(l, \sigma)$ decrements the reference count of l and returns the resultant store. The other two operations, $\mathsf{inc\text{-}env}(\rho, \sigma)$ and $\mathsf{dec\text{-}ptrs}(l, \sigma)$, increment or decrement the reference counts of multiple cells. The formal definition of the first of these is

$$\mathsf{inc\text{-}env}(\rho, \sigma) = \begin{cases} \sigma_n, & \text{where the domain of } \rho \text{ is } \{x_1, \ldots, x_n\}, \text{ and} \\ & \quad \sigma_1 = \mathsf{inc}(\rho(x_1), \sigma) \\ & \quad \vdots \\ & \quad \sigma_n = \mathsf{inc}(\rho(x_n), \sigma_{n-1}) \end{cases}$$

In words, $\mathsf{inc\text{-}env}(\rho, \sigma)$ increments the reference counts of the locations in the range of ρ and returns the resultant store. Note that a location's reference count may be incremented more than once by this operation, since two variables x_i, x_j may map to the same location l according to ρ.

The operation $\mathsf{dec\text{-}ptrs}(l, \sigma)$, which also returns an updated store, first decrements the reference count of location l. If the reference count falls to zero, it then recursively decrements the reference counts of all cells pointed to by l. The formal definition appears in Table 6; an example appears in Figure 4 where the left side of Figure 4 (assumed to be part of the graph of the store σ) is transformed into the right side by calling $\mathsf{dec\text{-}ptrs}(l, \sigma)$. The operation $\mathsf{dec\text{-}ptrs}(l, \sigma)$ is the single most complex operation used in the interpreter. Other operations are 'local' to parts of the memory graph and do not require a recursive definition. A key characteristic of our semantics is the fact that $\mathsf{dec\text{-}ptrs}(l,\sigma)$ is only used in the rule for evaluating ($\mathsf{dispose}$ M before N).

The basic laws that capture the relationships maintained by the reference-counting, allocation, and update operations on states are given in Table 7. Most of the laws are proven in the appendix, but we give the proof for the Attenuation Law A1 here to show how the proofs go. Suppose $\Re(l,\bar{l},\bar{\rho},\sigma)$, $\mathsf{refcount}(l,\sigma) = 1$ and $\sigma(l) = \mathsf{closure}(\lambda x.\,N,\rho)$, $\mathsf{recclosure}(\lambda x.\,N,\rho)$, or $\mathsf{thunk}(N,\rho)$. Note first that the state $S' = (\bar{l},\rho,\bar{\rho},\mathsf{dec}(l,\sigma))$ is count-correct: the environment ρ has been placed in the root set, accounting for the edges coming out of the closure or thunk which has now disappeared from the memory graph. Thus, property $\Re1$ holds of state S'. Since $\mathsf{dom}(\sigma) \supseteq \mathsf{dom}(\mathsf{dec}(l,\sigma))$, each of the properties $\Re2$–$\Re5$ follow directly from the hypothesis. Thus, $\Re(S')$. The property is called an "attenuation law" because pointers previously held inside the store are drawn out to the root set.

The next goal is to define an interpreter for the LL-based programming language. To understand the interpreter it is essential to appreciate how the invariants influence its design. We therefore describe the theorem that the interpreter is expected to satisfy, and mingle the proof of the theorem with the definition of the interpreter itself. The interpreter is a function `interp` which takes as its arguments a term M, an environment ρ, and a store σ. It is assumed that the domain of ρ is the set of free variables in M and that the image of ρ is contained in the domain of σ. The result of $\mathtt{interp}(M,\rho,\sigma)$ is a pair (l',σ') where σ' is a store and l' is a location in the domain of σ' such that $\sigma'(l')$ is a value, which can be viewed as the result of the computation. We use a binary infix @ for appending two lists. The theorem is stated as follows:

Theorem 6 *Let $S = (\rho,\sigma,\bar{l},\bar{\rho})$ be a state and suppose M is a typeable term. If $\Re(S)$ and $\mathtt{interp}(M,\rho,\sigma) = (l',\sigma')$, then $\Re(l',\sigma',\bar{l},\bar{\rho})$.*

Moreover, if $\bar{\rho} = \bar{\rho}_1 @ \bar{\rho}_2$, $\bar{l} = \bar{l}_1 @ \bar{l}_2$ and $l \in \mathsf{dom}(\sigma)$ is not reachable from $\rho :: \bar{\rho}_1$ or $\bar{l}_1$ in the memory graph induced by S, then the contents and reference count of l remain unchanged and l is not reachable from $\bar{\rho}_1$ or $l' :: \bar{l}_1$ in the memory graph induced by $(l',\sigma',\bar{l},\bar{\rho})$.

The first part of the theorem says that regularity is preserved under execution of typeable terms. The second part of the theorem expresses what we will call the **reachability property.** The special case of interest says that the evaluation of a program M in environment ρ and store σ does not affect locations in $\mathsf{dom}(\sigma)$ that are not reachable from ρ. The extra complexity of the statement is required to maintain a usable induction hypothesis in the proof of the property. A simplified version of Theorem 6 can be expressed as follows:

Corollary 7 *Suppose M is a closed, typeable term. If $\mathtt{interp}(M,\emptyset,\emptyset) = (l',\sigma')$, then $\Re(l',\sigma')$.*

The assumption that M is typeable is crucial in the proof of the theorem, because untypeable terms may not maintain reference counts correctly. For instance, the term

$$(\lambda x.\,(\mathsf{dispose}\ x\ \mathsf{before}\ x))\ (\mathsf{store}\ 1)$$

would cause a run-time error in the maintenance of reference counts—after the dispose, we would try to access a portion of memory with reference count zero and get a 'dangling pointer' error. This example shows that untypeable terms may cause premature deallocations. Another untypeable term

$$(\lambda x.\,(\mathsf{share}\ y,z\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{dispose}\ y\ \mathsf{before}\ 2)))\ (\mathsf{store}\ 1)$$

causes a 'space leak', *i.e.*, the reference count of the cell holding (store 1) is still greater than zero even though it is garbage at the end of the execution.

Interpreting the linear core.

The proof of Theorem 6 is by induction on the number of calls to the interpreter. The proof proceeds by considering each case for the program to be evaluated.

The interpretation of a variable is obtained by looking up the variable in the environment:

```
(1)    interp($x$, $\rho$, $\sigma$) = ($\rho(x)$, $\sigma$)
```

That the store $(\rho(x), \sigma', \bar{l}, \bar{\rho})$ is regular is a consequence of the Environment Law E because of the assumption that the domain of ρ is $\{x\}$. The reachability condition is clearly satisfied, since the output store is the same as the input store.

To evaluate an abstraction we create a new closure, place it in a new cell, and return the location together with the updated store:

```
(2)    interp($\lambda x.\, P$, $\rho$, $\sigma$) = new(closure($\lambda x.\, P$, $\rho$), $\sigma$)
```

To prove that regularity of the state is preserved, suppose that $(l', \sigma') = \mathsf{new}(\mathsf{closure}(\lambda x.\, P, \rho), \sigma)$, then $\Re(l', \sigma', \bar{l}, \bar{\rho})$ by Allocation Law N2. The reachability condition is satisfied because the output store differs from the input store only by extending it.

Given a term P and an environment ρ whose domain includes the free variables of P, let $\rho \,|\, P$ be the restriction of the environment ρ to the free variables of P. The evaluation of an application is given as follows:

```
(3)    interp(($P$ $Q$), $\rho$, $\sigma$) =
         let ($l_0$, $\sigma_0$) = interp($P$, $\rho \,|\, P$, $\sigma$)
             ($l_1$, $\sigma_1$) = interp($Q$, $\rho \,|\, Q$, $\sigma_0$)
         in  case $\sigma_1(l_0)$ of closure($\lambda x.\, N$, $\rho'$) or recclosure($\lambda x.\, N$, $\rho'$) =>
               if refcount($l_0$, $\sigma_1$) = 1
               then interp($N$, $\rho'[x \mapsto l_1]$, dec($l_0$, $\sigma_1$))
               else interp($N$, $\rho'[x \mapsto l_1]$, inc-env($\rho'$, dec($l_0$, $\sigma_1$)))
```

The reader may compare this rule to the rule for application given in Section 3. The key difference in the semantic clauses is the manipulation of reference counts: in the rule here, a conditional breaks the evaluation of the function body into two cases based on the reference count of the location that holds the value of the operator, and each branch of the conditional performs some reference-counting arithmetic. The resulting semantics clause looks similar to a denotational semantics such as that given in [Hud87] where information about reference counts is included in the semantics clauses. Note that the environment ρ has been split between the two subterms P and Q. The fact that $(P\ Q)$ is typeable implies that $\rho = (\rho \,|\, P) \cup (\rho \,|\, Q)$. In various forms this sort of property will be used repeatedly in the semantic clauses below.

To prove the preservation of regularity of the state for application, we start with the assumption that $\Re(\rho, \sigma, \bar{l}, \bar{\rho})$. This is equivalent to $\Re(\rho \,|\, P, \rho \,|\, Q, \sigma, \bar{l}, \bar{\rho})$. Now $\Re(l_0, \rho \,|\, Q, \sigma_0, \bar{l}, \bar{\rho})$ and $\Re(l_1, l_0, \sigma_1, \bar{l}, \bar{\rho})$ both hold by induction hypothesis (let us abbreviate 'induction hypothesis' as 'IH'). Now, there are two possibilities for the reference count of l_0 in σ_1, either it is equal to one or it is more than one. If $\mathsf{refcount}(l_0, \sigma_1) = 1$, then the first Attenuation Law, A1, says that $\Re(l_1, \rho', \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{\rho})$. By the Environment Law, E, this implies that $\Re(\rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{\rho})$

By IH, we have $\Re(l_0, \sigma_0, \bar{l}, \bar{\rho})$. Suppose $\sigma_0(l_0) = \mathsf{susp}(l_1)$ and $\sigma_0(l_1) = \mathsf{thunk}(R, \rho')$. If $\mathsf{refcount}(l_0, \sigma_0) = 1$, then $\Re(\rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0)), \bar{l}, \bar{\rho})$ by A1 and A2 so we are done by IH. Suppose, on the other hand, that $\mathsf{refcount}(l_0, \sigma_0) \neq 1$. By U2, $\Re(\rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])), \bar{l}, \bar{\rho})$ so $\Re(l_2, \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]), \bar{l}, \bar{\rho})$ by IH and U1; the reachability property is used to ensure the applicability of U1. More specifically, in σ_0 the location l_0 is not reachable from ρ'; thus, it is not reachable from l_2 in σ_1 either, and so $\sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]$ does not create an illegal loop in the memory graph. The cases when $\sigma_0(l_1)$ is a value or $\sigma_0(l_0) = \mathsf{rec}(l_1, f)$ are left to the reader.

The **share** command increments the reference count of a location:

```
(6)   interp((share $x, y$ as $P$ in $Q$),  $\rho$,  $\sigma$) =
        let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho \,|\, P$,  $\sigma$)
        in   interp($Q$,  $(\rho \,|\, Q)[x, y \mapsto l_0]$,  inc($l_0$,  $\sigma_0$))
```

$\Re(l_0, \rho \,|\, Q, \sigma_0, \bar{l}, \bar{\rho})$ by IH, so $\Re(l_0, l_0, \rho \,|\, Q, \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$ by I1. Thus it follows from the Environment Law E that $\Re((\rho \,|\, Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$, so the result follows from IH.

The **dispose** command decrements the reference count of a location. The requires calculating the consequences of possibly removing a node from the memory graph if its reference count of the disposed node falls to 0.

```
(7)   interp((dispose $P$ before $Q$),  $\rho$,  $\sigma$) =
        let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho \,|\, P$,  $\sigma$)
        in   interp($Q$,  $\rho \,|\, Q$,  dec-ptrs($l_0$,  $\sigma_0$))
```

Now, $\Re(l_0, \rho \,|\, Q, \sigma_0, \bar{l}, \bar{\rho})$ by IH, so $\Re(\rho \,|\, Q, \mathsf{dec\text{-}ptrs}(l_0, \sigma_0), \bar{l}, \bar{\rho})$ by D3. The result therefore follows from IH.

Interpreting PCF extensions.

The interpreter evaluates a constant simply by creating a cell holding the value of the constant.

```
(8)   interp($n, \rho, \sigma$) = new($n, \sigma$)

(9)   interp(true, $\rho, \sigma$) = new(true, $\sigma$)

(10)  interp(false, $\rho, \sigma$) = new(false, $\sigma$)
```

That regularity is preserved for these cases follows immediately from N1.

The rules for the arithmetic and boolean operations of PCF mimic the rules of the high-level operational semantics.

```
(11)  interp((succ $P$),  $\rho$,  $\sigma$) =
        let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho$,  $\sigma$)
        in   new($\sigma_0(l_0) + 1$,  dec($l_0$,  $\sigma_0$))

(12)  interp((pred $P$),  $\rho$,  $\sigma$) =
        let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho$,  $\sigma$)
             $n = \sigma_0(l_0)$
        in   if $n = 0$
             then new(0,  dec($l_0$,  $\sigma_0$))
             else new($n - 1$,  dec($l_0$,  $\sigma_0$))
```

5 Properties of the Semantics

In order for the reference-counting interpreter to make sense, it must satisfy a number of invariants and correctness criteria. In this section we describe these precisely.

No space leaks.

As a short example of the kind of property one expects the semantics to satisfy, let us consider how the idea that 'there are no space leaks' can be expressed in our formalism. Given a state $S = (\bar{l}, \bar{\rho}, \sigma)$, we say that a location l is reachable from $(\bar{l}, \bar{\rho})$ if it is reachable in $\mathcal{G}(S)$ from some $l_i \in \bar{l}$ or from some $\rho_j \in \bar{\rho}$. The desired property can now be expressed as follows:

Theorem 8 *Suppose* $(\rho, \sigma, \bar{l}, \bar{\rho})$ *is a regular state such that each* $l \in \mathsf{dom}(\sigma)$ *is reachable from* $(\rho, \bar{l}, \bar{\rho})$*. If* M *is typeable and* $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$*, then every* $l \in \mathsf{dom}(\sigma')$ *is reachable from* $(l', \bar{l}, \bar{\rho})$*.*

The theorem is proved by induction on the number of calls to the interpreter.

Invariance under different allocation relations.

If the design of the interpreter is correct, the exact memory usage pattern should be unimportant to the final answers returned by the interpreter. Since the allocation relation new completely determines memory usage—*i.e.*, which cell (with reference count 0) will be filled next—it should not matter which allocation relation is used. We set this up formally as follows: if f is an allocation relation, let $\mathtt{interp}_f$ be the partial interpreter function defined by using f in the place of new. Recall that the environment and store with empty domains are denoted by $\emptyset$. Then we would like to prove something like the following statement by induction on the number of calls to $\mathtt{interp}_f$:

> Suppose f and g are allocation relations. If $\mathtt{interp}_f(M, \emptyset, \emptyset) = (l_f, \sigma_f)$, then $\mathtt{interp}_g(M, \emptyset, \emptyset) = (l_g, \sigma_g)$. Moreover, if $\sigma_f(l_f) = n$, true, or false, then $\sigma_f(l_f) = \sigma_g(l_g)$.

A naive induction runs afoul, though, since the interpreter can return intermediate results that are neither numbers nor booleans. We therefore need to strengthen the induction hypothesis. If $\mathtt{interp}_f$ returns a closure or suspension, the result returned by $\mathtt{interp}_g$ may not literally be the same: for instance, $\mathtt{interp}_f$ may return a location holding $\mathsf{susp}(l_0)$ and $\mathtt{interp}_g$ may return a location holding $\mathsf{susp}(l_1)$. Nevertheless, these values should be the same up to a renaming of the locations in the domain of the returned store σ'_f.

Formalizing the notion of when two stores are 'equivalent' up to renaming of their locations can be done using the underlying graphs. Two stores are 'equivalent' if their underlying graph representations are isomorphic via some function h, and the values held at the cells are 'equivalent' under h. More formally,

Definition 9 Two states $S = (\bar{l}, \bar{\rho}, \sigma)$ and $S' = (\bar{l'}, \bar{\rho'}, \sigma')$ are **congruent** if there is an isomorphism $h : \mathcal{G}(\sigma) \rightarrow \mathcal{G}(\sigma')$ such that for any $l \in \mathsf{dom}(\sigma)$, $\mathsf{refcount}(l, \sigma) = \mathsf{refcount}(h(l), \sigma')$ and for any $l \in \mathsf{dom}(\sigma)$,

1. For all i, $h(l_i) = l'_i$;

2. For all i, $\mathsf{dom}(\rho_i) = \mathsf{dom}(\rho'_i)$ and for all $x \in \mathsf{dom}(\rho_i)$, $h(\rho_i(x)) = \rho'_i(x)$;

l_g	$\mathsf{closure}(\lambda h.\, \lambda y.\, (\mathsf{share}\ h_1, h_2\ \mathsf{as}\ h\ \mathsf{in}\ h_1(h_2\ y)),\ \emptyset)$
l	$\mathsf{closure}(\lambda f.\, ((g\ f)\ x),\ [g \mapsto l_g,\ x \mapsto l_x])$
l_x	$\mathsf{susp}(l'_x)$
l'_x	$\mathsf{thunk}((f\ \mathsf{true}),\ [f \mapsto l'])$
l'	$\mathsf{closure}(\lambda x.\, x,\ \emptyset)$

Figure 5: Store for Example of the valofcell Operation.

key definition missing here is the definition of 'related values'. One might attempt to extend the statement of the theorem directly—that is, for closed terms, $M \Downarrow c$ iff $\mathtt{interp}(M, \emptyset, \emptyset) = (l', \sigma')$ and $\mathsf{valofcell}(l', \sigma') = c$. While this statement holds for basic values, it *does not* hold for values of other types. The problem arises because the reference-counting interpreter memoizes the results of evaluating under store's whereas the natural semantics does not. For instance, evaluating the term

$$(\lambda x.\, (\mathsf{share}\ y, z\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{if}\ (\mathsf{zero?}\ (\mathsf{fetch}\ y))\ \mathsf{then}\ z\ \mathsf{else}\ z))\ (\mathsf{store}\ (\mathsf{succ}\ 5))$$

in the natural semantics returns the value (store (succ 5)), whereas evaluating the expression in the reference-counting semantics returns the value (after unwinding) (store 6). The proof thus requires relating terms that are 'less evaluated' to terms that are 'more evaluated'.

Definition 12 $M \geq N$, read 'N requires less evaluation than M', iff $M = C[M']$, $N = C[c]$, M' is closed, and $M' \Downarrow c$.

where $C[\,]$ denotes a term with a missing subterm and $C[M']$ the term resulting from using M' for that subterm. Let $\geq^*$ be the reflexive, transitive closure of $\geq$. This relation is necessary in order to express the desired property:

Theorem 13 *Suppose M is typeable,* $\mathsf{dom}(\rho) = FV(M)$, *$M'$ is closed, and* $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$. *Suppose also that* $\mathfrak{R}(\bar{l}', \rho, \bar{\rho}', \sigma)$.

1. *If $M' \Downarrow c$, then* $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$ *and* $c \geq^* \mathsf{valofcell}(l', \sigma')$.

2. *If* $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$, *then* $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$.

The extra assumptions about the state $(\bar{l}', \rho, \bar{\rho}', \sigma)$—namely that it satisfies the invariants above—are used in constructing an execution in the reference-counting interpreter. The proof is deferred to the appendix.

There is some help on this point to be found in the proof theory of linear logic. Note, that the problem with term M in (1) relies on having a term N of type $\mathsf{Nat}\multimap !\mathsf{Nat}$. From the stand-point of linear logic and its translation under the Curry-Howard correspondence, this is a suspicious assumption, however. The proposition $A \multimap !A$ is *not* provable in LL, and the situation illustrated by M runs contrary to proof-theoretic facts about what propositions are moved through 'boxes' in a proof net during cut elimination [Gir87]. This does not directly prove that a static property exists for the LL-based programming language, but it does suggest that there is hope.

To assert the desired property precisely, we will need some more terminology. Let us say that a storable object is **linear** if it is a numeral, boolean, closure, or recclosure and say that it is **non-linear** if it has the form $\mathsf{susp}(l)$, $\mathsf{rec}(l, f)$, or $\mathsf{thunk}(M, \rho)$. We say that a location l is **non-linear in store** σ if $\sigma(l)$ is a non-linear object; similarly, a location l is **linear in store** σ if $\sigma(l)$ is a linear object. The key property concerns the nature of the path in the memory graph between a location and the root set.

Definition 14 Suppose $S = (l, \sigma, \bar{l}, \bar{\rho})$ is a regular state and $\hat{l} \in \mathsf{dom}(\sigma)$. The location $\hat{l}$ is said to be **linear from** l **in** S if there is a path p from l to $\hat{l}$ in $\mathcal{G}(S)$ such that each l' on p satisfies the following two properties:

1. $\sigma(l')$ is linear and
2. $\mathsf{refcount}(l', \sigma) = 1$.

Note that the two conditions satisfied by the path p could only be satisfied by a *unique* path from l to $\hat{l}$; if there were more than one such path, condition (2) could not be satisfied. It will be convenient to say that a path satisfying these conditions is linear. Given a regular state $S = (\rho, \sigma, \bar{l}, \bar{\rho})$, we also say that $\hat{l}$ is linear from ρ in S if there is an x in the domain of ρ such that there is a (unique) linear path from $\rho(x)$ to $\hat{l}$.

To prove the desired property we will need to know some basic facts about types and evaluation. For the high-level semantics we already expressed the Subject Reduction Theorem 2 for the LL-based programming language. In conjuction with the Correctness Theorem 13 we have a version of the result for the low-level semantics as well:

Lemma 15 *Suppose* $S = (l, \sigma, \bar{l}, \bar{\rho})$ *is a regular state,* $\mathsf{dom}(\rho) = FV(M)$, $\vdash \mathsf{valof}(M, \rho, \sigma) : t$, *and* $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$. *Then* $\vdash \mathsf{valofcell}(l', \sigma') : t$.

The theorem we wish to express says that if a program is evaluated in an environment from which a location $\hat{l}$ is linear, then the value at the location is either used and deallocated or not used and linear from the location returned as the result of the evaluation. This statement is intended to formally capture the idea that a location that is linear from an environment is used once *or* left untouched with a reference count of one. Unfortunately, the assertion contains the term 'deallocate', which needs to be made precise. If we assert instead that the reference count of the location is 0 or linear from the result at the end of the computation, then there is a problem in the case where reference count falls to 0 because the allocation relation might *reallocate* the location $\hat{l}$ to hold a value that is unrelated to the one placed there originally. This would make it impossible to assert anything interesting about the outcome of the computation. To resolve this worry, we can make a restriction on the allocation relation insisting that $\hat{l}$ is not in its range. This assumption is harmless in a sense made precise by Theorem 10. The result of interest can now be asserted precisely as follows:

i. $\mathsf{refcount}(\hat{l}, \sigma_1) = 0$. By assumption $\hat{l}$ is never reallocated by new, so $\mathsf{refcount}(\hat{l}, \sigma') = 0$ as needed.

ii. $\mathsf{refcount}(\hat{l}, \sigma_1) = 1$, In this case, the IH implies that there is a linear path from l_1 to $\hat{l}$. There are now two subcases to consider: either $\mathsf{refcount}(l_0, \sigma_0) = 1$ or $\mathsf{refcount}(l_0, \sigma_0) > 1$. We consider only the second and leave the first to the reader. By laws D2, I2, and E, we know that the state

$$S' = (\rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l_0, \sigma_1)), \bar{l}, \bar{\rho})$$

is regular and it is not hard to check that $\hat{l}$ is linear from $\rho'[x \mapsto l_1]$ in S'. Since we must have

$$\mathtt{interp}(N, \rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l_0, \sigma_1))) = (l', \sigma')$$

we are done by IH.

2. $M = (\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n)$. In this case, $\hat{l}$ is reachable from exactly one of the environments $\rho \,|\, M_i$. In the evaluation of M, we have

$$\begin{aligned} &\mathtt{interp}(M_1, \rho \,|\, M_1, \sigma) = (l_1, \sigma_1) \\ &\qquad\vdots \\ &\mathtt{interp}(M_i, \rho \,|\, M_i, \sigma_{i-1}) = (l_i, \sigma_i) \end{aligned}$$

By IH, there are two possibilities for the regular state

$$(l_1, \ldots, l_i, \rho \,|\, M_{i+1}, \ldots, \rho \,|\, M_n, \sigma_i, \bar{l}, \bar{\rho})$$

arising after the evaluation of M_i. Either the reference count of $\hat{l}$ is zero in σ_i or it is one and there is a linear path from l_i to $\hat{l}$. If the first case holds, then we are done, since $\hat{l}$ is not reallocated in the remainder of the computation, and therefore the conclusion of the theorem is satisfied. On the other hand, the second case is impossible: by Lemma 15, $\mathsf{valofcell}(l_i, \sigma_i)$ has type $!t$ and $\sigma_i(l_i)$ is a value, so it has the form $\mathsf{susp}(l'')$ or $\mathsf{rec}(l'', f)$. This contradicts the assumption that $\hat{l}$ is linear from l_i. Therefore reference count of $\hat{l}$ must be 0 in σ_i and hence we are done, since new never reallocates $\hat{l}$.

3. $M = (\mathsf{share}\ x, y\ \mathsf{as}\ P\ \mathsf{in}\ Q)$. In the evaluation of M we compute

$$\begin{aligned} &\mathtt{interp}(P, \rho \,|\, P, \sigma) = (l_0, \sigma_0) \\ &\mathtt{interp}(Q, (\rho \,|\, Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0)) = (l', \sigma') \end{aligned}$$

Now $\hat{l}$ is reachable for exactly one of the environments $\rho \,|\, P$ or $\rho \,|\, Q$. We consider the two cases separately.

(a) $\hat{l}$ is reachable from $\rho \,|\, P$. For the same reasons discussed in the case for store above, IH implies that $\mathsf{refcount}(\hat{l}, \sigma_0) = 0$, and thus we are done since new never reallocates $\hat{l}$.

(b) $\hat{l}$ is reachable from $\rho \,|\, Q$. Then there is a linear path from $\rho \,|\, Q$ to $\hat{l}$ which, by Theorem 6, is unaffected by the evaluation of P. In particular, $\hat{l}$ is not reachable from l_0, so it is linear from $\rho \,|\, Q$ in the regular state $((\rho \,|\, Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$ so we are done by IH.

7 Discussion

For this paper we have chosen a particular natural deduction presentation of linear logic. Others have proposed different formulations of linear logic, and it would be interesting to carry out similar investigations for those formulations. For instance, Abramsky [Abr] has used the sequent formulation of linear logic. His system satisfies substitutivity because this is essentially a rule of the sequent presentation (the *cut rule* to be precise), but there is no clear means of doing type inference for his language. Others [Mac91, LM92] have attempted to reconcile the problems of type inference and substitutivity by proposing restricted forms of these properties. Another approach has been to modify linear logic by adding new assumptions. For instance, [Wad91a] and [O'H91] propose taking $!!A$ to be isomorphic to $!A$; from the perspective of this paper, such an identification would collapse two levels of indirection and suspension into one and hence fundamentally change the character of the language. Other approaches to the presentation of LL seem to have compatible explanations within our framework, but might yield slightly different results. For example, there is a way to present LL using judgements of the form $\Gamma; \Delta \vdash s$ where Γ is a set of 'intuitionistic assumptions' (types of non-linear variables) and Δ is a multi-set of 'linear assumptions' (types of linear variables). This approach might suit the results of Section 6 better than the presentation we used in this paper because it singles out the linear variables more clearly and provides what might be a simpler term language. On the other hand, the connection with reference counts is less clear for that formulation.

It is also possible to fold reference-counting operations into the interpretation of a garden variety functional programming language (that is, one based on intuitionistic logic). The ways in which the result differs from the semantics we have given for an LL-based language are illuminating. First of all, there are several choices about how to do this. One approach is to maintain the invariant that `interp` is evaluated on triples (M, ρ, σ) where the domain of ρ is exactly the set of free variables of M. When evaluating an application $M \equiv (P\ Q)$, for example, it is essential to account for the possibility that some of the free variables of M are shared between P and Q. This means that when P is interpreted, the reference counts of variables they have in common must be incremented (otherwise they may be deallocated before the evaluation of Q begins):

```
interp($(P\ \ Q)$, $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = interp($P$, $\rho \,|\, P$, inc-env($\rho \,|\, P \cap \rho \,|\, Q$, $\sigma$))
      ($l_1$, $\sigma_1$) = interp($Q$, $\rho \,|\, Q$, $\sigma_0$)
  in  case $\sigma_1(l_0)$ of closure($\lambda x.\, N$, $\rho'$) or recclosure($\lambda x.\, N$, $\rho'$) =>
        if refcount($l_0$, $\sigma_1$) = 1
        then interp($N$, $\rho'[x \mapsto l_1]$, dec($l_0$, $\sigma_1$))
        else interp($N$, $\rho'[x \mapsto l_1]$, inc-env($\rho'$, dec($l_0$, $\sigma_1$)))
```

The deallocation of variables is driven by the requirement that only the free variables of M can lie in the domain of ρ; this arises particularly in the semantics for the conditional:

```
interp(if $N$ then $P$ else $Q$, $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = interp($N$, $\rho \,|\, N$, inc-env($\rho|N \cap (\rho|P \cup \rho|Q)$, $\sigma$))
  in  if $\sigma_0(l_0)$ = true
      then interp($P$, $\rho \,|\, P$, dec($l_0$, dec-ptrs-env($(\rho|P) - (\rho|Q)$, $\sigma_0$)))
      else interp($Q$, $\rho \,|\, Q$, dec($l_0$, dec-ptrs-env($(\rho|Q) - (\rho|P)$, $\sigma_0$)))
```

An alternative approach to providing a reference-counting semantics for an intuitionistic language would be to delay the deallocation of variables until 'the last minute' and permit the application suite in [Gab85]. This problem is addressed by the technique of *strictness analysis* [AH87]: with strictness analysis the translation can be made more efficient or the translated program can be optimized. There are several techniques known for translating intuitionistic logic into linear logic. To illustrate, consider the combinator S (here written in ML syntax):

```
fn x => fn y => fn z => (x z)(y z)
```

When we apply Girard's translation, the result (using a syntax similar to the one in Table 1) is the following program:

```
fn x => fn y => fn z =>
  share z1,z2 as z in
    ((fetch x) (store (fetch z1)))
    (store ((fetch y) (store (fetch z2))))
```

However, another program having S as its 'erasure' is

```
fn x => fn y => fn z =>
  share z1,z2 as z in (x z1)(y z2)
```

which is evidently a much simpler and more efficient program. An analog of strictness analysis that applies to the LL translation is clearly needed if an LL intermediate language is to be of practical significance in analyzing 'intuitionistic' programs.

Our reference-counting interpreter and the associated invariance properties can easily be extended to the linear connectives &, ⊗, and ⊕ (although it is unclear how to handle the 'classical' connectives). Extending the results to dynamic allocation of references and arrays is not difficult if such structures do not create cycles. For instance, it can be assumed that only integers and booleans are assignable to mutable reference cells. To see this in a little more detail, if we assume that o is Nat or Bool, then typing rules can be given as follows:

$$\frac{\Gamma \vdash M : o}{\Gamma \vdash \mathsf{ref}(M) : \mathsf{ref}(o)} \qquad \frac{\Gamma \vdash M : \mathsf{ref}(o) \qquad N : o}{\Gamma \vdash M := N : \mathsf{ref}(o)} \qquad \frac{\Gamma \vdash M : \mathsf{ref}(o)}{\Gamma \vdash\, !M : o}$$

To create a reference cell initialized with the value of a term M, the term M is evaluated and its value is *copied* into a new cell:

```
(16)   interp(ref($M$), $\rho$, $\sigma$) =
         let ($l_0$, $\sigma_0$) = interp($M$, $\rho \,|\, M$, $\sigma$)
         in  new($\sigma_0(l_0)$, dec($l_0$, $\sigma_0$))
```

The location l_0 holds the *immutable* value of M; a new *mutable* cell must be created with the value of M as its initial value. Assignment mutates the value associated with such a cell:

```
(17)   interp($M := N$, $\rho$, $\sigma$) =
         let ($l_0$, $\sigma_0$) = interp($M$, $\rho \,|\, M$, $\sigma$)
             ($l_1$, $\sigma_1$) = interp($N$, $\rho \,|\, N$, $\sigma_1$)
         in  ($l_0$, dec($l_1$, $\sigma_1[l_0 \mapsto \sigma_1(l_1)]$))
```

To obtain the value held in a mutable cell denoted by M, the contents of the cell must be copied to a new immutable cell:

A Proofs of the Main Theorems

Verification of the Basic Laws in Table 7

Proposition 17 *Each of the laws A1, A2, D1, D2 given in Section 4 hold.*

Proof: The proof of A1 may be found in Section 4, and the proof of A2 is similar. We thus need only to verify D1 and D2.

D1 Suppose $S = (l, \bar{l}, \bar{\rho}, \sigma)$, $\mathfrak{R}(S)$ holds, $\sigma(l)$ is a numeral or boolean, and $S' = (\bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$. Note that there are no outgoing edges from l in the memory graph induced by S; thus, even if $l \notin \mathsf{dom}(\mathsf{dec}(l,\sigma))$, the state S' is count-correct. Since $\mathsf{dom}(\sigma) \supseteq \mathsf{dom}(\mathsf{dec}(l,\sigma))$, each of the properties $\mathfrak{R}2$–$\mathfrak{R}5$ follow directly from the hypothesis. Thus, $\mathfrak{R}(S')$.

D2 Suppose $\mathfrak{R}(l, \bar{l}, \bar{\rho}, \sigma)$ and $\mathsf{refcount}(l, \sigma) \neq 1$, and let $S' = (\bar{l}, \rho, \bar{\rho}, \mathsf{dec}(l, \sigma))$. By hypothesis, it follows that $\mathsf{refcount}(l, \sigma) > 1$ since l is in the root set. Thus, $\mathsf{refcount}(l, \mathsf{dec}(l, \sigma)) \geq 1$ and hence S' is count-correct, satisfying $\mathfrak{R}1$. Since $\mathsf{dom}(\sigma) = \mathsf{dom}(\mathsf{dec}(l, \sigma))$, each of the properties $\mathfrak{R}2$–$\mathfrak{R}5$ follow directly from the hypothesis. Thus, $\mathfrak{R}(S')$.

This completes the verification of each part. ■

Proposition 18 *Law D3 holds; more generally,*

1. *If* $\mathfrak{R}(l, \bar{l}, \bar{\rho}, \sigma)$, *then* $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.
2. *If* $\mathfrak{R}(\bar{l}, \rho, \bar{\rho}, \sigma)$, *then* $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \sigma))$.

Proof: By induction on the total number of calls to dec-ptrs and dec-ptrs-env. In the basis, suppose the number of calls is one; there are two cases:

1. dec-ptrs is called. Then there are three subcases:

   (a) $\sigma(l) = n$, true, or false. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec}(l, \sigma)$. By D1, $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.

   (b) $\sigma(l) = \mathsf{susp}(l')$, $\mathsf{thunk}(M, \rho)$, or $\mathsf{closure}(\lambda x.\, M, \rho)$, and $\mathsf{refcount}(l, \sigma) > 1$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec}(l, \sigma)$, and hence by D2, $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.

   (c) $\sigma(l) = \mathsf{rec}(l', f)$ or $\mathsf{recclosure}(\lambda x.\, N, \rho)$, and $\mathsf{refcount}(l, \sigma) > 2$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec}(l, \sigma)$, and hence by D2, $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.

2. dec-ptrs-env is called. Then since dec-ptrs is not called, $\mathsf{dom}(\rho)$ must be the empty set. Thus, $\mathsf{dec\text{-}ptrs\text{-}env}(\rho, \sigma) = \sigma$ and hence $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \sigma))$.

For the induction hypothesis, suppose the total number of calls to dec-ptrs and dec-ptrs-env is greater than one. There are again two main cases:

1. dec-ptrs is called. There are five subcases depending on the reference count and the value stored at l.

   (a) $\sigma(l) = \mathsf{susp}(l')$ and $\mathsf{refcount}(l, \sigma) = 1$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec\text{-}ptrs}(l', \mathsf{dec}(l, \sigma))$. By A2, $\mathfrak{R}(l', \bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$ and so by induction, $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l', \mathsf{dec}(l, \sigma)))$. Thus, $\mathfrak{R}(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.

N2 Suppose $\Re(\bar{l}, \rho, \bar{\rho}, \sigma)$, $(l', \sigma') = \mathsf{new}(\mathsf{closure}(N, \rho), \sigma)$ or $\mathsf{new}(\mathsf{thunk}(N, \rho), \sigma)$, $FV(N) = \mathsf{dom}(\rho)$, and N is typeable, and let $S' = (l', \bar{l}, \bar{\rho}, \sigma')$. Since new is an allocation relation, $\mathsf{refcount}(l', \sigma) = 0$, $\mathsf{refcount}(l', \sigma') = 1$, and for any location $l \neq l'$, $\sigma(l) = \sigma(l')$ and $\mathsf{refcount}(l, \sigma) = \mathsf{refcount}(l, \sigma')$. To see that property $\Re1$—namely count-correctness—holds of S', note that all of the pointers from ρ are accounted for in the closure or thunk stored in l', and that l' only has reference count 1. To see $\Re2$, $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \cup \{l'\}$ is finite because $\mathsf{dom}(\sigma)$ is. If l' is a thunk, then $\mathsf{refcount}(l', \sigma') = 1$, which together with the hypothesis guarantees property $\Re3$. No cycles are created in the induced memory graph by new, so $\Re4$ holds. Finally, $\Re5$ holds by hypothesis. Thus, $\Re(S')$.

N3 Suppose $\Re(l, \bar{l}, \bar{\rho}, \sigma)$ and $(l', \sigma') = \mathsf{new}(\mathsf{susp}(l), \sigma)$ or $\mathsf{new}(\mathsf{rec}(l, f), \sigma)$. Then $\Re(l', \bar{l}, \bar{\rho}, \sigma')$ follows in a manner similar to the previous case.

U1 Suppose $S = (\bar{l}, \bar{\rho}, \sigma)$ and $\Re(S)$, $\sigma(l)$ is a constant. We prove the first statement of U1 only; the first follows similarly. So suppose $l' \in \mathsf{dom}(\sigma)$, and l is not reachable from l' in the memory graph induced by S, and let $S' = (\bar{l}, \bar{\rho}, \mathsf{inc}(l', \sigma[l \mapsto \mathsf{susp}(l')]))$. In S' the in-degree of l' is now one greater than in S; the in-degree of all other nodes remains the same. Thus, S' satisfies property $\Re1$. Since $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$, the domain of σ' is finite, satisfying property $\Re2$. No new thunks are created, so property $\Re3$ holds of S'. Since l is not reachable from l' in S, there is no cycle through l in S'. Thus, S' satisfies property $\Re4$. Finally, property $\Re5$ holds since no thunks or closures are added to σ. Thus, $\Re(S')$.

U2 Suppose $S = (l, \bar{l}, \bar{\rho}, \sigma)$ and $\Re(S)$, $\mathsf{refcount}(l, \sigma) \neq 1$, $\sigma(l) = \mathsf{susp}(l')$, and $\sigma(l') = \mathsf{thunk}(N, \rho)$, and let $S' = (\rho, \bar{l}, \bar{\rho}, \mathsf{dec}(l', \mathsf{dec}(l, \sigma[l \mapsto c])))$. To verify property $\Re1$, note first that $\mathsf{refcount}(l', \sigma) = 1$ by hypothesis. Thus, since the pointers from $\sigma l'$ are mentioned in the root set of S', it follows that S' is count-correct. It is also clear that each of the properties $\Re2$–$\Re5$ hold of S'. Thus, $\Re(S')$.

This completes the verification of each part. ■

Proof of Lemma 10

Lemma 10 *Suppose $(\bar{l}', \rho_f, \bar{\rho}', \sigma_f)$ and $(\bar{l}'', \rho_g, \bar{\rho}'', \sigma_g)$ are congruent. If* $\mathtt{interp}_f(M, \rho_f, \sigma_f) = (l'_f, \sigma'_f)$, *then* $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l'_g, \sigma'_g)$ *and the resultant states $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ and $(l'_g, \bar{l}'', \bar{\rho}'', \sigma'_g)$ are congruent.*

Proof: By induction on the number of calls to `interp`. We cover the four cases in the core language and leave the other cases to the reader. To make the cases easier to read, let h be the isomorphism from $\mathcal{G}(\sigma_f)$ to $\mathcal{G}(\sigma_g)$ that makes the above states congruent.

1. $M = x$. Then $\mathtt{interp}_f(M, \rho_f, \sigma_f) = (\rho_f(x), \sigma_f)$. Then also $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (\rho_g(x), \sigma_g)$, and the resultant states $(\rho_f(x), \bar{l}', \bar{\rho}', \sigma'_f)$ and $(\rho_g(x), \bar{l}'', \bar{\rho}'', \sigma'_g)$ are congruent via h.

2. $M = (\lambda x.\, P)$. Then $\mathtt{interp}_f(M, \rho_f, \sigma_f) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P, \rho_f), \sigma_f) = (l'_f, \sigma'_f)$. Since f is an allocation relation,

   - $l'_f \notin \mathsf{dom}(\sigma_f)$ and $\mathsf{dom}(\sigma'_f) = \mathsf{dom}(\sigma_f) \cup \{l'_f\}$;
   - for all locations $l \in \mathsf{dom}(\sigma_f)$, $\sigma_f(l) = \sigma'_f(l)$ and $\mathsf{refcount}(l, \sigma_f) = \mathsf{refcount}(l', \sigma'_f)$; and

4. $M = (\mathsf{fetch}\ P)$. Then $\mathtt{interp}_f(P, \rho_f, \sigma_f) = (\ell_{f,0}, \sigma_{f,0})$. By induction, $\mathtt{interp}_g(P, \rho_g, \sigma_g) = (\ell_{g,0}, \sigma_{g,0})$ and the states $(\ell_{f,0}, \bar{l}', \bar{\rho}', \sigma_{f,0})$ and $(\ell_{g,0}, \bar{l}'', \bar{\rho}'', \sigma_{g,0})$ are congruent. Now there are two main cases: either $\sigma_{f,0}(\ell_{f,0}) = \mathsf{susp}(\ell_{f,1})$ or $\sigma_{f,0}(\ell_{f,0}) = \mathsf{rec}(\ell_{f,1}, x)$. We leave the second case to the reader since it is relatively straightforward and consider only the first case.

   Suppose $\sigma_{f,0}(\ell_{f,0}) = \mathsf{susp}(\ell_{f,1})$. By the definition of congruence, $\sigma_{g,0}(\ell_{g,0}) = \mathsf{susp}(\ell_{g,1})$. Now there are two subcases depending on the object held at $\ell_{f,1}$:

   (a) $\sigma_{f,0}(\ell_{f,1}) = \mathsf{thunk}(R, \rho'_f)$. Then by congruence, $\sigma_{g,0}(\ell_{g,1}) = \mathsf{thunk}(R, \rho'_f)$. There are two subcases depending on the reference count of $\ell_{f,0}$:

      i. $\mathsf{refcount}(\ell_{f,0}, \sigma_{f,0}) = 1$. Since the above tuples are congruent, $\mathsf{refcount}(\ell_{g,0}, \sigma_{g,0}) = 1$. Note that the states

      $$(\bar{l}', \rho'_f, \bar{\rho}', \mathsf{dec}(\ell_{f,1}, \mathsf{dec}(\ell_{f,0}\sigma_{f,0},)))$$
      $$(\bar{l}'', \rho'_g, \bar{\rho}'', \mathsf{dec}(\ell_{g,1}, \mathsf{dec}(\ell_{g,0}\sigma_{g,0},)))$$

      are congruent since ρ'_f and ρ'_g must have the same domain and must match via the multigraph isomorphism h on their domains. Thus, by induction, $\mathtt{interp}_g(R, \rho'_g, \mathsf{dec}(\ell_{g,1}, \mathsf{dec}(\ell_{g,0}\sigma_{g,0},))) = (l'_g, \sigma'_g)$ and the states $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ and $(l'_g, \bar{l}'', \bar{\rho}'', \sigma'_g)$ are congruent. Putting all the steps together, we also see that $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l'_g, \sigma'_g)$

      ii. $\mathsf{refcount}(\ell_{f,1}, \sigma_{f,1}) \neq 0$. Similar to the previous case.

   (b) $\sigma_{f,0}(\ell_{f,1}) \neq \mathsf{thunk}(R, \rho'_f)$. Then again there are two cases depending on the reference count of $\ell_{f,0}$:

      i. $\mathsf{refcount}(\ell_{f,0}, \sigma_{f,0}) = 1$; then $\mathsf{refcount}(\ell_{g,0}, \sigma_{g,0}) = 1$. Thus,

      $$\mathtt{interp}_g(M, \rho_g, \sigma_g) = (\ell_{g,1}, \mathsf{dec}(\ell_{g,0}, \sigma_{g,0}))$$

      and the states $(\ell_{f,1}, \bar{l}', \bar{\rho}', \mathsf{dec}(\ell_{f,0}, \sigma_{f,0}))$ and $(\ell_{g,1}, \bar{l}'', \bar{\rho}'', \mathsf{dec}(\ell_{g,0}, \sigma_{g,0}))$ are congruent.

      ii. $\mathsf{refcount}(\ell_{f,0}, \sigma_{f,0}) \neq 1$. Similar to the previous case.

This completes the induction and hence the proof. ■

Proof of Theorem 13

Recall from Section 5 that, in order to prove a correctness theorem, we needed a definition of how to unwind a term from a store. The definition of two mutually-recursive functions for performing this task, valof and valofcell, appears in Table 8. It is obvious from the definitions that only the reachable cells affect the value returned by valof and valofcell. For instance, if l' is not reachable from l in store σ and $\sigma' = \mathsf{dec}(l', \sigma)$, then $\mathsf{valofcell}(l, \sigma) = \mathsf{valofcell}(l, \sigma')$. We will use this fact throughout the arguments that follow.

Also essential to the proof of Theorem 13 is a notion of when one term is 'more evaluated' than another. Section 5 defines a relation $\geq^*$ between terms which expresses this relationship. We can prove three lemmas about the relationship of $\geq$ and canonical forms.

Lemma 20 *If $c \geq P$ and c is a canonical form, then P is a canonical form. Moreover, c and P have the same shape, i.e., if c is a numeral or boolean, then $c = P$; if $c = \lambda x.\, Q$, then $P = \lambda x.\, Q'$; and if $c = (\mathsf{store}\ Q)$, then $P = (\mathsf{store}\ Q')$.*

Proof: There are two cases to consider: either $c \Downarrow P$, or $c = C[M]$, $P = C[N]$, $C[\cdot]$ is nontrivial, and $M \Downarrow N$. In the first case, since c is canonical, $c = P$, and hence P is canonical. In the second case, for c to be canonical it must be the case that $C[\cdot] = n$, true, false, $\lambda x.\, D[\cdot]$, or (store $C[\cdot]$). Thus, P must be canonical as well, and must have the same shape as c. ■

Lemma 21 *If c is a canonical form and $M \geq c$, then $M \Downarrow d \geq c$.*

Proof: By the definition of $M \geq c$, we know that $M = C[M']$, $c = C[d]$, and $M' \Downarrow d$. In order for c to be canonical, it must be the case that either $C[\cdot] = [\cdot]$, n, true, false, $\lambda x.\, D[\cdot]$, or (store $D[\cdot]$). In the first case, $M' = M$ and $d = c$, so $M \Downarrow c \geq c$. For the other cases, $M \Downarrow M \geq c$. ■

Lemma 22 *If c is a canonical form and $M \geq^* c$, then $M \Downarrow d \geq^* c$.*

Proof: An easy induction on the length of $M = M_1 \geq \ldots \geq M_k \geq c$ using Lemma 20. ■

We need a similar definition of one state in the reference-counting interpreter being 'more evaluated' than another. Basically, one state is more evaluated than another if, tracing from the root set, the storable objects held at nodes are identical or thunks have been replaced by more evaluated forms. Formally,

Definition 23 We say $(\bar{l}, \bar{\rho}, \sigma) \geq^* (\bar{l}, \bar{\rho}, \sigma')$ if for all l reachable from the root set, $l \in \mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma')$ and

1. $\sigma(l) = n$, true, false, $\mathsf{closure}(\lambda x.\, N, \rho)$, or $\mathsf{recclosure}(\lambda x.\, N, \rho)$, and $\sigma(l) = \sigma'(l)$ and $(\rho, \sigma) \geq^* (\rho, \sigma')$;
2. $\sigma(l) = \mathsf{susp}(l_0)$ or $\mathsf{rec}(l_0, f)$, $\sigma(l_0)$ is not a thunk, $\sigma'(l) = \mathsf{susp}(l_0)$ and $(l_0, \sigma) \geq^* (l_0, \sigma')$; or
3. $\sigma(l) = \mathsf{susp}(l_0)$, $\sigma(l_0) = \mathsf{thunk}(R, \rho)$ and either
   (a) $\sigma'(l) = \mathsf{susp}(l_0)$, $\sigma'(l_0) = \mathsf{thunk}(R, \rho)$, and $(\rho, \sigma) \geq^* (\rho, \sigma')$; or
   (b) $\sigma'(l) = \mathsf{susp}(l')$, $\sigma'(l')$ is not a thunk, $\mathtt{interp}(R, \rho, \sigma) = (l', \sigma'')$, and $(l', \sigma'') \geq^* (l', \sigma')$

where $(\rho, \sigma) \geq^* (\rho, \sigma')$ if for every $x \in \mathsf{dom}(\rho)$, $(\rho(x), \sigma) \geq^* (\rho(x), \sigma')$.

It is not difficult to prove that $\geq^*$ is reflexive and transitive on states. It is also not difficult to prove the following two lemmas:

Lemma 24 *Suppose $\Re(\bar{l}, \rho, \bar{\rho}, \sigma)$ and $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$. Then $(\bar{l}, \bar{\rho}, \sigma) \geq^* (\bar{l}, \bar{\rho}, \sigma')$.*

Lemma 25 *If $Q' \geq^* \mathsf{valof}(Q, \rho, \sigma)$ and $(\bar{l'}, \rho, \bar{\rho'}, \sigma) \geq^* (\bar{l'}, \rho, \bar{\rho'}, \sigma')$, then $Q' \geq^* \mathsf{valof}(Q, \rho, \sigma')$.*

The proof of the first is an easy induction on the number of calls to $\mathtt{interp}$; the proof of the second is an easy induction on the definition of valof.

We now have enough machinery to prove the main correctness theorem.

Theorem 13 *Suppose M is typeable, $\mathsf{dom}(\rho) = FV(M)$, M' is closed, and $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$. Suppose also that $\Re(\bar{l'}, \rho, \bar{\rho'}, \sigma)$.*

1. *If $M' \Downarrow c$, then $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$ and $c \geq^* \mathsf{valofcell}(l', \sigma')$.*

2. $M' = (\mathsf{store}\ N'\ \mathsf{where}\ x_1 = M'_1, \ldots, x_n = M'_n)$, $c = (\mathsf{store}\ N'[x_1, \ldots, x_n := d_1, \ldots, d_n])$, and $M'_i \Downarrow d_i$. We only need to consider the case when $M = (\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n)$, where $N' \geq^* \mathsf{valof}(N, \emptyset, \sigma)$ and $M'_i \geq^* \mathsf{valof}(M_i, \rho \,|\, M_i, \sigma)$. Since M is typeable, the free variables of each M_i are disjoint. Since $M'_1 \geq^* \mathsf{valof}(M_1, \rho \,|\, M_1, \sigma)$, by induction

   $$\mathtt{interp}(M_1, \rho_1, \sigma) = (l_1, \sigma_1),$$

   where $d_1 \geq^* \mathsf{valofcell}(l_1, \sigma_1)$. By Lemma 24,

   $$(\bar{l}', \rho \,|\, M_2, \ldots, \rho \,|\, M_n, \sigma) \geq^* (\bar{l}', \rho \,|\, M_2, \ldots, \rho \,|\, M_n, \sigma_1)$$

   and by Theorem 6, $\Re(l_1, \bar{l}', \rho \,|\, M_2, \ldots, \rho \,|\, M_n, \sigma_1)$. Since $M'_2 \geq^* \mathsf{valof}(M_2, \rho \,|\, M_2, \sigma)$, by Lemma 25, $M'_2 \geq^* \mathsf{valof}(M_2, \rho \,|\, M_2, \sigma_1)$. Using similar repeated applications of the induction hypothesis,

   $$\mathtt{interp}(M_i, \rho_i, \sigma_{i-1}) = (l_i, \sigma_i)$$

   where $d_i \geq^* \mathsf{valofcell}(l_i, \sigma_i)$, and by Lemma 24,

   $$(l_1, \ldots, l_{i-1}, \bar{l}', \bar{\rho}', \sigma_{i-1}) \geq^* (l_1, \ldots, l_{i-1}, \bar{l}', \bar{\rho}', \sigma_i)$$

   Finally, let

   $$\begin{array}{l} \rho' = \emptyset[x_1, \ldots, x_n \mapsto l_1, \ldots, l_n] \\ \mathsf{new}(\mathsf{thunk}(N, \rho'), \sigma_n) = (l_{n+1}, \sigma_{n+1}) \\ \mathsf{new}(\mathsf{susp}(l_{n+1}), \sigma_{n+1}) = (l', \sigma') \end{array}$$

   Then using Lemma 25, we find that $(\mathsf{store}\ N'[x_1, \ldots, x_n := d_1, \ldots, d_n]) \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

3. $M' = (\mathsf{fetch}\ N')$, where $N' \Downarrow (\mathsf{store}\ Q')$ and $Q' \Downarrow c$. Then the only case to consider is $M = (\mathsf{fetch}\ N)$ where $N' \geq^* \mathsf{valof}(N, \rho, \sigma)$. By induction,

   $$\mathtt{interp}(N, \rho, \sigma) = (l_0, \sigma_0)$$

   where $(\mathsf{store}\ Q') \geq^* \mathsf{valofcell}(l_0, \sigma_0)$. By Theorem 6, $\Re(l_0, \bar{l}', \bar{\rho}', \sigma_0)$. Since $(\mathsf{store}\ Q') \geq^* \mathsf{valofcell}(l_0, \sigma_0)$ and $\sigma_0(l_0)$ must be a value, it follows from Lemma 20 that $\sigma_0(l_0) = \mathsf{susp}(l_1)$ or $\mathsf{rec}(l_1, f)$ and $Q' \geq^* \mathsf{valofcell}(l_1, \sigma_0)$. We consider only the case when $\sigma_0(l_0)$ is $\mathsf{susp}(l_1)$ and leave the other case to the reader. There are two subcases:

   (a) $\sigma_0(l_1) = \mathsf{thunk}(R, \rho')$. There are two subcases:

      i. $\mathsf{refcount}(l_0, \sigma_0) = 1$. First, note that neither l_0 nor l_1 is reachable from ρ'—if either were, the state $S = (l_0, \bar{l}', \bar{\rho}', \sigma_0)$ would have a cycle that was not composed solely of a rec and a $\mathsf{recclosure}$—and this contradicts the regularity of the state S. Thus,

      $$Q' \geq^* \mathsf{valof}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))).$$

      By laws A1 and A2, $\Re(\bar{l}', \rho', \bar{\rho}', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0)))$. Thus, it follows by induction that $\mathtt{interp}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))) = (l', \sigma')$ and $c \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

3. $M = (P\ Q)$. Since $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$, it follows that

$$\begin{aligned}&\mathtt{interp}(P, \rho\,|\,P, \sigma) = (l_0, \sigma_0)\\&\mathtt{interp}(Q, \rho\,|\,Q, \sigma_0) = (l_1, \sigma_1)\\&\sigma_1(l_0) = \mathsf{closure}(\lambda x.\, N, \rho') \text{ or } \mathsf{recclosure}(\lambda x.\, N, \rho')\end{aligned}$$

Since $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$, it must be that $M' = (P'\ Q')$ for some closed P' and Q', where $P' \geq^* \mathsf{valof}(P, \rho, \sigma)$ and $Q' \geq^* \mathsf{valof}(Q, \rho, \sigma)$. By induction,

$$\begin{aligned}P' &\Downarrow d' \geq^* \mathsf{valofcell}(l_0, \sigma_0)\\Q' &\Downarrow d \geq^* \mathsf{valofcell}(l_1, \sigma_1)\end{aligned}$$

By Lemmas 24 and 25, $d' \geq^* \mathsf{valofcell}(l_0, \sigma_1)$. Since $\sigma_1(l_0)$ is a closure, $\mathsf{valofcell}(l_0, \sigma_1)$ must be a λ-abstraction, and so by Lemma 20 it follows that $d' = (\lambda x.\, N')$ for some N'. If $\mathsf{refcount}(l_0, \sigma_1) = 1$, then $N'[x := d] \geq^* \mathsf{valof}(N', \rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1))$. If, on the other hand, $\mathsf{refcount}(l_0, \sigma_1) > 1$, then $N'[x := d] \geq^* \mathsf{valof}(N', \rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l_0, \sigma_1)))$. In either case, it follows by the induction hypothesis that

$$N'[x := d] \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma').$$

Thus, we conclude $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$.

4. $M = (\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n)$. Since M evaluates,

$$\begin{aligned}&\mathtt{interp}(M_1, \rho\,|\,M_1, \sigma) = (l_1, \sigma_1)\\&\mathtt{interp}(M_2, \rho\,|\,M_2, \sigma_1) = (l_2, \sigma_2)\\&\qquad\vdots\\&\mathtt{interp}(M_n, \rho\,|\,M_n, \sigma_{n-1}) = (l_n, \sigma_n)\\&\rho' = \emptyset[x_1, \ldots, x_n \mapsto l_1, \ldots, l_n]\\&(l_{n+1}, \sigma_{n+1}) = \mathsf{new}(\mathsf{thunk}(N, \rho'), \sigma_n)\\&(l', \sigma') = \mathsf{new}(\mathsf{susp}(l_{n+1}), \sigma_{n+1})\end{aligned}$$

Since $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$, it follows that $M' = (\mathsf{store}\ N'\ \mathsf{where}\ x_1 = M'_1, \ldots, x_n = M'_n)$ and $N' \geq^* \mathsf{valof}(N, \emptyset, \sigma)$ and $M'_i \geq^* \mathsf{valof}(M_i, \rho\,|\,M_i, \sigma)$. By induction, $M_1 \Downarrow c_1 \geq^* \mathsf{valofcell}(l_1, \sigma_1)$. To evaluate the next term in the sequence, note that

$$M'_2 \geq^* \mathsf{valof}(M_2, \rho_2, \sigma) \geq^* \mathsf{valof}(M_2, \rho_2, \sigma_1)$$

so by induction $M'_2 \Downarrow c_2 \geq^* \mathsf{valofcell}(l_2, \sigma_2)$. Extending the induction hypothesis further yields that $M_i \Downarrow c_i \geq^* \mathsf{valofcell}(l_i, \sigma_i)$. Note also that by Lemmas 24 and 25, $N' \geq^* \mathsf{valof}(N, \emptyset, \sigma_n)$; it follows that

$$(\mathsf{store}\ N'[x_1, \ldots, x_n := c_1, \ldots, c_n]) = c \geq^* \mathsf{valof}(N, \rho', \sigma_n) = \mathsf{valofcell}(l', \sigma').$$

Thus $M_i \Downarrow c_i$ and so $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

5. $M = (\mathsf{fetch}\ P)$. Since M evaluates, $\mathtt{interp}(P, \rho, \sigma) = (l_0, \sigma_0)$ and by Theorem 6, $\Re(l_0, \bar{l}', \bar{\rho}', \sigma_0)$. Since $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$, it follows that $M' = (\mathsf{fetch}\ P')$ for some P' and $P' \geq^* \mathsf{valof}(P, \rho, \sigma)$. By induction,

$$P' \Downarrow d' \geq^* \mathsf{valofcell}(l_0, \sigma_0).$$

References

[Abr] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science.* To appear.

[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[App92] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Bak88] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(7):11–20, 1988.

[BBdH92] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. Term assignment for intuitionistic linear logic. Announced on the **Types** electronic mailing list, 1992.

[BGS90] V. Breazu-Tannen, C. Gunter, and A. Scedrov. Computing with coercions. In M. Wand, editor, *Lisp and Functional Programming*, pages 44–60. ACM, 1990.

[BHY88] A. Bloss, P. Hudak, and J. Young. An optimizing compiler for a modern functional programming language. *Computer Journal*, 31(6), 1988.

[CGR92] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 139–150. ACM, 1992.

[Col60] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[DB76] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[Des86] Joëlle Despeyroux. Proof of translation in natural semantics. In *Proceedings, Symposium on Logic in Computer Science.* IEEE, 1986.

[Fel91] A. Felty. A logic program for transforming sequent proofs to natural deduction proofs. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, Lecture Notes in Artificial Intelligence, pages 157–178, Berlin, 1991. Springer-Verlag.

[Gab85] R. P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[GG92] B. Goldberg and M. Gloger. Polymorphic type reconstruction for garbage collection without tags. In W. Clinger, editor, *Lisp and Functional Programming*, pages 53–65. ACM, 1992.

[GH90] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Sci.*, 50:1–102, 1987.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Sci.*, 1:125–159, 1975.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[Pot77] Garrel Pottinger. Normalization as a homomorphic image of cut-elimination. *Annals of Mathematical Logic*, 12(3):223–357, 1977.

[PS91] S. Purushothaman and J. Seaman. An adequate operational semantics of sharing in lazy evaluation. Technical Report PSU-CS-91-18, Pennsylvania State University, 1991.

[Sco] D. S. Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. Unpublished manuscript, 1969.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods.* North Holland, 1990.

[Wad91a] P. Wadler. There is no substitute for linear logic. Manuscript, 1991.

[Wad91b] Philip Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273. ACM, 1991.

[WHHO92] D. S. Wise, C. Hess, W. Hunt, and E. Ost. Uniprocessor performance of reference-counting hardware heap. Unpublished manuscript, 1992.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing. The MIT Press, 1993.

[WO92] M. Wand and D. P. Oliva. Proving the correctness of storage representations. In W. Clinger, editor, *Lisp and Functional Programming*, pages 151–160. ACM, 1992.

[Zuc74] J. I. Zucker. Cut-elimination and normalization. *Annals of Mathematical Logic*, 1(1):1–112, 1974.