Operational aspects of linear lambda calculus

Patrick Lincoln*
Computer Science Department
Stanford University
Stanford, CA, 94305
lincoln@cs.stanford.edu

John Mitchell†
Computer Science Department
Stanford University
Stanford, CA, 94305
jcm@cs.stanford.edu

Abstract

Linear logic is a resource-aware logic that is based on an analysis of the classical proof rules of contraction (copying) and weakening (throwing away). Several previous researchers have studied functional programming languages derived from linear logic according to the "formulas-as-types" correspondence. In languages with linear logic types, one may hope that traditional implementation problems in functional languages such as update in place could be simplified by careful use of the type system. In this paper, we prove that the standard sequent calculus proof system of linear logic is equivalent to a natural deduction style proof system. Using the natural deduction system, we investigate the pragmatic problems of type inference and type safety for a linear lambda calculus. Although terms do not have a single most-general type (for either the standard sequent presentation or our natural deduction formulation), there is a set of most-general types that may be computed using unification. The natural deduction system also facilitates the proof that the type of an expression is preserved by any evaluation step. We also describe an execution model and implementation, using a variant of the "three-instruction machine" (TIM). A novel feature of the implementation is that we distinguish garbage-collected "non-linear" memory from "linear" memory, which does not require garbage collection and where it is possible to do secure update in place.

*Supported by AT&T Doctoral Scholarship and SRI.

†Supported in part by an NSF PYI Award, matching funds from Digital Equipment Corporation, the Powell Foundation, and Xerox Corporation; a gift from Mitsubishi Corporation and the Wallace F. and Lucille M. Davis Faculty Scholarship.

0-8186-2735-2/92 $3.00 © 1992 IEEE

1 Introduction

Linear logic is a resource-sensitive refinement of classical logic, introduced by Girard [Gir87]. The logic is more detailed than classical or intuitionistic logic, distinguishing A *and* A from A, in a way that suggests applications to reasoning about resource bounds or resource management in computation. Propositional linear logic is technically more expressive than either classical or intuitionistic logic since propositional linear logic is undecidable [LMSS90]. The extra expressive power of linear logic has enabled the study of practical problems in implementation of declarative programming languages. For example, Lafont has eliminated garbage collection in functional languages [Laf88] and Cerrito has analyzed Prolog using linear logic [Cer90].

In this paper we focus on the application of linear logic to functional programming type systems. Historically, intuitionistic logic has been the basis for type systems, via the Curry-Howard isomorphism, or "formulas-as-types" principle [How80]. Through this isomorphism, intuitionistic proofs of propositions may be viewed as functional programs, and logical propositions may viewed as types. A similar use of linear logic has been initiated by Girard and Lafont, and Abramsky [GL87, Abr90]. In [GL87], a linear calculus was developed which effectively determines reduction order, while explicitly marking the points where contraction and weakening are used. Abramsky further defined a type inference system, here called SEQ, which is discussed in Section 3.3. Abramsky went on to generalize this system into one incorporating quantifiers and full linear logic, a move which enabled him to interpret linear types in terms of concurrent computations. Recently Chirimar, Gunter, and Riecke [CGR92] have implemented a version of the linear calculus. In this paper we restrict our attention to intuitionistic linear logic.

One important property of type systems is *subject*

reduction, which states that if a term t has type A, then any term produced by any number of reduction (evaluation) steps still has type A. This is crucial if we wish to use types to statically determine execution properties of terms. While it may be possible to prove subject reduction for a type system based on sequent calculus rules, there is a significant technical obstacle. If we wish to reason about the effect of reduction, we need to understand the connection between the syntactic form of a term and the set of possible types. However, with sequent calculus rules such as SEQ, a single term may have typing proofs of many different forms. (This is because uses of **Cut**, which are essential for typing terms not in normal form, are not reflected in the syntax of terms.) To avoid this problem, we formulate an equivalent set of natural deduction style typing rules, called NAT. This system has the property that for each form of linear term, there is exactly one typing rule that may be used to give a type. Using the natural deduction typing rules, subject reduction may be proved by traditional means. In addition, with syntax-directed typing rules, it is possible to formulate a unification-based algorithm that determines the most general types of any linear lambda term.

An interesting property of NAT is that one essential rule, **!SR,** based on the modal operator ! of linear logic, involves substitution into terms. Since a term may be written as the result of substitution in many different ways, this rule gives us a system in which a term may have several different principal linear types. Of course, since NAT is equivalent to SEQ, this is not an idiosyncrasy of our presentation, but a property shared by Abramsky's system SEQ that seems inherent to linear logic. If we simplify the **!SR** typing rule of NAT, we obtain an *inequivalent* system, which we call NAT2. If we restrict reduction to closed terms, then subject reduction holds for this system. However, the provable typing judgements are not closed under substitution. Essentially this system has been studied by [Mac91], who proves the existence of unique most-general types. Walder also discusses alternative rules for **!SR**. Since these systems are not equivalent to SEQ and NAT, it seems an important research problem to evaluate the trade-offs between the systems.

In the final part of the paper we explore implementations of the linear lambda calculus. One problem with Lafont's method of eliminating garbage collection is that it requires a tremendous amount of duplication. Essentially, in comparison with a standard reference counting scheme, garbage collection is eliminated by making every datum have reference count one. This is achieved by copying the datum whenever we would otherwise increment the reference count. A consequence is that there is a significant increase in the amount of storage space required. We believe that in practice, it is useful to consider the trade-offs between copying and garbage collection. In particular, if a datum is large, then copying it even a small number of times may be prohibitive, and may outweigh the benefit of suspending garbage collection. In order to explore such trade-off's in a general setting, we have developed an implementation with two forms of memory, called "linear" and "non-linear" memory. Within this framework, we eliminate garbage collection in linear memory but retain traditional garbage collection techniques in non-linear memory. Similarly, we may perform array update in place on arrays in linear memory. Our implementation is based on an extension of the "three instruction machine" (TIM) [FW85] with additional operations of DELAY, FORCE, COPY, and DISCARD to provide explicit control over evaluation order and storage management, and arrays that are updated in linear memory. The implementation of our abstract machine is written in Common Lisp, with garbage collection in non-linear memory handled by the Lisp garbage collector.

In the next section, we describe the problems that arise in using linear logic formulas as types for pure lambda terms. This motivates the use of linear lambda calculus with explicit copy and discard primitives. In Section 3, we present the syntax of linear lambda calculus and the typing rules. SEQ and NAT are proved equivalent in Section 3, where we also prove complementary term subformula and type subformula properties for the two proof systems. A type inference algorithm and proof of most general typing are given in Section 4, with the subject reduction property proved in Section 5. The remaining sections of the paper discuss our execution model and TIM-based implementation.

2 Why explicit storage operations?

In the pure lambda calculus (typed or untyped), there are no explicit `store`, `read`, `copy`, or `discard` primitives. The usual implementations of languages based on lambda calculus perform these operations as needed, according to one of several possible strategies. In other words, these operations are implicit in the language but explicit in the implementation. Since `store`, `read`, `copy`, and `discard` are explicit in the proof system of linear logic, we might attempt to insert these operations into lambda terms as part of

inferring linear logic types. This was part of the program we started to follow in collaboration with Scedrov in 1989, before discovering that this seemed to require algorithms for deciding provability properties of propositional linear logic; this led to the study of decision problems reported in [LMSS90]. In the remainder of this motivational section, we sketch two particular problems that arise, namely, the lack of a natural form of principal type and the failure of subject reduction theorem. The first, along with the undecidability results of [LMSS90], suggests that the process of inferring types will be algorithmically tractable only if additional operations or typing constraints are added to lambda calculus. The failure of subject reduction reinforces this conclusion by showing that additional operations are needed in the language to determine the order of function application and discarding of data.

In this section only, we consider a type system derived from the SEQ rules, in Appendix B, by modifying each rule whose name begins with ! so that the term in the consequent is the same as the term in the antecedent. This has the effect of assigning $\multimap, !$ types to pure lambda terms in a way that allows ! operations to be done implicitly at any point in the evaluation of terms. There is an equivalent presentation of this type system derived from the NAT rules, but since our intent is only to give a few intuitive examples, it does not seem worthwhile to present the details. The main properties of this system are that a function of type $A \multimap B$ must use its argument of type A "exactly once" in producing a result of type B. However, if A is of the form $!C$, then the copy and discard rules associated with ! types allow us to define functions that use their argument zero or more times.

All of the intuitive points we will consider may be illustrated using the the λ term

$$\lambda\, q.\, \lambda\, r.\, \lambda\, s.\, (\lambda x.q)(r\, s)$$

The subterm $(\lambda x.q)$ must have a linear type of the form

$$(\lambda\, x.q) : (!B \multimap A),$$

since only arguments of ! type need not appear in the body of a function. Consequently, the application $(r\, s)$ must have type $!B$. There are two possible types of r and s. One is that r is a non-discardable function that produces discardable output. That is, $r : (C \multimap !B)$, $s : C$. The other possibility is more subtle and requires more detailed understanding of the type system. If r and s are both discardable resources, such as $r : !(!C \multimap B)$ and $s : !C$, then by the usual application rule we have $rs : B$. However, whenever we have an expression of type B such that all variables appearing in the term have a type beginning with !, the **!SR** rule allows use to conclude that the term has type $!B$. Using our concepts of linear and nonlinear memory, the **!SR** rule may be explained by saying that if we define a value of type B by referring only to values in non-linear memory, the value we define may reside in non-linear memory, and have type $!B$.

From the discussion above, we can see that there are two types for the example λ-term:

$$A \multimap !(!C \multimap B) \multimap !C \multimap A$$

$$A \multimap (C \multimap !B) \multimap C \multimap A$$

Associated with these types are two different orders of evaluation. The first is the type of the function that reduces the outermost application first; the application $(r\, s)$ is thrown away before it is ever evaluated. The second type is the type of the function that first reduces the inner term $(r\, s)$, to obtain a discardable value, and then reduces the outer term. In the first case neither r nor s are used at all. In the second case they are both used, but the result is not.

The first general conclusion that follows from this example is that not all terms have a most general type with respect to substitution. This is evident since we have two types for the example term such that neither is a substitution instance of the other, and it can be checked that no shorter type is derivable for this term. Moreover, the two types are disjoint in this system: we can find terms of each type that do not have the other type.

A second conclusion follows from comparing the informal operational readings of each typing with the reduction rules of lambda calculus. In particular, consider the type $A \multimap (C \multimap !B) \multimap C \multimap A$. We may understand the correctness of this type by saying that we apply the second argument to the third and then discard the resulting discardable value. However, this informal reading assumes a particular reduction order. By the usual reduction rules of lambda calculus, we may obtain a term

$$\lambda\, q.\, \lambda\, r.\, \lambda\, s.\, q$$

in which the second and third arguments do not occur. Since this term does not have the linear type $A \multimap (C \multimap !B) \multimap C \multimap A$, the subject reduction property fails for this simplified system. This is a serious problem, since we always expect types to be preserved by reduction. If types are not preserved by reduction, then reduction of well-typed terms may lead to terms that are not well-typed. Essentially, this means that static typing does not prevent run-time type errors.

Intuitively, the failure of subject reduction seems to result from the contrast between a careful accounting of resources in linear logic and the inherent ambiguity in reduction order in the λ calculus. This implies that reduction order must be restricted in some way. The most natural approach seems to be to introduce additional constants that indicate where the operations associated with ! types are performed. This restricts the set of types in a way that makes type inference possible and also provides a convenient framework for restricting evaluation order. In particular, using explicit discard, we may say explicitly, inside a term, whether a discard happens before or after a function application.

3 The Linear Calculus

We describe the terms of linear lambda calculus in Section 3.1 and give three sets of typing rules. The first, SEQ, given in Appendix B, is the standard set of rules given by applying the Curry-Howard isomorphism to Girard's sequent calculus proof system, restricted to intuitionistic linear logic. The second system, NAT, given in Appendix C, is based on a Gentzen-style sequent calculus presentation of natural deduction for intuitionistic linear logic. The third system, NAT2, is closely related to NAT, differing only in the **!SR** and **Subst** rules. The difference between NAT and NAT2 lies in the point of view taken on whether the **!SR** rule is a "left" rule or "right" rule of the sequent calculus: it introduces a type constructor on the right, so it appears to be a "right" rule, while it depends on the form of the context, or left side of a sequent, so it may also be a "left" rule. Traditionally, right rules of sequent calculus and introduction rules of natural deduction systems are analogous, while left rules of sequent calculus correspond to a combination of elimination rules and substitution. Consequently, the translation of left rules into natural deduction should be closed under substitution while the translation of right rules should be direct.

Other researchers have independently formulated similar typing rules, although none we know of incorporate a rule of the form of the **!SR** rule of NAT. Lafont, Girard, Abramsky, and others have studied systems very similar to SEQ [GL87, Abr90]. In recent unpublished notes [Abr91, Wad91b] and an MS thesis [Mac91], systems close to NAT2 have been studied. Walder also discusses alternative rules for **!SR** and the implications of syntaxless **Subst** rule in the context of a NAT-like system. We take this parallel development of ideas as evidence that these are natural formulations of type systems based on linear logic. We show that NAT and SEQ give the same set of types to each linear term in Section 3.6, while NAT2 provides equivalence only up to a point. Technical theorems showing the "term-driven" nature of NAT and NAT2 and the "type-driven" nature of SEQ are proved in Section 3.7.

3.1 Linear terms and reduction

The linear calculus may be considered a functional programming language with fine-grained control over the use of data objects. To a first approximation, no function may refer to an argument twice without explicitly copying, nor ignore an argument without explicitly discarding it. The reason we say, "to a first approximation" is that the notion of referring to an argument twice is somewhat subtle, especially in the presence of additive type connectives. For example, a variable should appear "once" in both branches of a case statement, which is an additive operator. A more precise understanding of the restrictions on use and discard may be gained from reading the typing rules. In our presentation of the linear calculus, we have not restricted reduction order completely (in contrast to [Abr90], for example.) However, as pointed out in Section 2, the order of certain reductions must be determinate.

Using x, y, z for term variables, and t, u, v for terms, the syntax of linear lambda terms are summarized by the grammar given in Figure 1.

All the `let A be B in C` constructs bind variables in A by pattern-matching A against the result of evaluating B, and then evaluate C. For example, consider the term `let` $x \times y$ `be` $(1 \times ((\lambda x.x)1))$ `in` $(x \times y)$. First the subject term is evaluated (to 1×1), then x and y are bound (both to 1), and finally $(x \times y)$ is evaluated (in the extended context) producing a final result of (1×1).

The term `store` u is a reusable, or delayed version of u. The `copy` operation inserts multiple copies of a term `store` u, while `discard` completely eliminates a `store` u term. These `copy` and `discard` operations may be implemented by pointer manipulations (implementing sharing) or by explicit copying. The `read` construct forces evaluation of a `store` d term. The interaction between `read` and `store` is the critical point where the linear calculus determines reduction order. In other terminology, `store` is a wrapper or box which is only opened when the term must be `read`.

The reduction rules for linear lambda calculus are given in Appendix A. A linear term t *reduces* to a linear term s if $t \twoheadrightarrow s$ can be inferred from the linear cal-

term	::=	x	variable
	|	`let` $x \times y$ `be` t `in` u	bind x to car and y to cdr of t in u
	|	$(t \times u)$	eager pair (like cons in ML or lisp)
	|	$(t\,u)$	application
	|	$(\lambda x.t)$	abstraction
	|	$inr(t)$	determines right branch of case
	|	$inl(t)$	determines left branch of case
	|	`case` t `of` $inl(x) \Rightarrow u, inr(y) \Rightarrow v$	evaluate t then branch
	|	$\langle t, u \rangle$	lazy pair
	|	`let` $\langle \_, x \rangle$ `be` t `in` u	bind x to cdr of lazy pair
	|	`let` $\langle x, \_ \rangle$ `be` t `in` u	bind x to car of lazy pair
	|	`let 1 be` t `in` u	evaluate t to 1, then become u
	|	`store` u	store or delay u
	|	`discard` t `in` u	throw away t
	|	`read store` x `as` t `in` u	evaluate t to `store` t', bind x to t'
	|	`copy` $x@y$ `as` t `in` u	binds x and y to t

Figure 1: Grammar of the Linear Lambda Calculus

culus evaluation rules. Abramsky has demonstrated determinacy for a more restricted form of reduction relation in [Abr90]. The reduction relation given in Appendix A only allows "nonlinear" reductions (involving the **!WL**, **!DL**, and **!CL** reduction rules) to apply at the "top level" of a term, in the empty context. However, the remaining "linear" reduction steps may be applied anywhere in a term. Thus reduction is neither a congruence with respect to all term formation rules, nor is it deterministic. With Mitschke's δ-reduction theorem [Bar84] this reduction system can be shown to be confluent on untyped terms, even though not all untyped terms have a normal form. For typed terms, the usual cut-elimination procedure provides a proof of weak normalization for this reduction system.

3.2 Typing preliminaries

We review some standard definitions. A *type multiset* or *type environment* is a multiset of pairs $x_i{:}A_i$ of variables x_i and linear logic formulas A_i. A *typing judgement* is a type multiset Γ, a single linear term t, and a single linear logic formula A, separated by a $\vdash$, constructed as follows: $\Gamma \vdash t{:}A$. A *typing deduction* is a tree, presented with the root at the bottom, and the leaves at the top. Each branch of a deduction is a sequence of applications of the proof rules, some of which, such as $\otimes R$ in SEQ, represent branching points in the deduction tree, some, such as $\multimap R$, which extend the length of a branch, and some, such as identity, which terminate a branch. Any branch not terminated by identity or 1 **R** is called an assumption. The leaves therefore embody the type assumptions and the root the conclusion. Such a structure is said to be a deduction of the conclusion from the assumptions. A *proof* is a typing deduction with no assumptions. That is, all the branches terminate with an application or identity or 1 **R**. A closed linear term t is said to be *linearly typable* if there exists some proof with conclusion $\vdash t{:}A$ for some linear formula A.

3.3 The SEQ typing rules

The first system we will study is called SEQ, the rules for which are given in Appendix B. This formulation is due mainly to Abramsky [Abr90]. We have modified the syntax used in the original presentation slightly, but the idea is the same: take the rules for (intuitionistic) propositional linear logic and decorate them with linear terms. In the system SEQ, cut-free derivations produce linear terms in normal form. Some derivations with cut correspond to linear terms in non-normal form, and cut-elimination steps transform the term, essentially performing beta-reduction and other linear reduction steps. Performing cut-elimination on an SEQ proof is analogous to reduction in the linear calculus, although the exact correspondence is somewhat complicated.

3.4 The NAT typing rules

The typing rules for the second system, called NAT, are given in Appendix C. The main difference between

the two systems is that NAT is more "term-driven", while SEQ is more "type-driven". In Section 3.6, we show that NAT is equivalent to SEQ for proving typing judgements. This is not surprising since NAT is based on a Gentzen-style sequent calculus presentation of natural deduction for intuitionistic linear logic, while SEQ is based on a sequent calculus presentation of the same logic. The term "decorations" have been chosen in NAT so that the provable sequents in these systems are the same.

In devising the NAT rules from SEQ, we were guided by the correspondence between intuitionistic sequent calculus and natural deduction [Pra65, Appendix A]. The main idea is to interpret sequent proof rules as instructions for constructing natural deduction proofs. The sequent rules acting on the left determine constructions on the top (hypotheses) of a natural deduction proof, while sequent rules acting on the right extend the natural deduction proof from the bottom (conclusion). The **Cut** rule is interpreted by substituting a proof for a hypothesis. In order for this to work, the natural deduction proof system must have the substitution property formalized by the derived **Subst** rule in Appendix C.

A simple example that illustrates the general pattern is the tensor rule acting on the left, $\otimes\mathbf{L}$. In the conclusion of the sequent rule, there is a new term variable $z{:}(A \otimes B)$. However, we want natural deduction proofs to be closed under the operation of substituting terms for variables (hypotheses). If we use **Cut** in a sequent proof to replace $z{:}(A \otimes B)$, we end up with a sequence of proof steps whose hypotheses and conclusion are identical to the antecedent and consequent of the natural deduction $\otimes\mathbf{L}$ rule in Appendix C.

The most unusual rule of the NAT system is the **!SR** rule. This may be understood by considering the **!SR** rule of SEQ and remembering that when we substitute proofs for hypotheses in NAT, we must still have a well-formed natural deduction proof. If we follow SEQ **!SR** with **Cut**, we may use $\Delta \vdash t{:}!B$ and $x{:}!B, !\Sigma \vdash u{:}A$ to prove $\Delta, !\Sigma \vdash \texttt{store}\ u[t/x]{:}!A$. Generalizing to any number of **Cut**'s, so that natural deduction proofs will be closed under substitution, we obtain the **!SR** rule in Appendix C. This rule may without loss of generality be restricted to the case where all the Δ_i contain some non-! type.

3.5 The NAT2 typing rules

The third and final system we consider is called NAT2. The NAT2 rules are generated from the NAT rules by removing the rules of **Subst** and **!SR** from NAT and replacing them with:

$$\frac{\Sigma \vdash t{:}A \quad x{:}A, \Gamma \vdash u{:}B}{\Sigma, \Gamma \vdash \mathit{let}_{cut}\, x\ \texttt{be}\ t\ \texttt{in}\ u{:}B}\ \mathit{let}_{cut}$$

$$\frac{!\Sigma \vdash t{:}A}{!\Sigma \vdash \texttt{store}\ t{:}!A}\ \mathbf{!SR}$$

The reason for the explicit syntax of let_{cut} is that **Subst** is not a derived rule of NAT2, and for Theorem 3.2 we need some cut-like rule. These differences lead to subtlely different properties of NAT and NAT2 systems. For example, since **Subst** is syntaxless in NAT, one can show that that one need not consider **Subst** in searching for type derivations. On the other hand, NAT2 is entirely driven by term syntax, leading to a unique principle type theorem.

3.6 Equivalence of SEQ and NAT

In Theorem 3.1 below, we prove that the two systems are equivalent, that is, that any linear type judgement provable in SEQ is also provable in NAT.

We should emphasize that although NAT and SEQ are equivalent in the sense that any typing judgement provable in SEQ is provable in NAT and vice-versa, they are not equivalent with respect to operations on proofs.

Theorem 3.1 (**SEQ equiv** NAT) *A type sequent* $\Gamma \vdash t{:}A$ *is provable in* SEQ *if and only* $\Gamma \vdash t{:}A$ *is provable in* NAT.

Proof. One can show that each rule in SEQ is derivable in NAT, and vice versa, using local transformations. All the right rules, identity, are the same in both systems, and **Cut** and **Subst** take the same form. For most left rules, one may simulate the NAT version of the rule in SEQ with one application of the rule of similar name and one application of **Cut**. One may simulate the SEQ version of most left rules in NAT by using the rule of the same name and identity. The multi-hypothesis **!SR** rule of NAT is derivable in SEQ with the use of **!SR** and multiple instances of **Cut**.

One may transform instances of the **Cut** rule in SEQ as applications of **Subst**, which is a derivable rule in NAT. Upon removal of **Subst**, one may see that **Cut** in SEQ corresponds to introduction-elimination pairs of rules in NAT. ∎

There is a somewhat looser correspondence between NAT2 and SEQ, than that just claimed between NAT and SEQ.

A term t' is *related* to a term t if t can be obtained from t' by replacing occurrences $\mathit{let}_{cut} x\ \texttt{be}\ t\ \texttt{in}\ v$ with $v[t/x]$.

Theorem 3.2 (SEQ **equiv** NAT2) *A type sequent* $\Gamma \vdash t{:}A$ *is provable in* SEQ *if and only* $\Gamma \vdash t'{:}A$ *is provable in* NAT2 *for some* t' *related to* t.

This theorem may be proven in the same manner as the above, although in some cases the extra syntax of let_{cut} is used in the NAT2 term t'. The reason for let_{cut} is that **Subst** is not a derived rule of NAT2.

3.7 Subformula properties

In this section, we demonstrate the main technical difference between the three sets of typing rules. If we imagine searching for a cut-free proof of a typing derivation, beginning with a prospective conclusion and progressing toward appropriate instances of the axioms, our search will be driven by the form of the term in NAT2 and the form of the type in SEQ. To state this precisely, we begin by reviewing the routine definitions of type subformula and term subformula.

A type A is a *type subformula* of a type B if A syntacticly occurs in B. Similarly, a term t is a *term subformula* of a term s if t syntacticly occurs in s.

Lemma 3.3 (NAT2 **Term Subformula Property**) *In any proof of* $\vdash u{:}B$ *in* NAT2, *every term* t *that appears anywhere in the proof is a subformula of* u.

Note that the system NAT fails to have this property because of the form of the **!SR** rule.

Lemma 3.4 (SEQ **Type Subformula Property**) *For any cut-free proof of* $\vdash t{:}B$ *in* SEQ, *any type* A *that appears anywhere in the proof is a subformula of* B.

Note that the cut rule violates the subformula property, and so Lemma 3.4 does not hold of SEQ proofs with cut.

4 Most General Linear Type

4.1 Most General Types in NAT and SEQ

In this section, we show that every linearly typable term has a finite set of most general types. This set may be used to decide the set of types of the term. More specifically, a linear formula, A, is *more general* than another, B, if there exists a substitution σ mapping linear propositions to linear formulas such that $(A)\sigma = B$. A set, S, of formulas is more general than another, T, if every element of T is a substitution instance of some element of S. Given a term t, the typing algorithm either returns a finite set of formulas more general than all types of t, or terminates with *failure* if t has no linear type. The number of formulas in the set of most general types is bounded by an exponential function of the number of uses of `store` in the term. Without `store`, every typable term has a single most general type.

Up to **!SR**, the rules of NAT that are used in a typing derivation, and the order of application, are totally determined by the syntactic structure of the linear term. For example, if the term is a variable then the only possible proof in NAT is one use of identity. If the term is $\lambda x.t$, then the only possible rule is $\multimap$ **R**. The only freedom, except for **!SR**, is in the choice of linear types for variables and the division of a type multiset among hypotheses (in the rules with multiple hypotheses). However, type judgements contain exactly the set of free variables of a term in the type context. This property determines the division of a multiset among hypotheses of the rule. Thus, for NAT without **!SR**, we may compute the most general typing by a simple Prolog program, obtained by translating the typing rules into Horn clauses in a straightforward manner.

An example that shows the complications associated with terms of the form `store` t is

$$\lambda a.\lambda b.\ \texttt{store}\ ((\ \texttt{read}\ \ \texttt{store}\ c\ \texttt{as}\ a\ \texttt{in}\ c)\, b)$$

which has the two incomparable NAT and SEQ types

$$!(!B{\multimap}C){\multimap}!B{\multimap}!C$$
$$!(B{\multimap}!C){\multimap}B{\multimap}!!C$$

The basic idea is very similar to the example in Section 2 that involves implicit `store`. If $a{:}!A$, then the expression (`read store` c `as` a `in` c) has type A. This gives us the two typing judgements

$$a{:}!(!B{\multimap}C),\ b{:}!B \vdash (\texttt{read}\ \ \texttt{store}\ c\ \texttt{as}\ a\ \texttt{in}\ c)\, b : C$$
$$a{:}!(B{\multimap}!C),\ b{:}B \vdash (\texttt{read}\ \ \texttt{store}\ c\ \texttt{as}\ a\ \texttt{in}\ c)\, b : !C$$

The common feature of these judgements is that they both allow us to apply ! to the expression, in the first case because all of the types of free variables begin with ! and in the second case because the type of the expression begins with !. In NAT, the two typings of the expression with `store`are derived using two different substitutions in the **!SR** rule. In SEQ, the two typings are derived using **Cut** to substitute into a `store` expression in two different ways.

Accounting for the possibility of several different substitutions in the **!SR** rules (some of which are provably unnecessary), all of the NAT rules are straightforward syntax-directed rules that may be translated into Prolog Horn clauses without complication. This gives us an algorithm that finds a finite set of most general types for each linearly typable term or (since the search is bounded) terminates with failure on untypable terms.

Theorem 4.1 (MGT) *Every* NAT *typable term has a finite set of most general types. There is a unification-based algorithm that, given any term, either computes a set of most general types or halts with* failure *if the term is not typable.*

4.2 Most General Types in NAT2

In the simplified NAT2 system, **!SR** is replaced by a simple syntax-directed rule with no possibility of substitution. Consequently, the most general type of any typable linear term may be computed by a unification-based algorithm or simple Prolog program. The following most general typing theorem is due independently to Mackie [Mac91].

Theorem 4.2 (MGT) *There is a unification-based algorithm that computes a most general linear type for any* NAT2 *typable term t and terminates in* failure *on any untypable term.*

5 Type Soundness

In this section we prove a technical property commonly called "subject reduction" for both NAT and SEQ. This property is that if linear term t has type A, and t reduces to t', then t' also has type A. The term t' may also have other types; rewriting a term may allow us to deduce more typing properties. However, if the typing rules allows us to derive some property of a term, this property remains as we reduce, or evaluate, the term. Without the subject reduction property, it might be possible for a typed term to become untypable during execution. In this case, we would consider the type system "unsound" as a method for determining the absence of type errors.

We prove subject reduction by considering NAT first and then deriving the result for SEQ as a corollary. The main reason that the proof is simpler for NAT can be seen by comparing the $\multimap$**L** rule of SEQ with the **A** (application) rule of NAT. When an application $(\lambda x.t)u$ is β-reduced, the structure of the NAT rules guarantee that this was typed by the **A** rule and t was typed subject to some hypothesis about the type of variable x. The same type may be given to $t[u/x]$ using a substitution instance of this proof. In SEQ, we would need a series of detailed lemmas giving us some information about the possible structure of the typing proof for $(\lambda x.t)u$. In particular, since the sequent rule $\multimap$**L** allows arbitrary substitution, the structure of the typing proof is not determined by the form of an application.

The following lemma is the key step in the inductive proof of Theorem 5.2. It covers the case of the very general **!SR** rule, essentially stating that for one-step linear reduction, terms of ! type do not interact with any other terms. Note that the reduction relation $\rightarrow_0^{\mathcal{C}}$ does not include any of the ! reductions.

Lemma 5.1 *If $t = r[s_1/x_1, \cdots, s_n/x_n]$, and $t \rightarrow_0^{\mathcal{C}} u$ and for $1 \leq i \leq n$: $\Delta_i \vdash s_i : !B_i$*
then ($u = r'[s_1/x_1, \cdots, s_n/x_n]$ and $r \rightarrow_0^{\mathcal{C}} r'$) or
($\exists j$: $u = r[s_1/x_1, \cdots, s'_j/x_j, \cdots, s_n/x_n]$ and $s_j \rightarrow_0^{\mathcal{C}} s'_j$).

Theorem 5.2 (NAT Subject Reduction) *If there is a proof of $\vdash t : A$ in* NAT, *and $t \twoheadrightarrow s$, then there is a proof of $\vdash s : A$ in* NAT.

Proof. Induction on the derivation $t \twoheadrightarrow s$. ∎

Corollary 5.3 *If there is a proof of $\vdash t : A$ in* SEQ, *and $t \twoheadrightarrow s$, then there is a proof of $\vdash s : A$ in* SEQ.

The stronger property that if there is any typing proof of $\Sigma \vdash t : A$ for term t with free variables, and $t \twoheadrightarrow s$, then there is a typing proof of $\Sigma \vdash s : A$ does not hold. As we have seen, control over evaluation order is of critical importance in maintaining linear type soundness, as the following example demonstrates.

$$x : !A \vdash \texttt{discard}\ x\ \texttt{in}\ 1$$

The above judgement is provable in SEQ, NAT, and NAT2, but after one step of reduction, the judgement becomes $x : !A \vdash 1$ which is not provable in any type system discussed in this paper. Chirimar, Gunter, and Riecke have also noticed this failure of subject reduction for open terms [CGR92].

On the other hand, we do have this more general form of subject reduction of the reflexive transitive closure of $\rightarrow_0$. That is, we may reduce using $\rightarrow_0$ anywhere in a term and still preserve the types.

With a slight modification of the systems we are working with, an intermediate form of these subject reduction theorems is possible. If Γ and Σ are multisets, then we write $\Sigma \subseteq \Gamma$ to mean that Σ may by

obtained from Γ by removing elements or adding duplicates. The following theorem holds in a version of these type systems where the restriction that every variable occur exactly once in binding and once in use is relaxed to the restriction that every variable occur as many times in binding as it occurs in use, and all occurrences of a variable have the same type.

Theorem 5.4 (Generalized Subject Reduction) *If there is a proof of* $!\Gamma \vdash t{:}A$ *in* NAT *or* SEQ, *and* $t \twoheadrightarrow s$, *then there is a proof of* $!\Sigma \vdash s{:}A$ *in* NAT *and* SEQ, *where* $!\Sigma \subseteq !\Gamma$.

Also, a weaker form of subject reduction theorem holds for NAT2. The restrictions on reduction order are sufficient to guarantee that whenever $(\lambda x.t)u$ is reduced, under the conditions given in the theorem below, $t[u/x]$ has the same type.

Theorem 5.5 *If there is a proof of* $\vdash t{:}A$ *in* NAT2, *and* $t \twoheadrightarrow s$, *then there is a proof of* $\vdash s'{:}A$ *in* NAT2, *for some term* s' *related to* s.

6 Implementation of LC

We now give an overview of a compiled implementation of the linear calculus based on insights provided by these studies of type systems. This implementation is based on a modified version (LTim) of the Three Instruction Machine (Tim).

The Tim is an extremely simple abstract machine designed to facilitate lazy reduction of super combinator expressions [FW85, WF89, Hug84, Pey87]. The LTim extendeds the Tim with four special combinators: DELAY, FORCE, COPY, and DISCARD, and modifies some of the internal data structures of Tim to also support eager evaluation and explicit storage management. The LTim implementation was pursued for two reasons. First, it provides further evidence that the linear calculus may be executed efficiently. Second, it embodies a natural dual space memory model well suited to the execution of linear calculus terms.

The key point of departure of our implementation from the previous implementations of the linear calculus is the memory model. The LTim implements two spaces, one linear, and one nonlinear. The idea is that objects in the linear space are purely linear, and thus have a reference count of exactly one at all times. Objects in the nonlinear space represent "stored" or reusable entities. Little or no static information is available about reference counts of objects in this space. The execution model we have in mind is that a `!` or `store` instruction (corresponding to the !SR rule of linear logic) ensures that objects reside in nonlinear space. Once an object is stored, it may be discarded or copied, a `discard` operation removes a pointer to a stored object, and a `copy` operation simply copies a pointer to an object in nonlinear space, thus implementing sharing, or call-by-need. However, in this nonlinear space, objects can be referenced any number of times (including 0), requiring some form of dynamic garbage collection. In the linear space, objects are never shared; there is always exactly one reference to all objects. Thus garbage collection is not needed, since objects in that space become garbage the first time they are used, and linear objects may always be updated in place. In other words, in our execution model dynamic garbage collection is never applied to objects in linear space, but may occasionally be applied to objects in nonlinear space. Update in place is always applicable to linear objects, but is never applied to nonlinear objects.

Other implementations of the linear calculus have effectively assumed a single memory space. A potential disadvantage of the single memory space is that it obfuscates the distinction between shared and unshared objects. Lafont built an implementation of the linear calculus with the fantastic property that dynamic garbage collection is never used: all terms effectively have exactly one reference to them, and thus become garbage the first (only) time they are referenced. However, Lafont avoids garbage collection by copying, a rather costly implementation technique [GH90]. Chirimar, Gunter, and Riecke have described an implementation which also focuses on the issue of garbage collection [CGR92]. In their implementation, objects may be shared, so dynamic garbage collection is potentially required on all objects. However, the linear types of terms may be used to identify potential times at which objects may become garbage. Their implementation does not include the additives, but is extended with a recursion operator and polymorphism. Abramsky has described the implementation of a linear SECD machine further studied by Mackie [Abr90, Mac91], and went on to generalize the linear calculus to one based on classical linear logic and described an implementation based on the chemical abstract machine [BB90]. Wadler [Wad91b] has also described several implementation issues regarding the linear calculus. He points out the importance of `!(!A)` being isomorphic to `!A` (which is true in our operational model), and suggests several extensions, including, for example, arrays, let! with read-only ac-

cess, the removal of syntax for weakening and contraction, etc. Wadler also discusses the separation of types into linear and nonlinear, giving the types different syntax, very similar to our two memory spaces. We have considered only the extension of the linear calculus to include recursion and arrays, essentially as mentioned by Wadler [Wad91a].

Our LTim implementation does not count the references of integers, continuation frames, nor code. In linear logic terms, it is assumed that code, continuations, and and base integer values are of ! type. That is, they are reusable. However, arrays, structures, and cons cells are not treated in this manner. Since integer values are assumed one word long, it is more efficient to copy them, rather than sharing. Code is always assumed to be nonlinear, and is shared. Code is traditionally assumed to be static and reusable, and it is difficult to imagine an implementation taking much advantage of code-space freed up when some code is executed for the last time. Continuations are assumed to be nonlinear, and are shared. This (mis)management of the storage for continuation frames could be a serious deficiency of this implementation. Continuation frames contain a sequence of pairs of pointers into code space and data. A continuation frame is created upon entry into every combinator, and must be preserved whenever a combinator suspends computation while control is transferred to some other combinator. That is, whenever a combinator pushes a label on the argument stack, the label contains a pointer to the current continuation frame, and the space the current frame takes up cannot be reclaimed. However, nonlinear values in frames are overwritten once computed (in the lazy style of the Tim), and whole frames can be shared, instead of being copied. The penalty for this is that some traditional garbage collection mechanism must be used on frames.

All other objects are handled with explicit sharing instructions. The objects handled in this way include arrays, structures, and cons cells. These are separated into two classes: linear and nonlinear. Linear objects are not copyable, and are never referenced by two pointers at the same time. In our implementation all arrays (even linear ones) have elements which are reusable (of ! type), although the arrays can be of arbitrary dimension. A nonlinear object is essentially handled in the a traditional way, with sharing of pointers to the same object. That is, nonlinear objects may be referenced by any number of pointers simultaneously.

7 Conclusion

We have presented a linear calculus and three type inference systems: SEQ, NAT, and NAT2. We have shown that SEQ and NAT equivalent, and that NAT2 is closely related. We have demonstrated the existence of most general types and the subject reduction theorem. The linear calculus and very closely related type systems have appeared elsewhere, perhaps most well known in [Laf88, Abr90].

Also, we have implemented a two-space abstract machine based on the three instruction machine which may be used to exploit the information available in linear types to generate more efficient code. For example, one may perform update in place on arrays in linear space. Although the study of opportunities for update in place in functional languages has a long history, the linear calculus and its type systems present a logical foundation for this kind of "resource-conscious" compiler optimization.

References

[Abr90] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 1990. Special Issue on the 1990 Workshop on Math. Found. Prog. Semantics. To appear.

[Abr91] S. Abramsky. Tutorial on linear logic. Lecture Notes from Tutorial at ILPS, 1991.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Proc. 17-th ACM Symp. on Principles of Programming Languages, San Francisco*, pages 81–94, January 1990.

[Cer90] S. Cerrito. A linear semantics for allowed logic programs. In *Proc. 5th IEEE Symp. on Logic in Computer Science, Philadelphia*, June 1990.

[CGR92] J. Chirimar, C. Gunter, and J. Riecke. Linear ML. In *Lisp and Functional Programming*, 1992. To Appear.

[FW85] J. Fairbairn and S. Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *3rd Conf. on Functional Programming and Computer Architecture, Lecture Notes in Computer Science 274*, New York, 1985. Springer-Verlag.

[GH90] J. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proc. 5th*

IEEE Symp. on Logic in Computer Science, Philadelphia, June 1990.

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

[GL87] J.-Y. Girard and Y. Lafont. Linear logic and lazy computation. In *TAPSOFT '87, Volume 2*, pages 52–66. Springer LNCS 250, 1987.

[How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[Hug84] R.J.M. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, PRG-40, Oxford, 1984.

[Laf88] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.

[LMSS90] P. Lincoln, J.C. Mitchell, A. Scedrov, and N. Shankar. Decision problems for propositional linear logic. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 662–671, 1990.

[Mac91] I.C. Mackie. Lilac - a functional programming language based on linear logic. Master's thesis, Imperial College, London, 1991.

[Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Pra65] D. Prawitz. *Natural Deduction.* Almquist and Wiksell, Stockholm, 1965.

[Wad91a] P. Wadler. Linear types can change the world! IFIP TC 2 Conf. on Prog. Concepts and Methods, 1991.

[Wad91b] P. Wadler. There's no substitute for linear logic. Draft, 1991.

[WF89] S. Wray and J. Fairbairn. Non-strict languages - programming and implementation. *Computer Journal*, 32(2):142–151, 1989.

A Linear Calculus

The reduction relation of interest here is $\twoheadrightarrow$, where the notation $t \twoheadrightarrow u$ is meant to be read "t evaluates in any number of steps to u". However, to facilitate the definition, we use an auxiliary relations $\rightarrow_0$, which captures the notion of one-step linear reduction without ! rules, and $\rightarrow_0^{\mathcal{C}}$, which denotes the congruence closure of $\rightarrow_0$. The following rules are universally quantified over terms t, u, v, w, and variables x, y.

$$\multimap\mathbf{L} \qquad (\lambda x.v)u \rightarrow_0 v[u/x]$$

$$\otimes\mathbf{L} \qquad \texttt{let } x \times y \texttt{ be } (s \times t) \texttt{ in } u \rightarrow_0 u[s/x, t/y]$$

$$\oplus\mathbf{L} \qquad \texttt{case } inl(t) \texttt{ of } inl(x) \Rightarrow u, inr(y) \Rightarrow v \rightarrow_0 u[t/x]$$

$$\oplus\mathbf{L} \qquad \texttt{case } inr(t) \texttt{ of } inl(x) \Rightarrow u, inr(y) \Rightarrow v \rightarrow_0 v[t/y]$$

$$\&\,\mathbf{L1} \qquad \texttt{let } \langle x, \_ \rangle \texttt{ be } \langle v, t \rangle \texttt{ in } u \rightarrow_0 u[v/x]$$

$$\&\,\mathbf{L2} \qquad \texttt{let } \langle \_, y \rangle \texttt{ be } \langle v, t \rangle \texttt{ in } u \rightarrow_0 u[t/y]$$

$$1\mathbf{L} \qquad \texttt{let } 1 \texttt{ be } 1 \texttt{ in } u \rightarrow_0 u$$

$$\mathbf{Cong} \qquad \frac{t \rightarrow_0 v}{\mathcal{C}[t] \rightarrow_0^{\mathcal{C}} \mathcal{C}[v]}$$

As formally stated by the **Cong** rule below, the above rules may be applied anywhere in a term. The ! rules, however, may only be applied to closed terms at top level.

$$\mathbf{I} \qquad \frac{}{t \twoheadrightarrow t}$$

$$\mathbf{Trans} \qquad \frac{t \twoheadrightarrow u \qquad u \twoheadrightarrow v}{t \twoheadrightarrow v}$$

$$\mathbf{Linear} \qquad \frac{t \rightarrow_0^{\mathcal{C}} v}{t \twoheadrightarrow v}$$

$$!\mathbf{WL} \qquad \texttt{discard } t \texttt{ in } u \twoheadrightarrow u$$

$$!\mathbf{CL} \qquad \texttt{copy } x@y \texttt{ as } t \texttt{ in } u \twoheadrightarrow u[t/x, t/y]$$

$$!\mathbf{DL} \qquad \frac{t \twoheadrightarrow \texttt{ store } v \qquad v \twoheadrightarrow w}{\texttt{read } \texttt{ store } x \texttt{ as } t \texttt{ in } u \twoheadrightarrow u[w/x]}$$

B SEQ Proof Rules

I $\dfrac{}{x{:}A \vdash x{:}A}$

Cut $\dfrac{\Sigma \vdash t{:}A \qquad x{:}A, \Gamma \vdash u{:}B}{\Sigma, \Gamma \vdash u[t/x]{:}B}$

$\multimap$L $\dfrac{\Sigma \vdash t{:}A \qquad \Gamma, x{:}B \vdash u{:}C}{\Sigma, \Gamma, f{:}(A \multimap B) \vdash u[(ft)/x]{:}C}$

$\multimap$R $\dfrac{\Sigma, x{:}A \vdash t{:}B}{\Sigma \vdash \lambda x.t{:}(A \multimap B)}$

$\otimes$L $\dfrac{\Sigma, x{:}A, y{:}B \vdash t{:}C}{\Sigma, z{:}(A \otimes B) \vdash \texttt{ let } (x \times y) \texttt{ be } z \texttt{ in } t{:}C}$

$\otimes$R $\dfrac{\Sigma \vdash t{:}A \qquad \Gamma \vdash u{:}B}{\Sigma, \Gamma \vdash (t \times u){:}(A \otimes B)}$

$\oplus$L $\dfrac{\Sigma, x{:}A \vdash u{:}C \qquad \Sigma, y{:}B \vdash v{:}C}{\Sigma, z{:}(A \oplus B) \vdash \texttt{case } z \texttt{ of } inl(x) \Rightarrow u, inr(y) \Rightarrow v{:}C}$

$\oplus$R1 $\dfrac{\Sigma \vdash t{:}A}{\Sigma \vdash inl(t){:}(A \oplus B)}$

$\oplus$R2 $\dfrac{\Sigma \vdash u{:}B}{\Sigma \vdash inr(u){:}(A \oplus B)}$

& L1 $\dfrac{\Sigma, x{:}A \vdash t{:}C}{\Sigma, z{:}(A \,\&\, B) \vdash \texttt{ let } \langle x, \_ \rangle \texttt{ be } z \texttt{ in } t{:}C}$

& L2 $\dfrac{\Sigma, y{:}B \vdash t{:}C}{\Sigma, z{:}(A \,\&\, B) \vdash \texttt{ let } \langle \_, x \rangle \texttt{ be } z \texttt{ in } t{:}C}$

& R $\dfrac{\Sigma \vdash t{:}A \qquad \Sigma \vdash u{:}B}{\Sigma \vdash \langle t, u \rangle{:}(A \,\&\, B)}$

!WL $\dfrac{\Sigma \vdash t{:}A}{\Sigma, z{:}!B \vdash \texttt{ discard } z \texttt{ in } t{:}A}$

!DL $\dfrac{\Sigma, x{:}A \vdash t{:}B}{\Sigma, z{:}!A \vdash \texttt{ read store } x \texttt{ as } z \texttt{ in } t{:}B}$

!CL $\dfrac{\Sigma, x{:}!A, y{:}!A \vdash t{:}B}{\Sigma, z{:}!A \vdash \texttt{ copy } x@y \texttt{ as } z \texttt{ in } t{:}B}$

!SR $\dfrac{!\Sigma \vdash t{:}A}{!\Sigma \vdash \texttt{ store } t{:}!A}$

1L $\dfrac{\Sigma \vdash t{:}A}{\Sigma, z{:}1 \vdash \texttt{ let } 1 \texttt{ be } z \texttt{ in } t{:}A}$

1R $\dfrac{}{\vdash 1{:}1}$

C NAT Proof Rules

I $\dfrac{}{x{:}A \vdash x{:}A}$

Subst $\dfrac{\Sigma \vdash t{:}A \quad x{:}A, \Gamma \vdash u{:}B}{\Sigma, \Gamma \vdash u[t/x]{:}B}$

A $\dfrac{\Delta \vdash u{:}(A \multimap B) \qquad \Sigma \vdash t{:}A}{\Sigma, \Delta \vdash (u\,t){:}B}$

$\multimap$R $\dfrac{\Sigma, x{:}A \vdash t{:}B}{\Sigma \vdash \lambda x.t{:}(A \multimap B)}$

$\otimes$L $\dfrac{\Delta \vdash u{:}(A \otimes B) \quad \Sigma, x{:}A, y{:}B \vdash t{:}C}{\Sigma, \Delta \vdash \texttt{ let } (x \times y) \texttt{ be } u \texttt{ in } t{:}C}$

$\otimes$R $\dfrac{\Sigma \vdash u{:}A \qquad \Gamma \vdash t{:}B}{\Sigma, \Gamma \vdash (u \times t){:}(A \otimes B)}$

$\oplus$L $\dfrac{\Delta \vdash t{:}(A \oplus B) \quad \Sigma, x{:}A \vdash u{:}C \quad \Sigma, y{:}B \vdash v{:}C}{\Sigma, \Delta \vdash \texttt{ case } t \texttt{ of } inl(x) \Rightarrow u, inr(y) \Rightarrow v{:}C}$

$\oplus$R1 $\dfrac{\Sigma \vdash t{:}A}{\Sigma \vdash inl(t){:}(A \oplus B)}$

$\oplus$R2 $\dfrac{\Sigma \vdash t{:}B}{\Sigma \vdash inr(t){:}(A \oplus B)}$

& L1 $\dfrac{\Delta \vdash u{:}(A \,\&\, B) \qquad \Sigma, x{:}A \vdash t{:}C}{\Sigma, \Delta \vdash \texttt{ let } \langle x, \_ \rangle \texttt{ be } u \texttt{ in } t{:}C}$

& L2 $\dfrac{\Delta \vdash u{:}(A \,\&\, B) \qquad \Sigma, y{:}B \vdash t{:}C}{\Sigma, \Delta \vdash \texttt{ let } \langle \_, y \rangle \texttt{ be } u \texttt{ in } t{:}C}$

& R $\dfrac{\Sigma \vdash t{:}A \qquad \Sigma \vdash u{:}B}{\Sigma \vdash \langle t, u \rangle{:}(A \,\&\, B)}$

!WL $\dfrac{\Delta \vdash u{:}!A \qquad \Sigma \vdash t{:}B}{\Sigma, \Delta \vdash \texttt{ discard } u \texttt{ in } t{:}B}$

!DL $\dfrac{\Delta \vdash u{:}!A \qquad \Sigma, x{:}A \vdash t{:}B}{\Sigma, \Delta \vdash \texttt{ read store } x \texttt{ as } u \texttt{ in } t{:}B}$

!CL $\dfrac{\Delta \vdash u{:}!A \qquad \Sigma, x{:}!A, y{:}!A \vdash t{:}B}{\Sigma, \Delta \vdash \texttt{ copy } x@y \texttt{ as } u \texttt{ in } t{:}B}$

!SR $\dfrac{\Delta_i \vdash t_i{:}!B_i \qquad x_i{:}!B_i, !\Sigma \vdash u{:}A}{\Delta_1 \cdots \Delta_n, !\Sigma \vdash \texttt{ store } u[t_1/x_1 \cdots t_n/x_n]{:}!A}$

1L $\dfrac{\Delta \vdash u{:}1 \qquad \Sigma \vdash t{:}A}{\Sigma, \Delta \vdash \texttt{ let } 1 \texttt{ be } u \texttt{ in } t{:}A}$

1R $\dfrac{}{\vdash 1{:}1}$

The **Subst** rule is derivable in the rest of the system.