CONJUNCTIVE POLYMORPHIC TYPE CHECKING WITH EXPLICIT TYPES

by

Kevin E. Flannery

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

APPROVED:

Johannes J. Martin, Chairman

James Arthur, Co-chairman

Patrick Bixler

Daniel Farkas

John Roach

April, 1989

Blacksburg, Virginia

Acknowledgments

It is a privilege to thank my advisor, Dr. Johannes J. Martin, for his invaluable guidance throughout the course of this research, and for his encouragement and unlimited patience. Special thanks go to Dr. James Arthur for serving as committee co-chairman during the latter part of this research, as well as to committee members Dr. Patrick Bixler, Dr. Daniel Farkas and Dr. John Roach for their exceptional cooperation and interest.

Table of Contents

Table of Notation

Symbol	Usual Interpretation
$\alpha, \beta, \gamma, \sigma, \tau, \rho, \zeta, \nu$	range over type expressions
a, b, c	type variables
x, y	lambda variables
A, B, C, D	range over type ideals
S, K, I, Y, B	functional combinators
e, f, g, h	range over computational expressions
D	semantic domain of computations
T	semantic domain of types
$\perp$	least element of D
$\perp_T$	least element of T
ρ	ranges over type environments
e f	e applied to f
e f g	(e f) g
$\alpha\rightarrow\beta\rightarrow\gamma$	$\alpha\rightarrow(\beta\rightarrow\gamma)$
$D\rightarrow D$	domain of continuous functions over D
$A\rightarrow B$	ideal of functions f such that $f[A] \subseteq B$
G	ranges over type assignments (assignments of types to computational variables)
$e : \tau$	e has type τ (in a given type language)
$G[x:\tau]$	the type assignment G changed to map x to τ

(Symbol)	(Usual Interpretation)
$e[x \leftarrow f]$	f replaced for all occurrences of the variable x in e
$\tau[a \leftarrow \sigma]$	the type σ replaced for the type variable a in τ
$[x_1:=e_1,...,x_n:=e_n]$	substitution function which simultaneously substitutes e_i for x_i
$[a_1:=\tau_1,...,a_n:=\tau_n]$	simultaneous substitution of types τ_i for type variables a_i
S	ranges over substitution functions
P	ranges over type substitution functions
$S[e\backslash x]$	substitution S changed to map x to e
$P[\tau\backslash a]$	type substitution changed to map a to τ
$\langle A_i \rangle_i$	infinite sequence of ideals A_1, A_2, ...
$\Diamond$	end of proof

1

Introduction

An expressive type language and the ability to do compile-time type inference are desirable goals in language design, but the attainment of the former may preclude the possibility of the latter. Specifically, the type conjunction operator (type intersection) induces a rich type language at the expense of decidability of the typeable expressions. Two extreme alternatives to this dilemma are to abandon type inference (and force the programmer to, essentially, supply a derivation for his type claims) or to abandon (or restrict) type conjunction. This work presents a third alternative in which the programmer, at times, may be required to supply explicit types in order for type inference to succeed. In this way, the power of conjunctive types is preserved, yet type inference can be done for many powerful functions.

In this dissertation, we give a typed combinator-based functional language whose types types are closed under function-type formation ($\rightarrow$), conjunction ($\cap$) and a restricted form of type fixedpoint (μ). The language features user-supplied type information and limited automatic type inference. The user-added type information is necessary, since without it type checking

for this system is undecidable, and automatic type inference is desirable since it allows the programmer to specify incomplete type information. This combination can be viewed as a compromise between a language in which automatic derivation of a user-supplied type is possible, and a system in which automatic *checking* of a user-supplied type derivation occurs -- the former places the task of the type derivation completely on the type checker, a strategy not possible for sufficiently powerful type systems, the latter completely on the programmer. Our language captures the typing power of an undecidable type system, yet type checking requires only a moderate amount of type information of the programmer.

1.1 Typed Languages

The notion of attributing a "type" to an object in mathematics dates back to Russell [Russell]. Russell discovered that some descriptions of sets which appear to be valid (grammatically) are in fact nonsense, e.g., the "Russell Paradox," the set of all sets not containing themselves. To rule out such constructions, Russell required that a set contain elements of some "type" different from the type of the set itself. Constructions of sets using well-typed elements were then guaranteed to be meaningful.

In programming languages, there is a similar motivation for types. An integer function, for example, will produce meaningless results when applied to character representations. It is more desirable to detect such misinterpretations of data before the program is executed, rather than at the moment they occur, hence we equate "type" and "static type" throughout. Once type checked, programs conform to type rules, thus functions will not misinterpret their arguments.

It is common to view a typed language as two languages: a language of computational expressions and a language of type expressions. The computational expressions specify computations to be performed, and the type expressions give information about those computations. Generally, the information a type gives about a primitive object (such as 4 or 'A') is how the representation of the object is to be interpreted, and the information a type gives about a function is how the representations of its domain and codomain are to be interpreted. In a typed language, there is always a relation "has type" between computational and type expressions, and this relation can usually be defined inductively using a set of typing rules. For example, all typed languages which accommodate functional application have the following obvious rule, in some form:

"If f is a function from values of type τ_1 to values of type τ_2, and x has type τ_1,then f(x) has type τ_2"

which may be given in an abbreviated form: $f(x) : \tau_2$ if $f : \tau_1 \rightarrow \tau_2, x : \tau_1$.

We call the noninductive rules, such as "+ : Integer × Integer → Integer," axioms, and we call the type associated with an expression e in an axiom the axiomatic type of e.

For our purposes, the term *typed language* (also called *type system*) will mean a set of computational expressions, a set of type expressions, and a set of type rules which define when a computational expression e "has type" τ (written $e:\tau$). *Type checking* is determining whether $e:\tau$ can be derived from the rules, given e and τ. *Type inference* is finding a type for a given expression. An expression is *typeable* if it has at least one type.

1.2 Functional Languages

Functional languages have a long history. One of the first was LISP, for LISt Processing language. LISP was introduced in 1958 by McCarthy as a language for artificial intelligence applications [McCar58] and has since enjoyed a wealth of popularity among AI researchers, due to its flexibility and the concise form of its programs. What distinguished LISP from languages before it (and many after) is the freedom afforded the programmer in composing and applying functions. The weak type structure of LISP has been viewed as both an advantage and disadvantage: the programmer is free to apply the primitive list manipulating functions in virtually any fashion, since the only types are lists, functions and atoms, but the price is the increased risk of a function misinterpreting its arguments at run time.

In 1977, Backus, in his Turing Award lecture, described a functional language FP which was based on primitive functions with function- and object-forming operators. In his paper [Back77], Backus identifies the assignment statement present in all procedural languages as the "bottleneck" in the design of program logic, and shows how an applicative language facilitates concise and verifiable programming. The syntactic and semantic simplicity of FP's variable-free programs makes FP easy to implement as well.

Around the same time, the language ML was designed in Edinburgh by Gordon, Milner and others [BMetc78] as a meta-language for LCF, an automatic theorem-proving tool, but soon became recognized as a general-purpose programming language. ML was the first language to feature strong typing for polymorphic functions, and its "parametric polymorphism" was adopted later on in (functional) languages such as FQL, HOPE, GALILEO, MIRANDA, POLY, and a typed version of Scheme (a dialect of LISP).

Most of the languages mentioned above are based on the lambda calculus and therefore include lambda abstraction in their syntax, i.e., expressions of the form $\lambda x.E$ where $(\lambda x.E)F = E[x \leftarrow F]$ = "E with F replaced for x". A notable exception is Backus' FP. In FP, the programmer combines a set of primitive functions using composition and application to form new functions. Given a sufficiently rich set of primitives, one achieves the power of the lambda calculus without the need of variables or lambda abstraction. This syntactic simplification yields a clean semantics and makes the design of the types and type rules as simple as possible.

1.3 Polymorphism

Type checking is easily done for languages with sufficiently limited expressive power, but it becomes nontrivial when the language contains polymorphic functions. A function is *polymorphic* if it can operate on values of families of types rather than values of a single type. An example is the pair-formation function p defined by p x y = <x,y> which has types $a \rightarrow (b \rightarrow (a \times b))$ for all types a and b. A more interesting example is Milner's function Map [Mil78]: Map f <x, y, z, ...> = <f(x), f(y), f(z), ...> having types $(a \rightarrow b) \rightarrow (\text{list of } a) \rightarrow (\text{list of } b)$ for all types a and b. A typed language with polymorphic functions provides the flexibility of a typeless language and the security of strong typing.

Type expressions containing quantified variables (such as a and b above) are called *polymorphic types* (or polytypes, as Milner called them), and those not containing variables are called *monomorphic types* (monotypes). Polymorphic type checking can be a difficult problem, depending on the axioms and inference rules, and may even be undecidable.

Although the term "polymorphism" was not used until the 60s (Strachey [Str67] seems to be the first), the notion of a function having many types was well known to Church and Curry, who studied abstract functional behavior [Chu40, Chu41, C&F58].

Church used the Lambda Calculus to study functional application. His language of lambda terms has the following syntax (Var is an infinite, countable set of variables):

Exp ::= Var | λ Var . Exp | Exp Exp

A term $\lambda x.e$ is interpreted as the function $f(x) = e$, hence an application $(\lambda x.e_1)e_2$ is "performed" by replacing all occurrences of x not bound by λ in e_1 by e_2, and the resulting term is denoted by $e_1[x \leftarrow e_2]$. This is called a *reduction*, and can occur anywhere inside an expression. An expression on which no reductions can be performed is said to be in *normal form.*. An expression e is said to be *normalizable* (respectively, *strongly normalizable*) if some (resp., every) reduction sequence starting with e results in a normal form.

Church noticed that a simple type system could be imposed on the lambda calculus. Let B be a nonempty set of basic types (with arbitrary interpretation) and build the set of simple types ST by closing B under '$\rightarrow$' formation, and let A be a finite assignment of types to variables. Inductively define a relation R of assignments A and lambda expressions e to simple types τ as follows:

1) $(A, x, \tau) \in R$ whenever $A(x) = \tau$

2) $(A, ef, \tau) \in R$ whenever $(A, e, \sigma \rightarrow \tau) \in R$ and $(A, f, \sigma) \in R$

3) $(A, \lambda x.e, \sigma\rightarrow\tau) \in R$ whenever $(A[x:\sigma], e, \tau) \in R$, where $A[x:\sigma]$ is the assignment A changed to map x to s

We write $A \mid e : \tau$ for $(A, e, \tau) \in R$, and simply $e : \tau$ when $A=\emptyset$. A lambda expression e is said to be *simply typed* if $A \mid e : \tau$ for some A and τ. The set of simply typed expressions does not depend on B Turing showed that all simply typed expressions have a normal form [see Gandy80], and this was later strengthened to the strong normalizability of the language [see FLO83].

It is easy to show that all simply typed lambda expressions have a principal form; e.g., all types for $\lambda x.x$ must be of the form $t\rightarrow t$ where t ranges over Texp. (This was known to Curry--he called the form "functional character".) Due to results of Hindley and Morris [Hind69, Mor68], one could use Robinson's Unification Algorithm [Rob65] to find the principal form for any simply typed lambda expression; thus, the simply typed lambda expressions are decidable.

Curry used Combinatory Logic to study functional behavior [C&F58]. A CL consists of a set of primitive objects called Pob (possibly containing indeterminates), a complete set of objects called Ob generated by closing Pob under free binary application (denoted by juxtaposition) and a set of equality axioms. CL's are simpler than the lambda calculus since they do not contain variable bindings. The simplest CL used by Curry consists of the primitive objects S and K and the axioms S A B C = AC(BC) and K A B = A. S and K naturally correspond to the closed lambda expressions $\lambda x\lambda y\lambda z.xz(yz)$ and $\lambda x\lambda y.x$, and the functional characters of S and K observed by Curry and Hindley are their principal types when interpreted as simply typed lambda expression. Thus, one can infer simple types for combinations in classical CL:

1) $S : (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c))$ for all $a,b,c \in ST$

2) $K : a \rightarrow (b \rightarrow a)$ for all $a,b,c \in ST$

3) $ef : \tau$ if $e : \sigma \rightarrow \tau$ and $f : \sigma$

The principal types in CL and the lambda calculus were viewed by many as "type schemes" from which all the "real" types could be generated; e.g., $t \rightarrow t$ denotes the family of types INT→INT, BOOL→BOOL, etc. Type schemes later came to be known as *parametric types*. We can add parametric types to the simply typed lambda calculus by adding an infinite set of type variables Tvar to B. Then, principal type forms are formally derivable as type expressions containing variables. The axioms for the parametric type system derived for CL can then be stated in the following way ($a,b,c \in$ Tvar):

1-a) $S : (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c))$

-b) $K : a \rightarrow (b \rightarrow a)$

2) $e : \tau$ if $e : \sigma$ and $\tau = P\sigma$ for some substitution P of types for type variables

3) $ef : \tau$ if $e : \sigma \rightarrow \tau$ and $f : \sigma$

Hindley suggests that functions possessing such types are not functions at all, but are a *family* of functions instead. In the context of programming languages, however, a piece of code which computes a well defined INT when given an INT, a well defined BOOL when given a BOOL, etc, *does* have the types INT→INT, BOOL→BOOL, etc, using the inference rule for application, and whether we semantically associate a single generic function or a set of functions with such a program is immaterial, provided that the *same* piece of code having type INT→INT also has type BOOL→BOOL, etc. (i.e., overloaded operators are not polymorphic).

Milner was the first to incorporate parametric types into a full programming language, ML [Mil78]. ML was based on the lambda calculus and included the usual primitive objects (lists, pairs, atoms) along with lambda abstraction, recursion and environment manipulation. Milner used a lattice of sets of computational values to define the semantics of his types. Polymorphic types were then interpreted as infinite intersections of monomorphic types. When restricted to the lambda calculus, typeability in ML is equivalent to typeability in the simply typed lambda calculus, and principal types for lambda expressions of ML can be derived using unification. The Milner-style polymorphism appears in such languages as B [Mee83], HOPE [BMS80], and a typed PROLOG [M&O84].

ML was a vast improvement over non-polymorphic typed languages, however, there were functions which could not be typed in ML (two famous examples are the self application function and the least fixed point function).

Girard [Gir72] and Reynolds [Rey74] independently proposed a theory of second-order types (called the second-order typed lambda calculus). Here, type variables are introduced and lambda abstraction of these variables over function and type expressions is allowed; e.g., the polymorphic identity function is written as $(\Lambda t.\lambda x{:}t.x)$ and has type $\Delta t.t \rightarrow t$. McCracken [McC79] gave a semantics for this language using a modification of Scott's domain of retracts [Scott76]. We say a lambda expression e has a second-order type if there is an e' in the second-order typed lambda calculus which is identical to e when the types and type abstractions are stripped away. Second-order types were an improvement over parametric types: all simply typed expressions were typeable with second-order types, and all lambda expressions in normal form have a second-order type (thus $\lambda x.xx$ is typeable). It is also true that any expression which is second-order typeable is strongly normalizable [FLO83] (thus the least fixed point function is not typeable), but there are many (strongly) normalizable

expressions which are not second-order typeable [see Lei83]. The second-order typeable expressions is a decidable set (see [McC79]). Type abstraction can be found in the languages Russell [D&D80], Pebble [B&L84], PASQUAL [Ten75] and [Car86].

In [Cop80], Coppo gave an extension of the parametric types of Milner in which conjunctions (intersections) of types may be formed. This system allows more lambda expressions to be typed than in the second-order lambda calculus, in fact, the set of typeable expressions are exactly the strongly normalizable ones [Cop80, Lei83]. The rules for type inference are simpler, too; they are just the rules for the parametric system with the following additions:

a) $A \mid e:\sigma \cap \tau$ if $A \mid e:\sigma$ and $A \mid e:\tau$

b) $A \mid e:\sigma$ and $A \mid e:\tau$ if $A \mid e:\sigma \cap \tau$

As in Milner's system, types are interpreted as sets, and polymorphic types are interpreted as infinite intersections. Since the set of strongly normalizable lambda expressions is undecidable, typeability of expressions in the conjunctive system is not decidable. This implies the undecidability of conjunctive type checking in general, since e typeable $\Leftrightarrow$ $(\lambda x.\lambda y.y)\ e : a \rightarrow a$.

A restricted version of Coppo's system is proposed by Ghosh-Roy [G-R88]. Ghosh-Roy extends the unification algorithm used in parametric type inference to accommodate many conjunctive types. In that language, the type checking algorithm *itself* defines when an expression has a given type.

MacQueen, Sethi and Plotkin [MS82, MPS84] describe a system featuring Π- and Σ-quantification (Π-quantification is a variant of Reynold's Δ-abstraction above), conjunction, union, and type fixedpoints.

Type fixedpoints satisfy equations of the form $t = e(t)$, for a restricted form of type expression $e(x)$. These fixedpoints were shown to be unique in a common semantic domain of types, and are denoted by $\mu x.e(x)$. For example, in that system a type of the self-application function $\lambda x.x(x)$ would be $\mu t.t \rightarrow \tau$. Type fixedpoints add considerable typing power to a language; e.g., they can be used to type the least fixedpoint operator $Y = \lambda f.(\lambda x.f(xx))(\lambda y.f(yy))$, which is not typeable in the conjunctive type discipline. In fact, type fixedpoints allow all pure lambda expressions to be typed (although many have the meaningless type $\mu t.t \rightarrow t$).

Type checking is not decidable in MacQueen's system, but undecidability does not come from the type fixedpoints, as one can add them to the parametric system and still do type inference using circular unification [see ASU86]. The main contribution of [MPS84] is in showing the semantic validity of the type fixedpoints.

In a general setting, Leivant [Lei83] gives an analysis of type conjunction, Π-quantification and (Reynolds') abstraction as an extension to Milner's parametric types and reveals that the typeable expressions in the conjunctive type discipline properly contain those in the quantificational and abstractional disciplines, making conjunction types more flexible, and therefore more desirable to implement.

In spite of their power and simplicity, at this time there exists no implementation of a polymorphic type system based on conjunctive types. The problem is that type checking in the conjunctive discipline is not decidable.

1.4 Explicit Types

In the face of undecidability, we could implement some decidable subset of the conjunctive type system. This subset can be determined by the type checking algorithm itself, e.g., [G-R88] extends the standard parametric polymorphic type checking algorithm [Mil78], which is based on unification, in order to accommodate (to a degree) conjunction types. This strategy, in which the typing rules are replaced by a type checking algorithm, has a disadvantage: the class of typeable expression in such a system is not easily defined, i.e., it is difficult to predict which expressions will pass type checking without actually applying the algorithm. More importantly, although more expressive than the parametric type system of ML, the resulting conjunction type system must fall short of its full expressive potential, that is, it is doomed to be a weaker type system than Coppo's. We will run into this problem, in fact, even if we choose a less ad-hoc approach of restricting the rules themselves in some way ([Lei83] suggests a restriction on the functional level in which an $\cap$ may occur).

There is an alternative to restricting the type system. Obviously, if the programmer gives a *derivation* (i.e., a proof) with each type claim, then it is a simple matter to check its correctness. Thus, any typed language can be made decidable by incorporating type derivations into the types. Is this a reasonable approach to take? Certainly we may expect the programmer to *be able to* specify a derivation for his type claim, since he must have had some reason for making the claim in the first place. A programming language which requires derivations to be given for all types, however, would make programming tedious. The

question is, should the programmer always have to give the type checker a *complete* derivation? The answer is "no."

Our approach to type checking requires explicit type information (actual types used in the derivation) from the programmer at certain points in his program. With this information, the type checker can infer a principal type (of sorts) for parts of the program, and hence can do type checking *at no cost to the expressive power of the type language.*

For simplicity, the language we propose is based on the combinators S and K from Combinatory Logic [C&F58] which form an ample basis for computation. Our type system includes a type conjunction operator $\cap$ and a type fixedpoint operator μ. The language without the fixedpoint operator is powerful enough to type all SK combinations typeable in the system of [Cop80], and with the fixedpoint operator can type (with nontrivial types) more SK combinations than any system implemented at this time.

1.5 Overview

In chapter 2, we give the typed language TCL (for Typed Combinatory Logic). A semantics is given for the types using MacQueen's model of ideals. We also show the semantic soundness of the typing rules.

In chapter 3, it is shown that the typeable expressions in TCL are the strongly-normalizable S-K combinations, and hence are an undecidable set. In doing this, we show that the standard isomorphisms between the lambda calculus and combinatory logic preserve strong normalizability.

In chapter 4, we incorporate user-supplied type information into the syntax and typing rules of TCL, giving XTCL, and a type checking algorithm for XTCL is given.

In chapter 5, we extend TCL and XTCL to include type fixedpoints and call these systems TCLμ and XTCLμ, respectively. Semantic soundness is shown for TCLμ, and a type checking algorithm for XTCLμ is discussed.

Chapter 6 is devoted to practical issues. We suggest a way to reduce the amount of user-supplied type information necessary for type checking. The incorporation of abstract types into the language is also addressed.

Chapter 7 contains a summary and directions for future research.

2

The Language TCL

In this chapter, we define the language TCL (for "Typed Combinatory Logic"), give its semantics, and prove a few of its properties, including the semantic soundness of the typing rules. The computational portion of TCL provides primitive combinators (S and K) and allows expressions to be constructed with arbitrary applications of the primitives. The typed portion of TCL is the parametric system extended by the type conjunction operator, $\cap$. The type system for TCL can be viewed as an adaptation of Coppo's conjunctive types to a combinator-based language (Coppo's type system was constructed for a language based on lambda abstraction). When restricted to SK combinations, $e:\tau$ in Coppo's system implies $e:\tau$ in TCL, but not conversely (see Chapter 3), however, the two systems assign nontrivial types to the same set of SK combinations.

Before defining TCL, we give a motivation for extending Milner's parametric types.

2.1 Extending Parametric Types

To illustrate the limitations of parametric types, a language is given which is based on lambda abstraction and which has parametric types. We show that type checking fails for a simple

expression, and that the problem can be corrected using one of two well-known extensions to parametric types.

The language we present uses the mechanism of lambda abstraction to denote functions. Essentially, $\lambda x.e$ denotes the function $f(x) = e$, thus the application $(\lambda x.e)u$ is performed by substituting unbound occurrences of x in e by u-- $(\lambda x.e)u = f(u) = e[x \leftarrow u]$. The advantage of this notation is that there is never a need to name functions in order to define them (i.e., "f" in f(x) is not needed).

The Parametric System (with Pairs, Booleans and Natural Numbers)

syntax The computational language E is the lambda calculus with the addition of

i) pairs and projection functions P_1 and P_2 with the meanings

$P_1 \langle e, f \rangle = e$ and $P_2 \langle e, f \rangle = f$

ii) natural numbers and the functions is0 and succ with the meanings

is0 n = {true if n=0, false if n > 0}

succ n = {the number (n + 1) }

iii) true and false, and a conditional function "if" with the meaning

((if b) e) f = {e if b=true, f if b=false}

(Note that the behavior of P_1, P_2, is0, succ and if when applied to values outside their domains (e.g., P_1 5) is undefined, and such applications would constitute an error.)

E ::= Var {an infinite set of variables}

| Nat {the natural numbers}

| P_1 | P_2 {the pair projection functions}

| is0 {the test for zero}

| succ {the successor function}

| true | false {the boolean constants}

| if {the conditional function}

| < E, E > {pair formation}

| λVar . E {lambda abstraction}

| E E {application, associating to the left}

(T is the language of type expressions)

T ::= Tvar {an infinite set of type variables}

| INT {the type constant for natural numbers}

| BOOL {the type constant for true and false}

| T → T {function type formation}

| T × T {pair type formation}

type inference rules These rules inductively define a ternary relation between type assignments A, expressions e and types τ, and the relation is written as "A | e : τ". The assignment A assigns a type expression σ in T to each free variable (a variable is free if it is not bound by a λ) in e. (The reason for the type assignment is that a statement of the form "e has type τ" is meaningless without making an assumption on the types of the free variables of e.) Of course, if there are no free variables in e, then "e : τ" is taken to mean "ø | e : τ" where ø is the empty assignment. For type assignments A, variables x in Var and types τ in T, we use "A[x:τ]" to mean the mapping obtained by changing A so that it maps x to τ. Precisely, for variables y in Var,

$A[x{:}\tau]\ y \quad = \quad \tau$, if $x=y$

$A\ y$, if $x \neq y$

For all

assignments A of types to variables in Var,

expressions e and f in E,

natural numbers n in Nat,

variables x in Var,

and type expressions σ and τ in T,

1-a) $A \mid n : INT$

-b) $A \mid true : BOOL$

-c) $A \mid false : BOOL$

-d) $A[x{:}\tau] \mid x : \tau$

-e) $A \mid P1 : (\sigma \times \tau) \rightarrow \sigma$

-f) $A \mid P2 : (\sigma \times \tau) \rightarrow \tau$

-g) $A \mid is0 : INT \rightarrow BOOL$

-h) $A \mid succ : INT \rightarrow INT$

-i) $A \mid if : BOOL \rightarrow (\tau \rightarrow (\tau \rightarrow \tau))$)

2) $A \mid \langle e, f \rangle : \sigma \times \tau$ if $A \mid e : \sigma$ and $A \mid f : \tau$

3) $A \mid \lambda x.e : \sigma \rightarrow \tau$ if $A[x{:}\sigma] \mid e : \tau$

4) $A \mid e\,f : \tau$ if $A \mid e : \sigma \rightarrow \tau$ and $A \mid f : \sigma$

As an example, consider the function Switch = $\lambda x.\langle P_2 x, P_1 x \rangle$ which takes a pair <e,f> and returns <f,e>. Using the type rules, one can derive that $A \mid Switch : INT \times BOOL \rightarrow BOOL \times INT$ for any assignment A:

Statement	*Rule*
A) $A[x:INT \times BOOL] \mid x : INT \times BOOL$	(1-d)
B) $A[x:INT \times BOOL] \mid P_1 : INT \times BOOL \rightarrow INT$	(1-e)
C) $A[x:INT \times BOOL] \mid P_2 : INT \times BOOL \rightarrow BOOL$	(1-f)
D) $A[x:INT \times BOOL] \mid P_1 x : INT$	(4)
E) $A[x:INT \times BOOL] \mid P_2 x : BOOL$	(4)
F) $A[x:INT \times BOOL] \mid \langle P_2 x, P_1 x \rangle : BOOL \times INT$	(2)
G) $A \mid \lambda x . \langle P_2 x, P_1 x \rangle : INT \times BOOL \rightarrow BOOL \times INT$	(3)

A property of the parametric system is that all closed expressions e (i.e., e has no free variables) which are typeable have principal types, that is, they have types from which all other types are generated via substitutions. Thus, if σ is a principal type for e, then

$e : \tau \Rightarrow \tau = S\sigma$ for some substitution S of types for type variables in σ.

For example, it is obvious from rule 1 that all types for P_1 are of the form $(u \times v) \rightarrow u$, hence $(u \times v) \rightarrow u$ is a principal type for P_1, where u and v are type variables. Similarly, P_2 has a principal type of $(w \times z) \rightarrow z$.

Also, since closed expressions have no free variables, any type statement $A \mid e : \tau$ derivable from the rules is also derivable with $A = \emptyset$, provided e is closed.

Consider the function $h = \lambda f. \lambda x. \lambda y. \langle fx, fy \rangle$. We will show that the expression

((h Switch) <5, 6>) <true, false>

has no type in the parametric system defined above. Our approach is straightforward:

1. Derive a principal type for Switch.
2. Derive a principal type for h.
3. Derive a principal type for h Switch.
4. Derive a principal type for (h Switch) <5,6>.
5. Show that no type $\sigma \rightarrow \tau$ exists for (h Switch) <5,6> where σ is a type for <true,false>.

1. Derive a principal type for Switch. Suppose Switch : ρ. Since Switch is a λ-expression, a derivation of any type for Switch must be based on rule 3, thus ρ must be of the form $\sigma \rightarrow \tau$, where $\emptyset[x:\sigma] \mid \langle P_2x, P_1x \rangle : \tau$. By rule 2, τ must be of the form $\tau_1 \times \tau_2$ where $\emptyset[x:\sigma] \mid P_2x : \tau_1$ and $\emptyset[x:\sigma] \mid P_1x : \tau_2$.

Now $\emptyset[x:\sigma] \mid P_2x : \tau_1$ is must be derived using rule 4, thus $\emptyset[x:\sigma] \mid P2 : \gamma \rightarrow \tau_1$ and $\emptyset[x:\sigma] \mid x:\gamma$ for some type γ. This means $\gamma = \sigma$ by rule 1, thus $\emptyset[x:\sigma] \mid P_2 : \sigma \rightarrow \tau_1$ and $\emptyset[x:\sigma] \mid x : \sigma$. But any type for P_2 is of the form $(u \times v) \rightarrow v$, thus $\sigma \rightarrow \tau_1 = (u \times v) \rightarrow v$, implying that $\sigma = u \times v$ and $v = \tau_1$ for some u and v.

Similarly, $\emptyset[x:\sigma] \mid P_1x : \tau_2$ implies $\emptyset[x:\sigma] \mid P_1 : \sigma \rightarrow \tau_2$ and $\emptyset[x:\sigma] \mid x : \sigma$. Since all types of P_1 must be of the form $(w \times z) \rightarrow w$, we get that $\sigma \rightarrow \tau_2 = (w \times z) \rightarrow w$, and thus $\sigma = w \times z$ and $w = \tau_2$ for some w and z. From the previous paragraph, $\sigma = u \times v = w \times z$, implying $u = w = \tau_2$ and $v = z = \tau_1$.

Substituting, we get $\emptyset \mid \lambda x.\langle P_2 x, P_1 x\rangle : \rho = \sigma \to \tau = \sigma \to (\tau_1 \times \tau_2) = (u \times v) \to (v \times u)$, therefore any type for Switch must be of the form $(u \times v) \to (v \times u)$. This means that $(u \times v) \to (v \times u)$ is a principal type for Switch, where u and v are type variables.

2. Derive a principal type for $h = \lambda f.\lambda x.\lambda y.\langle fx, fy\rangle$. By rule 3, any type for h must be of the form $\sigma \to (\tau \to (\rho \to \gamma))$ where $\emptyset[f{:}\sigma][x{:}\tau][y{:}\rho] \mid \langle fx, fy\rangle : \gamma$. By rule 2, γ must be of the form $\gamma_1 \times \gamma_2$ where $\emptyset[f{:}\sigma][x{:}\tau][y{:}\rho] \mid fx : \gamma_1$ and $\emptyset[f{:}\sigma][x{:}\tau][y{:}\rho] \mid fy : \gamma_2$. By rule 4, $\sigma = \tau \to \gamma_1 = \rho \to \gamma_2$, implying $\tau = \rho$ and $\gamma_1 = \gamma_2$. This means any type for h must be of the form $(\tau \to \gamma_1) \to (\tau \to (\tau \to (\gamma_1 \times \gamma_1)))$, thus $(a \to b) \to (a \to (a \to (b \times b)))$ is a principal type for h.

3. Derive a principal type for h Switch. Let $\emptyset \mid h\ \text{Switch} : \tau$. By rule 4, $\emptyset \mid h : \sigma \to \tau$ and $\emptyset \mid \text{Switch} : \sigma$ for some σ. Using the principal types for h and Switch, there are types a, b, u and v such that

$$(a \to b) \to (a \to (a \to (b \times b))) = \sigma \to \tau$$
$$(u \times v) \to (v \times u) = \sigma$$

which implies that

$$\sigma = a \to b = (u \times v) \to (v \times u) \quad \text{and} \quad \tau = a \to (a \to (b \times b))$$

implying $a = u \times v$ and $b = v \times u$. Thus, any type τ of h Switch is of the form $(u \times v) \to ((u \times v) \to ((v \times u) \times (v \times u)))$.

4. Derive a principal type for (h Switch) <5, 6>. By rule 4, $\emptyset \mid (h\ \text{Switch})\ \langle 5, 6\rangle : \tau$ implies $\emptyset \mid h\ \text{Switch} : \sigma \to \tau$ and $\emptyset \mid \langle 5, 6\rangle$ has type σ, for some type σ. By rules 2 and 1, the only

type for <5, 6> is INT × INT, thus ø | h Switch : (INT × INT) → τ. But all types for h Switch are of the form (u × v) → ((u × v) → ((v × u) × (v × u))), which implies τ = (INT × INT) → ((INT × INT) × (INT × INT)). Note that since this type is principal and monomorphic, it is the only type for (h Switch) <5, 6>.

5. In order for ((h Switch) <5,6>) <true, false> to have a type τ, rule 4 dictates that ø | (h Switch) <5, 6> : σ→τ and ø | <true, false> : σ. But the only type for (h Switch) <5, 6> is (INT × INT) → ((INT × INT) × (INT × INT)), and the only type for <true, false> is BOOL × BOOL (which is different from INT × INT). Hence, the expression ((h Switch) <5, 6>) <true, false> has no type.

The fact that ((h Switch) <5, 6>) <true, false> does not have a type in the language defined above is a symptom of the language's general inability to properly type functions which require polymorphic arguments. Note that ((h Switch) <5, 6>) <true, false> converts to (i.e., is equal to) < <6, 5>, <false, true> > which does have a type, hence, whether a function is typeable in the parametric system depends on the way it is defined.

The parametric system above can be extended so that ((h Switch) <5, 6>) <true, false> will have a (nontrivial) type. One well-known extension [MS82] adds the form ∀ Tvar . T to the syntax of type expressions, and adds the type inference rule

5-a) A | e : ∀t.τ if t a type variable not appearing in A and A | e : τ

-b) A | e : τ[t←σ] if A | e : ∀t.τ and σ a type expression

(The notation τ[t←σ] denotes the type obtained by substituting free occurrences of the type variable t in τ by the type expression σ.) In this system, the following derivation is possible:

$\emptyset \mid \text{Switch} : \forall a.\forall b.(a \times b) \rightarrow (b \times a)$

$\emptyset[f : \forall a.\forall b.(a \times b) \rightarrow (b \times a)][x : INT \times INT][y : BOOL \times BOOL] \mid fx : INT \times INT$

$\emptyset[f : \forall a.\forall b.(a \times b) \rightarrow (b \times a)][x : INT \times INT][y : BOOL \times BOOL] \mid fy : BOOL \times BOOL$

$\emptyset[f : \forall a.\forall b.(a \times b) \rightarrow (b \times a)][x : INT \times INT][y : BOOL \times BOOL] \mid$

$\langle fx, fy \rangle : (INT \times INT) \times (BOOL \times BOOL)$

$\emptyset \mid h = \lambda f.\lambda x.\lambda y.\langle fx,fy \rangle : (\forall a.\forall b.(a \times b) \rightarrow (b \times a)) \rightarrow$

$((INT \times INT) \rightarrow ((BOOL \times BOOL)$

$\rightarrow ((INT \times INT) \times (BOOL \times BOOL))))$

$\emptyset \mid h\ \text{Switch} : (INT \times INT) \rightarrow ((BOOL \times BOOL) \rightarrow ((INT \times INT) \times (BOOL \times BOOL)))$

$\emptyset \mid (h\ \text{Switch})\ \langle 5, 6 \rangle : (BOOL \times BOOL) \rightarrow ((INT \times INT) \times (BOOL \times BOOL))$

$\emptyset \mid ((h\ \text{Switch})\ \langle 5, 6 \rangle)\ \langle \text{true, false} \rangle : (INT \times INT) \times (BOOL \times BOOL)$

Another extension (due to Coppo) adds the form $T \cap T$ to the type syntax, with the inference rule

5-a) $A \mid e : \sigma \cap \tau$ if $A \mid e : \sigma$ and $A \mid e : \tau$

-b) $A \mid e : \sigma$ and $A \mid e : \tau$ if $A \mid e : \sigma \cap \tau$

Then one can derive

$\emptyset \mid \text{Switch} : ((INT \times INT) \rightarrow (INT \times INT)) \cap ((BOOL \times BOOL) \rightarrow (BOOL \times BOOL))$

$\emptyset[f : ((INT \times INT) \rightarrow (INT \times INT)) \cap ((BOOL \times BOOL) \rightarrow (BOOL \times BOOL))]$

$[x : INT \times INT]\ [y : BOOL \times BOOL] \mid fx : INT \times INT$

$\emptyset[f : ((INT \times INT) \rightarrow (INT \times INT)) \cap ((BOOL \times BOOL) \rightarrow (BOOL \times BOOL))]$

$[x : INT \times INT]\ [y : BOOL \times BOOL] \mid fy : BOOL \times BOOL$

$\emptyset[f : ((INT \times INT) \rightarrow (INT \times INT)) \cap ((BOOL \times BOOL) \rightarrow (BOOL \times BOOL))]$

$[x : INT \times INT]\ [y : BOOL \times BOOL] \mid \langle fx, fy \rangle : (INT \times INT) \times (BOOL \times BOOL)$

$\emptyset \mid h = \lambda f.\lambda x.\lambda y.\langle fx, fy \rangle :$

$(((INT \times INT) \rightarrow (INT \times INT)) \cap ((BOOL \times BOOL) \rightarrow (BOOL \times BOOL)))$

$\rightarrow ((INT \times INT) \rightarrow ((BOOL \times BOOL) \rightarrow$

$((INT \times INT) \times (BOOL \times BOOL))))$

$\emptyset \mid h\ Switch : (INT \times INT) \rightarrow ((BOOL \times BOOL) \rightarrow ((INT \times INT) \times (BOOL \times BOOL)))$

$\emptyset \mid (h\ Switch) \langle 5, 6 \rangle : (BOOL \times BOOL) \rightarrow ((INT \times INT) \times (BOOL \times BOOL))$

$\emptyset \mid ((h\ Switch) \langle 5, 6 \rangle) \langle true, false \rangle : (INT \times INT) \times (BOOL \times BOOL)$

Of the two extensions, Coppo's is known to be the more powerful in that it can assign nontrivial types to more expressions [Lei83]. In the following sections, we adopt Coppo's extension to a simple language of combinators TCL, give the semantics and type inference rules of this language, and show that the type inference rules are valid.

2.2 The Syntax and Semantics of TCL

TCL can be specified with a syntax for both computational expressions and type expressions and a set of typing rules. The computational expressions Exp are those from combinatory logic and are generated by closing the set of primitive constants {S,K} under functional application (denoted by juxtaposition). In the absence of parentheses, application associates to the left. The behavior of the constants is as follows: For all computational expressions X, Y, Z,

$$S\ X\ Y\ Z = XZ(YZ), \qquad K\ X\ Y = X$$

Applying one of these rules to a computational expression to give another computational expression is called performing a (weak) conversion, and applying a rule by substituting an expression in the form of a left-hand-side of one of the equations with the corresponding right-hand-side is called a (weak) reduction. A computational expression on which no reduction may be performed is said to be in (weak) normal form. It is well known that this language matches the expressive power of the lambda calculus--that is, the two languages denote the same class of functions.

The set of type expressions Texp is generated by a countable set of type variables Tvar along with a finite set of type constants Tcnst (possibly empty), and is closed under function type formation ("$\rightarrow$") and conjunction type formation ("$\cap$"). The idea behind a function f having type $A \rightarrow B$ is that whenever x has type A, then f(x) has type B. In the absence of parentheses, "$\rightarrow$" will associate to the right. An expression e having type $A \cap B$ means e has both type A and type B. As will be apparent from the typing rules, a computational expression e may have many types.

DEFINITION 2.2.1 (Syntax of TCL)

Exp	::=	S | K | Exp Exp
Texp	::=	Tvar | Tcnst | Texp $\rightarrow$ Texp | Texp $\cap$ Texp
Tvar	::=	{ an infinite supply of vars, including a,b,c }
Tcnst	::=	{ a finite set of constants, such as INT, BOOL, .. }

We call any type expression not of the form $\alpha \cap \beta$ a *term*. Any type expression τ can be expressed as an intersection of one or more terms $\tau_1 \cap \tau_2 \cap ... \cap \tau_n$, and we say each τ_i is a term of τ. Also, we identify two expressions if they are intersections of identical terms.

Occasionally, we will use the notation $\cap\{\tau \mid P(\tau)\}$, where P is some predicate having finitely many true values, to denote the type expression $\tau_1 \cap ... \cap \tau_n$ where $P(\tau_i)$ is true, e.g., $\cap\{\sigma \mid \sigma$ is a term of τ and $\sigma = \alpha \rightarrow \beta$ for some α and $\beta\}$ denotes the syntactic intersection of all arrow terms of τ. (If the predicate P is always false, then the expression is undefined.)

Given a denotational semantics of the lambda calculus [see Scott76, Stoy77] the semantics of Exp is arrived at by interpreting the constants S and K as the lambda expressions $\lambda x \lambda y \lambda z.xz(yz)$ and $\lambda x \lambda y.x$, respectively. We use the model of [MPS84, S&W77] for types, which assumes that the model D for computational expressions is a partial order under $<$ with least element $\perp$ and satisfies the following properties:

i) $x_0 < x_1 < ...$ where each $x_i \in D \Rightarrow LUB\{x_0, x_1, ...\} \in D$

ii) any subset G of D which has a bound in D has a LUB in D

iii) the set $D^\circ = \{y \in D \mid \forall$ increasing sequences $\langle x_i \rangle_i$ of D,

$$(y < LUB\{x_0, x_1, ...\} \Rightarrow \exists i \text{ such that } y < x_i)\ \},$$

is countable and forms a basis for D, that is, $\forall z \in D\ \exists G \subseteq D^\circ$ such that

$z = LUB(G)$

Note that the set of basis elements D° is a partial order under $<$. Elements in D° are also called *finite elements*. Scott's information systems [Scott82] satisfy the above criteria. We also require that D contain its own continuous function space $D \rightarrow D$ ordered by the usual ordering--$f<g$ iff $f(x)<g(x)$ for all x, and that application be continuous. Let E be the semantic function mapping computational expressions to elements in D (an environment is not necessary for a language without variables). We assume that the following is satisfied for all values X, Y, Z:

$$E[[XY]] = E[[X]]\,E[[Y]], \qquad E[[S]]\,X\,Y\,Z = XZ(YZ), \qquad E[[K]]\,X\,Y = X.$$

To model the types, we use the Hoare powerdomain construction over D (this is equivalent to the constructions in [MPS84, Scott82, Win86, S&W82], but not the one in [MS82]). A type is a non-empty set R of elements of D which is downward closed (i.e., $e \in R$ and $f < e$ (in D) implies $f \in R$) and closed under limits of increasing approximations (i.e., $e_1 < e_2 < ...$ and each $e_i \in R \Rightarrow LUB\{e_i \mid i>0\} \in R$). Such an R is called an *ideal*. Note that D itself is an ideal. The set of all ideals ordered under $\subseteq$ is our domain of types T, and is always a complete lattice with set intersection as the meet and set union as the join operation. Let $\perp_T$ denote the least element of T. Note that T embeds D.

Define $R_1 \rightarrow R_2$, where R_1 and R_2 are ideals, as $\{ f \in (D \rightarrow D) \mid x \in R_1 \Rightarrow f(x) \in R_2 \}$. $R_1 \rightarrow R_2$ is an ideal: $g<f$, $x \in R_1$ and $f \in R_1 \rightarrow R_2 \Rightarrow g(x)<f(x)$ and $f(x) \in R_2$, $\Rightarrow g(x) \in R_2 \Rightarrow g \in R_1 \rightarrow R_2$; also $x \in R_1$, $f_1 < f_2 < ...$all in $R_1 \rightarrow R_2 \Rightarrow f_1(x) < f_2(x) < ...$all in R_2, by continuity of the f_i's, $\Rightarrow LUB\{f_i(x) \mid i>0\} \in R_2$, but $LUB\{f_i(x) \mid i>0\} = LUB\{f_i \mid i>0\}(x)$ by continuity of apply, thus $LUB\{f_i \mid i>0\} \in R_1 \rightarrow R_2$.

The meaning of "has type" is "element of," that is, a computational expression e has type A iff $e \in A$. The universal type is the top of T (i.e., the set D), the inconsistent type (which belongs to the totally undefined function) is at the bottom, and the rest are somewhere in between. A type contained in another type is said to be a *subtype* of the larger set. Obviously, if $A \subseteq B$ then the statement "e has type A" is stronger (logically) than "e has type B," and e has type $A \cap B$ iff it has both type A and type B.

A function is *polymorphic* if it behaves indifferently with respect to type to one or more of its arguments (see Introduction) and hence has many types. A familiar example is the identity function I=SKK (the null operation) which leaves its argument alone. The identity function

has type $t \to t$ for all types t, and in our domain of types T, a valid type for I would therefore be $\cap\{t \to t \mid t \in T\}$. Another example is the constant function K which has type $\cap\{s \to (t \to s) \mid s \in T, t \in T\}$.

Following [Mil78, Cop80, S&W77, MS82, MPS84], all type expressions containing type variables will denote intersections over all substitutions of ideals for the variables. The semantic function M associates with a type expression τ the intersection of all ideals obtained by associating ideals with variables of τ. We define M in terms of an auxiliary function M' which maps a type expression and a type environment (a function from variables to ideals) to an ideal.

Let $Tvar_\perp$ be the flat domain obtained by adding a least element $\perp_{Tvar}$ to the set of type variables, and $Texp_\perp$ be the flat domain obtained by adding $\perp_{Texp}$ to the set of type expressions. As usual in denotational semantics, we use double brackets [[ ]] to enclose syntactic arguments.

DEFINITION 2.2.2. Let

$\sigma, \tau \in Texp$

$x \in Tvar$

$c \in Tcnst$

Dc = an ideal in T associated with the constant c

$\rho \in Tvar_\perp \to T$

$FV(\sigma)$ = set of variables occurring in σ.

Define the semantic function $M : Texp_\perp \to T$ in terms of $M' : Texp_\perp \to (Tvar_\perp \to T) \to T$:

$$M[[\tau]] = \cap\{\, M'[[\tau]]\rho \mid \rho \in Tvar_{\perp}\rightarrow T\},$$

where

$$M'[[x]]\rho = \rho[[x]]$$

$$M'[[c]]\rho = Dc$$

$$M'[[\sigma\cap\tau]]\rho = M'[[\sigma]]\rho \cap M'[[\tau]]\rho$$

$$M'[[\sigma\rightarrow\tau]]\rho = M'[[\sigma]]\rho \rightarrow M'[[\tau]]\rho$$

2.3 The Typing Rules

The interpretation of "e has type τ" as "E[[e]] is an element of M[[τ]]" gives us a good intuitive feel for the properties of types and subtypes, but unfortunately cannot serve as the definition for valid typings e:τ in TCL if we hope to do automatic type checking. To see this, assume we have an algorithm to determine when $E[[e]] \in M[[\sigma]]$ for any expression e and type expression σ. Now if t is a type variable, $M[[t]] = \perp_T$ which is the singleton set containing the totally undefined function Ω. Thus, given any expression e, our algorithm could tell if e is undefined (never halts) for all its arguments. This is a contradiction, since the problem of whether or not e is totally undefined is unsolvable in the lambda calculus and CL [Bar85].

Our rules make use of a binary relation $\leq$ over Texp which we call "weaker." The weaker relation is defined with the relation $\subseteq$ (subtype over the ideals T) in mind. Notice that for ideals A, B, C and D,

i) $A_1\cap\ldots\cap A_n \subseteq A_i$ for $i\leq n$

ii) $A\rightarrow B \subseteq C\rightarrow D$ if $C\subseteq A$ and $B\subseteq D$

iii) $A \subseteq C \cap D$ if $A \subseteq C$ and $A \subseteq D$

iv) $(A \to B) \cap (C \to D) \subseteq (A \cap C) \to (B \cap D)$

are all true. (In fact, the converse of iii is true, and also the converse of ii provided B and D are not the top element in T.) Using these properties, we define a relation $\leq$ which implies subtype in T.

DEFINITION 2.3.1. Let α and β be type expressions where $\alpha = \alpha_1 \cap \ldots \cap \alpha_m$ are intersections of 1 or more terms (non intersections). Then $\alpha \leq \beta$ iff one of the following is true:

i) $\beta = \alpha_i$ for some $i \leq m$ and β is atomic (a variable or constant)

ii) $\beta = \beta_1 \to \beta_2$, $\exists I \subseteq 1..n$ such that $(\forall i \in I \,.\, \alpha_i = \sigma_i \to \tau_i)$ and

$$\beta_1 \leq \cap\{\sigma_i \mid i \in I\} \quad \text{and} \quad \cap\{\tau_i \mid i \in I\} \leq \beta_2$$

iii) $\beta = \beta_1 \cap \ldots \cap \beta_n$ (each β_i a term, $n > 1$) and $\forall i \leq n \,.\, \alpha \leq \beta_i$

It can be shown that $\leq$ is reflexive, transitive and substitution invariant, i.e., $\alpha \leq \beta \Rightarrow P\alpha \leq P\beta$ for substitutions P of type expressions for type variables (see Appendix, page 191). Not surprisingly $\alpha \leq \beta \Rightarrow M[[\alpha]] \subseteq M[[\beta]]$, although the converse is certainly not true. Note that $\alpha \leq \beta \Rightarrow \alpha \cap \gamma \leq \beta$ for any α, β, γ, thus rule ii) in definition 2.3.1 could be rewritten as

ii') $\beta = \beta_1 \to \beta_2$ and $\cap\{\tau \mid \beta_1 \leq \sigma \to \tau,\ \sigma \to \tau \text{ a term of } \alpha\} \leq \beta_2$

This definition of the $\leq$ rules makes an algorithm for deciding the relation obvious. There is an equivalent definition for $\leq$ using the simpler, more numerous rules

i) $z \leq z$ (z atomic)

ii) $\sigma \leq \tau_1 \cap \tau_2$ if $\sigma \leq \tau_1$ and $\sigma \leq \tau_2$

iii) $\sigma_1 \cap \sigma_2 \leq \tau$ if $\sigma_1 \leq \tau$ or $\sigma_2 \leq \tau$

iv) $\sigma_1 \rightarrow \sigma_2 \leq \tau_1 \rightarrow \tau_2$ if $\tau_1 \leq \sigma_1$ and $\sigma_2 \leq \tau_2$

v) $(\sigma_1 \rightarrow \sigma_2) \cap (\tau_1 \rightarrow \tau_2) \leq (\sigma_1 \cap \tau_1) \rightarrow (\sigma_2 \cap \tau_2)$

vi) $\sigma \cap (\tau \cap \rho) \leq (\sigma \cap \tau) \cap \rho$ and $(\sigma \cap \tau) \cap \rho \leq \sigma \cap (\tau \cap \rho)$

vii) $\sigma \cap \tau \leq \tau \cap \sigma$

viii) $\sigma \leq \rho$ if $\sigma \leq \tau$ and $\tau \leq \rho$

Below, a,b and c are type variables, σ and τ are type expressions, e, f and g are computational expressions, P is a substitution of type expressions for type variables, and $P\tau$ denotes the type expression obtained by applying the substitution P to the expression τ.

DEFINITION 2.3.2 (Typing Rules for TCL)

1-a) $S : (a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c))$

-b) $K: a \rightarrow (b \rightarrow a)$

2) $e : P\tau$ if $e : \tau$

3) $fg : \tau$ if $f : \sigma \rightarrow \tau$ and $g : \sigma$

4-a) $e : \sigma \cap \tau$ if $e : \sigma$ and $e : \tau$

-b) $e : \sigma$ and $e : \tau$ if $e : \sigma \cap \tau$

5) $e : \tau$ if $e : \sigma$ and $\sigma \leq \tau$

A *type derivation* consists of applying a finite number of rules 1-5 in order to obtain a typing $e:\tau$. Any derivation for $e:\tau$ can be organized into a tree with root consisting of the statement "$e:\tau$" and the number of a rule which proves the statement, along with as many subtrees, representing subderivations, that the rule requires (0, 1 or 2, depending on the rule used).

Derivation trees will be used as a means of facilitating inductive proofs involving type statements.

As a sidenote, rules 1, 2 and 3 are essentially the rules for the parametric type system, restricted to S-K combinations, thus any derivation of $e:\tau$ using only those rules can be done in the language ML. In fact, one does not increase the set of typeable expressions even if rule 4 is added (provided lambda abstraction is not added). This indicates that considerable typing power lies in the $\leq$ rules.

Below are some examples of type derivations. For each line L, we give the steps and the rules from which the line L immediately follows. The typing in example i can be derived in the parametric system; the typing in examples ii and iii can not.

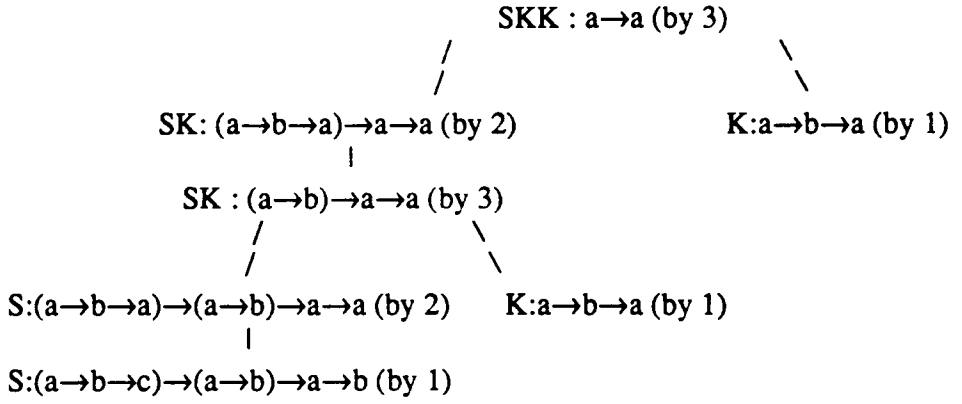
Notation: $[a_1:=\tau_1,...a_n:=\tau_n]$ denotes the substitution function mapping type expressions τ to the expression obtained by simultaneously substituting each occurrence of a_i in τ with the type expression τ_i.

i) Derivation of S K K : $a \rightarrow a$

A.	$S : (a\rightarrow(b\rightarrow c))\rightarrow((a\rightarrow b)\rightarrow(a\rightarrow c))$	{ 1-a }
B.	$S : (a\rightarrow(b\rightarrow a))\rightarrow((a\rightarrow b)\rightarrow(a\rightarrow a))$	{ A,2 using [c:=a] }
C.	$K : a\rightarrow(b\rightarrow a)$	{ 1-b }
D.	$SK: (a\rightarrow b)\rightarrow(a\rightarrow a)$	{ B,C,3 }
E.	$SK : (a\rightarrow(b\rightarrow a))\rightarrow(a\rightarrow a)$	{D,2 using [b:=b→a]
F.	$SKK : a\rightarrow a$	{ E,C,3 }

Notice that SKKX = X for all X, thus SKK is the identity function and will be referred to throughout as "I". Any derivation of a typing for an S-K-I combination will use I:a→a as a lemma (labelled 1-c), but will in fact represent the subderivation above.

The above derivation can be presented as a proof tree:



ii) Derivation of S I I : (x∩(x→y))→y

A.	S : (a→(b→c))→((a→b)→(a→c))	{ 1-a }
B.	S : ((x∩(x→y))→(x→y)) → (((x∩(x→y))→x)→((x∩(x→y))→y))	{ A,2 }
C.	I : a→a	{ 1-c }
D.	I : (x→y)→(x→y)	{ C,2 }
E.	I : (x∩(x→y))→(x→y)	{ D,5 }
F.	SI : ((x∩(x→y))→x) → ((x∩(x→y))→y)	{ B,E,3 }
G.	I : x→x	{ C,2 }
H.	I : (x∩(x→y))→x	{ G,5 }
I.	SII : (x∩(x→y))→y	{ F,H,3 }

iii) Derivation of S I I I : a→a

A.	SII : $(x \cap (x \to y)) \to y$	{ example ii }
B.	SII : $((a \to a) \cap ((a \to a) \to (a \to a))) \to (a \to a)$	{ A,2 }
C.	I : $a \to a$	{ 1-c }
D.	I : $(a \to a) \to (a \to a)$	{ C,2 }
E.	I : $(a \to a) \cap ((a \to a) \to (a \to a))$	{ C,D,4 }
F.	SIII : $a \to a$	{ B,E,3 }

Among the expressions which do not have a type in TCL are (SII)(SII) and the least fixedpoint combinator Y = S ((S(K(SK)S))(K(SII))) ((S(K(SK)S))(K(SII))).

2.4 Semantic Properties of TCL

LEMMA 2.4.1. For type expressions α, β and type environment ρ, $\alpha \leq \beta \Rightarrow M'[[\alpha]]\rho \subseteq M'[[\beta]]\rho$.

proof By induction on $|\alpha| + |\beta|$. The base case is covered when $\beta = t$, an atom. In this case $\alpha = t \cap \alpha'$ and therefore $M'[[\alpha]]\rho = M'[[t]]\rho \cap M'[[\alpha']]\rho \subseteq M'[[t]]\rho$. Now induct. If $\beta = \beta_1 \to \beta_2$, then $\alpha = \alpha_1 \cap ... \cap \alpha_n$ and $\exists I \subseteq 1..n$ such that $\forall i \in I\ \alpha_i = \sigma_i \to \tau_i$ (for some σ_i, τ_i) and $\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \beta_2$. By hypothesis, $M'[[\beta_1]]\rho \subseteq \cap\{M'[[\sigma_i]]\rho \mid i \in I\}$ and $\cap\{M'[[\tau_i]] \mid i \in I\} \subseteq M'[[\beta_2]]\rho$ which implies $\cap\{M'[[\sigma_i]]\rho \mid i \in I\} \to \cap\{M'[[\tau i]]\rho \mid i \in I\} \subseteq M'[[\beta_1]]\rho \to M'[[\beta_2]]\rho$. But $M'[[\alpha]]\rho = \cap\{M'[[\alpha_i]]\rho \mid i \in 1..n\} \subseteq \cap\{M'[[\sigma_i \to \tau_i]]\rho \mid i \in I\}$ $= \cap\{M'[[\sigma_i]]\rho \to M'[[\tau_i]]\rho \mid i \in I\} \subseteq \cap\{M'[[\sigma_i]]\rho \mid i \in I\} \to \cap\{M'[[\tau_i]]\rho \mid i \in I\} \subseteq M'[[\beta]]\rho$. If $\beta = \beta_1 \cap \beta_2$ (β_1 and β_2 may be intersections) then $\alpha \leq \beta_1$ and $\alpha \leq \beta_2 \Rightarrow$ (by hypothesis) $M'[[\alpha]]\rho \subseteq M'[[\beta_1]]\rho \cap M'[[\beta_2]]\rho = M'[[\beta_1 \cap \beta_2]]\rho$. ◊

COROLLARY 2.4.1. $\alpha \leq \beta \Rightarrow M[[\alpha]] \subseteq M[[\beta]]$.

proof Obvious from the definition of M and Lemma 2.4.1. ◊

LEMMA 2.4.2. $M'[[\ \alpha[x\leftarrow\beta]\ ]]\ \rho = M'[[\alpha]](\rho[M'[[\beta]]\rho\backslash x])$ for all ρ.

proof By induction on the size of α. $\Diamond$

THEOREM 2.4.1. (Semantic Soundness) $e : \tau \Rightarrow E[[e]] \in M[[\tau]]$.

proof Let $e:\tau$. Induct on the size of the derivation tree. The base case is when rule 1 is used and e is S or K. If e=S, then we must show that for all ideals A, B and C, $E[[S]] \in (A\rightarrow(B\rightarrow C))\rightarrow((A\rightarrow B)\rightarrow(A\rightarrow C))$. Following [Scott76], let $X \in A\rightarrow(B\rightarrow C)$. Show $E[[S]]X \in (A\rightarrow B)\rightarrow(A\rightarrow C)$. To this end, let $Y \in A\rightarrow B$ and show $E[[S]]XY \in A\rightarrow C$. Again, let $Z \in A$ and show $E[[S]]XYZ \in C$. $E[[S]]XYZ = XZ(YZ)$ by our assumptions on E, and $XZ(YZ) \in C$ by assumptions on X, Y and Z, and by the definition of $\rightarrow$. This proves the base case for e=S. The case for e=K is analogous. For the inductive part, take cases of the rule used at the root of the derivation. For rule 2, $e:\tau$, $\tau=P\sigma$ for some substitution $P=[x_1:=\tau_1,...,x_k:=\tau_k]$, and let $\rho\in Tvar_\perp\rightarrow T$. By hypothesis, $e: \sigma \Rightarrow E[[\sigma]] \in M'[[\sigma]]\rho'$ for all $\rho' \Rightarrow E[[\sigma]] \in M'[[\sigma]](\rho[M'[[\tau_1]]\rho\backslash x_1,...,M'[[\tau_k]]\rho\backslash x_k]) = M'[[P\sigma]]\rho$ by lemma 2.4.2. For rule 3, $e=fg:\tau$, $f:\sigma\rightarrow\tau$ and $g:\sigma \Rightarrow$ (by hypoth.) $E[[f]] \in M[[\sigma\rightarrow\tau]]$ and $E[[g]] \in M[[\sigma]]$ $\Rightarrow$ (rewriting) $E[[f]] \in \cap\{M'[[\sigma]]\rho\rightarrow M'[[\tau]]\rho \mid \text{any } \rho\}$ and $E[[g]] \in \cap\{M'[[\sigma]]\rho \mid \text{any } \rho\}$ $\Rightarrow$ (def. of $\rightarrow$) $E[[f]]\ E[[g]] \in \cap\{M'[[\tau]]\rho \mid \text{any } \rho\} \Rightarrow$ (rewriting) $E[[fg]] \in M[[\tau]]$. If rule 4 is used, then either $e:\tau=\sigma\cap\gamma$ is the root and $e:\sigma$ and $e:\gamma$ are the subtrees of the derivation, or $e:\tau$ is the root and $e:\tau\cap\sigma$ is the subtree. In both cases, the result follows immediately from the induction hypothesis. If rule 5 is used, then $e: \sigma$ and $\sigma\leq\tau$ are subtrees $\Rightarrow$ (hypoth.) $E[[e]] \in M[[\sigma]] \Rightarrow$ (by corollary 2.4.1) $E[[e]] \in M[[\tau]]$. $\Diamond$

3

Undecidability of TCL

In this chapter, we show that the typeable expressions in TCL are those whose counterparts in the lambda calculus are strongly normalizable, i.e., they are expressions e such that all reduction sequences from e terminate. It is well known that the set of strongly normalizable lambda expressions is undecidable, implying the undecidability of the set of typeable expressions in TCL.

We proceed to define a language Tλ whose set of typeable expressions contains the typeable expressions of TCL. It is then shown that the typeable expressions e in Tλ are all strongly normalizable (written SN(e)), implying the strong normalizability of typeable expressions in TCL. Using a well-known fact it is shown that SN(e) implies e typeable in a subsystem of Tλ equivalent to Coppo's system Cp. Then, we show that typeability of an S-K combination e in Cp implies its typeability in TCL. Letting unfac(e) represent the lambda expression corresponding to the SK combination e, we will have shown

e TCL-typeable ⇒ unfac(e) Tλ-typeable ⇒ SN(unfac(e)) ⇒ unfac(e) Cp-typeable ⇒ e TCL-typeable.

Finally, we show that the set of strongly normalizable unfactored SK combinations is undecidable. We do this by showing that the standard SKI factoring (fac) preserves strong normalizability, i.e., SN(e) $\Rightarrow$ SN(unfac(fac(e))), and since unfac(fac(e)) reduces to e, we have SN(e) $\Leftrightarrow$ SN(unfac(fac(e)). That is, a decision procedure for strongly normalizable unfactored S-K combinations would imply one for strong normalizability in general, and hence cannot exist. Hence, TCL-typeability is undecidable as well.

The problem of deciding if e : τ in TCL is also undecidable, since otherwise one could determine typeability of e by determining if, for example, KKe has type a$\rightarrow$b$\rightarrow$a.

The first section introduces the language Tλ which is an obvious adaptation of TCL to the lambda calculus. It is readily seen that an expression e having type τ in TCL has, when unfactored, type τ in Tλ. The notion of *reduced type expression* is introduced to facilitate subsequent proofs. (The term is taken from [Lei83].)

3.1 The Language Tλ

Here, the computational expressions are generated by closing a set of variables Var under lambda abstraction and application. The type expressions are the same as in TCL.

DEFINITION 3.1.1 (Syntax for Tλ)

Exp	::=	Var | λ Var . Exp | Exp Exp
Var	::=	{ an infinite supply of variables, including x, y and z }

Texp ::= Tvar | Tcnst | Texp → Texp | Texp ∩ Texp

Tvar ::= { an infinite supply of type variables }

Tcnst ::= { a finite set of type constants }

As usual, the intended meaning of an expression of the form λx.e is the function which when applied to some value denoted by f gives the value denoted by e[x←f] (e with all free occurrences of x replaced by f). Replacing an occurrence of (λx.e)f with e[x←f] in a lambda expression M is called *performing a (β-)reduction* (or contraction) on M, and (λx.e)f is called the *redex* of the reduction. An expression M on which no reduction is possible is said to be in *normal form.* All expressions do not have a normal form, but the normal form is unique (up to renaming of λ-bound variables) for an expression having one (by the Church-Rosser Theorem). An expression M having a normal form may also have an infinite reduction sequence, e.g., (λxλy.x) z ((λw.ww)(λw.ww)) has a normal form z, yet one can apply successive reductions to (λw.ww)(λw.ww) without reaching a normal form. Expressions which always normalize, regardless of which redexes are contracted, are called *strongly normalizable.*

The semantics of the type expressions is the same as for TCL. However, since there are variables in the language, the typing rules must be given with respect to a set of assumptions on the types of the free variables in an expression. We call such a set a type assignment, and is actually a finite function from Var to Texp. If G is a type assignment, then G[x:τ] is G updated to map x to τ. A typing for e is of the form G | e : τ and means that e has type τ under the type assumptions x:ρ in G for the free variables x in e. If some free variable x in e is not mapped by G, then G | e : τ is not a valid typing. We let Ø denote the empty type assignment. Below, G is a type assignment, e, f and g are expressions, σ and τ are type expressions, and x is a variable.

DEFINITION 3.1.2. (Typing Rules for $T\lambda$)

1) $G[x:\tau] \mid x : \tau$

2) $G \mid \lambda x.e : \sigma \to \tau$ if $G[x:\sigma] \mid e:\tau$

3) $G \mid fg : \tau$ if $G \mid f : \sigma \to \tau$ and $G \mid g : \sigma$

4-a) $G \mid e : \sigma \cap \tau$ if $G \mid e : \sigma$ and $G \mid e : \tau$

-b) $G \mid e : \sigma$ and $G \mid e : \tau$ if $G \mid e : \sigma \cap \tau$

5) $G \mid e : \tau$ if $G \mid e : \sigma$ and $\sigma \leq \tau$

Notice that no assumptions are needed for typings of closed lambda expressions (that is, expressions without free variables). For example, a derivation of $\varnothing \mid (\lambda x.xx) : (a \cap (a \to b)) \to b$ is

A.	$\{x:a \cap (a \to b)\} \mid x : a \cap (a \to b)$	{ 1 }
B.	$\{x:a \cap (a \to b)\} \mid x : a$	{ 4-b,A }
C.	$\{x:a \cap (a \to b)\} \mid x : a \to b$	{ 4-b,A }
D.	$\{x:a \cap (a \to b)\} \mid xx : b$	{ 3,B,C }
E.	$\varnothing \mid \lambda x.xx : (a \cap (a \to b)) \to b$	{ 2,D }

As we will see, any expression typeable in $T\lambda$ is typeable using a subset of types known as *reduced* types.

DEFINITION 3.1.3. A type τ is *reduced* if there are no intersections immediately to the right of any $\to$ occurring in τ.

We can always reduce a type by distributing the $\rightarrow$ over the intersection. Let d be the distribute function defined for type expressions τ:

$d(\tau) =$ if $\tau = \tau 1 \rightarrow \tau 2$, then $\bigcap_{i \leq n} (d(\tau 1) \rightarrow \beta i)$ where $d(\tau 2) = \beta 1 \cap \ldots \cap \beta n$ and each βi a non-$\cap$

if $\tau = \tau 1 \cap \tau 2$ then $d(\tau 1) \cap d(\tau 2)$

if τ in Tvar or Tcnst then τ

We can extend the function d in a natural way to map type assignments to reduced type assignments: $d(\varnothing) = \varnothing$, and $d(G[x{:}\tau]) = d(G)[x{:}d(\tau)]$.

LEMMA 3.1.1 In Tλ, G| e:τ $\Rightarrow$ there is a derivation involving only reduced types for d(G)|e:d(τ).

proof Induction on the derivation of G|e:τ. The base case is when rule 1 is used, and the result follows from the definition of d. For the inductive step, take cases on the rule applied at the root of the derivation. Rules 2, 3 and 4 follow immediately from the induction hypothesis. Rule 5 follows from the hypothesis and by another induction on the derivation of $\sigma \leq \tau$. $\Diamond$

Lemma 3.1.1 allows us to restrict the type expressions to reduced types.

If we restrict Tλ to typing rules 1 through 4, then we have the ordinary conjunctional system of Coppo, which is known to type the strongly normalizable lambda expressions [Cop79, Lei83].

We will call Coppo's system Cp, as defined below.

DEFINITION 3.1.4. (Typing Rules for Cp)

1) $G[x:\tau] \mid x : \tau$

2) $G \mid \lambda x.e : \sigma \to \tau$ if $G[x:\sigma] \mid e:\tau$

3) $G \mid fg : \tau$ if $G \mid f : \sigma \to \tau$ and $G \mid g : \sigma$

4-a) $G \mid e : \sigma \cap \tau$ if $G \mid e : \sigma$ and $G \mid e : \tau$

-b) $G \mid e : \sigma$ and $G \mid e : \tau$ if $G \mid e : \sigma \cap \tau$

Certainly e typeable in Cp implies e typeable in Tλ. We will see that the reverse is true as well. It is not true, however, that $G|e:\tau$ in Tλ implies τ is a type for e in Cp, e.g., $\lambda x.x : (a \to a) \to (a \cap b) \to a$ in Tλ but cannot be derived without using rule 5.

The following lemma is no surprise, but is necessary later on.

LEMMA 3.1.2. $G|e : \tau$ in Cp $\Rightarrow$ there is a derivation of $d(G)|e:d(\tau)$ using only reduced types.

proof Essentially the same as for lemma 3.1.1. $\Diamond$

3.2 Tλ-Typeability = SN

In this section, we show that every expression in Tλ is strongly normalizable and every strongly normalizable expression is typeable in Tλ. To help the proof along, we introduce a computational constant Δ into Exp with the type rule $G|\Delta:\tau$ for every τ and G. As far as reductions are concerned, Δ behaves as a free variable which may not be bound by a λ. Call

this new language $T\lambda\Delta$. Using a technique of [Barendregt85, p. 569], we show that every typeable expression in $T\lambda\Delta$ is strongly normalizable, implying the result for $T\lambda$.

For closed lambda expressions e, we write $e:\tau$ as a shorthand for $\varnothing|e:\tau$. Let SN(e) be true exactly when the lambda expression e is strongly normalizable, and define NS(e) = not SN(e).

DEFINITION 3.2.1. For all $t \in Tvar \cup Tcnst$ and reduced types σ and τ,

$Z(t) = \{ e \mid e$ a closed lambda expression, SN(e) and $e:t \}$

$Z(\sigma\to\tau) = \{ e \mid e$ a closed lambda expression, $e:\sigma\to\tau$ and $\forall f\in Z(\sigma).\ ef\in Z(\tau) \}$

$Z(\sigma\cap\tau) = Z(\sigma) \cap Z(\tau)$

Note that since Δ is closed, each $Z(\tau)$ is non-empty.

DEFINITION 3.2.2. Let S be a substitution of closed lambda expressions for variables and G a type assignment. Then $H(S,G) = (\ dom(S)=dom(G)$ and $S(x)\in Z(G(x))\ )$.

DEFINITION 3.2.3. For reduced types σ,

$Z^*(\sigma) = \{ e \mid e$ any lambda expression such that

$\forall$ type assignments G,

$\forall$ substitutions S of closed lambda expressions for variables,

$G|e:\sigma$ and $H(S,G) \Rightarrow S(e)\in Z(\sigma)$ $\}$.

LEMMA 3.2.1. $e\in Z(\tau) \Rightarrow SN(e)$.

proof Induct on the size of τ. Base case is immediate from definition 3.2.1. For $e \in Z(\sigma \cap \rho)$, apply the hypothesis to $Z(\sigma)$. $e \in Z(\sigma \to \rho) \Rightarrow \forall f \in Z(\sigma).ef \in Z(\rho) \Rightarrow$ (since $Z(\sigma) \neq \emptyset$) $\exists f.ef \in Z(\rho) \Rightarrow$ (by hypothesis) $\exists f.SN(ef) \Rightarrow SN(e)$. $\Diamond$

LEMMA 3.2.2. Let e be closed, $e:\sigma$, $\sigma = \sigma 1 \to ... \to \sigma n \to t$, where t is a Tvar or Tcnst. Then

$$e \in Z(\sigma) \Leftrightarrow e:\sigma \text{ and } \forall f1 \in Z(\sigma 1)...\forall fn \in Z(\sigma n).SN(e(f1)(f2)...(fn)).$$

proof ($\Rightarrow$) by definition of $Z(\sigma)$. ($\Leftarrow$) Induct on n. For n=0, we have $\sigma = t$ and the result follows from definition 3.2.1. Assume true for n=k-1, and assume n=k>0. $\forall f1 \in Z(\sigma 1)\forall ...\forall fk \in Z(\sigma k)$. $SN(e(f1)(f2)...(fk))$ and $e:\sigma 1 \to ... \to \sigma k \to t$, $\Rightarrow$ (by def. of $Z(\sigma 1)$) $\forall f1 \in Z(\sigma 1)$ $(\forall f2 \in Z(\sigma 2)\forall$ ... $\forall fk \in Z(\sigma k)$. $SN((e(f1))(f2)...(fk))$ and $e(f1):\sigma 2 \to ... \to \sigma k \to t)$, $\Rightarrow$ (by hypothesis) $\forall f1 \in Z(\sigma 1)$. $e(f1) \in Z(\sigma 2 \to ... \to \sigma k \to t)$, thus $e \in Z(\sigma 1 \to ... \to \sigma k \to t)$. $\Diamond$

COROLLARY 3.2.1. If $e[x \leftarrow f] \in Z(\sigma)$ and $f \in Z(\rho)$ for some ρ, then $(\lambda x.e)f \in Z(\sigma)$.

proof Let $f \in Z(\rho)$ and $e[x \leftarrow f]$ be in $Z(\sigma)$ where $\sigma = \sigma 1 \to ... \to \sigma n \to t$. By lemma 3.2.2, $\forall f1 \in Z(\sigma 1)$ ... $\forall fn \in Z(\sigma n).SN(e[x \leftarrow f](f1)...(fn))$. It must be that $SN((\lambda x.e)f(f1)...(fn))$, else $NS((\lambda x.e)f(f1)...(fn)) \Rightarrow$ contracting $(\lambda x.e)f$ destroys the existence of an infinite reduction, which is impossible since $(\lambda x.e)f$ is the leftmost redex and f is strongly normalizable [Barendregt85]. Thus, $SN((\lambda x.e)f(f1)...(fn)) \Rightarrow$ (by lemma 3.2.2) $(\lambda x.e)f \in Z(\sigma)$. $\Diamond$

LEMMA 3.2.3. $\sigma \leq \tau \Rightarrow Z(\sigma) \subseteq Z(\tau)$.

proof Induction on the derivation of $\sigma \leq \tau$. $\Diamond$

LEMMA 3.2.4. $Gle:\sigma$ and σ reduced $\Rightarrow e \in Z^*(\sigma)$.

proof Induct on the derivation of $G|e:\sigma$. For e=x, a variable, let $G(x)=\sigma$ and let S be a substitution such that H(S,G) is true. Then $S(x)\in Z(\sigma)$ by def. of H, and therefore $x\in Z^*(\sigma)$. If rule 2 is used at the root of the derivation, then $e=\lambda x.f$, $s=\alpha\rightarrow\beta$, and $G[x:\alpha]|f:\beta$. By the induction hypothesis, $f\in Z^*(\beta)$. Let H(S,G) be true. Then by def. of $Z^*(\beta)$, $\forall g\in Z(\alpha).S(f)[x\leftarrow g]\in Z(\beta) \Rightarrow$ (by corollary 3.2.1) $(\lambda x.S(f))g\in Z(\beta)$ for all $g\in Z(\alpha)$ $\Rightarrow S(\lambda x.f)g\in Z(\beta)$ for all $g\in Z(\alpha) \Rightarrow S(\lambda x.f)\in Z(\alpha\rightarrow\beta) \Rightarrow (\lambda x.f)\in Z^*(\alpha\rightarrow\beta)$. If rule 3 is used, then e=fg, $G|f:\rho\rightarrow\sigma$ and $G|g:\rho$, and by the induction hypothesis, $f\in Z^*(\rho\rightarrow\sigma)$ and $g\in Z^*(\rho)$. Therefore, for any S and G', $H(S,G') \Rightarrow S(f)\in Z(\rho\rightarrow\sigma)$ and $S(g)\in Z(\rho) \Rightarrow$ $S(f)S(g)\in Z(\sigma)$, but $S(f)S(g) = S(fg)$, $\Rightarrow fg\in Z^*(\sigma)$. If either of rules 4-a or -b is used, the result follows directly, since $Z^*(\alpha\cap\beta) = Z^*(\alpha) \cap Z^*(\beta)$. If rule 5 is used, the result follows from lemma 3.2.3. $\Diamond$

It follows from lemma 3.2.4 that any closed expression e in Tλ having type τ is contained in $Z(\tau)$ and hence is strongly normalizable. If $x_1, x_2, ...,x_n$ are free variables of some expression e in Tλ, then e typeable and e strongly normalizable $\Leftrightarrow \lambda x_1\lambda x_2...\lambda x_n.e$ typeable and strongly normalizable. Therefore, every expression typeable in Tλ is strongly normalizable.

THEOREM 3.2.1. $G|e:\tau$ in T$\lambda \Rightarrow$ SN(e).

We remark that it can also be shown that every strongly normalizable lambda expression has a type in Tλ. The easiest way is to use the fact that the typeable expressions in Coppo's conjunctive type system Cp are the strongly normalizable ones [Cop80]:

THEOREM 3.2.2 (Coppo) e typeable in Cp $\Leftrightarrow$ SN(e).

With theorem 3.2.1 we have that the typeable expressions in Tλ are exactly the strongly normalizable ones.

THEOREM 3.2.3. e typeable in Tλ $\Leftrightarrow$ SN(e)

3.3 TCL-Typeable = SN

In this section, we show that e is typeable in TCL iff e is strongly normalizable when interpreted as a lambda expression.

An SKI combination is "interpreted" as a lambda expression by translating occurrences of S to λxλyλz.xz(yz), K to λxλy.x, and I to λx.x.

DEFINITION 3.3.1. Let e be an SKI combination. Then

unfac(e) =

if e=x, a variable, then x

if e=λx.f then λx.unfac(f)

if e=S then λxλyλz.xz(yz)

if e=K then λxλy.x

if e=I then λx.x

if e=fg then unfac(f) unfac(g)

LEMMA 3.3.1. Let e be in TCL, α a reduced type. Then ∅⊢unfac(e):α in Cp $\Rightarrow$ e:α in TCL.

proof Since the typing rules in TCL contain the rules in Cp not dealing with lambda abstraction, we need only show that any non-intersection, reduced type derivable for an unfactored S or K combinator in Cp is derivable for S or K, resp., in TCL. // Comment:

Recall the notation $\cap\{\beta_i \mid i\in I\}$ is used to denote a *syntactic* intersection of one or more non-$\cap$ terms β_i for finite index set I. // First, consider unfac(S) = $\lambda x\lambda y\lambda z.xz(yz)$. To derive a type for unfac(S) in Cp, assumptions on the types of x, y and z must be used to first derive a type for xz(yz). Let G be those assumptions (i.e., a type assignment) and let G|xz(yz):τ. Let G(z) = $\cap\{\gamma_k \mid k\in K\}$ be an intersection of non-intersection types γk over a finite index set K. We can express all the types of (yz) used in the inference in the same way: $\cap\{\rho_h \mid h\in H\}$. From this, we conclude that the intersection of types of y (i.e., G(y)) must at least contain $\cap\{(\cap\{\gamma_k \mid k\in K_h\})\rightarrow\rho h \mid h\in H$ and each $K_h\subseteq K\}$, thus G(y) = $(\cap\{(\cap\{\gamma_k \mid k\in K_h\})\rightarrow\rho_h \mid h\in H$ and $K_h\subseteq K\}) \cap \xi$ for some type ξ and each $K_h\subseteq K$. Now the type inferred for (xz)(yz) cannot be an intersection (since we are assuming reduced types), therefore the type for (xz) cannot be an intersection, hence the type used for x in inferring the type for (xz) cannot be an intersection. The type of (xz) must be of the form $(\cap\{\rho_h \mid h\in H'\})\rightarrow\tau$ for some $H'\subseteq H$. Therefore, the type of x used in the derivation must be of the form $(\cap\{\gamma_k \mid k\in K'\})\rightarrow((\cap\{\rho_h \mid h\in H'\})\rightarrow\tau)$ where $H'\subseteq H$, $K'\subseteq K$, and we conclude that G(x) = $(\cap\{\gamma_k \mid k\in K'\})\rightarrow((\cap\{\rho_h \mid h\in H'\})\rightarrow\tau) \cap \eta$ for some type η. Finally, using typing rule 2, we conclude that the general reduced type for $\lambda x\lambda y\lambda z.xz(yz)$ in Cp is of the form

$$((\cap\{\gamma_k \mid k\in K'\})\rightarrow((\cap\{\rho_h \mid h\in H'\})\rightarrow\tau) \cap \eta) \rightarrow$$
$$((\cap\{(\cap\{\gamma_k \mid k\in K_h\})\rightarrow\rho_h \mid h\in H \text{ and } K_h\subseteq K\}) \cap \xi) \rightarrow ((\cap\{\gamma_k \mid k\in K\})\rightarrow\tau))$$

where K, H, K' are finite index sets such that $K'\subseteq K$ and $H'\subseteq H$, and where τ, γ_i, ρ_i, η and ξ are non-intersection types. We must now show that the general form above is derivable from the axiomatic type of S using the rules in TCL. This will be shown if we can find type expressions a, b and c such that

$(a\rightarrow(b\rightarrow c))\rightarrow((a\rightarrow b)\rightarrow(a\rightarrow c)) \leq$

$$((\cap\{\gamma_k \mid k\in K'\})\rightarrow((\cap\{\rho_h \mid h\in H'\})\rightarrow\tau)\cap\eta) \rightarrow$$
$$((\cap\{ (\cap\{\gamma_k \mid k\in K_h\})\rightarrow\rho_h \mid h\in H \text{ and } K_h\subseteq K\})\cap\xi) \rightarrow ((\cap\{\gamma_k \mid k\in K\})\rightarrow\tau))$$

Using "$\sigma_1\rightarrow\sigma_2 \leq \tau_1\rightarrow\tau_2$ if $\tau_1\leq\sigma_1$ and $\sigma_2\leq\tau_2$", this will be satisfied if

$$\cap\{\gamma_k \mid k\in K\} \leq a,\ a\leq\cap\{\gamma_k \mid k\in K'\},\ b\leq\cap\{\rho_h \mid h\in H'\},\ c = t,$$
$$\text{and } \cap\{ (\cap\{\gamma_k \mid k\in K_h\})\rightarrow\rho_h \mid h\in H \text{ and } K_h\subseteq K\} \leq a\rightarrow b.$$

These inequalities are satisfied by the substitution [a:=($\cap\{\gamma_k \mid k\in K\}$), b:=($\cap\{\rho_h \mid h\in H\}$), c:=$\tau$], hence the general reduced type for S (in Cp) is derivable in TCL. A similar argument shows the general reduced type for K is derivable in TCL. $\Diamond$

Certainly, e typeable in TCL $\Rightarrow$ unfac(e) typeable in Tλ (and in Cp); thus

THEOREM 3.3.1.

e typeable in TCL $\Leftrightarrow$ unfac(e) typeable in Tλ $\Leftrightarrow$ unfac(e) typeable in Cp $\Leftrightarrow$ SN(unfac(e)).

3.4 SKI Factorization Preserves SN

Here we show that, given the standard mappings between the lambda calculus and combinatory logic, the property of strong normalizability is preserved when a lambda term is mapped to its S-K-I factors in combinatory logic and the mapped back to a lambda calculus

term. This will show the undecidability of the set of strongly-normalizable unfactored S-K combinations.

Let λ be the set of lambda expressions and CL be the set of combinations of S, K and I. We give the standard mapping "fac" from λ to CL [C&F58]. As defined, fac is actually a mapping from the set of combinations of S, K, I and variables, closed under lambda abstraction (call this set λCL), to CL with free variables added. However, closed lambda expressions always map to expressions in CL.

DEFINITION 3.4.1. Let $e \in \lambda CL$. Then fac(e) is the S-K-I combination defined as follows:

fac(e) =

if e is a variable, S, K or I, then e

if e=fg then fac(f) fac(g)

if $e=\lambda x.x$ then I

if $e=\lambda x.g$ where g is S, K, I or a variable $y \neq x$ then Kg

if $e=\lambda x.fg$ then S fac($\lambda x.f$) fac($\lambda x.g$)

if $e=\lambda x \lambda y.g$ then fac($\lambda x.$fac($\lambda y.g$))

It is easy to see that fac is well defined and computable--the algorithm F based on the definition of fac always terminates since the number of lambdas in e is never less than that of any argument of a recursive call in F(e), and they are equal only when $e=\lambda x.fg$, a case which cannot repeat indefinitely.

(We are calling two lambda expressions "equal" if they are the same up to renaming of bound variables.) It is not hard to show that for any lambda expression e, unfac(fac(e)) reduces to e. We will show that unfac ∘ fac preserves strong normalizability.

Recall that for $e \in \lambda$, NS(e) means e is not strongly normalizable, i.e., there is an infinite reduction from e. Define a context C[] to mean a lambda expression with a single "hole" where a possible lambda expression can be placed. For example, C[] = $\lambda x.((\lambda y.x[])\ (xx))$ is a context. Write C[g], where g is a lambda expression, to mean C with g filled in for the hole, and $C_1[C_2[]]$ to mean the new context obtained by filling in $C_2[]$ for the hole in C_1.

Our strategy is to show that NS(unfac(fac(e))) $\Rightarrow$ NS(e). The argument makes use of the Conservation Theorem:

CONSERVATION THEOREM. Let e be a lambda expression containing the variable x, let f be any lambda expression, and let C[] be any context. Then

$$NS(\ C[(\lambda x.e)f]\ ) \Rightarrow NS(\ C[e[x \leftarrow f]]\ ).$$

proof [see Barendregt, 85] ◊

We also need a similar result for the case when e may not contain the variable x.

LEMMA 3.4.1. Let C[] be a context, e a lambda expression, and x and y variables. Then

$$NS(\ C[(\lambda x.e)y]\ ) \Rightarrow NS(\ C[e[x \leftarrow y]]\ ).$$

proof This follows directly from a theorem in [B&K82], but has also been proved independently by the author. ◊

LEMMA 3.4.2. Let e∈λCL and C[] be any context. Then

$$NS(C[unfac(fac(\lambda x.e))]) \Rightarrow NS(C[\lambda x.unfac(e)]).$$

proof Let F compute fac and U compute unfac by applying the definitions recursively. Induct on the number of calls to F in computing F(λx.e). The base case has 2 possibilities: 1) e=x, then NS(C[U(F(λx.x))]) ⇒ NS(C[λx.x]) ⇒ NS(C[λx.U(x)]). 2) e=S, K, I or a variable y≠x, then NS(C[U(F(λx.e))]) ⇒ NS(C[U(Ke)]) ⇒ NS(C[(λyλx.y)U(e)]) ⇒ (by Conservation Theorem) NS(C[λx.U(e)]). For the inductive part, assume there is more than 1 call to F. There are 2 more cases: 3) e=fg, then NS(C[U(F(λx.fg))]) ⇒ NS(C[U(S F(λx.f) F(λx.g))]) ⇒ NS(C[(λzλyλx.zx(yx)) U(F(λx.f)) U(F(λx.g))]) ⇒ (by Conservation Theorem) NS(C[λx.U(F(λx.f)x(U(F(λx.g))x)]) ⇒ (by hypothesis) NS(C[λx.(λx.U(f))x((λx.U(g))x)]) ⇒ (by lemma 3.4.1) NS(C[λx.U(f)U(g)]) ⇒ NS(C[λx.U(fg)]). 4) e=λy.f, then NS(C[U(F(λxλy.f))]) ⇒ NS(C[U(F(λx.F(λy.f)))]) ⇒ (by induction hypothesis) NS(C[λx.U(F(λy.f))]) ⇒ (by induction hypothesis) NS(C[λx.λy.U(f)]) ⇒ NS(C[λx.U(λy.f)]). ◊

Using lemma 3.4.2, it can be shown by induction on the number of top-level applications that for any context C[], NS(C[U(F(fg))]) ⇒ NS(C[fg]). Also, since unfac(fac(e)) reduces to e, we have that NS(C[e]) ⇒ NS(C[unfac(fac(e))]) for all e, which yields:

THEOREM 3.4.1. For any closed lambda expression e, SN(e) ⇔ SN(unfac(fac(e))).

3.5 TCL-Typeability Is Undecidable

It is well known that the strongly normalizable expressions (and the closed strongly normalizable expressions) is an undecidable set. Since each strongly normalizable expression has a computable strongly normalizable factorization (by theorem 3.4.1), the set of typeable expressions in TCL is undecidable.

Another approach to showing the undecidability of the typeable expressions in TCL without showing strong normalization involves the λ-I calculus. The λ-I calculus is the same as the λ calculus except that lambda abstraction of a variable x is only allowed over expressions e containing x--e.g., λxλy.x would not be allowed. Call Tλ restricted to the λ-I calculus Tλ-I. One can show that typeability in Tλ-I (but not in Tλ!) is preserved under *conversion* (e converts to f iff e reduces to f or f reduces to e). Then one extends the following theorem, due to Scott, to the λ-I calculus:

THEOREM 3.5.1. (Scott). Any proper subset of the λ calculus which is closed under conversion is undecidable.

proof [see Barendregt85]. ◊

to obtain

THEOREM 3.5.2. Any proper subset of the λ-I calculus which is closed under conversion is undecidable.

proof By modification of the proof of theorem 3.5 in [Barendregt85]. ◊

This shows the undecidability of the typeable expressions in Tλ. One can then show that in Tλ, G$\vdash$e:$\tau \Leftrightarrow$ G$\vdash$ unfac(fac(e)) : τ and that for S-K combinations e, $\emptyset\vdash$ unfac(e):τ in Tλ (τ reduced) $\Rightarrow$ e:τ in TCL. This forces the undecidability of the typeable expressions in TCL.

Note that the criterion that typeability be preserved under conversion is a desirable one: it means that functions which compute the same values (via reduction) have the same types, a property not present in ML, for example. Theorem 3.5.2 implies that this criterion imposes undecidability on type checking.

4

TCL with Explicit Types

Faced with the undecidability of type checking in a proposed type system (such as TCL), a language designer may impose restrictions on the system, that is, on the rules or syntax, in order to make things decidable. For example, it has been noted [Lei83] that restricting the use of the type-intersection operator so that it may only appear to the left of n arrows ($\rightarrow$) where $n \leq 2$ makes the conjunctive system decidable. Aside from the inevitable loss of typing power, such restrictions often create seemingly arbitrary cutoffs in the set of typeable expressions, making it difficult for the programmer to know what constitutes a well-typed program. This problem is even worse in languages whose well-typed expressions are defined by the type checking algorithm.

Another philosophy is to define a type as we have defined a type derivation. That is, the type checker receives "e:G", where G is a *type derivation tree* for some type (in the old sense) τ, and checks whether or not G is a valid derivation for $e:\tau$. Thus a programmer must give a

"proof" to the type checker that his program e has type τ. In principle there is nothing wrong with this, but in practice there are 2 drawbacks. First, a complete derivation can add substantially to the length of a program, even for short e and τ. Secondly, no type inference is done by the type checker. This means that type derivations must be explicitly given for every subexpression of e, a requirement that most programmers would certainly find objectionable. These impracticalities make this approach undesirable to implement.

Our approach incorporates the notion of *explicit typings* (i.e., user-supplied type information) into the type syntax and typing rules of TCL. Here, the programmer specifies explicit type information at key places in the program, and this information is used by the type checker to check type claims. Our language, which we call XTCL, has the typing power of TCL and allows for limited type inference. A consequence of the explicit type approach is that the programmer must have a particular derivation of his type claim $e:\tau$ in mind in order to specify the required explicit type information...that is, one cannot completely rely on intuition (with respect to types) in writing a well-typed program, rather, one must know the reasons that his program is well-typed in order to make that claim. Therefore, the underlying philosophy of XTCL is based on the "derivations as types" one of the previous paragraph, but its programs are not as long in comparison.

In this chapter, we give the system XTCL along with an algorithm for type checking. As motivation, a system TCL' is given which is equal in typing power to TCL, but which places restrictions on the form of a type derivation. TCL' serves as the basis for XTCL. It is shown that typeable expressions in XTCL have "principal" types (i.e., most general) with respect to a special relation << ("below"). We show that this relation is decidable, but the decision problem is NP-complete. A type checking algorithm for XTCL is derived. Finally, generalizations of XTCL and effects of adding other language features are briefly discussed.

4.1 Restricting the Substitution Rule in TCL

In this section, we show that the substitution rule in TCL (i.e., rule 2) may be arranged in any derivation so that it precedes occurrences of rules 3, 4 and 5. This distributive property of rule 2 becomes clear through the iterative use of transformations on a type derivation tree. Transformations on derivation trees also show that the intersection rule distributes over the weaker rule, that is, rule 4-a can be pushed below a rule-5 child in the derivation tree.

For reference, we recall the typing rules for TCL from chapter 2.

TCL Type Rules

1-a) $S : (a \to b \to c) \to (a \to b) \to a \to c$

-b) $K : a \to b \to a$

2) $e : P\tau$ if $e : \tau$ (P a substitution)

3) $fg : \tau$ if $f : \sigma \to \tau$ and $g : \tau$

4-a) $e : \sigma \cap \tau$ if $e : \sigma$ and $e : \tau$

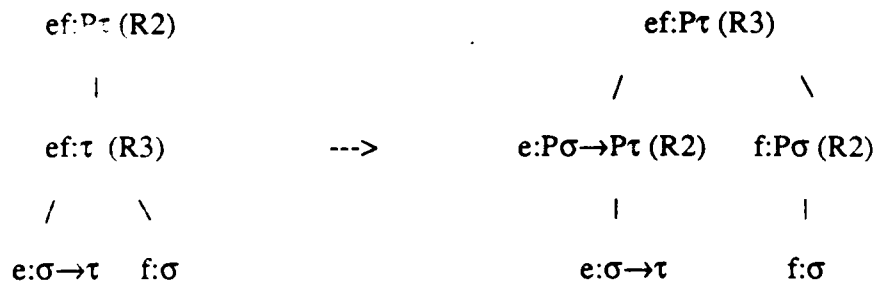
-b) $e : \sigma$ and $e : \tau$ if $e : \sigma \cap \tau$

5) $e : \tau$ if $e : \sigma$ and $\sigma \leq \tau$

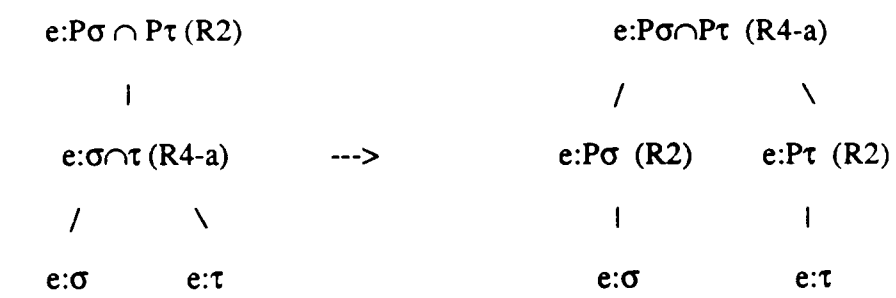
Throughout, we will use the notation Ax(b), where $b \in \{S, K\}$, to denote the axiom type of b, that is, the type given to b in rule 1 of the type system in question.

On examination of the typing rules of TCL, we see that in any derivation, if the substitution rule (rule 2) immediately follows rule 2, 3, 4 or 5, then there is a derivation in which the substitution rule is used before rule 2, 3, 4 or 5. To convince ourselves that this is the case, we note the validity of the following transformations:

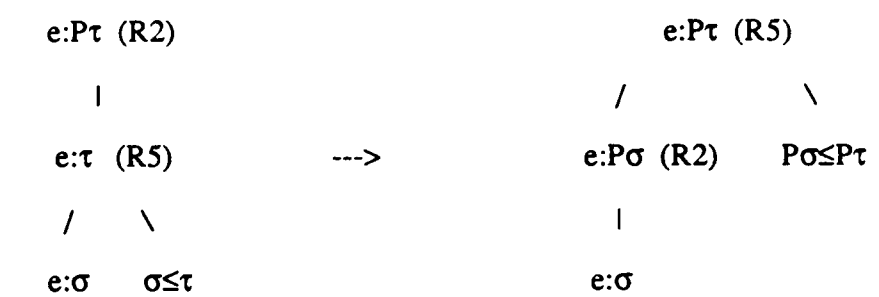
T1) Rule 2 can be exchanged with rule 3.



T2) Rule 2 can be exchanged with rule 4-a.



T3) By substitutivity of ≤, rule 2 can be exchanged with rule 5.



T4) Since "is a substitution of" is transitive, the following is valid:

By iteratively applying the transformations above, we have that if $e:\tau$, then there is a derivation in which the substitution rule is applied only to the types of the primitive combinators S and K, and is applied before any other rule (except 1) is applied. Hence, we may limit the use of the substitution rule to primitive combinators without destroying the relation between expressions and type expressions.

DEFINITION 4.1.1. (TCL') The type system TCL' has the same syntax and semantics of expressions and type expressions in TCL, but has the following rules instead:

1') $S : (a\to b\to c)\to(a\to c)\to b\to c$

$K : a\to b\to a$

2') $b : P\tau$ if $b : \tau$, $b\in\{S, K\}$, P a type substitution

3') $ef : \tau$ if $e : \sigma\to\tau$ and $f : \sigma$

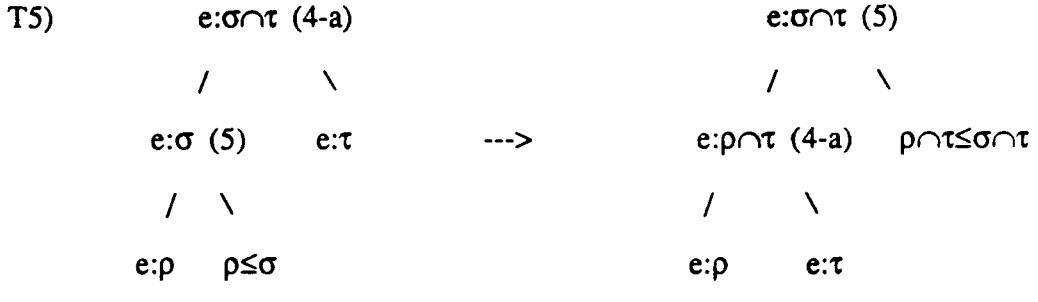
4') $e : \sigma\cap\tau$ if $e : \sigma$ and $e : \tau$

5') $e : \tau$ if $e : \sigma$ and $\sigma \le \tau$

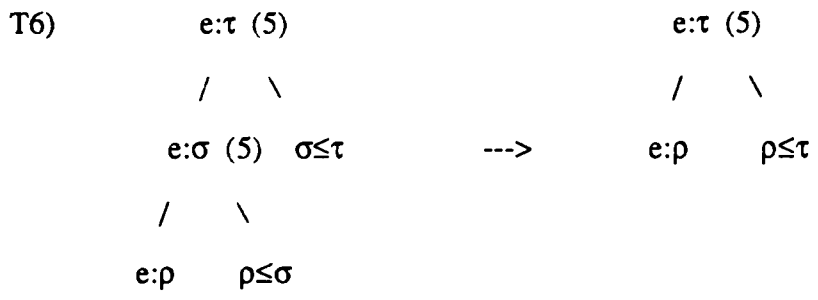
THEOREM 4.1.1 $e:\tau$ in TCL $\Leftrightarrow$ $e:\tau$ in TCL'.

proof $\Leftarrow$ is obvious. $\Rightarrow$ follows by the transformations 1--4 on the derivation of $e:\tau$ in TCL (note that rule 4-b is not needed since it is implied by rule 5). $\Diamond$

We may also show that in TCL and in TCL', rule 4-a distributes over rule 5.



Transitivity of ≤ gives us the following transformation.



The transformations tell us that any type τ derivable for a primitive combinator b in TCL is derivable by first applying separately some number n of substitutions (P_1, ..., P_n) to the axiomatic type Ax(b) of b, then applying rule 4-a successively to get b : $P_1Ax(b)\cap...\cap P_nAx(b)$, and finally applying rule 5 where $P_1Ax(b)\cap...\cap P_nAx(b) \leq \tau$. Formally, we have the following theorem.

THEOREM 4.1.2 Let b∈ {S, K}. Then b:τ in TCL $\Leftrightarrow$ $\exists n \geq 1$ $\exists$ substitutions P_1, ..., P_n such that $P_1Ax(b)\cap...\cap P_nAx(b) \leq \tau$

proof $\Leftarrow$ is trivial. $\Rightarrow$ by the distributivity of rule 2 over 3 and 4 and rule 4 over 3. ◊

For convenience, we define a relation << ("below") on TCL type expressions which captures the condition to the right of the $\Leftrightarrow$ in theorem 4.1.2.

DEFINITION 4.1.1. For types σ and τ in TCL,

$\sigma << \tau$ means $\exists n$ $\exists$ substitutions $P_1,...,P_n$ with $P_1\sigma\cap...\cap P_n\sigma \leq \tau$.

It can easily be shown that << is reflexive, transitive and substitution invariant (in fact, $\sigma << \tau \Rightarrow \sigma << P\tau$ and $P\sigma << P\tau$ for any substitution P). With this notation, theorem 4.1.2 can be

stated succinctly: $b{:}\tau \Leftrightarrow Ax(b) << t$. (The << relation will facilitate the definition of our explicitly-typed system XTCL, as well as provide a notion of principal type in XTCL.)

4.2 XTCL

It can be shown that if we required that the programmer explicitly supply any substitutions used in deriving types for the primitive combinators occurring in an expression e, then it can be decided automatically whether or not a TCL' derivation exists for a given $e{:}\tau$ which uses those substitutions. Rather than requiring all primitive combinators to be explicitly typed, we opt for a less severe restriction, namely, that the user explicitly attach a type to primitive combinators which appear on the left side of an application--all other types can be deduced by the type checker. Before giving the formal system, some definitions are required.

DEFINITION 4.2.1. (f-expression) Let e be an expression in TCL. An occurrence of a subexpression e' of e is called an f-expression (in e) if that occurrence of e' appears as the left hand side of an application in e. If in addition, e' is a primitive combinator, then e' is called a primitive f-expression (in e).

For example, K and (KS) are f-expressions in KSS, K is a primitive f-expression, but neither S's nor KSS itself are f-expressions.

Next, we introduce the notion of explicit typing.

DEFINITION 4.2.2. Given an expression e of TCL, an *explicit typing* of e is an association of a type expression τ with each primitive f-expression b of e. Equivalently, we can think of an

explicit typing of e as an expression e' derived from e by replacing each primitive f-expression b by [b::τ], where t is some type expression.

We are now ready to define XTCL. The computational expressions are TCL expressions with explicit types included in primitive f-expressions. The syntax of the types is the same as before, and the typing rules are based on the relation <<.

DEFINITION 4.2.3. XTCL is the language with syntax and typing rules described below. Note the use of brackets as terminals in the syntax. As before, Ax(b) refers to the type for b in rule 1.

Syntax

Exp ::= S | K | Pexp Exp

Pexp ::= [S::Texp] | [K::Texp] | Pexp Exp

Texp ::= Tvar | Tcnst | Texp → Texp | Texp ∩ Texp

Tvar ::= { an infinite supply of variables }

Tcnst ::= { a finite set of constants }

Typing Rules

1-a) S : $(a\to(b\to c))\to((a\to b)\to(a\to c))$

-b) K : $a\to(b\to a)$

-X) [b::τ] : τ if $Ax(b) << \tau$

2) e : Pτ if e : τ and e not an f-expression (P a substitution)

3) fg : τ if f:σ→τ and g:σ

4) e:σ∩τ if e:σ and e:τ

5) e : τ if e : σ and $\sigma \leq \tau$

The type inference rules deserve some explanation. The new rule 1-X ensures that explicit types given for any combinator must actually be derivable for that combinator in TCL. Notice that the substitution rule of TCL has been restricted to non-f-expressions. This essentially forces derivations of f-expressions to be done without applying substitutions, but by using the explicit type of the primitive f-expression (i.e., the leftmost combinator). On the average, half of the primitive combinators in an expression are primitive f-expressions, assuming we view an average expression as a random binary tree of applications.

Some examples of derivations in XTCL follow.

1) Self application.

Let τ denote the type $((a\cap(a\to b))\to(a\to b))\to((a\cap(a\to b))\to a)\to((a\cap(a\to b))\to b)$.

$[S::\tau] : ((a\cap(a\to b))\to(a\to b))\to((a\cap(a\to b))\to a)\to((a\cap(a\to b))\to b)$

$I : x\to x$

$I : (a\cap(a\to b))\to(a\to b)$ {since $I : (a\to b)\to(a\to b) \le (a\cap(a\to b))\to(a\to b)$ }

$[S::\tau]\ I : ((a\cap(a\to b))\to a)\to((a\cap(a\to b))\to b)$

$I : (a\cap(a\to b))\to a$ {since $I : a\to a \le (a\cap(a\to b))\to a$ }

$[S::\tau]\ I\ I :\ (a\cap(a\to b))\to b$

2) Composition. (Recall $\to$ associates to the right.)

Let

$\tau = ((y\to z)\to(x\to y\to z)\to(x\to y)\to x\to z)\to((y\to z)\to x\to y\to z)\to(y\to z)\to(x\to y)\to x\to z$

and $\sigma = ((a\to b\to c)\to(a\to b)\to a\to c)\to d\to(a\to b\to c)\to(a\to b)\to a\to c$.

$[K::\sigma] : ((a\to b\to c)\to(a\to b)\to a\to c)\to d\to(a\to b\to c)\to(a\to b)\to a\to c$

$S : (a\to b\to c)\to(a\to b)\to a\to c$

$[K::\sigma]\ S : d\to(a\to b\to c)\to(a\to b)\to a\to c$

$[K::\sigma]\ S : (y\to z)\to(x\to y\to z)\to(x\to y)\to x\to z$

$[S::\tau]: ((y\to z)\to(x\to y\to z)\to(x\to y)\to x\to z)\to((y\to z)\to x\to y\to z)\to(y\to z)\to(x\to y)\to x\to z$

$[S::\tau]\ ([K::\sigma]\ S) : ((y\to z)\to x\to y\to z)\to(y\to z)\to(x\to y)\to x\to z$

$K : a\to b\to a$

$K : (y\to z)\to x\to y\to z$

$[S::\tau]\ ([K::\sigma]\ S)\ K : (y\to z)\to(x\to y)\to x\to z$

We now show that XTCL has the typing power of TCL; that is, if we ignore the explicit types, TCL and XTCL define the same relation between expressions and types.

THEOREM 4.2.1. Let $e \in$ Exp. Then $e:\tau$ in TCL $\Leftrightarrow \exists$ an explicit typing e' of e such that $e':\tau$ in XTCL.

proof $\Leftarrow$ is trivial, since the rules of TCL are less restrictive than the rules for XTCL (ignoring explicit types). $\Rightarrow$, by induction on the size of the derivation of $e:\tau$ in TCL'. Base case is trivial, since both systems agree on rules 1-a and 1-b. Now take cases of the rule used at the root of the TCL' derivation. If rule 2 is used, then e is a primitive combinator and the result follows by theorem 4.1.2 using rule 1-X. Suppose rule 3 is used at the root. Then $e=fg$ and $f:\sigma\to\tau$, $g:\sigma$ in TCL'. By hypothesis, there is an explicit typing g' of g such that $g':\sigma$ in XTCL. There are 2 cases for f:

i) f is an application. By hypothesis, there is an explicit typing f' of f with $f':\sigma\to\tau$ in XTCL, and thus f'g' is an explicit typing of fg having type t.

ii) f is a primitive combinator. By theorem 4.1.2, $f:\sigma\rightarrow\tau$ means $Ax(f) << \sigma\rightarrow\tau$, thus $[f::\sigma\rightarrow\tau] : \sigma\rightarrow\tau$ in XTCL, and hence $[f::\sigma\rightarrow\tau]\ g'$ is an explicit typing of fg having type τ in XTCL.

If rule 4 is used, then $\tau=\sigma\cap\rho$ and $e:\sigma$, $e:\rho$ in TCL'. By hypothesis, there are explicit typings e' and e'' of e such that $e':\sigma$ and $e'':\rho$ in XTCL. Certainly if $[b::\gamma]$ appears in e', then we may replace it by $[b::\gamma\cap\gamma']$, provided $Ax(b) << \gamma'$, and still derive the type σ (since $Ax(b) << \gamma$ and $Ax(b) << \gamma' \Rightarrow Ax(b) << \gamma\cap\gamma'$, and $\gamma\cap\gamma' \leq \gamma$). Thus, combine e' and e'' by intersecting the corresponding explicit types, giving e''' which has types σ and ρ, and hence has type $\sigma\cap\rho$ in XTCL. Finally, if rule 5 is used at the root, $e:\sigma$ and $\sigma \leq \tau$ in TCL', which implies by hypothesis that there is an explicit typing e' of e having type s in XTCL, and $\sigma\leq\tau$ implies $e':\tau$ in XTCL. ◊

As in TCL, we use I to denote SKK and $[I::\tau]$ to denote $[S::\sigma]KK$ where σ is an explicit type required to derive $[S::\sigma]KK : \tau$ in XTCL (there must be such a σ by the previous lemma). As before, we will use $I:a\rightarrow a$ as Ax(I) in typing S-K-I combinations of XTCL.

4.3 Principal Types

At this point, it is useful to introduce the notion of a principal type for an expression e in XTCL. In general mathematical terms, an object is "principal" with respect to a given operation if every other object can be generated by or derived from it. Milner showed that every typeable expression in the parametric system has a principal type with respect to substitution [Mil78], and Coppo showed that each typeable expression in the conjunctive system has a principal type with respect to a certain operation [Cop80].

DEFINITION 4.3.1. A type τ is *principal* for e with respect to $<<$ if $e:\tau$ and for all types σ of e, $\tau << \sigma$.

We define an algorithm PT for computing a principal type for a computational expression e in XTCL. If a principal type for e does not exist, PT returns "error." Recall that Ax(e) for e = S, K is the type given to e in rule 1 of the typing rules.

ALGORITHM 4.3.1

PT(e) = if $e \in \{S,K\}$ then Ax(e)

else if $e=[b:\tau]$ and $Ax(b) << \tau$ then τ else error

else if e=fg and $PT(g) \neq$error

and $PT(f)=(\sigma_1 \rightarrow \tau_1) \cap ... (\sigma_n \rightarrow \tau_n) \cap$(non-$\rightarrow$'s) and $\exists i.\ PT(g) << \sigma_i$

then $\cap\{\tau_i \mid PT(g) << \sigma_i, i \leq n\}$

else error

We now show that PT maps computational expressions of XTCL to principal types. To push the induction through, we prove a stronger claim for f-expressions.

THEOREM 4.3.1 $e:\tau$ in XTCL $\Leftrightarrow$ $PT(e) << \tau$.

proof ($\Leftarrow$) Show by induction on the size of e that e:PT(e) provided $PT(e) \neq$error. ($\Rightarrow$) We show in addition to the claim that if e is an f-expression then $PT(e) \leq \tau$. Induct on the derivation of $e:\tau$. Base case is when rule 1 is used (at the root of the derivation). If rule 1-a or 1-b is used, $e=b:\tau = Ax(b) << Ax(b) = PT(e)$ (rule 1-a and 1-b are never used on f-expressions). For rule 1-X, $e=[b:\tau]:\tau$ is an f-expression and $Ax(b) << \tau \leq \tau = PT(e)$. If rule 2 is used, $e:\sigma$, $P\sigma = \tau \Rightarrow \sigma << \tau \Rightarrow$ by hypothesis $PT(e) << \sigma << \tau$. For rule 3, let e=fg, $f:\sigma \rightarrow \tau$, $g:\tau$. Then $f:\sigma \rightarrow \tau$ is an f-expression $\Rightarrow$ (by hypoth) $PT(f) = \cap\{\sigma_i \rightarrow \tau_i \mid i \leq n\} \cap$(non-

$\cap$'s) $\leq \sigma \rightarrow \tau$, thus $\cap\{\tau_i \mid i \leq n, \sigma \leq \sigma_i\} \leq \tau$. Also (by hypoth on g:$\sigma$) PT(g) $<<$ σ and $\sigma \leq \sigma_i$ implies PT(g) $<<$ σ, thus $\cap\{\tau_i \mid i \leq n, PT(g) << \sigma_i\} \leq \tau$ (and $<< \tau$, too). For rule 4, e:$\tau = \tau_1 \cap \tau_2$, e:$\tau_1$ and e:τ_2. By hypothesis, PT(e) $<< \tau_1$ and PT(e) $<< \tau_2 \Rightarrow$ PT(e) $<< \tau_1 \cap \tau_2$. If e is an f-expression, then by hypothesis PT(e) $\leq \tau_1$ and PT(e) $\leq \tau_2 \Rightarrow$ PT(e) $\leq \tau_1 \cap \tau_2$. If rule 5 is used, then e: σ and $\sigma \leq \tau \Rightarrow$ by hypothesis PT(e) $<<$ σ and $\sigma \leq \tau \Rightarrow$ PT(e) $\leq \tau$. If e is an f-expression then by hypothesis PT(e) $\leq \sigma \leq \tau$. $\Diamond$

It would be a nice result if we could show that all typeable expressions in TCL have principal types.

4.4 Decidability of Explicit Type Checking

Certainly PT is computable provided $<<$ is computable, and PT computable implies that type checking in XTCL is computable by theorem 4.3.1. In this section, we give a bound on the size of substitutions which need to be searched in deciding $\alpha << \beta$. This bound is a computable function of $|\alpha| \times |\beta|$, which forces the decidability of $<<$. The bound $(|\alpha| 2^{|\beta|})$ is not optimal, but is sufficiently small to show the existence of a polynomial-time nondeterministic algorithm which decides a $<<$ b (see section 4.5).

DEFINITION 4.4.1. (Monotonic and Amonotonic Occurrences) Let τ be an occurrence of an expression in α. Then that occurrence of τ occurs *monotonically* in α if it appears to the left of an even number of $\rightarrow$'s. Symmetrically, if τ appears to the left of an odd number of $\rightarrow$'s, then it is said that τ occurs *amonotonically* in α.

LEMMA 4.4.1.

I) If x occurs once and monotonically in α, $\alpha[x\leftarrow\tau] \leq \beta$ and $\tau'\leq\tau$, then $\alpha[x\leftarrow\tau'] \leq \beta$.

II) If x occurs once and amonotonically in α, $\alpha[x\leftarrow\tau] \leq \beta$ and $\tau\leq\tau'$, then $\alpha[x\leftarrow\tau'] \leq \beta$.

III) If x occurs once and monotonically in β, $\alpha \leq \beta[x\leftarrow\tau]$ and $\tau\leq\tau'$, then $\alpha \leq \beta[x\leftarrow\tau']$.

IV) If x occurs once and amonotonically in β, $\alpha \leq \beta[x\leftarrow\tau]$ and $\tau'\leq\tau$, then $\alpha \leq \beta[x\leftarrow\tau']$.

proof Straightforward induction on $|\alpha| + |\beta|$ using transitivity of $\leq$. $\Diamond$

Lemma 4.4.1 (I) is useful since it allows us to add terms to monotonic occurrences of subexpressions $\rho_1, ..., \rho_k$ in α, where $\alpha\leq\beta$, and the resulting expression is still weaker than β. In particular, we can replace the monotonic occurrences of each ρ_i in α with $\rho_1\cap...\cap\rho_k$ and the expression obtained is weaker than β. We will use this technique later to construct size-bounded solutions to inequalities.

The << problem, which can be stated as

"Given type expressions α and β, is there an $n\geq1$ such that $\exists\ \alpha_1, \alpha_2, ..., \alpha_n$ each identical to α up to renaming of variables, $\exists$ a substitution P such that $P(\alpha_1\cap...\cap\alpha_n) \leq \beta$?"

can be reduced to,

"Given type expressions α and β, is there a substitution P such that $P\alpha \leq \beta$?"

provided that we can restrict the size that n has to be. We show that n need not be larger than the number of maximal rightmost arrow paths in the expression b.

DEFINITION 4.4.2. (NRP) Let β be a type expression. Viewing β as an expression tree, NRP(β) is the number of maximal paths from the root of β which always take the right branch of any $\rightarrow$, i.e.,

$$\begin{aligned} NRP(\beta) = \quad & \text{if } \beta \text{ is an atom then } 1 \\ & \text{if } \beta = \beta_1 \rightarrow \beta_2 \text{ then } NRP(\beta_2) \\ & \text{if } \beta = \beta_1 \cap \beta_2 \text{ then } NRP(\beta_1) + NRP(\beta_2) \end{aligned}$$

For example, NRP($(c \cap d) \rightarrow (b \cap (a \cap (e \cap f)))$) = 3.

We now use NRP to place a bound on the number of copies of a needed to show $\alpha << \beta$.

LEMMA 4.4.2. Let α and β by type expressions, $\alpha = \alpha_1 \cap ... \cap \alpha_n$ where each α_i is a term, and $\alpha \leq \beta$. Then $\exists H \subseteq 1..n$ such that $|H| \leq NRP(\alpha)$ and $\cap\{\alpha_i \mid i \in H\} \leq \beta$.

proof Induct on $|\beta|$. For $\beta = t$, an atom, $\exists k \leq n$ such that $t = \alpha_k$. Pick $H = \{k\}$ and the result follows. If $\beta = \beta_1 \rightarrow \beta_2$, $\exists I \subseteq 1..n$ such that $\forall i \in I\ \alpha_i = \sigma_i \rightarrow \tau_i$, and $\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \beta_2$, $\Rightarrow$ (by hypothesis) $\exists H \subseteq I$ such that $|H| \leq NRP(\beta_2)$ and $\cap\{\tau_i \mid i \in H\} \leq \beta_2 \Rightarrow |H| \leq NRP(\beta)$ and $\cap\{\alpha_i \mid i \in H\} \leq \beta$. For $\beta = \beta_1 \cap \beta_2$, $\alpha \leq \beta_1$ and $\alpha \leq \beta_2 \Rightarrow$ (by hypothesis) $\Rightarrow H_1$ and H_2 subsets of 1..n such that $|H_i| \leq NRP(\beta_i)$ and $\cap\{\alpha_j \mid j \in H_i\} \leq \beta_i$ (i = 1, 2). Choose H $= H_1 \cup H_2$ and the result follows by $NRP(\beta) = NRP(\beta_1) + NRP(\beta_2)$ and by lemma 4.4.1 (I). $\Diamond$

It follows from lemma 4.4.2 that to decide if $\alpha << \beta$, we need only look for a substitution P making $P(\alpha_1 \cap ... \cap \alpha_n) \leq \beta$ where each α_i is a renaming of α, and $n \leq NRP(\beta)$. By lemma 4.4.1, we can assume that $n = NRP(\beta)$.

We now seek a bound on the size of solutions $\rho_1, ..., \rho_k$ for variables $x_1, ..., x_k$ in α needed to satisfy "$\alpha \leq \beta$", that is, such that $\alpha[x_1 \leftarrow \rho_1, ..., x_k \leftarrow \rho_k] \leq b$.

LEMMA 4.4.3. Let α and β be types, let K be a finite index set indexing variables x_i ($i \in K$), and let $P = \{ [x_i := \rho_i] \mid i \in K \}$ be a substitution of x_i with types ρ_i.

I) If each x_i ($i \in K$) occurs at most once and monotonically in α, and $P\alpha \leq \beta$, then $\exists K' \subseteq K$ and substitution P' having domain $\{x_i \mid i \in K'\}$ such that $\forall i \in K'$ $P(x_i) \leq P'(x_i)$ and $|P'(x_i)| \leq |\beta|$, and $P'\alpha \leq \beta$.

II) If each x_i ($i \in K$) occurs at most once and amonotonically in β, and $\alpha \leq P\beta$, then $\exists K' \subseteq K$ and substitution P' having domain $\{x_i \mid i \in K'\}$ such that $\forall i \in K'$ $P(x_i) \leq P'(x_i)$ and $|P'(x_i)| \leq |\alpha|$, and $\alpha \leq P'\beta$.

proof Induct on $|\alpha| + |\beta| + |P|$. Base is when α, β and ρ_i are atoms ($K = \{i\}$). For I) and II) we can choose $K' = K$, $P' = P$ and the result trivially follows. For the induction part, take cases on β:

<u>β=t, an atom</u>.

I) Then $P\alpha \leq \beta \Rightarrow P\alpha$ contains t as a term. If t a term of α, then choose $K' = \emptyset$ and P' as the null substitution, and the result follows. If t not a term of α, then x_k is a term of α for some $k \in K$, and t is a term of $P(x_k)$. Choose $K' = \{k\}$, $P' = [x_k := t]$ and the result follows.

II) Then $\alpha \leq Pt$. Now t can not be equal to any $x_i \in K$ (since they occur amonotonically in β), hence we can choose $K' = \emptyset$, P' as the null substitution, and the result follows.

$\underline{\beta=\beta_1\to\beta_2}$

I) Then $P\alpha \le \beta$. WLOG, assume all terms of a are needed in the proof of $P\alpha \le \beta$ (otherwise, prune away the unused term of α, giving α', and apply the induction hypothesis to $\alpha' \le \beta$). Let $H \subseteq K$ index the variables which are terms of α, and let I index the other terms of α, which are, by assumption, all arrow terms, that is, $\alpha = \cap\{x_i \mid i\in H\} \cap \cap\{\sigma_i\to\tau_i \mid i\in I\}$. $P\alpha \le \beta$ implies $\cap\{\rho_i \mid i\in H\} \cap \cap\{P\sigma_i \to P\tau_i \mid i\in I\} \le \beta$. For each $i\in H$, let J_i index the terms of ρ_i, hence $\rho_i = \cap\{\rho_{ij} \mid j \in J_i\}$. By the same reasoning as before, we can assume that each ρ_{ij} is used in the proof of $P\alpha \le \beta$, hence, for each $i\in H$ and $j\in J_i$, $\rho_{ij} = \gamma_{ij}\to\delta_{ij}$ and

$$\beta_1 \le \cap\{\ \cap\{\gamma_{ij} \mid j\in J_i\} \mid i\in H\ \} \cap \cap\{P\sigma_i \mid i\in I\} \quad \text{and}$$

$$\cap\{\ \cap\{\delta_{ij} \mid j\in J_i\} \mid i\in H\ \} \cap \cap\{P\tau_i \mid i\in I\} \le \beta_2.$$

Bipartition $K\backslash H$ into K_1 and K_2 such that K_1 contains every index i of x_i appearing in $\cap\{\sigma_i \mid i\in I\}$ and K_2 contains every index i of x_i which appears in $\cap\{\tau_i \mid i\in I\}$. Let P_1 be P restricted to K_1. Certainly $\beta_1 \le P_1(\sigma_i)$ for all $i\in I$. Note that $|b_1| + |\cap\{x_i \mid i\in K\}| + |P_1| < |\beta| + |\alpha| + |P|$, and that each x_i ($i\in K_1$) occurs at most once and amonotonically in $\cap\{\sigma_i \mid i\in I\}$, thus by hypothesis (II), $\exists K_1' \subseteq K\ \exists P_1'$ with domain $\{x_i \mid i\in K_1'\}$ such that $\forall i\in K_1'\ P1(x_i) \le P_1'(x_i)$ and $|P_1'(x_i)| \le |\beta_1|$, and $\beta_1 \le P_1'(\cap\{\sigma_i \mid i\in I\}) = \cap\{P_1'(\sigma_i) \mid i\in I\}$. Let $P_2 = \{[x_i := \cap\{\delta_{ij} \mid j\in J_i\}] \mid i\in H\} \cap \{[x_i := P(x_i)] \mid i\in K_2\}$, thus $P_2(\cap\{x_i \mid i\in H\} \cap \cap\{\tau_i \mid i\in I\}) \le \beta_2$. Note that $|\cap\{x_i \mid i\in H\} \cap \cap\{\tau_i \mid i\in I\}| + |\beta_2| + |P_2| < |\alpha| + |\beta| + |P|$ and each x_i ($i\in$domain(P2)) appears at most once and monotonically in $\cap\{x_i \mid i\in H\} \cap \cap\{\tau_i \mid i\in I\}$, thus by hypothesis (I), $\exists M \subseteq H\cup K_2\ \exists P_2'$ with domain $\{x_i \mid i\in M\}$ such that $\forall i\in M\ P_2(x_i) \le P_2'(x_i)$ and $|P_2'(x_i)| \le |\beta_2|$, and $P_2'(\cap\{x_i \mid i\in H\} \cap \cap\{\tau_i \mid i\in I\}) \le \beta_2$. Construct $K' = M\cup K_1'$, $P' = \{[x_i := P_1'(x_i)] \mid i\in K_1'\} \cup \{[x_i := P_2'(x_i)] \mid i\in M\backslash H\} \cup \{[x_i := \beta_1\to P_2'(x_i)] \mid i\in M\backslash K_2\}$. Certainly $K' \subseteq K$ and P' has domain $\{x_i \mid i\in K'\}$. For all $i\in K'\backslash H$, obviously $P(x_i) \le P'(x_i)$. For $i\in K'\cap H$ ($= M\backslash K_2$), $P'(x_i) = \beta_1\to P_2'(x_i)$, and $P(x_i) = \cap\{\gamma_{ij} \to \delta_{ij} \mid j\in J_i) \le \cap\{\gamma_{ij} \mid j\in J_i\} \to \cap\{\delta_{ij} \mid j\in J_i\} \le \beta_1 \to P_2'(x_i)$. Check the sizes: $\forall i\in K_1'$, $|P_1'(x_i)| \le |\beta_1| < |\beta|$; $\forall i\in M\backslash H$ $|P_2'(x_i)| \le |\beta_2| < |\beta|$; $\forall i\in M\backslash K2$ $|P_2'(x_i)| \le |\beta_2| \Rightarrow |\beta_1 \to P_2'(x_i)| \le |\beta_1 \to \beta_2|$; hence $\forall i\in K'$

$|P'(x_i)| \leq |\beta|$. We must check if $P'\alpha \leq \beta$. First note that $P'(\alpha) \leq (\beta_1 \cap \cap\{P_1'(\sigma_i) \mid i \in I\}) \rightarrow (\cap\{P_2'(x_i) \mid i \in H \cap K'\} \cap \cap\{P_2'(\tau_i) \mid i \in I\})$. Certainly $\beta_1 \leq \cap\{P_1'(\sigma_i) \mid i \in I\}$, and we know that $\cap\{P_2'(x_i) \mid i \in H\} \cap \cap\{P_2'(\tau i) \mid i \in I\} \leq \beta_2$. Rewrite $\cap\{P_2'(x_i) \mid i \in H\}$ as $\cap\{x_i \mid i \in H \backslash K'\}$ $\cap \cap\{P_2'(x_i) \mid i \in K' \cap H\}$. Since no x_i occurs in β, the x_i's ($i \in H \backslash K'$) are not needed in the proof of $\cap\{P_2'(x_i) \mid i \in H\} \cap \cap\{P_2'(\tau_i) \mid i \in I) \leq \beta_2$, hence $\cap\{P_2'(x_i) \mid i \in H \backslash K'\} \cap \cap\{P_2'(\tau_i) \mid i \in I\} \leq \beta_2$. But this implies $P'\alpha \leq \beta$.

II) $\alpha \leq P\beta \Rightarrow \alpha \leq P\beta_1 \rightarrow P\beta_2$. Bipartition K into K_1 and K_2 containing the x_i's appearing in β_1 and β_2, respectively, and let P_1 and P_2 be the restrictions of P to K_1 and K_2, respectively. Now let $\alpha = \alpha_1 \cap \ldots \cap \alpha_n$ and let $I \subseteq 1..n$ such that $\alpha_i = \sigma_i \rightarrow \tau_i$ (for $i \in I$) and $P_1\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \beta_2$. By hypothesis (I), $\exists K_1' \subseteq K_1\ \exists P_1'$ with domain $\{x_i \mid i \in K_1'\}$ such that $\forall i \in K_1'\ P_1(x_i) \leq P_1'(x_i)$, $|P_1'(x_i)| \leq |\cap\{\sigma_i \mid i \in I\}|$, and $P_1'\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$. Similarly, by hypothesis (II) $\exists K_2' \subseteq K_2\ \exists P_2'$ with domain $\{x_i \mid i \in K_2'\}$ such that $\forall i \in K_2'\ P_2(x_i) \leq P_2'(x_i)$, $|P_2'(x_i)| \leq |\cap\{\tau_i \mid i \in I\}|$, and $\cap\{\tau_i \mid i \in I\} \leq P_2'\beta_2$. Simply construct $K' = K_1' \cap K_2'$, $P' = P_1' \cap P_2'$ and the result follows.

<u>$\beta = \beta_1 \cap \beta_2$</u>

I) $P\alpha \leq \beta_1 \cap \beta_2 \Rightarrow P\ \alpha \leq \beta_1$ and $P\alpha \leq \beta_2$. Invoke the hypothesis to get, for j=1,2, $\exists K_j\ \exists P_j$ with domain $\{x_i \mid i \in K_j\}$ such that $\forall i \in K_j\ P(x_i) \leq P_j(x_i)$, $|P_j(x_i)| \leq |b_j|$ and $P_j\alpha \leq \beta_j$. Construct $K' = K_1 \cup K_2$, $P' = \{[x_i := P_1(x_i)] \mid i \in K_1 \backslash K_2\} \cup \{[x_i := P_2(x_i)] \mid i \in K_2 \backslash K_1\} \cup \{[x_i := P_1(x_i) \cap P_2(x_i)] \mid i \in K_1 \cap K_2\}$. Certainly $K' \subseteq K$ and P' has domain K'. For $i \in K_1 \cap K_2$ $|P_1(x_i) \cap P_2(x_i)| \leq |\beta_1 \cap \beta_2|$ and $P(x_i) \leq P_1(x_i) \cap P_2(x_i)$, hence $P(x_i) \leq P'(x_i)$ and $|P'(x_i)| \leq |\beta|$. Show $P'\alpha \leq \beta$. $P_1\alpha \leq \beta_1 \Rightarrow (P_1 \cup \{[x_i := P_2(x_i)] \mid i \in K_2 \backslash K_1\})\ \alpha \leq \beta_1$ by substitutivity of $\leq$ (since no x_i's occur in β). Since each x_i is in a monotonic position in α, lemma 4.4.1 (I) allows us to replace occurrences of $P_1(x_i)$ with $P_1(x_i) \cap P_2(x_i)$ in $P_1\alpha$, hence $P'\alpha \leq \beta_1$. A similar argument shows that $P'\alpha \leq \beta_2$.

II) $\alpha \leq P(\beta_1) \cap P(\beta_2)$. This is easily proven using the technique for the case when $\beta=\beta_1 \rightarrow \beta_2$--by bipartitioning K into K_1, K_2, defining P_1 and P_2, invoking the hypothesis to get P_1' and P_2', and finally defining $K' = K_1 \cup K_2$ and $P' = P_1 \cup P_2$. $\Diamond$

Lemma 4.4.3 will allow us to place a bound on the size of a solution to an inequality, provided a solution exists. We first prove a lemma which bounds the depth that solutions need to have. The depth we are talking about here is the depth of the expression tree, ignoring $\cap$ nodes.

DEFINITION 4.4.3. (Arrow Level, Arrow Depth) Let α be a type expression containing a particular occurrence ρ. The alevel ("arrow level") of that occurrence of ρ in α is defined as the number of $\rightarrow$ nodes above r in a. The adepth ("arrow depth") of α, written "AD(α)," is defined as the maximum alevel of any subexpression occurrence of α.

LEMMA 4.4.4. Let $\alpha \leq \beta$, and let $\rho=\rho_1 \rightarrow \rho_2$.

I) if ρ occurs in α at an alevel $\geq$ AD(β), then that occurrence of ρ may be replaced in α by any type expression ρ', and the resulting expression is weaker than β.

II) if ρ occurs in β at an alevel $\geq$ AD(α), then that occurrence of ρ may be replaced in β by any type expression ρ', and the resulting expression is stronger than α.

proof Induct on $|\alpha| + |\beta|$. Take cases on β.

<u>β=t, an atom</u> (this covers the base case).

I) $\alpha = t \cap \alpha'$ where ρ occurs in α'. Let ρ' be any expression. Then $t \cap \alpha'' \leq \beta$ where $\alpha'' = \alpha'[\rho \leftarrow \rho']$ (α' with ρ' replaced for α).

II) Not applicable.

<u>$\beta=\beta_1 \rightarrow \beta_2$</u>

I) $\alpha = \alpha_1 \cap ... \cap \alpha_n$. Let $I \subseteq 1..n$ such that $\alpha_i = \sigma_i \rightarrow \tau_i$ $(i \in I)$ and $\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \beta_2$. Let $k \leq n$ such that ρ occurs in α_k. If $k \notin I$, then the result follows trivially, since ρ is not used in the subproofs involving β_1 and β_2. Suppose $k \in I$. Note $\rho \neq \alpha_k$, since then ρ would occur at an alevel in a less than AD(β). Suppose ρ occurs in σ_k. Then ρ occurs at an alevel in $\cap\{\sigma_i \mid i \in I\}$ of one less than its alevel in α, and $AD(\beta_1) \leq AD(\beta) - 1$ implies ρ occurs at an alevel in $\cap\{\sigma_i \mid i \in I\}$ which is greater than or equal to $AD(\beta_1)$. By hypothesis, ρ can be replaced in $\cap\{\sigma_i \mid i \in I\}$ by any ρ' and we get a type expression stronger than β_1. It follows that we can replace ρ by ρ' in α and get an expression weaker than β. A similar argument shows the result for when ρ occurs in τ_k.

$\underline{\beta = \beta_1 \cap \beta_2}$

II) Let $\alpha = \alpha_1 \cap ... \cap \alpha_n$, $I \subseteq 1..n$ such that each $\alpha_i = \sigma_i \rightarrow \tau_i$ $(i \in I)$ and $\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \beta_2$. Now either ρ occurs in β_1 or it occurs in β_2--it can not be equal to β by our assumption on the adepth of α. Suppose it occurs in β_2. Then ρ occurs in β_2 at an alevel of one less than it occurs in α, and $AD(\cap\{\tau_i \mid i \in I\}) \leq AD(\alpha) - 1$, thus by hypothesis, we can replace ρ in β_2 by ρ' and get an expression stronger than $\cap\{\tau_i \mid i \in I\}$. Therefore, replacing ρ by ρ' in β gives a type expression stronger than α. A similar argument shows the result when r occurs in β_1.

$\underline{\beta = \beta_1 \cap \beta_2}$

I) $\alpha \leq \beta_1$ and $\alpha \leq \beta_2$. Since $AD(\beta) = MAX(AD(\beta_1), AD(\beta_2))$, ρ occurs at an alevel in α which is greater than or equal to $AD(\beta_1)$ and $AD(\beta_2)$, $\Rightarrow$ (by hypothesis) any ρ' can be replaced for ρ in α and we get an expression that is weaker than both β_1 and β_2, and hence weaker than $\beta_1 \cap \beta_2$.

II) $\alpha \leq \beta_1$ and $\alpha \leq \beta_2$. Suppose ρ occurs in β_1. Then it occurs in β_1 at the same alevel as it occurs in β, thus (by hypothesis) we can replace ρ in β_1 by any ρ' and get a type stronger than α. Replacing ρ in $\beta_1 \cap \beta_2$ by ρ' gives the same results, since $\alpha \leq \beta_2$. The argument is symmetric when ρ occurs in β_2. $\Diamond$

An immediate consequence of lemma 4.4.4 is that a solution ρ for x in an inequality $\alpha(x) \leq \beta$ need never have adepth more than that of β. To see this, let $\alpha[x \leftarrow \rho] \leq \alpha$ and let z be any atom. If ρ has occurrences $\tau_1,...,\tau_k$ of expressions at an alevel in ρ greater than or equal to $AD(\beta)$, then $\tau_1,...,\tau_k$ occur at places in $\alpha[x \leftarrow \rho]$ at an alevel in $\alpha[x \leftarrow \rho]$ greater than or equal to $AD(\beta)$--thus, each τi can be replaced by z in $a[x \leftarrow \rho]$ and we get a type expression weaker than β, hence $\alpha[x \leftarrow \rho'] \leq \beta$ where ρ' is α with each τ_i replaced by z. Certainly $AD(\rho) \leq AD(\beta)$. We state this as a lemma:

LEMMA 4.4.5. Let $\alpha[x_1 \leftarrow \rho_1,...,x_m \leftarrow \rho_m] \leq \beta$. Then there are types ρ_1', ..., ρ_k' such that $\alpha[x_1 \leftarrow \rho_1'$, ..., $x_m \leftarrow \rho_m'] \leq \beta$ and $AD(\rho_i') \leq AD(\beta)$ for each $i \leq k$.

proof By the preceding paragraph, and iteration on each x_i. ◊

We are now ready to place a restriction on the size that any solution P needs to be in order for $P\alpha \leq \beta$.

LEMMA 4.4.6. Let the variable x occur k places in α, let k_m be the number of monotonic occurrences of x in α, let k_a be the number of amonotonic occurrences of x in α, and let $\alpha[x \leftarrow \rho] \leq \beta$. Then if $k_m > 0$, there exists ρ' such that $|\rho'| < k_m|b| + k_m$, and such that $\alpha[x \leftarrow \rho'] \leq \beta$. If $k_m = 0$, then there exists $\rho' = \sigma_1 \rightarrow \sigma_2 \rightarrow ... \rightarrow \sigma_n \rightarrow t$ for some $0 \leq n \leq AD(\beta)$, t an atom, such that for each σ_i, $|\sigma_i| < k|\beta| + k$, and $\alpha[x \leftarrow \rho'] \leq \beta$.

proof First assume $k_m > 0$. Let $x_1, ...,x_{km}$ be new variables, and let α' be derived from α by replacing each monotonic occurrence of x with a unique xi, hence $\alpha'[x_1 \leftarrow x,...,x_{km} \leftarrow x] = \alpha$. Let $P = [x_1 := \rho,...,x_{km} := \rho]$. Then $P(\alpha'[x \leftarrow \rho]) \leq \beta$, and by lemma 4.4.3 (I) there is a $K \subseteq 1..k_m$ and P' having domain K such that for all $i \in K$ $P(x_i) \leq P'(x_i)$ and $|P'(x_i)| \leq |\beta|$, and such that $P'(\alpha'[x \leftarrow \rho]) \leq \beta$. Let $\rho' = \cap\{P'(x_i) \mid i \in K\}$. Now $|\rho'| = (\sum_{i \in K} P'(x_i)) + |K| - 1 <$

$k_m\,|\beta| + k_m$. Also, $\rho' \leq P'(x_i)$ for all $i \in K \Rightarrow$ (since each x_i in $\alpha'[x \leftarrow \rho]$ is monotonic) $P''(\alpha[x \leftarrow \rho]) \leq \beta$ where $P'' = \{[x_i := \rho'] \mid i \in K\}$. Now each occurrence of x in $P''(\alpha')$ is amonotonic (assuming that ρ' does not contain x, which, of course, is perfectly valid since x does not occur in β), and since $\rho \leq \rho'$, by lemma 4.4.2 $P''(\alpha'[x \leftarrow \rho']) \leq \beta$. Also, by substitutivity, we may replace any x_i ($i \in (1..k_m) \backslash K$) remaining in $P''(\alpha'[x \leftarrow \rho'])$ with ρ' and obtain an expression weaker than β (since no x_i's are in β), hence, $\alpha'[x \leftarrow \rho', x_1 \leftarrow \rho', \ldots, x_{km} \leftarrow \rho'] \leq \beta$, or equivalently $\alpha[x \leftarrow \rho'] \leq \beta$. Suppose $k_m = 0$, that is, there are no monotonic occurrences of x in α, and $\alpha[x \leftarrow \rho] \leq \beta$. If ρ contains an atom term t, then $\alpha[x \leftarrow t] \leq \beta$, and the result follows trivially with n=0. Assume there is not an atomic term t in ρ. By lemma 4.4.1, we may assume that there are no intersections in r in monotonic positions (since they can be removed to get ρ', and $\rho \leq \rho' \Rightarrow \alpha[x \leftarrow \rho'] \leq \beta$ holds). Then ρ is of the form $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow t$ for some atom t. By lemma 4.4.3, we may assume that $n \leq AD(\beta)$. Let $y_1, \ldots, y_n$ be new variables. Then $\alpha[x \leftarrow (y_1 \rightarrow y_2 \rightarrow \ldots \rightarrow y_n \rightarrow t)]\ [y_1 \leftarrow \sigma_1, \ldots, y_n \leftarrow \sigma_n] \leq \beta$. Note that $y_1, \ldots, y_n$ each appears k times and monotonically in $\alpha[x \leftarrow (y_1 \rightarrow \ldots \rightarrow y_n \rightarrow t)]$. By the previous part of this proof, there is a type σ_1' such that $|\sigma_1'| < k|\beta| + k$ and $\alpha[x \leftarrow (\sigma_1' \rightarrow y_2 \rightarrow \ldots \rightarrow y_n \rightarrow t)]\ [y_2 \leftarrow \sigma_2, \ldots, y_n \leftarrow \sigma_n] \leq \beta$. Iteration gives n expressions $\sigma_1', \ldots, \sigma_n'$ all of size less then $k|\beta| + k$, such that $\alpha[x \leftarrow (\sigma_1' \rightarrow \sigma_2' \rightarrow \ldots \rightarrow \sigma_n' \rightarrow t)] \leq \beta$.◊

By repeated applications of this lemma, it follows that if $\alpha[x_1 \leftarrow \rho_1, \ldots, x_n \leftarrow \rho_n] \leq \beta$, then there are types $\rho_1', \ldots, \rho_n'$ each of size not more than $|\alpha|\,|\beta|^2$ and arrow depth not more than $AD(\beta)$, such that $\alpha[x_1 \leftarrow \rho_1', \ldots, x_n \leftarrow \rho_n'] \leq \beta$. Therefore, given two types α and β, we may restrict our search, accordingly, for the existence of substitutions P such that $P\alpha \leq \beta$. We of course only need to look at substitutions of the form $[x_1 := \rho_1, \ldots, x_n := \rho_n]$ where each x_i is a variable occurring in α, and by substitutivity, we may assume any atom occurring in some ρ_i ($i \leq n$) also occurs in β. Since there is a finite number of type expression trees which can by

built by $|\alpha|\,|\beta|^2$ nodes (maximum) using $\cap$, $\rightarrow$ and the atoms in β, there is a finite number of ρ_i candidates for the x_i's ($i \leq n$) such that $\alpha[x_1 \leftarrow \rho_1,\ldots,x_n \leftarrow \rho_n]$ could possibly be weaker than β. And since $\leq$ is decidable, one can simply generate solutions and test to see if the weaker relation holds.

As we have said earlier, we can decide if $\alpha << \beta$ by testing for the existence of P such that $P(\alpha_1 \cap \ldots \cap \alpha_m) \leq \beta$, where $m = NRP(\beta)$ (which is also easily computed). Hence, we have:

THEOREM 4.4.1. The below relation << is decidable.

proof By the previous two paragraphs. ◊

Since the computation of a principal type for an expression in XTCL is possible provided one can decide the << relation, it follows that type checking (and typeability) in XTCL is decidable.

THEOREM 4.4.2. The problems

"Given an explicitly typed expression e and a type $\tau \in$ Texp, does $e{:}\tau$ in XTCL?"

and "Given an explicitly typed expression e, does e have a type in XTCL?"

are decidable.

proof By theorem 4.3.1 and theorem 4.4.1. ◊

A final remark: The bounds placed on the sizes that a solution need not exceed are by no means optimal. With some work it can be shown that no solution for x making α weaker than β need be more than $|\alpha|$ in size. Our bound is easier to prove, and it is still a polynomial function of $|\alpha| + |\beta|$. This fact is used in the next section in which we show that although decidable, the question "Is $\alpha << \beta$?" is NP-complete.

4.5 Deciding << is NP-Complete

Although decidable, the problem of determining if there exists a substitution P which makes $P\alpha \leq \beta$ is difficult to compute. We show that it is an NP-complete problem.

Recall that P is the class of problems which can be solved by a deterministic Turing machine in polynomial time -- that is, in time f(n) where f(x) is a polynomial and n is the size of the input -- and NP is the class of problems which may be solved by a nondeterministic Turing machine in polynomial time. (Certainly $P \subseteq NP$, but it is not known at this time whether P = NP.) A problem p is NP-complete if it is in NP and if ($p \in P \Rightarrow P = NP$). A problem p' in NP can be shown to be NP-complete by translating an NP-complete problem p into p' using an algorithm which runs in polynomial time.

An example of an NP-complete problem is the satisfiability of a boolean expression in 3-conjunctive normal form (3CNF) [H&U79]. Recall that a boolean expression in 3CNF has the following syntax:

BE ::= Disj | Disj $\wedge$ BE

Disj ::= V $\vee$ V $\vee$ V

V ::= Bvar | Bvar '

Bvar ::= {an infinite supply of boolean variables}

The meanings of $\wedge$, $\vee$ and ' are AND (infix), OR (infix) and NOT (postfix). An example of a boolean expression in 3CNF is $(x \vee y \vee z) \wedge (x' \vee y' \vee z)$. A boolean expression w is *satisfiable* if we can replace each variable in w by a constant, either TRUE or FALSE, such that the resulting expression is TRUE when evaluated. (The example above is satisfiable,

since x = y = z = TRUE makes the expression true, but $(x \vee x \vee x) \wedge (x' \vee x' \vee x')$ is not satisfiable.)

The 3CNF satisfiability problem is known to be NP-complete. We show that deciding if $\alpha << \beta$, given α and β, is NP-complete. To do this, we first show that this problem is NP-hard by reducing the problem of 3CNF satisfiability to the problem of deciding <<. Then we show that the << problem is in NP by giving a nondeterministic algorithm which decides if $\alpha << \beta$ in polynomial time.

The reduction of the 3CNF problem is accomplished by showing that every instance of the 3CNF problem can be translated into an equivalent question "Does there exist a substitution P such that $P\alpha \leq \beta$" for appropriate α and β. Suppose we wish to determine the satisfiability of a boolean expression w in 3CNF having m variables $x_1, ..., x_m$ and n disjunction clauses $w = Z_1 \wedge ... \wedge Z_n$ where each $Z_i = z_{i1} \vee z_{i2} \vee z_{i3}$ and each z_{ij} is either x_k or x_k' for some $k \leq m$. Let $x_1, x_1', x_2, x_2', ..., x_m, x_m'$ be distinct type variables, and let t, f and a be distinct atoms (i.e., in Tvar $\cup$ Tcnst) and different from any x_k or x_k'. Notice that $t \leq x$ for some x iff $x = t$ or $x = t \cap t \cap ... \cap t$. WLOG, we can assume that solutions for x do not contain duplicate terms (since both solutions are equivalent with respect to $\leq$), hence, $t \leq x$ for some x iff $x = t$. Also note that since $t \neq f$, the inequalities $t \leq x$ and $f \leq x$ have no simultaneous solution for x. By the definition of $\leq$, for any x_j and x_j', the inequality $(x_j \rightarrow (x_j' \rightarrow a)) \cap (x_j' \rightarrow (x_j \rightarrow a)) \leq t \rightarrow (f \rightarrow a)$ has a solution for x_j and x_j' iff one of the following holds:

1) $t \leq x_j$ and $f \leq x_j'$, or
2) $f \leq x_j$ and $t \leq x_j'$, or
3) $t \leq x_j$ and $t \leq x_j'$ and $(x_j \rightarrow a) \cap (x_j' \rightarrow a) \leq f \rightarrow a$.

Since 3) is not satisfiable for any x_j and x_j', the only possible solutions the above inequality has (ignoring duplicate terms) are 1) $x_j = t$ and $x_j' = f$, or 2) $x_j = f$ and $x_j' = t$.

Let A_j ($j \leq m$) denote the type expression $(x_j \rightarrow (x_j' \rightarrow a)) \cap (x_j' \rightarrow (x_j \rightarrow a))$ and C denote $t \rightarrow (f \rightarrow a)$. Then by the previous paragraph, and the definition of $\leq$, the inequality

$$C \rightarrow a \leq (A_1 \rightarrow a) \cap (A_2 \rightarrow a) \cap \ldots \cap (A_m \rightarrow a) \qquad \text{(E0)}$$

has as solutions for $x_1, x_1', \ldots, x_m, x_m'$ all assignments of atoms in {t,f} to the variables such that for all $j \leq m$ either $x_j = t$ and $x_j' = f$ or $x_j = f$ and $x_j' = t$.

Now for each $Z_i = z_{i1} \vee z_{i2} \vee z_{i3}$ in w, construct the type $\rho_i = z_{i1} \cap z_{i2} \cap z_{i3}$ (remember each z_{ij} denotes some x_k or x_k'). Notice that the solutions to $\rho_i \leq t$ which assign atoms in {t,f} to the variables denoted by z_{i1}, z_{i2}, and z_{i3} are exactly those which assign "t" to at least one variable denoted by z_{i1}, z_{i2} or z_{i3}. Consider the following set of inequalities:

$$C \rightarrow a \leq (A_1 \rightarrow a) \cap (A_2 \rightarrow a) \cap \ldots \cap (A_m \rightarrow a) \qquad \text{(E0)}$$
$$\rho_1 \leq t \qquad \text{(E1)}$$
$$\rho_2 \leq t \qquad \text{(E2)}$$
$$\ldots \qquad \ldots$$
$$\rho_n \leq t \qquad \text{(En)}$$

In order for E0, ..., En to have a simultaneous solution for the variables x_1, x_1', ...etc., there must be an assignment of atoms in {t, f} to these variables such that

1) for each $j \leq m$, either $x_j = t$ and $x_j' = f$ or $x_j = f$ and $x_j' = t$, and

2) for each $i \leq n$, at least one of z_{i1}, z_{i2} or z_{i3} must denote a variable which is assigned "t".

Conversely, if 1) and 2) hold, then equations E0 through En are satisfied. Hence, E0, E1, ..., En have a simultaneous solution iff w is satisfiable. By the definition of $\leq$, E0, ..., En have a simultaneous solution for the x_j and x_j' variables iff there is a substitution P such that

$$P[\ (\ (A_1 \to a) \cap (A_2 \to a) \cap \ldots \cap (A_m \to a)\ ) \to (\rho_1 \to a) \to \ldots \to (\rho_{n-1} \to a) \to \rho_n\ ]$$

$$\leq \quad (C \to a) \to (t \to a) \to (t \to a) \to \ldots \to (t \to a) \to t$$

$$|\leftarrow \quad n - 1 \text{ repetitions} \quad \to|$$

Of course, this translation can be done for boolean expressions in CNF (rather than 3CNF): the only difference is that the ρi's may have more (or less) than 3 terms. We give two examples of the translation.

Example 1. Translate $w = (x_1 \vee x_2') \wedge (x_1' \vee x_2')$ into appropriate types α and β such that $\alpha << \beta$ iff w is satisfiable. Let x_1, x_2, x_1' and x_2' be distinct type variables. Using the formula above, we get

$$A_1 = (x_1 \to x_1' \to a) \cap (x_1' \to x_1 \to a)$$

$$A_2 = (x_2 \to x_2' \to a) \cap (x_2' \to x_2 \to a)$$

$$\rho_1 = x_1 \cap x_2'$$

$$\rho_2 = x_1' \cap x_2'$$

Hence,

$$\alpha = [\ (\ ((x_1 \to x_1' \to a) \cap (x_1' \to x_1 \to a)) \to a\ ) \cap (\ ((x_2 \to x_2' \to a) \cap (x_2' \to x_2 \to a)) \to a\ )\ ]$$

$$\to\ [\ (x_1 \cap x_2') \to a]\ \to\ (x_1' \cap x_2')$$

$\beta = [\,(t \to f \to a) \to a\,] \to [\,t \to a\,] \to t$

By the $\leq$ rules, $\alpha \leq \beta$ if and only if

1) $(t \to f \to a) \to a \leq ((x_1 \to x_1' \to a) \cap (x_1' \to x_1 \to a)) \to a$

2) $(t \to f \to a) \to a \leq ((x_2 \to x_2' \to a) \cap (x_2' \to x_2 \to a)) \to a$

3) $t \to a \leq (x_1 \cap x_2') \to a$

4) $(x_1' \cap x_2') \leq t$

which implies that

1') $(x_1 \to x_1' \to a) \cap (x_1' \to x_1 \to a) \leq t \to f \to a$

$\Leftrightarrow$ ($t \leq x_1$ and $f \leq x_1'$) or ($t \leq x_1'$ and $f \leq x_1$)

2') $(x_2 \to x_2' \to a) \cap (x_2' \to x_2 \to a) \leq t \to f \to a$

$\Leftrightarrow$ ($t \leq x_2$ and $f \leq x_2'$) or ($t \leq x_2'$ and $f \leq x_2$)

3') $x_1 \cap x_2' \leq t$

4') $x_1' \cap x_2' \leq t$

which has a solution x_1=t, x_1'=f, x_2'=f, x_2'=t (among others). Hence w has solution x_1=TRUE, x_2=FALSE.

Example 2. Translating $w = x_1 \wedge x_1'$ into appropriate α and β, we get that

$\alpha = [\,((x_1 \to x_1' \to a) \cap (x_1' \to x_1 \to a)) \to a] \to [\,x_1 \to a\,] \to x_1'$

$\beta = [\,(t \to f \to a) \to a\,] \to [\,t \to a\,] \to t$

and $\alpha \leq \beta$ means

1) $(t \leq x_1$ and $f \leq x_1')$ or $(t \leq x_1'$ and $f \leq x_1)$
2) $x_1 \leq t$
3) $x_1' \leq t$

which can never hold, for any x_1 and x_1'. Hence, w is not satisfiable.

We have shown that any instance w of the 3CNF satisfiability problem can be converted into an equivalent question of the existence of a P such that $P\alpha \leq \beta$, where α and β are derived from w as described above. Let α, β and w have some representations suitable for a Turing machine, and let $s(\alpha)$, $s(\beta)$ and s(w) be their sizes (i.e., the number of cells they occupy on the tape). The construction of α and β from w can be done on a 2-tape Turing machine in time directly proportional to s(w), which implies a time bound of $k\ s(w)^2$ (for appropriate k) on a single-tape Turing machine. Furthermore, $s(\alpha) + s(\beta) \leq c\ s(w)$ for fixed c, hence, if the problem of the existence of a P such that $P\alpha \leq \beta$ were solvable in time $f(s(\alpha) + s(\beta))$, where f is a (monotone) polynomial, then we could test the satisfiability of a w in 3CNF in time bounded by $f(c\ s(w)) + k\ s(w)^2$, a polynomial function of s(w). This shows that the existence of a P such that $P\alpha \leq \beta$ is NP-hard.

Since $NRP((C \rightarrow a) \rightarrow (t \rightarrow a) \rightarrow (t \rightarrow a) \rightarrow \ldots \rightarrow (t \rightarrow a) \rightarrow t) = 1$, by lemma 4.4.2 we have that

$$((A_1 \rightarrow a) \cap (A_2 \rightarrow a) \cap \ldots \cap (A_m \rightarrow a)) \rightarrow (\rho_1 \rightarrow a) \rightarrow \ldots \rightarrow (\rho_{n-1} \rightarrow a) \rightarrow \rho_n$$

$$<< \quad (C \rightarrow a) \rightarrow (t \rightarrow a) \rightarrow (t \rightarrow a) \rightarrow \ldots \rightarrow (t \rightarrow a) \rightarrow t$$

$$|\longleftarrow\ n - 1\ \longrightarrow|$$

if and only if there exists a substitution P such that

$$P\ \{\ (\ (A_1 \rightarrow a) \cap (A_2 \rightarrow a) \cap \ldots \cap (A_m \rightarrow a)\ )\ \rightarrow (\rho_1 \rightarrow a) \rightarrow \ldots \rightarrow (\rho_{n-1} \rightarrow a) \rightarrow \rho_n\ \}$$

$$\leq\quad (C \rightarrow a) \rightarrow (t \rightarrow a) \rightarrow (t \rightarrow a) \rightarrow \ldots \rightarrow (t \rightarrow a) \rightarrow t$$

$$|\longleftarrow\ n - 1\ \longrightarrow|$$

It follows that the problem of determining if $\alpha << \beta$, given α and β, is NP-hard as well.

THEOREM 4.5.1. The problems,

"Given types α and β, does there exist P such that $P\alpha \leq \beta$?" and

"Given types α and β, is $\alpha << \beta$?"

are NP-hard.

To show that the << problem is NP-complete, we must show that it is in NP. We take the direct approach of nondeterministically generating a possible solution for the variables in α which may satisfy $\alpha \leq \beta$, and simply checking the solution (deterministically). Our nondeterministic algorithm which generates candidates for solutions is called GT (for "generate type"). GT takes as parameters a list A of atoms and a number n and returns a type expression having no more than n nodes and whose atoms are on the list A. It is shown how a nondeterministic single-tape Turing machine can be constructed which decides "Given types α and β, is there a P such that $P\alpha \leq \beta$?" in polynomial time, and hence that << is decidable in nondeterministic polynomial time.

The nondeterminism in GT is completely controlled by the function "Pick" which, given a non negative integer d, nondeterministically returns some integer d' such that $0 \leq d' \leq d$.

Pick can easily be implemented as a nondeterministic Turing machine which runs in time and space proportional to log d (i.e., the length of a binary representation of d).

LEMMA 4.5.1 There is a nondeterministic Turing machine PICK which, given a binary representation of a positive integer n as input, always halts, and whose set of possible computed values for input n is $\{m \mid 0 \le m \le n\}$. Moreover, PICK runs in time proportional to log n.

proof Let PICK = $<Q, \Sigma, G, d, q_0, B, F>$ where

- $Q = \{q_0, q_1, q_2, q_3\}$ is the set of states
- $\Sigma = \{0, 1\}$ is the set of input symbols
- $G = \{0, 1, ' '\}$ is the set of tape symbols
- $B = ' '$ is the blank symbol
- q_0 is the initial state
- $F = \{q_3\}$ is the set of final states
- $d : ((Q \setminus \{q_3\}) \times \Sigma) \rightarrow 2^{(Q \times \Gamma \times \{L,R\})}$ is the nondeterministic state transition function

defined by the table

	0	**1**	**' '**
$\mathbf{q_0}$	$\{<q_0, 0, R>\}$	$\{<q_0, 1, R>, <q_1, 0, R>\}$	$\{<q_2,' ',L>\}$
$\mathbf{q_1}$	$\{<q_1, 0, R>, <q_1, 1, R>\}$	$\{<q_1, 0, R>, <q_1, 1, R>\}$	$\{<q_2,' ',L>\}$
$\mathbf{q_2}$	$\{<q_2, 0, L>\}$	$\{<q_2, 1, L>\}$	$\{<q_3,' ',R>\}$

PICK starts in state q_0 and stays in q_0, scanning right until it either changes a 1 to a 0 (making the number computed less than n, no matter how the remaining bits are changed) and switches to state q_1, or scans passed the number n (making the number computed equal to n) and switches to state q_2, which positions the head back to the leftmost bit of the result and halts in state 3. Once in state q_1, any of the lower-order bits of n may be changed; thus all $m \le n$ are

possible results. PICK takes a total of 2k + 2 moves, where k is the number of nonblank cells on the tape. Certainly k is of order $\log_2 n$. ◊

We now define the algorithm GT for generating types. The specification of GT uses the nondeterministic function Pick(n) computed by the nondeterministic Turing machine PICK.

ALGORITHM 4.5.1. Let A be a non-null list of atoms, let "head" and "tail" be list functions such that head($<a_1,a_2,...,a_p>$) = a_1 and tail($<a_1,a_2,...a_p>$) = $<a_2,...a_p>$, and let $n \geq 1$.

GT(A, n) = GenExact(A, OddCeiling(Pick(n-1)))

GenExact(A, z) =

if z = 1 then head(tail^k(A)), where k =Pick(Len(A)-1)

else T_1 op T_2

where T_1 = GenExact(A, m)

T_2 = GenExact(A,z-1-m)

m = OddCeiling(Pick(z-2))

op = (if Pick(1) = 0 then $\rightarrow$ else $\cap$)

OddCeiling(x) = if x odd then x else x+1

Len($<a_1,...,a_n>$) = n

LEMMA 4.5.2. Let ρ be a type expression having not more than n nodes, k of which are atoms, and such that each atom in ρ is on the list A. Then there is an execution sequence of GT(A, |ρ|) which returns ρ. Furthermore, ρ is computed using | ρ| calls to GenExact, k calls to Pick(Len(A)-1), |ρ|-k calls each to Pick(z - 2) (always for z<n) and Pick(1), and one call to Pick(n-1).

proof The result follows after showing by induction on z (z odd),

i) GenExact(A, z) may return any type expression ρ having z nodes and where each atom of z is on the list A,

ii) GenExact(A,z) is computed by a total of z calls to GenExact,

iii) Pick(Len(A)-1) is called in GenExact iff an atom is returned,

iv) Pick(z - 2) and Pick(1) are called in GenExact iff a non-atom is returned. $\Diamond$

It is also easy to see that if GT(A, n) returns a type ρ, then $|\rho| \leq n$ and each atom of ρ must be on the list A.

What about the time complexity of GT? It is safe to assume that each atom on the list A has a length of no more than log n, therefore head and tail run in log n time, and thus for k<n, $\text{head}(\text{tail}^k(A))$ runs in Len(A) (log n) time. Certainly each invocation of OddCeiling runs in time proportional to log n (OddCeiling(x) is nothing more than changing the low order bit of the binary representation of x to '1'), as does the numeric additions and subtractions. Pick(1) of course is of constant time, thus an upper bound on the time required to compute GT(A,n) is $T_{GT}(A,n) = c\ n\ \text{Len}(A) \log n$, where c is an appropriate constant. If, however, s(A) and s(n) are the sizes of the input representations of A and n, respectively (that is, $s(A) \leq \text{Len}(A) \log n$, $s(n) = \log n$), then the bound becomes $T_{GT}(A,n) = c\ 2^{s(n)}\ s(A)$.

Now suppose we are given two type expressions α and β and are asked if "$\alpha \leq \beta$" has a solution for variables of α. Let $s(\alpha)$ and $s(\beta)$ be the size of the input representations of α and β, respectively. The following 5 steps will certainly answer the question:

1) Compute the list A of atoms in β
2) Compute $n = |\alpha|\ |\beta|^2$
3) Compute the list V of variables in α

4) For every variable x on V

i) Compute $\rho = GT(A, n)$

ii) Substitute ρ for all occurrences of x in α

5) Test if $\alpha \leq \beta$, and return the result

Certainly each step is effective, hence this algorithm always terminates. By lemma 4.5.2 and results of the previous section, we have for any α and β,

Execution of steps 1 - 5 will never produce a TRUE result

iff There is no substitution P such that $P\alpha \leq b$.

If we use a single tape Turing machine to implement this algorithm (the TM can be deterministic except for the GT algorithm), it is easy to verify that for some constants c_1, c_2, ...,

i) Step 1 executes in time bounded by $c_1\ s(\beta)^2$ and computes A of size $s(A) \leq 2\ s(\beta)$

ii) Step 2 executes in time bounded by $c_2\ s(\alpha) + c_3\ s(\beta)$ {the multiplications do not contribute, since they are polynomial in *log* $(s(\alpha) + s(\beta))$ }, giving n of size $s(n) \leq \log (s(\alpha) + 2\ s(\beta))$

iii) Step 3 executes in time bounded by $c_4\ s(\alpha)^2$

iv) For each x on V, the GT(A,n) executes in time $T_{GT}(A,n) = c\ s(A)\ 2^{s(n)} \leq c\ (\ s(\alpha) + 2\ s(\beta))\ 2\ s(\beta)$, which is bounded by $c_5\ (s(\alpha) + s(\beta)^2)$ for appropriate constant c_5. GT(A,n) computes a type ρ of size $s(\alpha)\ s(\beta)^2$ which is inserted into the current α of size no more than $s(\alpha)^2\ s(\beta)^2$ every place an x occurs. We can assume a representation of types for which finding an x in a type γ can be done in linear time, and for which inserting a type ρ for x in γ can be done in $s(\rho)\ s(\gamma)$ time, no matter how many occurrences of x there are. Thus, for an appropriate c_6 we get that the insertion of ρ can be done in time $c_6\ s(\alpha)^3\ s(\beta)^4$ provided we

require α to be the rightmost datum on the tape (a reasonable requirement). Since $s(\alpha)$ bounds the number of x's in V, the time for the loop to execute is bounded by $c_7\ s(\alpha)^4\ s(\beta)^4$ for appropriate constant c_7, and the resulting α is no more than $s(\alpha)^2\ s(\beta)^2$ in size.

The asymptotic time complexity through step 4 is determined by the execution of the fourth step, i.e., $c_7\ s(\alpha)^4\ s(\beta)^4$ head moves. Note that the overhead involved in moving results from one step so they may be used in the next step need never be more than of fourth degree order (since the total of the sizes of the results is bounded quadratically), hence it is absorbed in the fourth degree asymptotic time complexity.

The weaker relation $\leq$ can be decided using the following (deterministic) algorithm:

ALGORITHM 4.5.3.

$W(\alpha,\beta) =$ if β=t, an atom, then if t a term of α then TRUE else FALSE
else if $\beta=\beta_1\rightarrow\beta_2$ then $W(\cap\{\tau \mid \sigma\rightarrow\tau$ a term of α, and $W(\beta_1,\sigma)\},\ \beta_2)$
else $W(\alpha,\ \beta_1)$ AND $W(\alpha,\ \beta_2)$ where $\beta = \beta_1\cap\beta_2$

It is easy to see that an upper bound for the number of calls to W in computing W(u,v) is |u| times |v|. (More precisely, it can be shown by induction on the number of calls to W that the total number of calls NC(u,v) to W in computing W(u,v) is less than (NI(u)+1) (NI(v)+1) + MIN(|u|,|v|), where $NI(\tau)$ is the total number of intersections in τ.) W can be implemented directly by a single-tape Turing machine which, essentially, uses the tape as a stack: The machine copies the arguments of any recursive call to the right of the current arguments and changes to the initial state. After a call to W is completed, the machine does a "return true" or "return false" which blanks out the arguments, scans left for a special symbol marking the

place where the returned value is to be placed, writes the value ("true" or "false"), scans back to another special symbol marking an encoding of the state which is to be entered, and enters that state. The size of the active portion of the stack is never more than the order of $s(u)\ s(v)\ (s(u) + s(v))$, and the scanning and copying required for the call and return linkage is therefore of the order $\{s(u)\ s(v)\ (s(u) + s(v))\}^2$. The operations of testing if an atom t is a term of some expression and extracting the terms from an expression can certainly be done in time bounded by $(s(u) + s(v))^2$, thus $c_8\ \{s(u)\ s(v)\ (s(u) + s(v))\}^2$ is a time bound for computing W(u,v).

From iv) above, W is called with arguments u, v of sizes $s(\alpha)2\ s(\beta)2$ and $s(\beta)$, hence it is computed in time bounded by $c_9\ s(\alpha)^8\ s(\beta)^{10}$ for suitable constant c_9. Since this function asymptotically bounds the execution time of step 4, it is a polynomial time bound for the entire algorithm.

We conclude that the problem, "Given α and β, is there a substitution P of types for variables in a such that $P\alpha \leq \beta$?," is in NP, and by theorem 4.5.1 it is NP-complete. Since $NRP(\beta) \leq s(\beta)$ is easily computed in deterministic polynomial time, we can use the 5-step algorithm to decide $\alpha << \beta$ by checking if $\alpha_1 \cap ... \alpha_{NRP(\beta)} \leq \beta$ is satisfiable -- in increased polynomial time -- where each α_i is a copy of α with fresh variables. Thus, we conclude that deciding $\alpha << \beta$ is NP-complete as well.

THEOREM 4.5.2 The problem "Given types α and β, is $\alpha << \beta$?" is NP-complete.

proof From the above discussion. ◊

4.6 A Type Checking Algorithm for XTCL

In this section we give a deterministic algorithm for deciding <<, and hence an algorithm for checking type claims in XTCL.

Our strategy to determine if $\alpha << \beta$, given α and β, uses the fact that $\sigma\rightarrow(\tau\cap\rho)$ and $(\sigma\rightarrow\tau)\cap(\sigma\rightarrow\rho)$ are equivalent with respect to $\leq$, hence we may assume that β is reduced, that is, that β has no $\cap$ appearing immediately to the right of any $\rightarrow$. Let $\beta = \beta_1\cap...\cap\beta_n$ be such that each β_i is a reduced term. It is easy to see that $\alpha << \beta$ iff $\alpha << \beta_1$ and $\alpha << \beta_2$ and ... and $\alpha << \beta_n$. Now $\alpha << \beta_k$ ($k\leq n$) iff there exists a substitution P such that $P(\alpha) \leq \beta_k$ (because $NRP(\beta_k) = 1$--see lemma 4.4.2), hence, for reduced $\beta = \beta_1\cap...\cap\beta_n$, $\alpha << \beta$ iff $\forall k\leq n\ \exists P$ such that $P\alpha \leq \beta_k$. Thus we reduce the problem to finding a solution for the variables of α which makes $\alpha \leq \beta$ where β is a reduced term (non-intersection). We can also assume that the variables in α are disjoint from those in β.

Notice that if $\alpha_1 \cap \alpha_2 \leq \beta$ and β is a reduced term, then either $\alpha_1 \leq \beta$ or $\alpha_2 \leq \beta$. Again, this is because $NRP(\beta) = 1$ and by lemma 4.4.1. This will be useful later, so we state it as a lemma:

LEMMA 4.6.1. Let α_1, α_2 and β be types where β is a reduced term. Then $\alpha_1\cap\alpha_2 \leq \beta$ iff $(\alpha_1\leq\beta)$ or $(\alpha_2\leq\beta)$.

proof Because $NRP(\beta) = 1$ and by lemma 4.4.1. ◊

Let β be a reduced term, let A be the set of atoms occurring in β, and let $fv = \{x_1, ..., x_n\}$ be the set of variables occurring in α. To determine if $\alpha << \beta$, we proceed by decomposing the proposition "$\alpha\leq\beta$" into a set of sets of propositions $\{Z_1, ..., Z_k\}$ such that $\alpha\leq\beta$ has a

solution (by substituting types for vars in fv) iff there is a solution (again, a type substitution for vars in fv) simultaneously satisfying the propositions in at least one set Z_j. That is, we transform the proposition "$\alpha \geq \beta$" into an equivalent disjunction of conjunctions (logically) of propositions. Ultimately, each conjunction will be a set of "simple" propositions "$u \leq v$" where either u or v is atomic (a variable or constant) and v is a term. The decomposition is done by recursively applying the $\leq$-rules to propositions which are not simple, using lemma 4.6.1 whenever possible. Each solvable conjunctive set C of propositions resulting from the decomposition has the property that "$u \leq v" \in C$ and u (resp. v) not atomic $\Rightarrow$ u (resp. v) contains no variable in fv (in other words, propositions like "$x \rightarrow \alpha \leq$ INTEGER" can not be true for any x, and their presence in C implies that C is unsatisfiable). To determine the solvability of a conjunctive set C, one can first test the truth of each proposition not containing variables in fv (for example, propositions like "INTEGER $\leq$ INTEGER" can be eliminated, and those like "$y \leq$ INTEGER $\rightarrow$ x", where $y \in A$, imply unsatisfiability of the conjunction). Then, having successfully eliminated from C propositions not containing variables in fv, for each $x \in$ fv we consider the solvability of subsets C_x of C where the propositions are of the form $x \leq a$ or $a \leq x$ (Cx is the set of propositions in C whose truth depends on x). There are 3 cases for any C_x:

1) The set of expressions σ below x is empty (i.e., $\{\sigma \mid "\sigma \leq x" \in C_x\} = \emptyset$),

2) the set of expressions above x is empty, or

3) neither sets of expressions above x or below x are empty.

The solvability of cases 1) and 3) is easily determined by $\leq$ computations; case 2) is solved by determining if a $\leq$ upper bound exists for the set of expressions below x in C_x.

Following this strategy, we give an algorithm for decomposing propositions. A proposition is defined syntactically as " Texp ≤ Texp ". (We use quotes to emphasize the syntactic treatment of propositions.) The decomposition algorithm DP ("Decompose Proposition") takes a proposition "$\alpha \leq \beta$" and a set of variables fv and returns a set of sets of propositions (henceforth called a "disjunction of conjunctions", or simply "disjunction"). DP calls DU ("Distribute Union"), a function which combines 2 disjunctions by pairing and uniting all combinations of conjunctions in the respective sets. $FV(\alpha)$ denotes the set of variables in α.

ALGORITHM 4.6.1

DP("$\alpha \leq \beta$", fv) =

if $\beta = \beta_1 \cap \beta_2$ then DU(DP("$\alpha \leq \beta_1$",fv), DP("$\alpha \leq \beta_2$",fv))

else if β an atom, $\beta \notin$ fv, and $\alpha = \alpha_1 \cap \alpha_2$

then DP("$\alpha_1 \leq \beta$",fv) $\cup$ DP("$\alpha_2 \leq \beta$",fv)

else if α or β an atom then { {"$\alpha \leq \beta$"} }

else // let $\beta = \beta_1 \rightarrow \beta_2$ //

if $\alpha = \alpha_1 \rightarrow \alpha_2$ then DU(DP("$\beta_1 \leq \alpha_1$",fv), DP("$\alpha_2 \leq \beta_2$",fv))

else // let $\alpha = \alpha_1 \cap \alpha_2$ //

if $FV(\beta) \cap fv = \emptyset$ and β reduced, then DP("$\alpha_1 \leq \beta$",fv) $\cup$ DP("$\alpha_2 \leq \beta$",fv)

else // there are no assignable variables in α //

$\cup$ {DU(DP("$\beta_1 \leq \sigma$",fv),DP("$\tau \leq \beta_2$",fv)) | $\sigma \rightarrow \tau$ = CombineArrows(B),

$B \subseteq$ ArrowTerms(α), $B \neq \emptyset$ }

DU(U,V) = { $u \cup v$ | $u \in U$, $v \in V$ }

ArrowTerms(α) = { $\sigma \rightarrow \tau$ | $\sigma \rightarrow \tau$ a term of α }

CombineArrows({ $\sigma_1 \rightarrow \tau_1$, ..., $\sigma_n \rightarrow \tau_n$ }) = $(\sigma_1 \cap ... \cap \sigma_n) \rightarrow (\tau_1 \cap ... \cap \tau_n)$

Given a proposition and a set of free variables, DP always terminates, yielding a disjunction. We now show that assuming $FV(\alpha)\cap FV(\beta)=\varnothing$, $P\alpha\leq\beta$ for some substitution P of types for variables in $\alpha \Leftrightarrow \exists C\in DP("\alpha\leq\beta",FV(\alpha))$ such that $\forall"\sigma\leq\tau"\in C\ P\sigma\leq P\tau$. To make the inductive proof work, we actually show something stronger.

Notation: For C a set of propositions and P a substitution, define SOLVES(P,C) to mean $P\sigma\leq P\tau$ for each $"\sigma\leq\tau"\in C$. Note that SOLVES(P,C_1) and SOLVES(P,C_2) $\Leftrightarrow$ SOLVES(P,$C_1\cup C_2$).

LEMMA 4.6.2. Let $FV(\alpha)\cap FV(\beta)=\varnothing$. Assume P is a substitution of types for variables of α, Q a substitution of types for variables of β. Let R be any set of variables not occurring in α or β. Then

i) $P\alpha\leq\beta \Leftrightarrow \exists C\in DP("\alpha\leq\beta",FV(\alpha)\cup R)$. SOLVES(P,C), and

ii) $\alpha\leq Q\beta \Leftrightarrow \exists C\in DP("\alpha\leq\beta",FV(\beta)\cup R)$. SOLVES(Q,C).

proof Let fva = $FV(\alpha)$, fvb = $FV(\beta)$. Induct on $|\alpha| + |\beta|$. If α and β are atomic, then $DP("\alpha\leq\beta",fva\cup R) = \{\ \{"\alpha\leq\beta"\}\ \}$ and i) follows. The same is true for ii), which disposes of the base case. Now take cases on α and β:

1) Suppose $\beta=\beta_1\cap\beta_2$.

i) $DP("\alpha\leq\beta_1\cap\beta_2",fva\cup R) = DU(DP("\alpha\leq\beta_1", fva\cup R), DP("\alpha\leq\beta_2",fva\cup R)) = \{\ u\cup v \mid u\in DP("\alpha\leq\beta_1",fva\cup R),\ v\in DP("\alpha\leq\beta_2",fva\cup R)\ \}$. By hypoth, $P\ \alpha\leq\beta_1$ and $P\alpha\leq\beta_2 \Leftrightarrow \exists C_1\in DP("\alpha\leq\beta_1",fva\cup R)\ \exists C_2\in DP("\alpha\leq\beta_2",fva\cup R)$ such that SOLVES(P,C_1) and SOLVES(P,C_2), i.e., such that SOLVES(P,$C_1\cup C_2$). Thus $P\alpha\leq\beta \Leftrightarrow \exists C \in \{\ u\cup v \mid u\in DP("\alpha\leq\beta_1",fva\cup R),\ v\in DP("\alpha\leq\beta_2",fva\cup R)\ \}$ such that SOLVES(P,C).

ii) is true by virtually the same argument.

2) If α or β is atomic, then if $\alpha=\alpha_1\cap\alpha_2$, $\beta\notin$ fv, then $\alpha\leq\beta$ satisfiable iff $\alpha_1 \leq \beta$ or $\alpha_2 \leq \beta$ is satisfiable, and the result follows as in case 1. Otherwise, DP("$\alpha\leq\beta$",?) = {{"$\alpha\leq\beta$"}}, and i) and ii) follow trivially.

3) Assume $\alpha=\alpha_1\rightarrow\alpha_2$ and $\beta=\beta_1\rightarrow\beta_2$.

i) $P\alpha \leq \beta \Leftrightarrow \beta_1\leq P\alpha_1$ and $P\alpha_2\leq\beta_2 \Leftrightarrow$ (by hypothesis) ($\exists C_1\in$ DP("$\beta_1\leq\alpha_1$", FV(α1) $\cup$ (fva\FV(α_1)) $\cup$ R) such that SOLVES(P,C_1)) and ($\exists C_2\in$ DP("$\alpha_2\leq\beta_2$", FV(α_2) $\cup$ (fva\FV(α_2)) $\cup$R) .such that SOLVES(P,C_2)) $\Leftrightarrow$ (as in case 1) $\exists$ C $\in$ { u $\cup$ v | u $\in$ DP("$\beta_1\leq\alpha_1$",fva$\cup$R), v $\in$ DP("$\alpha_2\leq\beta_2$",fva$\cup$R) } . SOLVES(P,C).

ii) Same as i)

4) Assume $\beta=\beta_1\rightarrow\beta_2$ and $\alpha=\alpha_1\cap\alpha_2$.

i) Here, β is reduced (by original assumption) and contains no assignable variables (i.e., variables in fva), hence $\alpha\leq\beta$ has a solution exactly when $\alpha_1 \leq \beta$ or $\alpha_2 \leq \beta$ has a solution (by lemma 4.6.1). Now DP("$\alpha_1\cap\alpha_2\leq\beta$",fva$\cup$R) = DP("$\alpha_1\leq\beta$", FV($\alpha_1$) $\cup$ (fva \ FV(α_1)) $\cup$ R) $\cup$ DP("$\alpha_2\leq\beta$", FV(α_2) $\cup$ (fva \ FV(α_2)) $\cup$ R) and by hypothesis, $P\alpha\leq\beta \Leftrightarrow$ ($P\alpha_1\leq\beta$ or $P\alpha_2\leq\beta$) $\Leftrightarrow$ (($\exists C\in$ DP("$\alpha_1\leq\beta$", fva $\cup$ R) . SOLVES(P,C)) or ($\exists C\in$ DP("$\alpha_2\leq\beta$",fva $\cup$ R) such that SOLVES(P,C))) $\Leftrightarrow$ $\exists$C $\in$ DP("$\alpha_1\leq\beta$", fva $\cup$ R) $\cup$ DP("$\alpha_2\leq\beta$", fva $\cup$ R) such that SOLVES(P,C).

ii) From the definition of $\leq$, $\alpha \leq Q\beta = Q\beta_1\rightarrow Q\beta_2$ iff $\exists$ a nonempty subset B of Arrowterms(α) such that CombineArrows(B) $\leq Q\beta_1\rightarrow Q\beta_2$. If Arrowterms($\alpha$=$\emptyset$ then no Q satisfies $\alpha \leq \beta$, and DP("$\alpha\leq\beta$",?) = $\emptyset$ (this shows $\Leftarrow$). Now suppose B is a subset of Arrowterms(α) such that CombineArrows(B) $\leq Q\beta_1\rightarrow Q\beta_2$. Let $\sigma\rightarrow\tau$ = CombineArrows(B). Note that $|\sigma|$ and $|\tau|$ are less than |a|, thus, by hypothesis, $\exists C_1\in$ DP("$\beta_1\leq\sigma$", fvb $\cup$ R) such that SOLVES(Q,C_1), and $\exists C_2\in$ DP("$\tau\leq\beta_2$",fvb $\cup$ R) such that SOLVES(Q,C_2) $\Rightarrow$ $\exists$ $C_1\cup C_2\in$ DU(DP("$\beta_1\leq\sigma$", fvb $\cup$ R), DP("$\tau\leq\beta_2$", fvb $\cup$ R)) such that SOLVES(Q, $C_1\cup C_2$). $\Diamond$

It is easy to see that any proposition in any conjunction of DP("$\alpha \leq \beta$", FV(α)) must be of the form $u \leq v$ where v is a term and either u or v is an atom. It is also easy to see that if a conjunction contains a proposition containing a non-atomic expression in which appear one or more assignable variables (e.g., "INT $\leq x \rightarrow y$" or "BOOL$\rightarrow x \leq$ INT"), then that conjunction has no solution. Of course, those propositions containing no assignable variables (e.g., "INT $\leq$ INT", "g $\leq$ BOOL$\rightarrow$BOOL" where g not in FV(α)) can be checked by the algorithm W, and if they are true, eliminated from the conjunction, otherwise the conjunction has no solution and can be eliminated from the disjunction. Thus, DP("$\alpha \leq \beta$", FV(α)) can be reduced to a set of conjunctions C such that each proposition in C is of the form $\sigma \leq x$ or $x \leq \sigma$, where x is an assignable variable (i.e., in FV(α)) and σ contains no assignable variables. For assignable variable x, let C_x denote the set of propositions in C containing x, hence $\{C_x \mid x \in FV(a)$, x appears in C$\}$ is a partition of C, and C is solvable iff each C_x is (independently) solvable. For a given C_x, there are three situations that could arise:

1) There is no "$\sigma \leq x$" in C_x. Then C_x is solvable by $x = \cap\{\tau \mid "x \leq \tau" \in C_x\}$.

2) There is at least one "$\sigma \leq x$" and at least one "$x \leq \tau$" in C_x. By transitivity of $\leq$, C_x is solvable iff $\forall \sigma$ such that "$\sigma \leq x$" $\in C_x$, $\sigma \leq \cap\{\tau \mid "x \leq \tau" \in C_x\}$. This is computable by W.

3) There is no "$x \leq \tau$" in C_x. Then C_x is solvable iff $\exists\, \rho$ such that $\forall \sigma$ such that "$\sigma \leq x$" $\in C_x$, $\sigma \leq \rho$.

This analysis tells us that the solvability of a set of decomposed propositions, given a set of assignable variables, can be determined provided we can solve the following problem:

"Given type expressions $\tau_1, \tau_2, \ldots, \tau_n$, is there a type expression ρ such that $\tau_i \leq \rho$ for all $i \leq n$?"

We now how that this problem has a solution which runs in expected time N log N, where N $= |\tau_1| + ... + |\tau_n|$. At this time it is useful to define *upper bound* with respect to $\leq$.

DEFINITION 4.6.1. Two type expressions σ and τ have a $\leq$-upper bound iff $\exists\rho$ such that $\sigma\leq\rho$ and $\tau\leq\rho$. A type ρ is a least $\leq$-upper bound of σ and τ iff $\rho\leq\rho'$ for all $\leq$-upper bounds ρ' of σ and τ.

Note that least $\leq$-upper bounds for σ and τ are not unique ($\leq$ is a preorder), but they must be unique modulo $\leq$ and $\geq$. This implies that if a $\leq$-upper bound exists for σ and τ, then a least $\leq$-upper bound exists. (Of course, a greatest $\leq$-lower bound for σ and τ is just $\sigma\cap\tau$.)

Consider the problem of determining if σ and τ have an upper bound (UB). First, if σ and τ have a UB, then they have a UB that is a non-intersection. Thus, if $\sigma=\sigma_1\cap\sigma_2$ and σ and τ have a UB, then either σ_1 and τ have a UB or σ_2 and τ have a UB. Next, note that 2 non-intersection types σ and τ have a UB only if they are atomic and equal, or if $\sigma=\sigma_1\rightarrow\sigma_2$, $\tau=\tau_1\rightarrow\tau_2$, and σ_2 and τ_2 have a UB. This suggests the following algorithm for testing the existence of a $\leq$-upper bound.

ALGORITHM 4.6.2.

$$
\begin{aligned}
EUB(\sigma,\tau) \;=\; & \text{if } \sigma=\sigma_1\cap\sigma_2 \text{ then } EUB(\sigma_1,\tau) \text{ OR } EUB(\sigma_2,\tau)\\
& \text{else if } \tau=\tau_1\cap\tau_2 \text{ then } EUB(\sigma,\tau_1) \text{ OR } EUB(\sigma,\tau_2)\\
& \text{else if } \sigma=\sigma_1\rightarrow\sigma_2 \text{ and } \tau=\tau_1\rightarrow\tau_2 \text{ then } EUB(\sigma_2,\tau_2)\\
& \text{else if } \sigma \text{ atomic and } \tau=\sigma \text{ then TRUE}\\
& \text{else FALSE}
\end{aligned}
$$

LEMMA 4.6.3 σ and τ have a UB $\Leftrightarrow$ EUB(σ,τ) returns TRUE.

proof By induction on the number of calls to EUB. $\Diamond$

We would like to generalize algorithm 4.6.2 to handle an arbitrary set of types A = $\{a_1,...,a_n\}$. (Note that computing the AND of ESUB(a_i,a_j) over all i,j doesn't work.) First, notice that if $\rho=\rho_1\rightarrow\rho_2\rightarrow...\rightarrow\rho_k\rightarrow(\tau_1\cap\tau_2)$ is a bound for any subset of A, then so is $\rho_1\rightarrow...\rightarrow\rho_k\rightarrow\tau_1$, therefore we need only look for bounds of the form $\rho_1\rightarrow...\rightarrow\rho_k\rightarrow z$ where z is atomic ($\rightarrow$ associates to the right, so z need not be interpreted as an $\rightarrow$ expression). Types in this form we shall call *right reduced*, the rightmost atom z we call the *end* and the number k of $\rightarrow$'s to the left of z the *length*. An inductive argument shows that any right reduced bound of an $a_i = \sigma_1\rightarrow...\rightarrow\sigma_t\rightarrow(\gamma_1\cap\gamma_2)$ must be of the form $\rho_1\rightarrow...\rightarrow\rho_t\rightarrow\eta$ where η is right reduced and a bound for either γ_1 or γ_2. If γ_1 is atomic, then its (non-intersection) bound must be atomic, thus a right reduced bound for ai may be of the form $\rho_1\rightarrow...\rightarrow\rho_t\rightarrow\gamma_1$. By iteration, we get a finite set R_i of types for a_i which describes the possible right reduced bounds for ai in a way such that for any right reduced bound ρ of a_i, there is a type $\sigma\in R_i$ having the same length and end as ρ. It follows that if ρ bounds each a_i, then there must be types $\tau_1\in R_i,...,\tau_n\in R_n$ all having the same length and end as ρ. Conversely, if there are elements $\tau_i=\xi_{i1}\rightarrow...\rightarrow\xi_{ik}\rightarrow z \in R_i$ all of the same length k and end z, then $v_1\rightarrow...\rightarrow v_k\rightarrow z$ bounds A, where $v_i = \cap\{\ \xi_{ij} \mid j\ \}$. Therefore, a simple algorithm for testing the existence of a bound for A is to compute for each $a_i\in A$ a set L_i of (length,end) pairs, one for each element in R_i, and simply test if $\cap\{\ L_i \mid i\leq n\ \} = \varnothing$.

ALGORITHM 4.6.3 (A = $\{a_1,...a_n\}$ is a set of types)

EUB'(A) = if $\cap\{$ LEP(0,a) $\mid a\in A\ \} \neq \varnothing$ then TRUE else FALSE

LEP(m,σ) = if s atomic then {<m,σ>}

else if $\sigma=\sigma_1\rightarrow\sigma_2$ then LEP(m+1,σ_2)

else if $\sigma=\sigma_1\cap\sigma_2$ then $LEP(m,\sigma_1) \cup LEP(m,\sigma_2)$

LEP stands for Length-End Pairs and takes an initial length argument in addition to a type argument. Assuming set union is done in constant time, LEP runs in time proportional to $|\sigma|$, thus it takes time proportional to $|a_1| + ... + |a_n|$ to form { LEP(0,a) | $a\in A$ }. If we use a list representation for each $LEP(0,a_i)$, then to test for empty intersection, we could unite all sets of pairs, sort them (using an NlogN expected time sort) and check for consecutive repeats. This will not work if there are non-unique pairs in an $LEP(0,a_i)$. To fix this, for each $LEP(0,a_i)$ we merely add "i" to each pair, yielding a set of triples T = {<len,end,i> | <len,end>$\in$ $LEP(0,a_i)$, $i\le n$} which can now be sorted, giving SORTED(T). Testing for consecutive entries in SORTED(T) is a linear process in |T| which is bounded by $|a_1| + ... + |a_n|$, thus the expected execution time of EUB'(A) is BigO(NlogN) where $N=|a_1|+...+|a_n|$.

Least upper bounds can also be computed.

ALGORITHM 4.6.4 (Assume that σ and τ are reduced.)

LUB(σ,τ) = if $\sigma=\sigma_1\cap\sigma_2$ then

if EUB(σ_1,τ) and EUB(σ_2,τ) then LUB(σ_1,τ)$\cap$LUB(σ_2,τ)

if EUB(σ_1,τ) and ~EUB(σ_2,τ) then LUB(σ_1,τ)

if ~EUB(σ_1,τ) and EUB(σ_2,τ) then LUB(σ_2,τ)

else "no LUB exists"

else if $\tau=\tau_1\cap\tau_2$ then LUB(τ,σ)

else if $\sigma=\sigma_1\rightarrow\sigma_2$ and $\tau=\tau_1\rightarrow\tau_2$ then $(\sigma_1\cap\tau_1)\rightarrow$LUB($\sigma_2$,$\tau_2$)

else if σ atomic and $\tau=\sigma$ then σ

else "no LUB exists"

(A simpler algorithm for LUB results if we assume a type constant TOP such that $\tau \leq TOP$ for all types τ.)

It can be shown that LUB returns "no LUB exists" iff EUB returns FALSE, and that ρ a UB for σ and $\tau \Leftrightarrow LUB(\sigma,\tau) \leq \rho$. Note that LUB can also be extended to finite sets $A = \{a_1,...,a_n\}$:

$LUB'(\{a_1,...,a_n\})$ = if $EUB'(\{a_1,...,a_n\})$ exists
then $LUB'(a_1,LUB'(a_2,...,LUB'(a_{n-1},a_n)...)$
else "no LUB exists"

We now give the algorithm for determining if a solution exists for a disjunction of conjunctions. ESD (Exists Solution for Disjunction) takes a set of sets of propositions D and a set of variables fv, ESC (Exists Solution for Conjunction) takes a set of propositions C and a set of variables fv, and ES takes a set of propositions, all containing some $x \in fv$. ESD, ESC and ES all return boolean values.

ALGORITHM 4.6.5

ESD(D,fv) = if $\exists C \in D$ such that ESC(C,fv) then TRUE else FALSE

ESC(C,fv) =
if there is no $x \in fv$ appearing in a non-atomic expression in C
then if W(a,b) for all "$a \leq b$" $\in$ {"$a \leq b$" | "$a \leq b$" $\in$ C, $FV(a \rightarrow b) \cap fv = \emptyset$}
then if $ES(C_x,x)$ for all $x \in fv$,
where $C_x = \{p \in C \mid x \text{ appears in } p\}$

then TRUE

else FALSE

else FALSE

else FALSE

ES(C,x) = if ~∃"a≤x"∈C then TRUE

else if ~∃"x≤a"∈C then

if EUB'({a | "a≤x"∈C}) exists

then TRUE

else FALSE

else if W(a, ∩{b | "x≤b"∈C}) for all a∈ {a | "a≤x"∈C}

then TRUE

else FALSE

LEMMA 4.6.4. $\exists P \,.\, P\alpha \leq \beta \Leftrightarrow$ ESD(DP("$\alpha \leq \beta$",FV(α)), FV(α)) returns TRUE.

proof This follows from lemma 4.6.2. ◊

It is interesting to note that for each satisfiable conjunction in DP("$\alpha \leq \beta$", FV(α)), the solutions for the variables in FV(α) are completely independent of each other. That is, C_x and C_y ($x \neq y$) are independent sets of propositions. Furthermore, the set of solutions for x satisfying any C_x can be expressed as either 1) the expressions beneath a particular type (in the $\leq$ ordering), 2) the expressions above a particular type, or 3) the expressions between 2 particular types. Using '*' and '-*' to mean having no $\leq$ bound above and no $\leq$ bound below, respectively (i.e., they are artificial constants denoting the top and bottom of the lattice of ideals), the solutions for any set of primitive propositions C_x involving only x can be expressed as an interval $[\tau_1, \tau_2]$ where ρ is a solution for x satisfying C_x iff $\tau_1 \leq \rho \leq \tau_2$.

Using the LUB algorithm, the bounds τ_1 and τ_2 for the interval can easily be computed, hence ESD could be modified to return a complete solution (i.e., a set of sets of variable-interval pairs).

Now we can construct an algorithm to decide $\sigma << \tau$, which can be used to compute PT. A simple algorithm for checking a type claim $e{:}\tau$ in XTCL is to test if $PT(e) << \tau$. Below, TC (for "Type Check") takes an explicitly typed expression and a claimed type expression, TCR ("Type Check Reduced") takes an explicitly typed expression and a reduced type expression, and Reduce takes a type expression and returns an equivalent reduced type expression (see the definition of "d" in section 3.1).

ALGORITHM 4.6.6

$TC(e,\tau)$ = $TCR(e, Reduce(\tau))$

$TCR(e,\tau)$ = if $PT(e)\neq$error

then if $\tau=\tau_1\cap\tau_2$ then $TCR(e,\tau_1)$ AND $TCR(e,\tau_2)$

else ESD(DP("$\sigma\leq\tau$", $FV(\sigma)$), $FV(\sigma)$) where $\sigma = PT(e)$

else FALSE

THEOREM 4.6.1. $TC(e,\tau) \Leftrightarrow e{:}\tau$ in XTCL

proof From lemma 4.6.4 and theorem 4.3.1. ◊

4.7 Generalizations

XTCL is based on the combinators S and K and their axiomatic types, but clearly any set of combinators and their types could be used with virtually no change to the types or type rules.

In fact, the ≤-rules may even be changed to accommodate new type forming operators (such as × or +) or to allow for subtype relations among type constants.

For example, the ≤-rules for a language with pairs may have the extra rule

$$\alpha \le \beta_1 \times \beta_2 \text{ if } \cap\{\sigma \mid \sigma \times \tau \text{ a term of } \alpha\} \le \beta_1 \text{ and } \cap\{\tau \mid \sigma \times \tau \text{ a term of } \alpha\} \le \beta_2$$

The necessary modifications to the algorithms to accommodate such features are for the most part obvious, and have no real effect on the computing time. We must, however, modify our definition of "reduced": A reduced expression with types of pairs has no intersection immediately to the right or left of a ×, as well as none to the right of an →. The algorithm DP can then be extended:

DP("$\alpha \le \beta$", fv) =

if $\beta = \beta_1 \cap \beta_2$ then DU(DP("$\alpha \le \beta_1$",fv), DP("$\alpha \le \beta_2$",fv))

else if β an atom, $\beta \notin$ fv, and $\alpha = \alpha_1 \cap \alpha_2$

then DP("$\alpha_1 \le \beta$",fv) ∪ DP("$\alpha_2 \le \beta$",fv)

else if α or β an atom then { {"$\alpha \le \beta$"} }

else if $\beta = \beta_1 \times \beta_2$

if $\alpha = \alpha_1 \times \alpha_2$ then DU(DP("$\alpha_1 \le \beta_1$",fv), DP("$\alpha_2 \le \beta_2$",fv))

else // let $\alpha = \alpha_1 \cap \alpha_2$ //

if FV(β)∩fv=Ø and β reduced, then DP("$\alpha_1 \le \beta$",fv) ∪ DP("$\alpha_2 \le \beta$",fv)

else // there are no assignable variables in α //

// Let B = CrossTerms(a) //

if B = Ø then Ø

else DU(DP("$\sigma \le \beta_1$",fv),DP("$\tau \le \beta_2$",fv))

where $\sigma \times \tau$ = CombineCross(B)

else // let $\beta = \beta_1 \to \beta_2$ //

if $\alpha = \alpha_1 \to \alpha_2$ then DU(DP("$\beta_1 \le \alpha_1$",fv), DP("$\alpha_2 \le \beta_2$",fv))

else // let $\alpha = \alpha_1 \cap \alpha_2$ //

if $FV(\beta) \cap fv = \emptyset$ and β reduced, then DP("$\alpha_1 \le \beta$",fv) $\cup$ DP("$\alpha_2 \le \beta$",fv)

else // there are no assignable variables in α //

$\cup$ {DU(DP("$\beta_1 \le \sigma$",fv),DP("$\tau \le \beta_2$",fv)) | $\sigma \to \tau$ = CombineArrows(B),

$B \subseteq$ ArrowTerms(α), $B \neq \emptyset$ }

DU(U,V) = $\{u \cup v \mid u \in U, v \in V\}$

ArrowTerms(α) = $\{\sigma \to \tau \mid \sigma \to \tau$ a term of $\alpha\}$

CombineArrows($\{\sigma_1 \to \tau_1, ..., \sigma_n \to \tau_n\}$) = $(\sigma_1 \cap ... \cap \sigma_n) \to (\tau_1 \cap ... \cap \tau_n)$

CrossTerms(α) = $\{\sigma \times \tau \mid \sigma \times \tau$ a term of $\alpha\}$

CombineCross($\{\sigma_1 \times \tau_1, ..., \sigma_n \times \tau_n\}$) = $(\sigma_1 \cap ... \cap \sigma_n) \times (\tau_1 \cap ... \cap \tau_n)$

Noticing that $\alpha \to \beta$ and $\sigma \times \tau$ have no common upper bound (w.r.t. $\le$), LUB and EUB' can be modified easily to accommodate $\times$. Naturally, we modify PT so that $PT(\sigma \times \tau) = PT(\sigma) \times PT(\tau)$, thus TC stands without modification.

In our formulation of TCL, we have assumed that type constants are unrelated with respect to $\le$, but subtype relations such as ZERO $\le$ INTEGER can be added directly into the $\le$ rules. Adding subtype relations among the constants only entails a minor modification to W and EUB'. In general, assume that there is a decidable relation $WA(t_1,t_2)$ among atoms which is reflexive, transitive and substitution invariant, and where $WA(x, y) \Leftrightarrow x = y$ (for x and y variables). Also, let EUBA be an algorithm which decides whether a set of atoms possesses a WA-upper bound. If there is a finite number of constants (which will usually be the case),

the computation of WA and EUBA is easily done. The generalized algorithms for W and EUB' then become

$W(\alpha,\beta)$ = if β an atom, then if $\exists$t an atomic term of α such that WA(t,β)
then TRUE
else FALSE
else if $\beta=\beta_1\rightarrow\beta_2$ then $W(\cap\{\tau \mid \sigma\rightarrow\tau$ a term of α, and $W(\beta_1,\sigma)\},\beta_2)$
else $W(\alpha, \beta_1)$ AND $W(\alpha, \beta_2)$ where $\beta = \beta_1\cap\beta_2$

EUB'(A) = if $\exists$n such that EUBA($\{z \mid <n,z> \in LEP(0,\alpha), \alpha\in A\}$)
then TRUE
else FALSE

LEP(m,σ) = if σ atomic then $\{<m,\sigma>\}$
else if $\sigma=\sigma_1\rightarrow\sigma_2$ then $LEP(m+1,\sigma_2)$
else if$\sigma=\sigma_1\cap\sigma_2$ then $LEP(m,\sigma_1) \cup LEP(m,\sigma_2)$

The rest of the type checking algorithm remains the same.

5

TCL with Type Fixedpoints

5.1 Why Type Fixedpoints?

Typeability of an expression in TCL implies its strong normalizability. While the set of combinators with strongly normalizable lambda calculus counterparts includes many powerful and useful functions, it does not include the least fixedpoint function Y (i.e., $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ in the lambda calculus) which is necessary for recursion in a purely functional language. Since a functional language which does not allow recursion is of little practical use, this problem must be solved. There are at least two solutions:

1) Add the Y combinator as a primitive with axiomatic type $(a \rightarrow a) \rightarrow a$ [Wand87], or, equivalently, change the syntax of the computational expressions to allow for recursive function definitions (such as adding the "letrec" construct found in ML [Mil78] or the μ operator found in [Cop80]).

2) Expand the typing system to allow an S-K factorization of Y to be typed.

The first approach is certainly the easiest, but induces nonuniform typeability among equivalent computational expressions. More importantly, with this approach, it is not clear

which lambda calculus expressions can be factored into typeable S-K-Y combinations, or even what the factoring algorithm should be.

Choosing 2), there are several ways to expand the typing system so that Y and other non-strongly normalizable expressions can be typed. In a version of the conjunctive system [Cop80a], a universal type TOP exists which is the type of all expressions. In this system, the set of expressions having a type other than TOP is precisely the set of solvable lambda expressions (a closed lambda expression e is *solvable* iff $\exists f_1,f_2,...,f_n$ such that $e\ f_1\ f_2\ ...\ fn$ converts to an expression in normal form) of which Y is a member. The problem, however, is that Y then does not have the desired type $(a\rightarrow a)\rightarrow a$, but has the type $(TOP\rightarrow a)\rightarrow a$ instead.

In [MS82,MPS84] it is shown how type fixedpoints can be used to properly type the Y combinator. A type fixedpoint is a type τ satisfying t in an equation $t = \sigma$ where t appears properly in σ, and is denoted by $\mu t.\sigma$. Using the type $\mu t.t\rightarrow s$, the type $(a\rightarrow a)\rightarrow a$ is derivable for Y. The semantics for these types was given in [MPS84] by imposing a metric on the space of ideals and showing that unique fixed points exist for compositions of type formation functions (such as $\rightarrow$ and $\cap$) which yield contractive functions (a function is *contractive* if the distance between two images of any points x and y is a constant proper fraction of the distances between x and y).

Aside from allowing the Y combinator to be typed, a system with type fixedpoints allows many useful data structures, such as infinite lists, to be typed as well.

In this chapter, we first show that the metric used in [MPS84] can be constructed for the space of ideals built from any underlying domain D provided a function $r:D^*\rightarrow N$ exists and

has a certain property. Under this metric, the set of ideals becomes a complete metric space with a unique limit for each Cauchy sequence. We then define a language TCLμ which is TCL expanded to handle certain recursively-defined type expressions. A semantics Mμ for TCLμ is given such that M and Mμ agree on the non recursively-defined type expressions and such that the recursively-defined expressions correspond semantically to limits of Cauchy sequences in the space of ideals. Semantic soundness is shown for TCLμ. Explicit types are added, yielding XTCLμ, and a type-checking algorithm is given.

5.2 The Metric Space Construction of MacQueen, Plotkin and Sethi

Most of the results in this section are taken from or are strongly motivated by [MPS84]. Our presentation, however, does not require that a specific domain D be used, and we do not require the use of the Banach Fixed Point Theorem.

Note that the lattice of ideals $\langle T,\subseteq\rangle$ is isomorphic to the lattice $\langle T',\subseteq\rangle$ where $T'=\{I\cap D^\circ \mid I\in T\}$, because every ideal is closed under upward limits. (This is also shown in [Win86].) Henceforth, we will assume that ideals are simply downward-closed sets of finite elements of D.

DEFINITION 5.2.1. Let $r:D^\circ\rightarrow N$ be a function mapping finite elements of D to the natural numbers. Then r is a *rank function* of D if $r(\perp) = 0$ and for all functions $f\in D^\circ$ and for all $x\in D^\circ$, $x\in dom(f) \Rightarrow r(f) > r(x)$ and $r(f) > r(fx)$. (Recall $x\in dom(f)$ means $f(x)\neq\perp$).

[MSP84] gives a rank function for a domain of functions, pairs and disjoint sums. As an example, we will construct a rank function for the finite elements of Pω, used in [Scott76, Stoy77] as a domain closed under function space construction.

Recall that Pω is the complete lattice of all subsets of the natural numbers ordered under set inclusion with finite basis elements the finite sets of natural numbers. Functions of the lambda calculus are encodings of sets of pairs $\langle n,m \rangle$ where n is the encoding of a finite element of Pω, m is a natural number. f(x) is defined as the set of integers $\{m \mid p\langle n,m \rangle \in f, b^{-1}(n) \subseteq x\}$, where p is the pair encoding function and b is the encoding of the finite elements. (Lambda calculus expressions semantically correspond to continuous monotonic functions, but this fact has no bearing on the construction of the rank function.) We choose $p\langle n,m \rangle = 2^n 3^m$ and $b\{n_1,...,n_k\} = \sum_{i \leq k} 2^{n_i}$ as our encoding functions.

Continuing, we say that an integer k *has interpretation* K if K=k or if $K=\langle K_1,K_2 \rangle$, $p\langle n,m \rangle=k$, m has interpretation K_2, and $K_1 = \{H \mid l \text{ has interpretation } H, l \in R\}$ where bR=n. For example, using p and b defined above, the number 18 has three interpretations: itself, $\langle \{0\},2 \rangle$ and $\langle \{0\},\langle \{0\},0 \rangle \rangle$. In fact, it can be shown that for our encoding functions there is a finite number of interpretations for any integer k (in general, this is satisfied if we use encodings for pairs and sets which are strictly greater than any component). For encoding functions with this property, we can define a function NI from natural numbers to natural numbers so that NI(k) is the number of interpretations for k. Now for any finite set v of integers, define r(v) = MAX $\{NI(k) \mid k \in v\}$, where MAX $\{\} = 0$. It is easy to verify that r is a rank function for Pω.

Following [MPS84], any rank function for an underlying domain D can be used to define a metric on the ideals T' of D° such that T' is a complete metric space. For ideals A, B where

$A \neq B$, define the "closeness" $C(A,B) = MIN \{ r(v) \mid v \in A \Delta B \}$ where $A \Delta B$ denotes the symmetric difference of A and B, $(A \cup B) \setminus (A \cap B)$, and define the distance function d to be $d(A,B) = 2^{-C(A,B)}$ if $A \neq B$, otherwise $d(A,B) = 0$. One can verify that d is a metric:

i) $d(A,B) = d(B,A) \geq 0$

ii) $d(A,B) = 0 \Leftrightarrow A = B$

iii) $d(A,C) \leq d(A,B) + d(B,C)$

In fact, by a property of symmetric difference, iii) can be strengthened to

iii)' $d(A,C) \leq MAX(d(A,B), d(B,C))$.

d induces a topology on T' by defining for each point x and real number z a set of open neighborhoods $X_z = \{y \mid d(x,y) < z\}$, thus T' can be interpreted as a (topological) metric space $\langle T',d \rangle$.

Recall that a sequence $A_1, A_2, \ldots$ is *Cauchy* if $\forall \varepsilon > 0 \; \exists k$ such that $\forall i,j > k$, $d(A_i,A_j) < \varepsilon$. Provided we interpret C(A,A) as indefinitely large, an equivalent definition of a Cauchy sequence $A_1, A_2, \ldots$, for our metric d, is $\forall M \; \exists k$ such that $\forall i,j > k \; C(A_i,A_j) > M$. This definition is the one we will use most often.

It has been shown that every Cauchy sequence in this metric space converges to a unique limit. Recall that L is a limit of the sequence $A_1, A_2, \ldots$ iff $\forall \varepsilon > 0 \; \exists k \; \forall i > k$, $d(L, A_i) < \varepsilon$. Equivalently, L is a limit of $A_1, A_2, \ldots$ iff $\forall M \; \exists k \; \forall i > k$, $C(L,A_i) > M$.

LEMMA 5.2.1 (MacQueen, et. al.). Every Cauchy sequence in $\langle T',d\rangle$ converges to a unique limit.

proof (See Appendix, page 192)

Also shown by [MSP84] is that the function type formation operator "$\rightarrow$" when viewed as a binary operation over the metric space is contractive, and that the intersection operator is non expansive. Thus we have

LEMMA 5.2.2 (MacQueen, et. al.). Let A_1,A_2,B_1 and B_2 be ideals.

i) $C(A_1\rightarrow A_2, B_1\rightarrow B_2) > MIN(\ C(A_1,B_1), C(A_2,B_2)\ )$ provided $A_1\rightarrow A_2 \neq B_1\rightarrow B_2$

ii) $C(A_1\cap A_2, B_1\cap B_2) \geq MIN(\ C(A_1,B_1), C(A_2,B_2)\ )$

iii) $C(\cap\{A_i \mid i\in I\}, \cap\{B_i \mid i\in I\}) \geq MIN\{\ C(A_i,B_i) \mid i\in I\ \}$

proof (See Appendix, page 193)

COROLLARY 5.2.1. Let $\langle A_i\rangle_i$ and $\langle B_i\rangle_i$ be Cauchy sequences with limits a and b, resp. Then $\langle A_i\rightarrow B_i\rangle_i$ is Cauchy with limit $a\rightarrow b$.

proof by lemma 5.2.2-i. $\diamond$

The preceding lemmas provide us with a sufficient condition for the existence of a limit for a sequence of ideals denoted by type expressions. We define the $\rightarrow$-level of a subexpression σ of τ as the number of $\rightarrow$ nodes above σ in τ (viewing σ and τ as expression trees). Next, we say that two type expressions σ and τ are *level-k compatible* if they are identical expressions up to $\rightarrow$-level k. Formally, we have the following.

DEFINITION 5.2.2. Let τ, τ' be type expressions, $k \geq 0$. Then

$\mathrm{Comp}(\tau,\tau',k)$=if $k = 0$ then TRUE

else if τ atomic then $(\tau=\tau')$

else if $\tau = \tau_1 \rightarrow \tau_2$, $\tau' = \tau_1' \rightarrow t_2'$ then

$\mathrm{Comp}(\tau_1,\tau_1',k-1)$ AND $\mathrm{Comp}(\tau_2,\tau_2',k-1)$

else if $\tau = \tau_1 \cap \tau_2$, $\tau' = \tau_1' \cap \tau_2'$ then

$\mathrm{Comp}(\tau_1,\tau_1',k)$ AND $\mathrm{Comp}(\tau_2,\tau_2',k)$

else FALSE

Note that for fixed k, Comp is an equivalence relation, and for $k>0$, $\mathrm{Comp}(\tau,\tau',k) \Rightarrow \mathrm{Comp}(\tau,\tau',k-1)$.

LEMMA 5.2.3. Let σ and τ be type expressions in TCL. If $\mathrm{Comp}(\sigma,\tau,k)$, then

a) for any ρ, $C(M'[[\sigma]]\rho, M'[[\tau]]\rho) > k$

b) $C(M[[\sigma]], M[[\tau]]) > k$.

proof a) follows by the previous lemma and induction on $|\sigma| + |\tau|$. b) Let R be any index set for type environments ρ. By lemma 5.1.2-iii, $C(\cap\{ M'[[\sigma]]\rho_v \mid v \in R \}, \cap\{ M'[[\tau]]\rho_v \mid v \in R \}) > \mathrm{MIN}\{C(M'[[\sigma]]\rho_v, M'[[\tau]]\rho_v) \mid v \in R\} > k$. $\diamond$

Now, if σ_1, σ_2, ... is a sequence of type expressions with the property that for each k there exists n such that $\forall i>n$, $\mathrm{Comp}(\sigma_n,\sigma_i,k)$, then for all r the sequence $M'[[\sigma_1]]\rho$, $M'[[\sigma_2]]\rho$, .. is Cauchy, and so is the sequence $M[[\sigma_1]]$, $M[[\sigma_2]]$,... . This follows directly from lemma 5.2.3 and by the fact that $C(A,C) \geq \mathrm{MIN}(C(A,B),C(B,C))$ for all A, B, C.

LEMMA 5.2.4. If σ_1, σ_2, ... is a sequence of type expressions such that there are integers k_i with the property $Comp(\sigma_i,\sigma_{i+1},k_i)$ for each i and where k_1, k_2, ... is non-decreasing and unbounded, then

a) for all ρ, $M'[[\sigma_1]]\rho$, $M'[[\sigma_2]]\rho$, ... is a Cauchy sequence with unique limit,

b) $M[[\sigma_1]]$, $M[[\sigma_2]]$, ... is a Cauchy sequence with unique limit.

proof Obvious from the preceding paragraph, and the fact that $Comp(\sigma,\tau,k) \Rightarrow Comp(\sigma,\tau,k\text{-}1)$. ◇

This lemma will be used later on when we give meaning to recursively-defined type expressions.

We conclude this section by showing that if a set of Cauchy sequences are intersected, term by term, then under certain conditions the resulting sequence is Cauchy with limit equal to the intersection of the limits of the initial sequences. It is shown that this result implies that if a Cauchy sequence is contained in another Cauchy sequence (term by term), then containment holds for the limits as well. This is used in the next section to show the semantic soundness of the ≤ rules for recursively-defined type expressions.

LEMMA 5.2.5. Let $\{ s_i = \langle A_{i1},A_{i2},...\rangle \mid i \in I \}$ be a set of Cauchy sequences, where $Lim(s_i) = a_i$, and such that for all M, the set

$$\{k_i \mid i \in I, k_i \text{ is the smallest index such that } \forall j,h>k_i \,.\, C(A_{ij},A_{ih}) > M\}$$

has a maximal element. Then $\cap\{A_{i1} \mid i\in I\}, \cap\{A_{i2} \mid i\in I\}, \ldots$ is Cauchy with limit $\cap\{a_i \mid i\in I\}$.

proof Pick M. Each s_i Cauchy $\Rightarrow \forall i\in I\ \exists$ a minimal k_i such that $\forall j,h>k_i$, $C(A_{ij},A_{ih}) > M$. Let $k = MAX\{ k_i \mid i\in I \}$. Then $\exists k$ such that $\forall i\in I\ \forall j,h>k$, $C(A_{ij},A_{ih}) > M$, which implies $\exists k\ \forall j,h>k$, $MIN\{C(A_{ij},A_{ih}) \mid i\in I\} > M$. By lemma 5.2.2 iii), $\exists k\ \forall j,h>k$, $C(\cap\{A_{ij} \mid i\in I\}, \cap\{A_{ih} \mid i\in I\}) > M$. This shows the sequence is Cauchy. The same approach shows that $\cap\{a_i \mid i\in I\}$ is the limit. ◊

Lemma 5.2.5 has the consequence that term by term set inclusion of Cauchy sequences is carried over to the limits.

COROLLARY 5.2.2. Let $A_1, A_2, \ldots$ and $B_1, B_2, \ldots$ be Cauchy sequences with limits A and B, respectively. If $A_i \subseteq B_i$ for all i, then $A \subseteq B$.

proof $A_i = A_i \cap B_i$ and has limit $A = A \cap B$ by lemma 5.2.5, thus $A \subseteq B$. ◊

Also, from lemmas 5.2.4 and 5.2.5 we have the following corollary:

COROLLARY 5.2.3. If $\sigma_1, \sigma_2, \ldots$ is a sequence of type expressions such that there are integers k_i with the property $Comp(\sigma_i,\sigma_{i+1},k_i)$ for each i and where $k_1, k_2, \ldots$ is non-decreasing and unbounded, then

$$Lim\ M[[\sigma_1]], M[[\sigma_2]], \ldots = \cap\{\ Lim\ M'[[\sigma_1]]\rho, M'[[\sigma_2]]\rho, \ldots \mid \rho \in Tvar_\perp \to T\}$$

proof Immediate from lemmas 5.2.4 and 5.2.5. ◊

5.3 TCLμ

In this section, we extend the type expressions of TCL to include recursively-defined ones, yielding TCLμ. The added expressions are of the form $\mu x.\tau$ where x is a type variable and τ is a type expression such that each occurrence of x appears in a subexpression of τ of the form $\alpha \rightarrow \beta$. The meaning of such a $\mu x.\tau$ is the type x such that $x = \tau$, that is, the fixedpoint of the function mapping the ideal x to τ. Equivalently, one can view $\mu x.\tau$ as the infinite type expression obtained by repeatedly replacing x in τ by τ. We choose this approach and define the semantics of $\mu x.\tau$ as the limit of the meanings of successively larger finite portions of the infinite type expression tree. The $\leq$ relation is extended in a similar manner, and the resulting relation is used to define typing rules for the new language of type expressions. It is then shown that the typing rules for TCLμ are semantically sound.

We now expand the syntax of type expressions in TCL to allow for expressions of the form $\mu x.\tau$ meeting the criterion that all free occurrences of x in τ must be contained in some subexpression of τ of the form $\alpha \rightarrow \beta$.

DEFINITION 5.3.1. (Texpμ) The set of type expressions Texpμ contains all expressions of the forms

$$\text{Texp}\mu \ ::= \ \text{Tcnst} \mid \text{Tvar} \mid \text{Texp}\mu \rightarrow \text{Texp}\mu \mid \text{Texp}\mu \cap \text{Texp}\mu \mid \text{Tvar} \,.\, \text{Texp}\mu$$

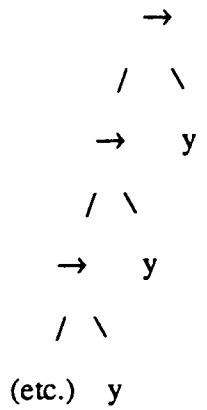
where each $x.\tau$ in Texpμ is such that all free occurrences of x in τ are contained in a subexpression of τ of the form $\alpha \rightarrow \beta$.

As in the lambda calculus, in the absence of parentheses the τ in $\mu x.\tau$ is taken to be everything following the "." up until the first unmatched closing parenthesis or the end of the entire expression, e.g., $\mu x.(a \cap x) \rightarrow \mu y.y \rightarrow a$ is parsed as $\mu x.((a \cap x) \rightarrow (\mu y.(y \rightarrow a)))$. Examples of expressions in Texpμ are

$$\mu x.z, \quad x \rightarrow \mu y.(y \cap x) \rightarrow y, \quad \mu x.(x \rightarrow y) \cap (y \rightarrow x) \quad \text{and} \quad \mu x.\mu y.z \cap (x \rightarrow y \rightarrow x),$$

whereas $\mu t.(\mu s.t \cap (s \rightarrow t))$ and $(\mu x.\mu y.x) \rightarrow t$ are not in Texpμ.

Expressions of the form $\mu x.\tau$ yield infinite expressions when we repeatedly replace x with τ in τ. Replacing x with τ in τ is called "unrolling" $\mu x.\tau$. For example, $\mu x.x \rightarrow y$ unrolled (once) yields $(\mu x.x \rightarrow y) \rightarrow y$. If unrolled indefinitely, $\mu x.x \rightarrow y$ gives us the infinite expression



One meaning for such a $\mu x.\tau$ is "the type x such that $x = \tau t$." By allowing such types, we increase the set of typeable expressions. For example, using $\mu x.x \rightarrow y$, we can give a derivation of SII(SII) : y. Let $\tau = \mu x.x \rightarrow y$ below.

A.	$S : (a\to b\to c)\to(a\to b)\to a\to c$	{1-a}
B.	$S : (\tau\to\tau\to y)\to(\tau\to\tau)\to\tau\to y$	{A,2}
C.	$I : a\to a$	{1-c}
D.	$I : \tau\to\tau = \tau\to\tau\to y$	{C,2}
E.	$SI : (\tau\to\tau)\to\tau\to y$	{B,D,3}
F.	$SII : \tau\to y = \tau$	{E,D,3}
G.	SII(SII) : y	{F,3}

Since $M[[y]]\emptyset = \{\bot\}$, the implication of SII(SII) : y is that SII(SII) is the totally undefined function, i.e., it never terminates, no matter what argument(s) it gets. It can be shown that the least fixedpoint combinator Y can also be typed using $\mu x.x\to y$ [see MSP84].

We now define the truncation of a member of $Texp\mu$ at level k using atom z.

DEFINITION 5.3.2. (Trunc) Let $\tau\in Texp\mu$, $k\geq 0$ and z be an atom. The expression $Trunc(\tau,k,z) \in Texp$ is defined by

$$
\begin{aligned}
Trunc(\tau, k, z) \;=\; & \text{if } k = 0 \text{ then } z \\
& \text{else if } \tau \text{ atomic then } \tau \\
& \text{else if } \tau = \tau_1\to\tau_2 \text{ then } Trunc(\tau_2,k\text{-}1,z) \to Trunc(\tau_2,k\text{-}1,z) \\
& \text{else if } \tau = \tau_1\cap\tau_2 \text{ then } Trunc(\tau_1,k,z) \cap Trunc(\tau_2,k,z) \\
& \text{else /* let } \tau = \mu x.\sigma \text{ */} \\
& \qquad Trunc(\ \sigma[x\leftarrow\tau],\ k,\ z\ )
\end{aligned}
$$

Note that since all occurrences of x in σ are contained in a left or right side of some $\to$-expression in σ, $\mu x.\sigma$ cannot be unrolled indefinitely in the computation of $Trunc(\mu x.\sigma,$ k,

z), hence Trunc is a well defined total function. Trunc(τ, k, z) has the effect of truncating the infinite expression corresponding to τ at $\rightarrow$-level k using the atom z. For example, Trunc($\mu x.x\rightarrow int$, 3, z) = ((z$\rightarrow$int)$\rightarrow$int)$\rightarrow$int, and Trunc((a$\rightarrow$b)$\cap\mu x.\mu y.x\rightarrow(b\cap y)$, 3, z) = (a$\rightarrow$b) $\cap$ (((z$\rightarrow$z)$\rightarrow$(b$\cap$(z$\rightarrow$z))) $\rightarrow$ (b$\cap$((z$\rightarrow$z)$\rightarrow$(b$\cap$(z$\rightarrow$z))).

Notice that Trunc(τ,k,z) and Trunc(τ,k+1,z) only differ (if at all) at $\rightarrow$-level k; that is, Comp(Trunc(τ,k,z), Trunc(τ,k+1,z), k) for all k. This gives us the following lemma:

LEMMA 5.3.1. Let τ be a type expression in Texpμ and let z be atomic. Then the sequences

a) (for all ρ) M'[[Trunc(τ,0,z)]]ρ, M'[[Trunc(τ,1,z)]]ρ, ... and

b) M[[Trunc(τ,0,z)]], M[[Trunc(τ,1,z)]], ...

are Cauchy and have a limit independent of z. Furthermore, the limit for b) is equal to the intersection over all ρ of the limit of the sequence $\langle$M'[[Trunc(τ,k,z)]]$\rho\rangle_k$.

proof First, it can be shown by induction on the computation of Trunc(τ,k_1,z) that Comp(Trunc(τ,k_1,z), Trunc(τ,k_2,z), k_1) holds whenever $k_1 \leq k_2$. By lemma 5.2.4, M[[Trunc(τ,0,z)]], M[[Trunc(τ,1,z)]], ... is Cauchy and therefore has a unique limit. Let seq1 be the sequence having i^{th} term M[[Trunc(τ,i,z_1)]] and seq2 have i^{th} term M[[Trunc(τ,i,z_2)]]. Note that Comp(Trunc(τ,k,z_1), Trunc(τ,k,z_2), k) holds for all k$\geq$0, thus C(M[[Trunc(τ,k,z_1)]], M[[Trunc(τ,k,z_2)]]) > k for all k, implying any limit for seq1 is also a limit for seq2. The final result follows from corollary 5.2.2. $\diamond$

This lemma suggests a semantics for expressions in Texpμ.

DEFINITION 5.3.3. (Semantics of Texpμ)

$$M\mu[[\tau]] = \cap\{M\mu'[[\tau]]\rho \mid \rho \in Tvar_\perp \rightarrow T\}$$

where $M\mu'[[\tau]]\,\rho = \lim \langle M'[[Trunc(\tau,k,z)]]\rho\rangle_k$

For α and β in Texpμ we define $\alpha \leq_\mu \beta$ to mean Trunc(α,k,z) $\leq$ Trunc(β,k,z) for all k. To show that this is a consistent extension of $\leq$, we show that $\leq_\mu = \leq$ when restricted to Texp.

LEMMA 5.3.2. For $\alpha,\beta \in$ Texp, $\alpha \leq_\mu \beta \Leftrightarrow \alpha \leq \beta$.

proof $\Rightarrow$ is trivial, since Trunc is the identity for large enough k. ($\Leftarrow$) Induct on the size of $|\alpha| + |\beta|$. The result is trivial for k=0, so assume k>0. If β atomic then Trunc(α,k,z) has β as a term, hence Trunc(α,k,z) $\leq \beta$ = Trunc(β,k,z). If $\beta = \beta_1 \cap \beta_2$, then by hypothesis, Trunc(α,k,z) $\leq$ Trunc(β_1,k,z) and Trunc(α,k,z) $\leq$ Trunc(β_2,k,z), thus Trunc(α,k,z) $\leq$ Trunc(β_1,k,z)$\cap$Trunc(β_2,k,z) = Trunc(β,k,z). If $\beta = \beta_1 \to \beta_2$, then let $\alpha = \alpha_1 \cap \ldots \cap \alpha_n$, let I $\subseteq$ 1..n such that for each i$\in$I, $\alpha_i = \sigma_i \to \tau_i$ and $\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \beta_2$. Then by hypothesis, Trunc(β_1,k-1,z) $\leq$ Trunc($\cap\{\sigma_i \mid i \in I\}$,k-1,z) and Trunc($\cap\{\tau_i \mid i \in I\}$,k-1,z) $\leq$ Trunc(β_2,k-1,z), thus by def of Trunc, Trunc($\cap\{\sigma_i \to \tau_i\} \cap \gamma$,k,z)$\leq$Trunc($\beta_1 \to \beta_2$,k,z) for any γ, $\Rightarrow$ Trunc(α,k,z)$\leq$Trunc(β,k,z). $\diamond$

Next we show that like $\leq$, the relation $\leq_\mu$ translates to $\subseteq$ in the domain of ideals T.

LEMMA 5.3.3. $\alpha \leq_\mu \beta \Rightarrow M\mu'[[\alpha]]\rho \subseteq M\mu'[[\beta]]\rho$, for all type maps ρ.

proof $M\mu'[[\alpha]]\rho = \lim \langle M'[[\text{Trunc}(\alpha,k,z)]]\rho \rangle_k$ and $M\mu'[[\beta]]\rho = \lim \langle M'[[\text{Trunc}(\beta,k,z)]]\rho \rangle_k$. Since Trunc($\alpha$,k,z) $\leq$ Trunc(β,k,z) for all k, we have that M'[[Trunc(α,k,z)]]$\rho \subseteq$ M'[[Trunc(β,k,z)]]ρ for all k (by lemma 2.4.1), which implies that lim $\langle$M'[[Trunc(α,k,z)]]$\rho\rangle_k \subseteq$ lim $\langle$M'[[Trunc(β,k,z)]]$\rho\rangle_k$ by corollary 5.2.2. $\diamond$

The following lemma is easily proven by induction on the computation of Trunc of the types involved.

LEMMA 5.3.4. Let α_1, $\alpha 2$, β_1, and $\beta 2$ be in Texpμ.

a) $\alpha_1 \to \alpha_2 \leq_\mu \beta_1 \to \beta_2 \Leftrightarrow \beta_1 \leq_\mu \alpha_1$ and $\alpha_2 \leq_\mu \beta_2$

b) $\alpha_1 \leq_\mu \beta_1 \cap \alpha_2 \Leftrightarrow \alpha_1 \leq_\mu \beta_1$ and $\alpha_1 \leq_\mu \beta_2$

c) $\mu t.\alpha_1 \leq_\mu \alpha_1[t \leftarrow \mu t.\alpha_1]$ and $\alpha_1[t \leftarrow \mu t.\alpha_1] \leq_\mu \mu t.\alpha_1$

proof by induction on the number of calls to Trunc. ◇

We now give the rules for type inference in TCLμ.

1-a) $S : (a \to b \to c) \to (a \to b) \to a \to c$

-b) $K : a \to b \to a$

2) $e : P\tau$ if $e : \tau$

3) $fg : \tau$ if $f : \sigma \to \tau$ and $g : \sigma$

4-a) $e : \sigma \cap \tau$ if $e : \sigma$ and $e : \tau$

-b) $e : \sigma$ and $e : \tau$ if $e : \sigma \cap \tau$

5) $e : \tau$ if $e : \sigma, \sigma \leq_\mu \tau$

As with TCL, we show that these rules are semantically sound.

LEMMA 5.3.5. $M\mu'[[\tau[t \leftarrow \sigma]]]\rho = M\mu'[[\tau]]\rho[M\mu'[[\sigma]]\rho \backslash t]$ for all σ, τ in Texpμ, and all type environments ρ.

proof First show that for σ and τ in Texp, $C(M'[[Trunc(\tau[t \leftarrow \sigma],k,z)]]\rho, M'[[Trunc(\tau,k,z)]]\rho[M'[[\sigma]]\rho \backslash t]) > k$ for all k, implying the limits of the two sequences are equal. Note that $Comp(Trunc(\tau[t \leftarrow \sigma],k,z), Trunc(\tau,k,z)[t \leftarrow \sigma], k)$, thus $C(M'[[Trunc(\tau[t \leftarrow \sigma],k,z)]]\rho, M'[[Trunc(\tau,k,z)[t \leftarrow \sigma]]]\rho) > k$ by lemma 5.2.2. But $M'[[Trunc(\tau,k,z)[t \leftarrow \sigma]]]\rho = M'[[Trunc(\tau,k,z)]]\ \rho[M'[[\sigma]]\rho \setminus t]$ by lemma 2.4.2. Next show by induction on $|\alpha|$ that for ideals A and B in T, if $C(A, B) > k$ then for any $\alpha \in$ Texp, $C(M'[[\alpha]]\ \rho[A \backslash t], M'[[\alpha]]\rho[B \backslash t]) > k$. The base case is trivial. Now suppose $\alpha = \alpha 1 \to \alpha 2$.

Then $M'[[\alpha]]\rho[A\backslash t] = M'[[\alpha 1]]\rho[A\backslash t] \rightarrow M'[[\alpha 2]]\rho[A\backslash t]$ and $M'[[\alpha]]\rho[B\backslash t] = M'[[\alpha 1]]\rho[B\backslash t] \rightarrow M'[[\alpha 2]]\rho[B\backslash t]$. By hypothesis, $C(M'[[\alpha 1]]\rho[A\backslash t], M'[[\alpha 1]]\rho[B\backslash t]) > k$ and $C(M'[[\alpha 2]]\rho[A\backslash t], M'[[\alpha 2]]\rho[B\backslash t]) > k$, which implies the result by lemma 5.2.2. A similar argument shows the result for $\alpha = \alpha 1 \cap \alpha 2$. It now follows that for any σ and τ in $Texp\mu$,

$C(M' [[Trunc(\tau,k,z)]] \rho [M' [[Trunc(\sigma,k,z)]] \rho\backslash t], M' [[Trunc(\tau,k,z)]] \rho[\lim \langle M'[[Trunc(\sigma,m,z)]]\rho\rangle_m \backslash t]) > k$ for all k, and therefore that $\lim \langle M'[[Trunc(\tau,k,z)]] \rho[M'[[Trunc(\sigma,k,z)]]\rho \backslash t]\rangle_k$ and $\lim \langle M'[[Trunc(\tau,k,z)]]\rho[M\mu'[[\sigma]]\rho \backslash t]\rangle_k$ are equal. But $C(M'[[Trunc(\tau,k,z) [t \leftarrow Trunc(s,k,z)]]]\rho, M'[[Trunc(\tau[t\leftarrow\sigma],k,z)]]\rho) > k$, thus $\lim \langle M'[[Trunc(\tau[t\leftarrow\sigma],k,z)]]\rho\rangle_k = \lim \langle M'[[Trunc(\tau,k,z)]]\rho[M\mu'[[\sigma]]\rho \backslash t]\rangle_k$. ◇

THEOREM 5.3.1. (Semantic Soundness) $e : \tau$ in $TCL\mu \Rightarrow E[[e]] \in M\mu'[[\tau]]\rho$ for all type environments ρ.

proof Induct on the derivation of $e : \tau$ in $TCL\mu$. Base case is when e = b, a primitive combinator, and $\tau = Ax(b)$. Then $M\mu'[[\tau]]\rho = M'[[\tau]]\rho$ since $\tau = Trunc(\tau, k, z)$ for all $k \geq 3$, and the result follows from the semantic soundness of TCL. Now take cases on the root of the derivation. If rule 2 is used, then $\tau = P\sigma$ for some substitution $P = [x_1 := \gamma_1,...,x_n := \gamma_n]$, and $e : \sigma$. By hypothesis, for all ρ, $E[[e]] \in M\mu'[[\sigma]]\rho$, which implies $E[[e]] \in M\mu'[[\sigma]] \rho[M\mu'[[\gamma_1]]\rho \backslash x_1, ..., M\mu'[[\gamma_n]]\rho \backslash x_n]$ for all ρ, which implies (by lemma 5.3.5) $E[[e]] \in M\mu'[[P\sigma]]\rho$ for all ρ. The result follows routinely when rule 3 or 4 is used, and the case for rule 5 follows by lemma 5.3.3. ◇

To compute when $\alpha \leq_\mu \beta$ holds, we can view α and β as expressions which possibly contain themselves as subexpressions. The algorithm to tell if $\alpha \leq_\mu \beta$ is the same as W except we return TRUE when a loop is detected in the recursive calls of W. This is similar to Knuth's algorithm for testing the equality of two circular lists [Knuth69]. To facilitate the definition of

the algorithm, we define the function Expose which unrolls an expression in Texpμ so that all its terms are non-μ's.

DEFINITION 5.3.4. (Expose) Let $\alpha \in$ Texpμ.

Expose(α) = if $\alpha = \alpha_1 \rightarrow \alpha_2$, or α is atomic, then α

else if $\alpha = \alpha_1 \cap \alpha_2$ then Expose(α_1) $\cap$ Expose(α_2)

else // let $\alpha = \mu x.\tau$ // Expose($\tau[x \leftarrow \alpha]$)

Henceforth, we will say that σ is a term of α if Expose(α) = $\sigma \cap \beta$ for some β, and σ is not a μ or $\cap$.

ALGORITHM 5.3.1. Let $\alpha, \beta \in$ Texpμ, α' = Expose(α), β' = Expose(β), let V be a finite subset of Texpμ × Texpμ, and let V' = V $\cup$ {$\langle \alpha, \beta \rangle$}.

Wμ(α,β,V) = if $\langle \alpha, \beta \rangle \in$ V then true

else if β' an atom then "β is a term of α' "

else if $\beta' = \beta_1 \cap \ldots \cap \beta_n$ (n > 1) then

Wμ(α,β_1,V') AND ... AND Wμ(α,β_n,V')

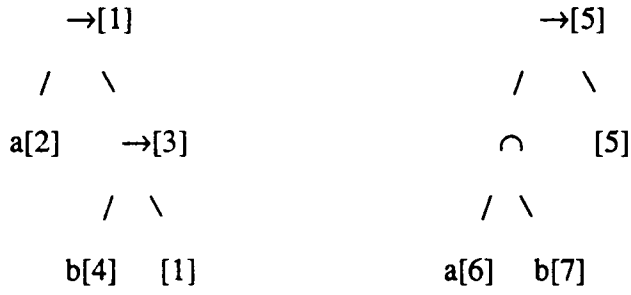
else // let $\beta' = \beta_1 \rightarrow \beta_2$ //

if {τ | $\sigma \rightarrow \tau$ a term of α', and Wμ(β_1,σ,V')} = Ø then false

else Wμ($\cap${τ | $\sigma \rightarrow \tau$ a term of α', and Wμ(β_1,τ,V')}, β_2, V')

If we represent α and β in Texpμ as trees which possibly have loops in them, we eliminate the need for computing Expose. To represent the set V of expression pairs, we may uniquely number all non-$\cap$ nodes which appear in either α or β, and represent any $\langle \sigma, \tau \rangle \in$ V as a pair of sets of node numbers $\langle \{M_1,\ldots,M_m\}, \{N_1,\ldots,N_n\} \rangle$ where each N_i is a term of σ and each M_i a term of τ.

For example, let $\alpha = \mu xa \to b \to x$, and $\beta = \mu x.(a \cap b) \to x$. To compute $W\mu(\alpha,\beta,\emptyset)$, we draw α and β as type expression trees with loops, and we number the non-$\cap$ nodes:



We then use these graphs along with sets of node numbers which describe a set of terms of subexpressions in either graph. For our example, we abbreviate the set {l, m, n, ...} by lmn... to save space. The computation of $W\mu(\alpha,\beta,\emptyset)$ where α is represented by 1 and β by 5 proceeds in this manner:

Call $W\mu(1,5,\emptyset)$

Compute $T = \{\tau \mid \sigma \to \tau$ a term of 1, and $W\mu(67,\sigma,\langle 1,5\rangle)\}$

Call $W\mu(67, 2, \langle 1,5\rangle)$

return true since 2 a term of 67

$T = \{3\}$ since $1 = 2 \to 3$

Compute $W\mu(3,5, \langle 1,5\rangle)$

Compute $T = \{\tau \mid \sigma \to \tau$ a term of 3, and $W\mu(67,\sigma, \langle 1,5\rangle \langle 3,5\rangle)\}$

Call $W\mu(67,4, \langle 1,5\rangle \langle 3,5\rangle)$

return true since 4 a term of 67

$T = \{1\}$ since $3 = 4 \to 1$

Compute $W\mu(1,5, \langle 1,5\rangle \langle 3,5\rangle)$

return true since $\langle 1,5\rangle \in V$

return true

return true

Notice that since α and β have a finite number of nodes, and no loop can occur in the tree of calls to Wμ, it follows that Wμ always terminates. Before showing that Wμ decides the $\leq_\mu$ relation (when initially given the two type expressions and the empty set as arguments), we prove a few useful intermediate results.

LEMMA 5.3.6. If W$\mu(\alpha.\beta,V)$ returns true and for all $\langle\sigma,\tau\rangle \in V$, W$\mu(\sigma,\tau,\emptyset)$ returns true, then W$\mu(\alpha,\beta,\emptyset)$ returns true.

proof Picture the tree of recursive calls in the computation of W$\mu(\alpha,\beta,V)$. Modify the call tree by replacing leaves returning true for W$\mu(\sigma,\tau,V\cup V')$ where $\langle\sigma,\tau\rangle \in V$ with a call tree for W$\mu(\sigma,\tau,V')$ returning true (there must be one, since W$\mu(\sigma,\tau,\emptyset) \Rightarrow$ W$\mu(\sigma,\tau V'))$. Then replace all other nodes of the form W$\mu(\gamma,\rho,V\cup V')$ with W$\mu(\gamma,\rho,V')$, and we have the tree of calls for W$\mu(\alpha,\beta,\emptyset)$ returning true. $\diamond$

LEMMA 5.3.7. $(\forall k\ \text{Trunc}(\alpha,k,z) \leq \text{Trunc}(\beta,k,z)) \Rightarrow \forall V$ W$\mu(\alpha,\beta,V)$ returns true.

proof Show for any V, if $\text{Trunc}(\alpha,k,z) \leq \text{Trunc}(\beta,k,z)$ for all k, then W$\mu(\alpha,\beta,V)$ returns true. Induct on the number of calls to Wμ in computing W$\mu(\alpha,\beta,V)$. Let α' = Expose(α), β' = Expose(α). Base case is one call. If W$\mu(\alpha,\beta,V)$ is not true, then either β' is an atom that is not a term of α', or $\beta' = \beta_1 \rightarrow \beta_2$ and α' has no $\rightarrow$ terms. Either way, $\text{Trunc}(\alpha,1,z) \leq \text{Trunc}(\beta,1,z)$ must be false. For the induction part, take cases on β'. If $\beta' = \beta 1 \cap \ldots \cap \beta n$, then for all k and for all $i \leq n$ $\text{Trunc}(\alpha',k,z) \leq \text{Trunc}(\beta_i,k,z)$. Since W$\mu(\alpha',\beta_i,V\cup\{\langle\alpha,\beta\rangle\})$ takes fewer calls to compute than W$\mu(\alpha,\beta,V)$, it returns true by our hypothesis (for each $i \leq n$), which implies W$\mu(\alpha,\beta,V)$ is true. If $\beta' = \beta_1 \rightarrow \beta_2$, then let α' =

$\alpha_1 \cap ... \cap \alpha_n$, let I_k ($k \geq 1$) be such that for all $i \in I_k$, $\alpha_i = \sigma_i \rightarrow \tau_i$ and $Trunc(\beta_1,k,z) \leq Trunc(\sigma_i,k,z)$ and $\cap\{Trunc(\tau_i,k,z) \mid i \in Ik\} \leq Trunc(\beta_2,k,z)$. Since $Trunc(\gamma,k,z) \leq Trunc(\rho,k,z)$ always implies $Trunc(\gamma,k-1,z) \leq Trunc(\rho,k-1,z)$ for $k \geq 1$, we get that the index set I_k which proves $Trunc(\alpha',k,z) \leq Trunc(\beta_1 \rightarrow \beta_2,k,z)$ also proves $Trunc(\alpha',k-1,z) \leq Trunc(\beta_1 \rightarrow \beta_2,k-1,z)$, and so for k-2, etc. It follows that there is a single index set $I \subseteq 1..n$ which proves $Trunc(\alpha',k,z) \leq Trunc(\beta_1 \rightarrow \beta_2,k,z)$ for all $k \geq 1$ (see lemma 5.4.2). Thus, for all $k \geq 1$, for all $i \in I$, $Trunc(\beta_1,k-1,z) \leq Trunc(\sigma_i,k-1,z)$ and $\cap\{Trunc(\tau_i,k-1,z) \leq Trunc(\beta_2,k-1,z) \Rightarrow$ (by hypoth.) $W\mu(\beta_1,\sigma i,V \cup \{\langle \alpha,\beta \rangle\})$ and $W\mu(\cap\{\tau_i \mid i \in I\},\beta_2,V \cup \{\langle \alpha,\beta \rangle\})$ return true for all $i \in I$, hence $W\mu(\cap\{\tau_i \mid \sigma_i \rightarrow \tau_i$ a term of α', $W\mu(\beta_1,\sigma_i,V \cup \{\langle \alpha,\beta \rangle\})$, β_2, $V \cup \{\langle \alpha,\beta \rangle\})$ must be true. $\diamond$

LEMMA 5.3.8. If $Trunc(\alpha,k,z) \leq Trunc(\beta,k,z)$ is NOT true for some k, then $W\mu(\alpha,\beta,\emptyset)$ returns false.

proof By contradiction. Let k be the smallest k such that $Trunc(\alpha,k,z) \leq Trunc(\beta,k,z)$ is not true and $W\mu(\alpha,\beta,\emptyset)$ returns true (for some α,β). For this k, choose β so that $Expose(\beta)$ has the least number of terms as possible. Note that $k \geq 1$. Let $\beta' = Expose(\beta)$. If $\beta' = \beta_1 \cap ... \cap \beta_n$ ($n>1$) then $Trunc(\alpha,k,z) \leq Trunc(\beta_i,k,z)$ is false for some $i \leq n$, and $W\mu(\alpha,\beta_i,\{\langle \alpha,\beta \rangle\})$ is true $\Rightarrow$ (by lemma 5.3.6) $W\mu(\alpha,\beta_i,\emptyset)$ is true, contradicting the minimality of the number of terms of β'. If $\beta' = \beta_1 \rightarrow \beta_2$, let $\alpha' = Expose(\alpha) = \alpha_1 \cap ... \cap \alpha_n$. If $\{\tau \mid \sigma \rightarrow \tau$ a term of α' and $Trunc(\beta_1,k-1,z) \leq Trunc(\sigma,k-1,z)\}$ is empty, then since $W\mu(\alpha,\beta,\emptyset)$ is true, there exists a term $\sigma \rightarrow \tau$ of α' with $W\mu(\beta_1,\sigma,\{\langle \alpha,\beta \rangle\})$ returning true, implying $W\mu(\beta_1,\sigma,\emptyset)$ returns true, by lemma 5.3.6. This contradicts the minimality of k. So $\cap\{Trunc(\tau,k-1,z) \mid \sigma \rightarrow \tau$ a term of α' and $Trunc(\beta_1,k-1,z) \leq Trunc(\sigma,k-1,z)\}$ has at least one term and is not weaker than $Trunc(\beta_2,k-1,z)$. Rewriting, $Trunc(\cap\{\tau \mid \sigma \rightarrow \tau$ a term of α' and $Trunc(\beta_1,k-1,z) \leq Trunc(\sigma,k-1,z)\}$ is not weaker than $Trunc(\beta_2,k-1,z)$. But $W\mu(\cap\{\tau \mid \sigma \rightarrow \tau$ a term of α', and $W\mu(\alpha_1,\sigma,\{\langle \alpha,\beta \rangle\})\}$, β_2, $\{\langle \alpha,\beta \rangle\})$ returns true $\Rightarrow$ (by lemma 5.3.6)

$W\mu(\cap\{\tau \mid \sigma\rightarrow\tau$ a term of α', $W\mu(\beta_1,\sigma,\emptyset)\}$, β_2, $\emptyset)$ is true. Now $Trunc(\beta_1,k\text{-}1,z) \leq Trunc(\sigma, k\text{-}1,z)$ holds if $W\mu(\beta_1,\sigma,\emptyset)$, by minimality of k, implying $W\mu(\cap\{\tau \mid \sigma\rightarrow\tau$ a term of α', $Trunc(\beta_1,k\text{-}1,z) \leq Trunc(\sigma,k\text{-}1,z)\}$, β_2, $\emptyset)$ returns true. But this again contradicts our choice of k. The remaining case is when β is atomic, which leads us to an immediate contradiction. $\diamond$

The preceding two lemmas give show the validity of $W\mu$ as a means of deciding $\leq_\mu$.

THEOREM 5.3.2. Let α, β $Texp\mu$. Then $\alpha \leq_\mu \beta \Leftrightarrow W\mu(\alpha,\beta,\emptyset)$ returns true.

proof Immediate from lemmas 5.3.7 and 5.3.8. $\diamond$

5.4 XTCLμ

In this section, explicit types are incorporated into TCLμ giving XTCLμ. The algorithms used in type checking XTCL are extended to accommodate a restricted form of μ-expressions.

In XTCLμ we use the auxiliary relation $<<_\mu$ among type expressions in Texpμ, as we have used << in XTCL.

DEFINITION 5.4.1. (Below) Let $\sigma,\tau \in$ Texpμ. Then $\sigma <<_\mu \tau$ means

$\exists n \exists$ substitutions $P_1,...,P_n$ with $P_1\sigma \cap ... \cap P_n\sigma \leq_\mu \tau$

DEFINITION 5.4.2. The language XTCLμ has the following syntax and typing rules:

Syntax

Exp ::= S | K | Pexp Exp

Pexp ::= [S::Texp] | [K::Texp] | Pexp Exp

Texp ::= { the set of all valid μ-expressions }

Tvar ::= { an infinite supply of variables }

Tconst ::= { a finite set of constants }

Typing rules

1-a) $S : (a\to b\to c)\to(a\to b)\to a\to c$

-b) $K : a\to b\to a$

-X) $[b:\tau] : \tau$ if $Ax(b) <<_\mu \tau$

2) $e : P\tau$ if $e : \tau$ and e not an f-expression (P a substitution)

3) $fg : \tau$ if $f : \sigma\to\tau$ and $g : \sigma$

4) $e : \sigma \cap \tau$ if $e : \sigma$ and $e : \tau$

5) $e : \tau$ if $e : \sigma, \sigma \leq_\mu \tau$

We use the term explicit typing for an expression e of TCLμ as we have used it for expressions in TCL, meaning an association of a type in Texpμ with each primitive f-expression of e. As might be expected, we have the following extension of theorem 4.2.1.

THEOREM 5.4.1. Let $e \in$ Exp. Then $e:\tau$ in TCLμ $\Leftrightarrow$ ∃ an explicit typing e' of e such that $e':\tau$ in XTCLμ.

proof Since $\leq_\mu$ is substitutive, the derivation tree transformations T1 through T6 of section 4.1 apply to derivations in TCLμ as well, and so the substitution rule (2) can be restricted to types of primitive combinators without altering the type relation. As in section 4.1, call this restricted system TCLμ'. One then shows, as before, that $b:\tau$ in TCLμ iff $Ax(b) <<_\mu \tau$. The result is finally proved using virtually the same proof of theorem 4.2.1 (i.e., induction on the derivation in TCLμ'). ◊

Principal types also exist for typeable expressions in XTCLμ. We use the obvious extension to algorithm 4.3.1:

ALGORITHM 5.4.2. (PTμ)

PTμ(e) = if $e \in \{S,K\}$ then Ax(e)

else if $e=[b:\tau]$ and $Ax(b) <<_\mu \tau$ then τ

else if e = fg and PTμ(g)≠error and PTμ(f)≠error

then $\cap\{\tau \mid PT\mu(g) <<_\mu \sigma, \sigma \rightarrow \tau$ a term of Expose(PTμ(f))$\}$

else error

THEOREM 5.4.2. $e:\tau$ in XTCLμ $\Rightarrow$ $PT(e) <<_{\mu} \tau$.

proof Completely analogous to the proof of theorem 4.3.1. $\diamond$

In order to extend the type checking algorithm of XTCL to XTCLμ we require a simplifying assumption. Notice that some type expressions in TCLμ can not be reduced, e.g., $\mu x.a\rightarrow(x\cap b)$ has an infinite number of reduced terms: $a\rightarrow b \cap a\rightarrow a\rightarrow b \cap$ etc. Since the type checking algorithm for TCL entails reducing type expressions, we restrict all type expressions $\mu x.\tau$ in TCLμ to be reducible to a finite number of terms. Syntactically, this means that we allow only $\mu x.\tau$ where τ is reduced and $\tau[x\leftarrow\tau]$ is reduced as well.

RESTRICTION ON μ-EXPRESSIONS. $\mu x.\ \tau$ is allowed if no subexpression of the form $\sigma \rightarrow (\rho\cap\gamma)$ occurs in τ or in $\tau[x\leftarrow\tau]$.

For type expressions restricted in this manner, we extend NRP to NRPμ by adding the case $NRP\mu(\mu x.\tau) = NRP\mu(\tau)$, and similarly extend Reduce to Reduceμ:

$$
\begin{aligned}
NRP\mu(\beta) = \quad & \text{if } \beta \text{ is an atom then } 1 \\
& \text{if } \beta = \beta_1\rightarrow\beta_2 \text{ then } NRP\mu(\beta_2) \\
& \text{if } \beta = \beta_1\cap\beta_2 \text{ then } NRP\mu(\beta_1) + NRP\mu(\beta_2) \\
& \text{if } \beta = \mu x.\tau \text{ then } NRP\mu(\tau)
\end{aligned}
$$

$$
\begin{aligned}
Reduce\mu(\beta) = \quad & \text{if } \beta \text{ is an atom then } \beta \\
& \text{if } \beta=\beta_1\rightarrow\beta_2 \text{ then } \cap\{Reduce\mu(\beta_1) \rightarrow \tau \mid \tau \text{ a term of } Reduce\mu(\beta_2)\} \\
& \text{if } \beta = \beta_1\cap\beta_2 \text{ then } Reduce\mu(\beta_1) \cap Reduce\mu(\beta_2) \\
& \text{if } \beta = \mu x.\tau \text{ then } \mu x.Reduce(\tau)
\end{aligned}
$$

Applying this restriction to μ-expressions, the technique then used in deriving an extension of the type checking algorithms in chapter 4 is to apply them, as is, to infinite expressions and to terminate with an appropriate answer when a loop is detected in the recursive calls. Of course $\leq_\mu$ can be decided by a direct implementation of Wμ. DP is extended by taking as extra argument a set of μ-expression pairs denoting propositions which have already been put into simple form. PT, ESD, ESC and ES remain essentially the same.

ALGORITHM 5.4.2. (DPμ) Let fv be a set of variables possibly occurring free in type expressions α or in β, but not in both, let V be a set of pairs of type expressions in TCLμ, and let $V' = V \cup \{\langle\alpha,\beta\rangle\}$.

DPμ("$\alpha \leq_\mu \beta$", fv, V) =

if $\langle\alpha,\beta\rangle \in$ V then {Ø}

else // let α' = Expose(α), β' = Expose(β) //

if $\beta' = \beta_1 \cap \beta_2$

then DU(DPμ("$\alpha' \leq_\mu \beta_1$", fv, V'), DPμ("$\alpha' \leq_\mu \beta_2$",fv,V'))

else if β' an atom, $\beta \notin$ fv, and $\alpha' = \alpha_1 \cap \alpha_2$

then DPμ("$\alpha_1 \leq_\mu \beta'$",fv,V') ∪ DPμ("$\alpha_2 \leq_\mu \beta'$",fv,V')

else if α' or β' an atom then { {"$\alpha' \leq_\mu \beta'$"} }

else // let $\beta' = \beta_1 \rightarrow \beta_2$ //

if $\alpha' = \alpha_1 \rightarrow \alpha_2$ then

DU(DPμ("$\beta_1 \leq_\mu \alpha_1$",fv,V'), DPμ("$\alpha_2 \leq_\mu \beta_2$",fv,V'))

else // let $\alpha' = \alpha_1 \cap \alpha_2$ //

if FV(β)∩fv = Ø and β reduced,

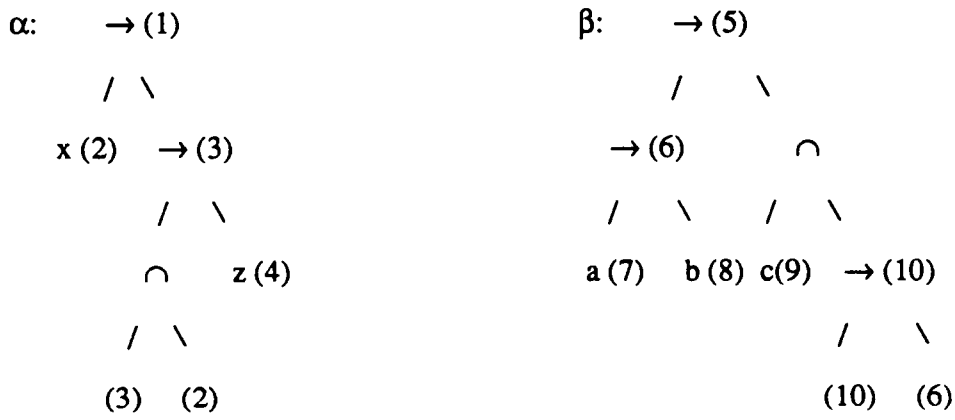
then DPμ("$\alpha_1 \leq \mu\beta$",fv,V') ∪ DPμ("$\alpha_2 \leq \mu\beta$",fv,V')

else // there are no assignable variables in α //

$$\cup\{DU(DP\mu(\text{"}\beta_1 \leq_\mu \sigma\text{"}, fv, V'),\ DP\mu(\text{"}\tau \leq_\mu \beta_2\text{"}, fv, V')) \mid$$
$$\sigma \to \tau = CombineArrows(B),$$
$$B \subseteq ArrowTerms(\alpha'),\ B \neq \emptyset\ \}$$

$DP\mu(\text{"}\alpha \leq_\mu \beta\text{"}, fv, V)$ always halts, for the same reason that $W\mu$ always halts: Since there are a finite number of (distinct) terms in any subexpression of α or β, and recursive calls to $DP\mu(\text{"}\sigma \leq_\mu \tau\text{"}, fv, V')$ are always such that σ and τ are intersections of terms in some subexpressions of α and β, an infinite path in the tree of calls to $DP\mu$ would have to repeat in the first parameter of $DP\mu$ (but loops in the first parameter are detected by testing for membership in the third parameter).

As an example, we will trace the algorithm for the proposition $\alpha \leq_\mu \beta$ where $\alpha = x \to \mu x.(x \cap y) \to z$ and $\beta = (a \to b) \to (c \cap \mu y.y \to a \to b)$. As before, our μ-expressions will be represented as graphs, and we number the terms of each distinct subexpression:



Each subexpression of α and β will be represented by a set of node numbers denoting the terms of the subexpression. When the set of node numbers is a singleton, we omit the surrounding braces. Let $F=FV(\alpha)=\{x,z\}$. The trace of $D = DP\mu(1 \leq_\mu 5, F, \emptyset)$ follows:

$DP\mu(1 \leq_\mu 5, F, \emptyset) = DU(D1, D2)$, where

$D1 = DP\mu(6 \leq_\mu 2, F, \{<1,5>\})$

$D2 = DP\mu(3 \leq_\mu \{9,10\}, F, \{<1,5>\})$

$D1 = \{ \{6 \leq_\mu 2\} \}$

$D2 = DU(D3, D4)$ where

$D3 = DP\mu(3 \leq_\mu 9, F, \{<1,5>, <3,\{9,10\}>\})$

$D4 = DP\mu(3 \leq_\mu 10, F, \{<1,5>, <3,\{9,10\}>\})$

$D3 = \{ \{3 \leq_\mu 9\} \}$

$D4 = DU(D5, D6)$ where

$D5 = DP\mu(10 \leq_\mu \{2,3\}, F, \{<1,5>, <3,\{9,10\}>, <3,10>\})$

$D6 = DP\mu(4 \leq_\mu 6, F, \{<1,5>, <3,\{9,10\}>, <3,10>\})$

$D6 = \{ \{4 \leq_\mu 6\} \}$

$D5 = DU(D7, D8)$ where

$D7 = DP\mu(10 \leq_\mu 2, F, \{<1,5>, <3,\{9,10\}>, <3,10>, <10,\{2,3\}>\})$

$D8 = DP\mu(10 \leq_\mu 3, F, \{<1,5>, <3,\{9,10\}>, <3,10>, <10,\{2,3\}>\})$

$D7 = \{ \{10 \leq_\mu 2\} \}$

$D8 = DU(D9, D10)$ where

$D9 = DP\mu(\{2,3\} \leq_\mu 10, F, \{<1,5>, <3,\{9,10\}>, <3,10>, <10,\{2,3\}>, <10,3>\})$

$D10 = DP\mu(6 \leq_\mu 4, F, \{<1,5>, <3,\{9,10\}>, <3,10>, <10,\{2,3\}>, <10,3>\})$

$D10 = \{ \{6 \leq_\mu 4\} \}$

$D9 = D11 \cup D12$ where

$D11 = DP\mu(2 \leq_\mu 10, F,$

$\{<1,5>, <3,\{9,10\}>, <3,10>, <10,\{2,3\}>, <10,3>, <\{2,3\},10>\})$

$D12 = DP\mu(3 \leq_\mu 10, F,$

$\{<1,5>, <3,\{9,10\}>, <3,10>, <10,\{2,3\}>, <10,3>, <\{2,3\},10>\})$

$D11 = \{ \{2 \leq_\mu 10\} \}$

$D12 = \{\emptyset\}$

...thus

$D9 = \{ \{2 \leq_\mu 10\}, \emptyset \}$

$D8 = \{ \{6 \leq_\mu 4, 2 \leq_\mu 10\}, \{6 \leq_\mu 4\} \}$

$D5 = \{ \{6 \leq_\mu 4, 10 \leq_\mu 2, 2 \leq_\mu 10\}, \{10 \leq_\mu 2, 6 \leq_\mu 4\} \}$

$D4 = \{ \{6 \leq_\mu 4, 4 \leq_\mu 6, 10 \leq_\mu 2, 2 \leq_\mu 10\}, \{10 \leq_\mu 2, 4 \leq_\mu 6, 6 \leq_\mu 4\} \}$

$D2 = \{ \{3 \leq_\mu 9, 6 \leq_\mu 4, 4 \leq_\mu 6, 10 \leq_\mu 2, 2 \leq_\mu 10\}, \{3 \leq_\mu 9, 10 \leq_\mu 2, 4 \leq_\mu 6, 6 \leq_\mu 4\} \}$

$D = \{ \{3 \leq_\mu 9, 6 \leq_\mu 2, 6 \leq_\mu 4, 4 \leq_\mu 6, 10 \leq_\mu 2, 2 \leq_\mu 10\},$

$\{3 \leq_\mu 9, 10 \leq_\mu 2, 6 \leq_\mu 2, 4 \leq_\mu 6, 6 \leq_\mu 4\} \}$

(We remark that D has no solution. As we will see, this means β is not a compatible type for α.)

To show the correctness of DPμ, we require an extension of lemma 4.6.1 to $\leq_\mu$.

LEMMA 5.4.1. Let α_1, α_2 and β be in Texpμ and let NRP$\mu(\beta) = 1$. Then

$$\alpha_1 \cap \alpha_2 \leq_\mu \beta \iff \alpha_1 \leq_\mu \beta \text{ or } \alpha_2 \leq_\mu \beta$$

proof From lemma 4.6.1, the result holds for every k-level truncation of the types involved. ($\Rightarrow$) Assume there are k_1 and k_2 such that the k_1-level truncation of α_1 is not $\leq$ than the k_1-level truncation of β, and the k_2-level truncation of α_2 is not $\leq$ than the k_2-level truncation of β. Since in general, Trunc(α,k,z) $\leq$ Trunc(β,k,z) implies Trunc(α,k-1,z) $\leq$ Trunc(β,k-1,z), k=Max(k_1,k_2) would be a k such that NOT(Trunc(α_1,k,z) $\leq$ Trunc(β,k,z) or Trunc(α_2,k,z) $\leq$ Trunc(β,k,z)), implying NOT($\alpha_1 \cap \alpha_2 \leq_\mu \beta$). ($\Leftarrow$) By definition of $\leq_\mu$. $\diamond$

We state as a lemma a similar result for the case when $\beta = \beta_1 \rightarrow \beta_2$, and may not be reduced.

LEMMA 5.4.2. If Expose(α) = $\alpha_1 \cap ... \cap \alpha_n$, $\beta = \beta_1 \rightarrow \beta_2$, then

$\alpha \leq_\mu \beta \Leftrightarrow \exists I \subseteq 1..n\ \forall i \in I\ \alpha_i = \sigma_i \rightarrow \tau_i$ and $\beta_1 \leq_\mu \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq_\mu \beta_2$.

proof ($\Rightarrow$) For each $k \geq 1$ $\exists I_k \subseteq 1..n$ such that $\forall i \in I_k\ \alpha_i = \sigma_i \rightarrow \tau_i$ and $Trunc(\beta_1,k,z) \leq \cap\{ Trunc(\sigma_i, k, z) \mid i \in I_k \}$ and $\cap\{Trunc(\tau_i, k, z) \mid i \in I_k\} \leq Trunc(\beta_2,k,z)$. Note that $Trunc(\beta_1,k',z) \leq \cap\{Trunc(\sigma_i,k',z) \mid i \in I_k\}$ and $\cap\{Trunc(\tau_i,k',z) \mid i \in I_k\} \leq Trunc(\beta_1,k',z)$ for all $k' < k$. Certainly there must be some I_k equal to an infinite number of I_j's. But this implies we may assume all I_j's are equal. ($\Leftarrow$) By definition of $\leq_\mu$. $\diamond$

We now show that DPμ(p, R, $\emptyset$) computes a set of sets of propositions logically equivalent to the proposition p when viewed as a disjunction of conjunctions. If C is a set of propositions and P a type substitution, then let SOLVEμ(C,P) mean $P\sigma \leq_\mu P\tau$ for all "$\sigma \leq_\mu \tau$" in C. As in chapter 4, we are only interested in the solvability of propositions p = "$\sigma \leq_\mu \tau$" where $FV(\sigma) \cap FV(\tau) = \emptyset$ using type substitutions P having domain either $FV(\sigma)$ or $FV(\tau)$, but not both. We first show that DPμ(p, R, $\emptyset$) is at least as easy to satisfy as p.

LEMMA 5.4.3. Let $\alpha, \beta \in$ Texpμ, let R be a set of variables, let P be a type substitution with domain R, and let $FV(\alpha) \cap FV(\beta) = \emptyset$.

I) β reduced and $FV(\alpha) \subseteq R$ and $R \cap FV(\beta) = \emptyset$ and $P\alpha \leq_\mu \beta \Rightarrow$

$\forall V\ \exists C \in DP\mu(\alpha \leq_\mu \beta, R, V)$ such that SOLVESμ(P,C)

II) β reduced and $FV(\beta) \subseteq R$ and $R \cap FV(\alpha) = \emptyset$ and $\alpha \leq_\mu P\beta \Rightarrow$

$\forall V\ \exists C \in DP\mu(\alpha \leq_\mu \beta, R, V)$ such that SOLVESμ(P,C)

proof Induct on the number of calls to DPμ in computing D = DP$\mu(\alpha \leq_\mu \beta, R, V)$. Let V' = $V \cup \{\langle \alpha, \beta \rangle\}$, α' = Expose(α) and β' = Expose(β), and consider the possible cases which could arise in the computation of D.

1. If $\langle \alpha, \beta \rangle \in V$, then D=$\emptyset$ is trivially satisfied in both I and II.

2. else, if $\beta' = \beta_1 \cap \beta_2$, then $D = DU(DP\mu(\alpha' \leq_\mu \beta_1, R, V'), DP\mu(\alpha' \leq_\mu \beta_2, R, V'))$.

I) $P\alpha \leq_\mu \beta_1 \cap \beta_2 \Rightarrow P\alpha' \leq_\mu \beta_1 \cap \beta_2 \Rightarrow P\alpha' \leq_\mu \beta_1$ and $P\alpha' \leq_\mu \beta_2 \Rightarrow$ (by hypothesis) $\exists C_1 \in DP\mu(\alpha' \leq_\mu \beta_1, R, V')$ and $\exists C_2 \in DP\mu(\alpha' \leq_\mu \beta_2, R, V')$ such that $SOLVES\mu(P, C_1 \cup C_2) \Rightarrow \exists C \in D$ with $SOLVES\mu(P,C)$.

II) <same argument as in I>

3. else, if β' is atomic, $\beta' \notin R$ and $\alpha' = \alpha_1 \cap \alpha_2$ then $D = DP\mu(\alpha_1 \leq_\mu \beta', R, V') \cup DP\mu(\alpha_2 \leq_\mu \beta', R, V)$.

I) $P\alpha \leq_\mu \beta \Rightarrow P\alpha' \leq_\mu \beta' \Rightarrow P\alpha_1 \cap P\alpha_2 \leq_\mu \beta' \Rightarrow$ (lemma 5.4.2) $P\alpha_1 \leq \mu \beta'$ or $P\alpha_2 \leq_\mu \beta' \Rightarrow$ (by hypothesis) $\exists C \in DP\mu(\alpha_1 \leq_\mu \beta', R, V') \cup DP\mu(\alpha_2 \leq_\mu \beta', R, V')$ such that SOLVES(P, C).

II) <same argument as in I>

4. else, if α' or β' is atomic then $D = \{\langle \alpha', \beta' \rangle\}$, and both I and II follow trivially.

5. else, if $\beta' = \beta_1 \rightarrow \beta_2$ and $\alpha' = \alpha_1 \rightarrow \alpha_2$ then $D = DU(DP\mu(\beta_1 \leq_\mu \alpha_1, R, V'), DP\mu(\alpha_2 \leq_\mu \beta_2, R, V'))$.

I) $P\alpha' \leq_\mu \beta' \Rightarrow \beta_1 \leq_\mu P\alpha_1$ and $P\alpha_2 \leq_\mu \beta_2 \Rightarrow$ (by hypothesis) $\exists C_1$ in $DP\mu(\beta_1 \leq_\mu \alpha_1, R, V')$ and $\exists C_2$ in $DP\mu(\alpha_2 \leq_\mu \beta_2, R, V')$ such that $SOLVES\mu(P, C_1 \cup C_2)$.

II) <similar argument as for I>

6. else, if $\beta' = \beta_1 \rightarrow \beta_2$ and $FV(\beta) \cap R = \emptyset$ and $\alpha' = \alpha_1 \cap \alpha_2$ and β reduced, then both I and II follow by lemma 5.4.2, as in case 3.

7. else, $\beta' = \beta_1 \rightarrow \beta_2$ and α not reduced, and only II applies. The result follows from lemma 5.4.3:

II) $\alpha' \leq_\mu P\beta' \Rightarrow \alpha' = \alpha_1 \cap \ldots \cap \alpha_n$ and $\exists I \subseteq 1..n$ $\forall i \in I$ $\alpha_i = \sigma_i \rightarrow \tau_i$ and $P\beta_1 \leq_\mu \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq_\mu P\beta_2 \Rightarrow$ (by hypothesis) $\exists C_1$ in $DP\mu(\beta_1 \leq_\mu \cap\{\sigma_i \mid i \in I\}, R, V')$ and $\exists C_2$ in $DP\mu(\cap\{\tau_i \mid i \in I\} \leq_\mu \beta_2, R, V')$ such that $SOLVES(P, C_1 \cup C_2)$; that is, for some subset I of Arrowterms(α'), $SOLVES(DU(DP\mu(\beta_1 \leq_\mu \sigma, R, V'), DP\mu(\tau \leq_\mu \beta_2, R, V')))$ where $\sigma \rightarrow \tau = CombineArrows(I)$. $\diamond$

Next we must show that $DP\mu(\alpha \leq_\mu \beta, R, \emptyset)$ does not return propositions which are easier to solve than "$\alpha \leq_\mu \beta$" itself.

LEMMA 5.4.4. Let α and β be in $Texp\mu$, $FV(\alpha) \cap FV(\beta) = \emptyset$, let R be a set of variables such that if $FV(\alpha) \cap R \neq \emptyset$ then β is reduced and R contains no free variable of β, and if $FV(\beta) \cap R \neq \emptyset$ then α is reduced and R contains no free variables of α, and let P be a type substitution with domain R. Then if $SOLVES\mu(P,C)$ for some C in $DP\mu(\alpha \leq_\mu \beta, R, V)$ then $W\mu(\alpha,\beta,PV)$ returns true, where PV denotes $\{\langle P\sigma, P\tau\rangle \mid \langle\sigma,\tau\rangle \in V\}$.

proof Induct on the number of calls to $DP\mu$. If $\langle\alpha,\beta\rangle \in V$, then $\langle P\alpha, P\beta\rangle \in PV$, and hence $W\mu(P\alpha,P\beta,PV)$ is true. Let $\alpha' = Expose(\alpha)$, $\beta' = Expose(\beta)$ and $V' = V \cup \{\langle\alpha,\beta\rangle\}$. If $\beta' = \beta_1 \cap \beta_2$, then P solves some C_1 in $DP\mu(\alpha \leq_\mu \beta_1, R, V')$ and some C_2 in $DP\mu(\alpha \leq_\mu \beta_2, R, V')$ $\Rightarrow$ (hypothesis) $W\mu(P\alpha,P\beta_1,PV')$ and $W\mu(P\alpha,P\beta_2,PV') \Rightarrow W\mu(P\alpha,P\beta,PV')$. If β' reduced and $FV(\beta) \cap R = \emptyset$ and $\alpha' = \alpha_1 \cap \alpha_2$ then P solves some C in $DP\mu(\alpha_1 \leq_\mu \beta', R, V')$ or in $DP\mu(\alpha_2 \leq_\mu \beta', R, V')$ $\Rightarrow$ (hypothesis) $W\mu(P\alpha_1,P\beta',PV')$ or $W\mu(P\alpha_2, P\beta', PV') \Rightarrow$ $W\mu(P\alpha,P\beta,PV)$. If $DP\mu$ returns $\{\{\langle\alpha',\beta'\rangle\}\}$, then $P\alpha' \leq_\mu P\beta' \Rightarrow P\alpha \leq \mu\ P\beta$ $\Rightarrow$ (by theorem 5.3.1) $W\mu(P\alpha,P\beta,\emptyset) \Rightarrow W\mu(P\alpha,P\beta,PV)$. The other cases for $\beta' = \beta_1 \rightarrow \beta_2$ follow directly from the definitions of $W\mu$ and $DP\mu$. $\diamond$

Our theorem stating the correctness of $DP\mu$ now follows directly from the preceding lemmas.

THEOREM 5.4.3. (Correctness of $DP\mu$) Let $\alpha, \beta \in Texp\mu$, β reduced, and $FV(\alpha) \cap FV(\beta) = \emptyset$. Then

$$(\exists C \in DP\mu(\alpha \leq_\mu \beta, FV(\alpha), \emptyset) \text{ such that } SOLVES(P,C)) \Leftrightarrow P\alpha \leq_\mu \beta$$

proof Immediate from the preceding two lemmas. $\diamond$

As with the algorithm DP, $DP\mu(\alpha \leq_\mu \beta, FV(\alpha), \emptyset)$ returns a set of sets of propositions each of the form $\sigma \leq_\mu \tau$ where either σ or τ are atomic, and such that it is never the case that both σ and τ contain assignable variables (i.e., variables in $FV(\alpha)$). As before, those propositions not containing any assignable variables can be checked for satisfiability by $W\mu$, and those propositions containing a non-atomic expression (on the left or right of the "$\leq_\mu$") which has at least one assignable variable are not satisfiable. Thus, each set in $DP\mu(\alpha \leq \mu \beta, FV(\alpha), \emptyset)$ can be reduced to a set C such that each proposition in C is of the form $\sigma \leq_\mu x$ or $x \leq_\mu \sigma$, where x is an assignable variable and σ contains no assignable variables. Again, if we let C_x denote the set of propositions in C containing the assignable variable x, then we can solve C iff we can solve each C_x. If there is at least one "$x \leq_\mu \tau$" in C_x, then C_x is satisfiable iff $\forall \sigma$ such that "$\sigma \leq_\mu x$" $\in C_x$, $\sigma \leq_\mu \cap \{\tau \mid$ "$x \leq \mu \tau$" $\in C_x\}$ (which is easily checked using $W\mu$). If no proposition of the form "$x \leq_\mu \tau$" is in C_x, then C_x is solvable iff there exists a $\leq_\mu$-upper bound for the set $\{\sigma \mid$ "$\sigma \leq_\mu x$" $\in C_x\}$.

If we take advantage of the fact that for any proposition "$\sigma \leq_\mu x$" $\in C_x$, every term γ of σ is such that $NRP\mu(\gamma) = 1$ (remember σ is reduced), then the set $\{\sigma \mid$ "$\sigma \leq_\mu x$" $\in C_x\} = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ has a $\leq_\mu$-upper bound iff either each σ_i has a term with an infinite rightmost $\rightarrow$ path (e.g., a term like $\mu x.a \rightarrow x$) or the σ_i's have a common finite rightmost $\rightarrow$ path having the same length and ending with the same atom. This can be checked in time proportional to N log N, where N is the total number of nodes in the rightmost $\rightarrow$ paths of all expressions in $\{IgnoreMu(\sigma) \mid$ "$\sigma \leq_\mu x$" $\in C_x\}$ ($IgnoreMu(\sigma)$ is σ with all $\mu x.\tau$ replaced by τ).

However, we will give a more general algorithm $EUB\mu$ for detecting if a $\leq_\mu$-upper bound exists for an arbitrary set A of expressions in $Texp\mu$. $EUB\mu$ is derived from EUB' of chapter 4 by modifying LEP' to return a finite representation of a (possibly) infinite set of length-end pairs. Recall that a length-end pair represents a "rightmost $\rightarrow$-path" of a type expression tree

σ which is constructed by starting at the root and moving right at any $\rightarrow$ node until a leaf is encountered. LEP$\mu(m,\sigma)$ returns a set of triples $\langle S,k,z\rangle$ where S is a finite set of natural numbers, k is a length (natural number) and z is an atom (the "end" of a rightmost $\rightarrow$ path from σ). Each triple $\langle S,k,z\rangle$ represents all the length-end pairs $\langle N+k,z\rangle$ where N is any sum of 0 or more elements (possibly repeated) from S, i.e., the pairs that would be produced if LEP' were applied to the infinite expression given by unrolling σ indefinitely. An integer n in a set S of a triple$\langle S,k,z\rangle$ implies that an occurrence of z is to the right of k $\rightarrow$'s in σ and is contained in the scope τ of some $\mu x.\tau$, where an occurrence of x appears to the right of n $\rightarrow$'s in τ; hence, by repeatedly unrolling $\mu x.\tau$ in σ, we get occurrences of z at lengths k, n + k, 2n + k, etc. from the root of σ. The set of all such triples represents all terminating rightmost $\rightarrow$-paths from the root of σ. Of course, any triple $\langle S,k,z\rangle$ with $S\neq\emptyset$ also includes the infinite rightmost path $\rightarrow\rightarrow\rightarrow$...etc.

Having computed the set L_i of LEPμ triples for each $\alpha_i\in A$, a bound for A exists iff either

1) each L_i is either empty or represents an infinite set, in which case $\mu x.\alpha\rightarrow x$ is a bound for a sufficiently weak α, or

2) some of the L_i represent finite sets (i.e., all triples are of the form $\langle\emptyset,k,a\rangle$), and there is a triple $\langle\emptyset,k,z\rangle$ in each "finite" set representing the length-end pair $\langle k,z\rangle$ which is also represented by a triple $\langle S,m,z\rangle$ in each of the "non-finite" sets.

ALGORITHM 5.4.3.

$$\text{EUB}\mu(A) \;=\; \text{HasCommon}(\,\{\text{LEP}\mu(0,\alpha)\mid \alpha\in A\}\,)$$

$LEP\mu(m,\sigma) =$ if σ atomic then $\{ \langle \emptyset,m,\sigma \rangle \}$

else if $\sigma = \sigma_1 \cap \sigma_2$ then $LEP\mu(m,\sigma_1) \cup LEP\mu(m,\sigma_2)$

else if $\sigma = \sigma_1 \rightarrow \sigma_2$ then $LEP\mu(m+1,\sigma_2)$

else // let $\sigma = \mu x.\tau$,

$Q = LEP\mu(0,\tau)$,

$Sx = \{ n \mid \langle S,k,x \rangle \in Q$ and $n \in S \cup \{k\} \}$ //

$\{ \langle S \cup Sx, n+m, z \rangle \mid \langle S,n,z \rangle \in Q,\ z \neq x \}$

HasCommon(H) =

// let $H^f = \{ Q \in H \mid Q \neq \emptyset$ and $\forall \langle S,n,z \rangle \in Q,\ S = \emptyset \}$, $H^\infty = H \setminus H^f$ //

if $H^f = \emptyset$ then true

else $(\exists \langle \emptyset,n,z \rangle \in \cap H^f\ \forall Q \in H^\infty\ \exists \langle S,m,z \rangle \in Q$ such that $n = k+m$

for some finite sum k (possibly 0) of integers in S)

LEMMA 5.4.5. $EUB\mu(A) \Leftrightarrow \exists \tau \in Texp\mu\ \forall \sigma \in A\ \sigma \leq_\mu \tau$

proof ($\Rightarrow$) If $EUB\mu(A)$ is true, then either there is an infinite rightmost $\rightarrow$-path in each σ in A (i.e., each σ contains a subexpression $\mu x.\rho$ not appearing to the left of any $\rightarrow$, and such that x terminates a rightmost $\rightarrow$ path in ρ) or in each σ there is a common rightmost $\rightarrow$-path of length k terminating with the same atom z. In the former case, $\rho = \mu x.\alpha \rightarrow x$ is a $\leq_\mu$-upper bound for A, where α is the intersection of all distinct subexpressions of all $\sigma \in A$. In the latter case, $\rho = \alpha \rightarrow \alpha \rightarrow \ldots \rightarrow z$ bounds A, where the rightmost $\rightarrow$ path of ρ is k $\rightarrow$'s long and α is as before. ($\Leftarrow$) If $EUB\mu(A)$ is false, then there is a $\sigma \in A$ with no infinite rightmost $\rightarrow$-path, and there is a $\tau \in A$ having no rightmost $\rightarrow$-path in common with σ. It can be shown by induction that $\sigma \leq_\mu \rho$ implies every rightmost $\rightarrow$-path of ρ is also a rightmost $\rightarrow$-path of σ, hence, there is no ρ such that $\sigma \leq_\mu \rho$ and $\tau \leq_\mu \rho$. $\diamond$

Another way to determine the existence of a $\leq_\mu$-upper bound for a set A is to convert each $\alpha_i \in A$ to a nondeterministic finite automaton nfa(α_i) which accepts a string of k 1's followed by the atom z iff $\langle N,n,z\rangle \in LEP\mu'(0, \alpha_i)$, where k = n + <linear combination of elements of N>. This construction is easily done by converting μ-expressions to finite trees with loops and deleting the left sides of all $\rightarrow$'s. Using each remaining node and arc,

1) use the root node as the initial state,

2) create for each atom z an accepting state Acc-z,

3) connect each occurrence of z to Acc-z by an arc labeled with 'z',

4) label each arc leaving an $\rightarrow$ node with '1',

5) label each arc leaving an $\cap$ with ε, denoting a null transition.

This defines the NFA associated with αi. Then, there is bound for A iff the NFAs nfa(α_i) accept a common string, or each nfa(α_i) accepts an infinite language.

The type checking algorithm is now the obvious extension of TC given in chapter 4.

ALGORITHM 5.4.4.

$TC\mu(e,\tau) = TCR\mu(e, Reduce\mu(\tau))$

$TCR\mu(e,\tau) =$ if $PT\mu(e) \neq$ error then $Below\mu(PT\mu(e), t)$

$Below\mu(\sigma, \tau) =$ if $\tau=\tau_1 \cap \tau_2$ then $Below\mu(\sigma,\tau_1)$ AND $Below\mu(\sigma,\tau_2)$

else if $\tau=\mu x.\rho$ then $Below\mu(\sigma,\rho[x \leftarrow \tau])$

else $ESD\mu(DP\mu("\sigma' \leq_\mu \tau", FV(\sigma'),\emptyset), FV(\sigma'))$

where σ' = σ with free variables renamed so that

$FV(\sigma') \cap FV(\sigma) = \emptyset$

$ESD\mu(D,fv)$ = if $\exists C \in D$ such that $ESC\mu(C,fv)$ then TRUE else FALSE

$ESC\mu(C,fv)$ = if $W\mu(\alpha,\beta)$ for all $"\alpha \leq_\mu \beta" \in \{"\alpha \leq_\mu \beta" \in C \mid FV(\alpha \rightarrow \beta) \cap fv = \emptyset\}$

then if $ES\mu(C_x,x)$ for all $x \in fv$, where $C_x = \{p \in C \mid x \text{ appears in } p\}$

then TRUE

else FALSE

else FALSE

$ES\mu(C_x,x)$ = if $\sim\exists"\alpha \leq_\mu x" \in C_x$ then TRUE

else if $\sim\exists"x \leq_\mu \alpha" \in C_x$

then if $EUB\mu(\{\alpha \mid "\alpha \leq_\mu x" \in C_x\})$

then TRUE

else FALSE

else if $W\mu(\alpha, \cap\{\beta \mid "x \leq_\mu \beta" \in C_x\})$ whenever $"\alpha \leq_\mu x" \in C_x$

then TRUE

else FALSE

THEOREM 5.4.4. $TC\mu(e,\tau) \Leftrightarrow e{:}\tau$ in $XTCL\mu$

proof This follows from the preceding lemma and theorems 5.4.2 and 5.4.3. ◊

6

Adding Practical Features to the Language

In this chapter, we add default explicit typing and abstract type declarations to XTCLμ. Default explicit typing refers to the capability of the type checker to supply types for primitive f-expressions when they are not given by the programmer (i.e., type inference, in the usual sense). Our goal here is for the type checker to at least infer a parametric type (if possible) for an expression containing no explicit types.

By abstract type declarations, we mean a facility for defining new combinators and types, and for hiding the implementation details from the rest of the program. This entails adding value and type identifiers to the language, along with a definition construct which changes the environment in which expressions are evaluated and typed. A language is proposed that accommodates these extensions.

6.1 Default Explicit Typing

In general, explicit types (or something along that line) are necessary to make our type system decidable, however, many times type checking can be done without them, e.g., recall from

chapter 4 the derivation of the composition function S(KS)K which had a type $(b\rightarrow c)\rightarrow(a\rightarrow b)\rightarrow a\rightarrow c$. This type can easily be inferred from the axiomatic types of S and K in the parametric system using unification. Therefore, we can relax the requirement that explicit types be supplied for all primitive f-expressions, but only for those which are required to have types not derivable in the parametric system (i.e., the system without $\cap$'s). It is a simple matter to extend the algorithm for computing principal types (with respect to $<<_\mu$) for explicitly typed expressions to one which also computes parametric principal types (with respect to substitution) for non-explicitly typed expressions having parametric types. When an f-expression f in $e = f\ e_1 \ldots e_n$ is encountered without an explicit type, the type checker can infer a principal type for e by using the axiomatic type of f and unifying the appropriate parts with the principal types of $e_1, \ldots, e_n$, thus capturing the type inference power of parametric type systems.

We start by giving a unification algorithm for type expressions in Texpμ. Ideally, the algorithm should find the most general substitution P such that $P\alpha \leq_\mu P\beta$, given types α and β, but the existence of such an inference algorithm would imply the decidability of the type system without explicit types. Hence, we extend the standard unification algorithm (found, for example, in [ASU85]) to compute a set of substitutions $\{P_1, \ldots, P_n\}$ such that $P_i\alpha \leq_\mu P_i\beta$ holds for all i. When α and β are parametric types (i.e., contain no intersections), the set computed is a singleton $\{P\}$ where $P\alpha = P\beta$, and P is the most general unifier of α and β.

Our algorithm UW takes a pair of type expressions α, β, a set V of expression pairs representing the comparisons considered already, and a set of equivalence classes of expressions (representing the unifications done so far) and returns a set of sets of equivalence classes, each representing a possible unification of variables with subexpressions of α and β which would force $\alpha \leq_\mu \beta$. The equivalences classes are such that there is at most one non-

variable expression per class--hence, the classes define a (possibly circular) substitution. For each equivalence class, we pick a distinguished expression (the representative) which is never a variable unless all expressions in the class are variables. The function Find then takes an expression α and a set of equivalence classes E and returns the distinguished element of the class in E which contains α. Given a set E of equivalence classes, the corresponding substitution P which makes $\alpha \leq_\mu \beta$ is obtained by replacing variables in a given class by the representative of that class, and performing that replacement on the other representatives, making sure that if x is to be replaced by an expression τ containing x, then the replacement is actually $\mu x.\tau$.

ALGORITHM 6.1.1. (Unify Weaker) Let $\alpha,\beta \in Texp\mu$, V be a set of pairs of type expressions, and let E be a partition of some set of expressions containing α and β and closed under subexpressions.

$UW(\alpha, \beta, V, E)=$ // let $\alpha' = Find(\alpha, E)$, $\beta' = Find(\beta, E)$, $V' = V \cup \{<\alpha,\beta>\}$ //

if $<\alpha,\beta> \in V$ then $\{E\}$

else if $\alpha' = \beta'$ then $\{ E \}$

else if $\beta'=\mu x.\tau$ then $UW(\alpha',\tau[x \leftarrow \beta'],V,E)$

else if $\alpha'=\mu x.\tau$ then $UW(\tau[x \leftarrow \alpha'],\beta',V,E)$

else if $\beta' = \beta_1 \cap \beta_2$ then

$$\cup\{UW(\alpha',\beta_2,V',E') \mid E' \in UW(\alpha',\beta_1,V',E)\}$$

else if $\alpha' = \alpha_1 \cap \alpha_2$ then $UW(\alpha_1,\beta',V',E) \cup UW(\alpha_2,\beta',V',E)$

else if α' or β' a variable then $\{Merge(\alpha',\beta',E)\}$

else if $\alpha' = \alpha_1 \rightarrow \alpha_2$ and $\beta' = \beta_1 \rightarrow \beta_2$ then

$$\cup\{UW(\alpha_2,\beta_2,V',E') \mid E' \in UW(\beta_1,\alpha_1,V',E)\}$$

else $\emptyset$

$\text{Merge}(\alpha,\beta,E) = (E \setminus \{[\alpha]_E, [\beta]_E\}) \cup \{ [\alpha]_E \cup [\beta]_E \}$

where $[n]_E$ denotes the equivalence class of n in E

UW always terminates since the maximum depth of calls for any computation UW(α, β, V', E), not counting unrolling of μ's, is limited by n^2, where n = total number of nodes of α and β. Initially, of course, we pass UW the finest partition of the subexpressions (nodes) involved, that is, the set of singletons E0 = { {n} | n a node of α or β }. That UW produces a set of equivalence classes, each forcing $\alpha \leq_\mu \beta$, is shown by induction on the number of calls to UW, and is sketched below. To state this properly, we say that a substitution P *respects* a set of equivalence classes E if whenever σ and τ are in the same class, then $P\sigma = P\tau$ (are identical when viewed as infinite expressions), and we say that P respects a set V of pairs of type expressions $<\sigma,\tau>$ if $P\sigma \leq_\mu P\tau$ for all $<\sigma,\tau>$ in V.

THEOREM 6.1.1. Let α, β be valid type expressions in Texpμ, and let E0 = { {σ} | σ a subexpression of α or β }. If P respects some $E \in \text{UW}(\alpha,\beta,\emptyset,E0)$, then $P\alpha \leq_\mu P\beta$.

proof Define a relation $\leq'$ over Texp as follows:

$\alpha \leq' \beta$ means $\alpha=\beta$, α and β are atomic, or

$\beta = \beta_1 \cap \beta_2$ and $\alpha \leq' \beta_1$ and $\alpha \leq' \beta_2$, or

$\alpha = \alpha_1 \cap \alpha_2$ and ($\alpha_1 \leq' \beta$ or $\alpha_2 \leq' \beta$), or

$\alpha=\alpha_1 \to \alpha_2$, $\beta=\beta_1 \to \beta_2$, $\beta_1 \leq' \alpha_1$ and $\alpha_2 \leq' \beta_2$

By induction on the sizes of α and β, it is easily shown that $\alpha \leq' \beta$ implies $\alpha \leq \beta$ (as defined in chapter 2). Then define $\alpha \leq_\mu' \beta$ to mean Trunc(α,k,z) $\leq'$ Trunc(β,k,z) for all k. It follows that $\alpha' \leq_\mu' \beta$ implies $\alpha \leq_\mu \beta$. Consider the following algorithm W':

W'(α, β, V) =

if $<\alpha,\beta> \in$ V then true

else if $\beta=\beta_1 \cap \beta_2$ then W'(α,β_1,V') and W'(α,β_2,V')

else if $\beta=\mu x.\tau$ then $W'(\alpha,\tau[x\leftarrow\beta],V)$

else if $\alpha=\mu x.\tau$ then $W'(\tau[x\leftarrow\alpha'],\beta,V)$

else if $\alpha = \alpha_1 \cap \alpha_2$ then $W'(\alpha_1,\beta,V')$ or $W'(\alpha_2,\beta,V')$

else if $\alpha=\beta$, α and β variables then true

else if $\alpha = \alpha_1 \rightarrow \alpha_2$ and $\beta = \beta_1 \rightarrow \beta_2$

then $W'(\beta_1,\alpha_1,V')$ and $W'(\alpha_2,\beta_2,V')$

else false

W' always terminates, provided α and β are valid (as defined in section 5.3). A simple induction on the number of calls to W' shows $W'(\alpha,\alpha,V)$ is true for all V (i.e., W' is reflexive). Using virtually the same proof of lemma 5.3.8, it can be shown that $W'(\alpha,\beta,\emptyset)$ implies $\alpha\leq_\mu'\beta$ (and therefore that $\alpha\leq_\mu\beta$). Let R(P,E) mean that P respects the partition E. Now show that ($E\in UW(\alpha,\beta,V,E')$ and R(P,E)) implies $W'(P\alpha, P\beta, PV)$, where PV is the list of pairs $<P\sigma,P\tau>$ where $<\sigma,\tau>$ is on V (hence $E\in UW(\alpha,\beta,\emptyset,E0)$ implies $W'(P\alpha,P\beta,\emptyset)$, and thus $P\alpha\leq_\mu P\beta$) . Induct on the number of calls to W': (let α'=find(α,E), β'=find(β,E))

base cases

1. $<\alpha,\beta>\in V$ implies $<P\alpha,P\beta>\in PV$ implies $W'(P\alpha,P\beta,PV)$

2. $\alpha' = \beta'$ implies $P\alpha' = P\beta'$, but also $P\alpha=P\alpha'$ and $P\beta=P\beta'$, thus $W'(P\alpha,P\beta,PV)$ by the reflexive property of W'.

3. α' or α' a variable, R(P, merge(α',β',E)) implies $P\alpha'=P\beta'$, and result follows as in 2.

inductive cases (let $V' = V\cup\{<\alpha,\beta>\}$)

1. $\beta' = \beta_1\cap\beta_2$, $E\in UW(\alpha,\beta,V,E)$ implies $E\in UW(\alpha',\beta_2,V',E')$ for some E' in $UW(\alpha',\beta_1,V',E1)$. Since E, E' and E1 are successively finer partitions, R(P,E) implies R(P,E') implies R(P,E1), thus by hypothesis $W'(P\alpha', P\beta_1, PV')$ and $W'(P\alpha', P\beta_2, PV')$, and since $P\alpha=P\alpha'$ and $P\beta=P\beta'$, we get $W'(P\alpha', P\beta', PV)$ and $W'(P\alpha, P\beta, PV)$.

2. α' or β' a μ-expression, then the result follows immediately by hypothesis.

3. $\alpha'=\alpha_1\rightarrow\alpha_2$, $\beta'=\beta_1\rightarrow\beta_2$, then $E\in UW(\alpha_2,\beta_2,V',E')$ for some E' in $UW(\beta_1,\alpha_1,V',E1)$, and the result follows as in case 1.

4. $\alpha'=\alpha_1\cap\alpha_2$, then $E\in UW(\alpha_1,\beta',V',E1)$ or $E\in UW(\alpha_2,\beta',V',E1)$, which implies by hypothesis $W'(P\alpha_1,P\beta',PV')$ or $W'(P\alpha_2,P\beta',PV')$, which implies $W'(P\,\alpha',P\beta',PV)$ and $W'(P\alpha,P\beta,PV)$. ◊

The substitution derived from a set of equivalence classes computed by UW is obtained by replacing each variable x with Find(x) and applying this substitution to the variables in Find(x). Loops in the iteration of substitutions (e.g., x substituted by $a\rightarrow x$) are "closed" with the μ operator (e.g., substitute x with $\mu x.a\rightarrow x$, instead of $a\rightarrow x$). An algorithm Sub which derives the substitution associated with a set of equivalence classes of type expressions is given below (ø denotes the identity substitution):

Sub(E) = if E contains no class c which contains variables then ø
else // let $c\in E$ contain variables $x_1, x_2, ..., x_2$
let R = Sub(E \ {c}), let τ=R(Find(x_1,E)),
let $\tau'=\mu x_1.\mu x_2...\mu x_n.\tau$,
let S = $[x_1:=\tau,...,x_n:=\tau]$, S'=$[x_1:=\tau',...,x_n:=\tau']$ //
if τ is a variable then S ∘ R
else if FV(τ) contains some x_i then S' ∘ R
else S ∘ R

The following lemma states that Sub computes a most general substitution from E.

LEMMA 6.1.1 Let E be a partition of subexpressions of some types α and β. If P respects E then $P = P \circ Sub(E)$.

proof By induction on the size of E. If E is empty, then Sub(E) is the identity substitution, and $P = P \circ Sub(E)$. The same is true when E contains no classes with variables. Suppose E contains a class c having variables $x_1,...,x_n$. Let $R = Sub(E\backslash\{c\})$. It is easily shown that R fixes each x_i. By the induction hypothesis, $P = P \circ R$. Let τ=find(x_1,E). Suppose $R\tau$ is a variable, or no x_i appears in $R\tau$. Let $S = [x_1:=R\tau,...,x_n:=R\tau]$. For any x_i, we have that $P(S(R(x_i))) = P(S(x_i))$ since R fixes the x_i's. For any $y \notin c$, let ρ=R(y). Note that ρ may contain some x_i's. Let z be a variable in ρ.

case1. if z is fixed by S, then P(z)= P(S(z))

case2. if $z=x_i$ for some i, then $P(S(z)) = P(R(\tau)) = P\tau = P(z)$ (since P unifies c)

This means that applying $P \circ S$ to R(y) is the same as applying P to R(y), thus $P \circ S \circ R = P \circ R = P$ for all variables. Now consider when some x_i appears in $R\tau$, where $R\tau$ is not a variable. Let S' assign each x_i the type $\mu x_1...\mu x_n.R(\tau)$, hence $S'(x_i) = S'(R(x_i))$ by unrolling of μ's. By induction on n, show that $P(x_1) = P(x_2) = ... = P(x_n) = P(R(\tau))$ implies that P maps each x_j to $P(\mu x_1...\mu x_n.R(\tau))$. For any x_i, $P(S'(R(x_i))) = P(S'(x_i)) = P(\mu x_1...\mu x_n.R(\tau)) = P(x_i)$. Suppose y not in c, then let ρ=R(y), z be a variable in ρ.

case1. if z is fixed by S' then P(z) = P(S'(z))

case2. if z=xi for some i, then P(S'(z)) = P(z), as stated above

Therefore $P \circ S' \circ R = P$ for all variables. $\Diamond$

For α and β parametric types (having no intersections), UW always computes a singleton, hence we can reduce UW to the algorithm U defined below, which returns either a partition of the subexpressions involved in the unification or error (error plays the part of $\emptyset$ in UW):

$U(\alpha, \beta, V, E) =$ // let $\alpha' = Find(\alpha, E)$, $\beta' = Find(\beta, E)$, $V' = V \cup \{<\alpha,\beta>\}$ //

if $<\alpha,\beta> \in V$ then E

else if $\alpha' = \beta'$ then E

else if $\beta' = \mu x.\tau$ then $U(\alpha', \tau[x \leftarrow \beta'], V, E)$

else if $\alpha' = \mu x.\tau$ then $U(\tau[x \leftarrow \alpha'], \beta', V, E)$

else if α' or β' a variable then $Merge(\alpha', \beta', E)$

else if $\alpha' = \alpha_1 \rightarrow \alpha_2$ and $\beta' = \beta_1 \rightarrow \beta_2$

then if $U(\beta_1, \alpha_1, V', E)$= error then error

else $U(\alpha_2, \beta_2, V', U(\beta_1, \alpha_1, V', E))$

else error

By inspection of the $\leq$ rules, it can be seen that if α and β have no intersections, then $\alpha \leq \beta$ iff α and β are identical, hence $\alpha \leq_\mu \beta$ iff α and β are identical when viewed as infinite trees. Theorem 6.1.1 therefore implies that any substitution P of parametric types for variables which respects $E = U(\alpha,\beta,\emptyset,E0)$, where α and β parametric, must unify α and β. This of course means that such a P must also unify corresponding subexpressions of α and β, in particular, P unifies any pair of expressions in any set V of any subcall $U(\sigma,\tau,V,E)$ of $U(\alpha,\beta,\emptyset,E0)$, as well as unifying σ and τ. Thus, if $\sigma=\alpha_1 \rightarrow \sigma_2$ and $\tau=\tau_1 \rightarrow \tau_2$ and P is a substitution of parametric types respecting partitions E and $E'=U(\sigma,\tau,V,E)$, then (since P unifies σ and τ) P must also respect $merge(\sigma,\tau,E)$ (and in fact $merge(\sigma,\tau,E')$). Hence, any P which respects $U(\alpha,\beta,\emptyset,E0)$ also respects $U'(\alpha,\beta,\emptyset,E0)$ defined as

$U'(\alpha,\beta, V, E) =$ // let $\alpha' = Find(\alpha, E)$, $\beta' = Find(\beta, E)$, $V' = V \cup \{<\alpha,\beta>\}$ //

if $<\alpha,\beta> \in V$ then E

else if $\alpha' = \beta'$ then E

else if $\beta' = \mu x.\tau$ then $U'(\alpha', \tau[x \leftarrow \beta'], V, E)$

else if $\alpha'=\mu x.\tau$ then U'($\tau[x\leftarrow\alpha']$,β',V,E)

else if α' or β' a variable then Merge(α',β',E)

else if $\alpha' = \alpha_1 \rightarrow \alpha_2$ and $\beta' = \beta_1 \rightarrow \beta_2$

then if U'(β_1,α_1,V',Merge(α',β',E))= error then error

else U'(α_2,β_2,V',U'(β_1,α_1,V',Merge(α',β',E)))

else error

and conversely, since U computes a finer partition than U'. Note that a pair $<\sigma,\tau>$ is added to the list of pairs V on a recursive call to U' only when σ and τ have been merged, so we may eliminate the list of pairs parameter altogether, since it is covered by the current partition parameter E. This gives the following equivalent version of U':

U''(α,β, E) = // let α' = Find(α, E), β' = Find(β, E) //

if $\alpha' = \beta'$ then E

else if $\beta'=\mu x.\tau$ then U''(α',$\tau[x\leftarrow\beta']$,E)

else if $\alpha'=\mu x.\tau$ then U''($\tau[x\leftarrow\alpha']$,β',E)

else if α' or β' a variable then Merge(α',β',E)

else if $\alpha' = \alpha_1 \rightarrow \alpha_2$ and $\beta' = \beta_1 \rightarrow \beta_2$

then if U''(β_1,α_1,Merge(α',β',E))= error then error

else U''(α_2,β_2,U''(β_1,α_1,Merge(α',β',E)))

else error

If we represent μ-expressions as circular trees, then the steps

else if $\beta'=\mu x.\tau$ then U''(α',$\tau[x\leftarrow\beta']$,E)

else if $\alpha'=\mu x.\tau$ then U''($\tau[x\leftarrow\alpha']$,b',E)

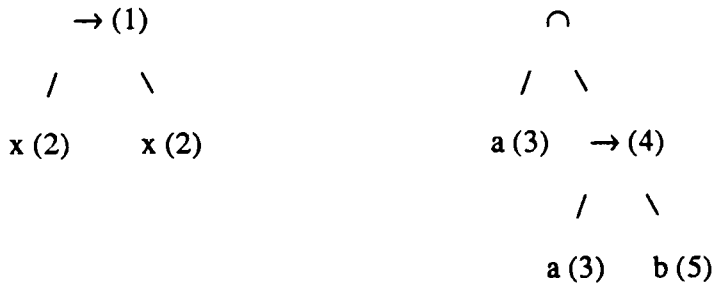
may be omitted, and U'' becomes the algorithm Unify given in [ASU85, pg. 378] (where $\rightarrow$ is the only operator). Unify(α,β,E0) computes a partition E such that Sub(E) is a most general substitution unifying α and β. From our discussion above, Sub(E) respects U(α,β,$\emptyset$,E0), which implies by lemma 6.1.1 that Sub(E) = Sub(E) $\circ$ Sub(U(α,β,$\emptyset$,E0)), implying that Sub(U(α,β,$\emptyset$,E0)) is also a most general substitution unifying α and β. That is, UW is the standard circular unification algorithm when restricted to parametric types.

THEOREM 6.1.2. If α and β are parametric and unifiable by P, then there is a substitution Q such that P = Q $\circ$ Sub(E), where E = UW(α,β,$\emptyset$,R0) and E0 is the finest partition of the subexpressions of α and β.

proof From the above discussion. $\Diamond$

Below are some examples of UW computations.

Example 1. Unify Weaker of $x\rightarrow x$ and $a\cap(a\rightarrow b)$. Once again, number the non-$\cap$ nodes:



Let E0 = { {1}, {2}, {3}, {4}, {5} }

V1 = { <x→x,a∩(a→b)> }

V2 = { <x→x, a∩(a→b)>, <x→x,a→b> }

UW(x→x, a∩(a→b), $\emptyset$, E0) = $\cup${UW(x→x, a→b, E') | E'$\in$ UW(x→x, a,$\emptyset$, E0)}

$UW(x\rightarrow x, a, \emptyset, E0) = \{ \{ \{1,3\}, \{2\}, \{4\}, \{5\} \} \}$

Compute $UW(x\rightarrow x, a\rightarrow b, V1, \{ \{1,3\}, \{2\}, \{4\}, \{5\} \})$

$= \cup\{UW(x,b,V2,E') \mid E' \in UW(a, x, V2, \{ \{1,3\}, \{2\}, \{4\}, \{5\} \}) \}$

$UW(a, x, V2, \{ \{1,3\}, \{2\}, \{4\}, \{5\} \}) = \{ \{ \{1,2,3\}, \{4\}, \{5\} \} \}$

The final answer is $UW(x,b, V2, \{ \{1,2,3\}, \{4\}, \{5\} \}) = \{ \{ \{1,2,3,5\}, \{4\} \} \}$. Thus, $x=a=b=\mu t.t\rightarrow t$ is the substitution obtained.

Example 2. Unify Weaker $a\cap(a\rightarrow b)$ and $x\rightarrow x$.

Let $E0 = \{ \{1\}, \{2\}, \{3\}, \{4\}, \{5\} \}$

$V1 = \{ <a\cap(a\rightarrow b), x\rightarrow x> \}$

$V2 = \{ <a\cap(a\rightarrow b), x\rightarrow x>, <a\rightarrow b, x\rightarrow x> \}$

Using the node numbering in example 1,

$UW(a\cap(a\rightarrow b), x\rightarrow x, \emptyset, E0) = UW(a, x\rightarrow x, V1, E0) \cup UW(a\rightarrow b, x\rightarrow x, V1, E0)$

Compute $UW(a, x\rightarrow x, V1, E0) = \{ \{ \{1,3\}, \{2\}, \{4\}, \{5\} \} \}$

Compute $UW(a\rightarrow b, x\rightarrow x, V1, E0)$

$= \cup\{UW(b,x,V2,E') \mid E' \in UW(x,a,V2,E0) \}$

$UW(x,a, V2, E0) = \{ \{ \{1\}, \{2,3\}, \{4\}, \{5\} \} \}$

$UW(b,x, \{ \{1\}, \{2,3\}, \{4\}, \{5\} \}) = \{ \{ \{1\}, \{2,3,5\}, \{4\} \} \}$

Hence, our final answer is $\{ \{ \{1,3\}, \{2\}, \{4\}, \{5\} \}, \{ \{1\}, \{2,3,5\}, \{4\} \} \}$, which gives the 2 substitutions $[a := x\rightarrow x]$ and $[a := x, b := x]$.

We can specify a type inference algorithm which works for explicitly-typed expressions and non-explicitly typed expressions, and which gives us the power of the parametric system (with μ-types) for the latter case. The algorithm IT("Infer Type") computes a type for an expression in which possibly not all primitive f-expressions are given explicit types. If IT is given an explicitly typed expression, then it calls IET ("Infer Explicit Type") which behaves

as PTμ. When given an application fg and the explicit type for f is not given, IT infers a type α for f and a type β for g. If α contains a variable term, then f has all types, and hence fg has all types (this only occurs when f is nonterminating, as in the case SII(SII)). Otherwise, each term of α (when μ-s are unrolled) of the form $\sigma\to\tau$ yields a set of substitutions obtained by (effectively) computing UW(ρ,σ,Ø,E0) where ρ is an intersection of k renamings of β, where k is the number of terms of σ (this is done by UT, "Unify Terms"). Below, the function Rename takes a type expression and a list of variables and renames any variables in the type expression which are on the list with new variables.

ALGORITHM 6.1.2. (Infer Type) e∈Exp, b a primitive combinator, Ax(b) is the axiomatic type for b.

IT(e) = if e = b then Ax(b)
else if e = [b::τ] e_1 e_2 ... e_n ($n\geq 0$) then EPT(e)
else // let e = fg //
if IT(f) or IT(g) are error then error
else // let $\alpha_1\cap \ldots \cap\alpha_n$ = Expose(IT(f)), β = IT(g) //
if α_i is a variable (some i) then α_i
else // let Z={Sub(E)τ | $\sigma\to\tau=\alpha_i$ (some i), E∈ UT(β,σ,E0)
where E0 is the finest partition
of the nodes of σ } //
if Z = Ø then error else $\cap$Z

IET(e) = if e = b then Ax(b)
else if e = [b::τ] and Below(Ax(b), τ) then τ
else if e = fg and IET(f)≠error and IET(g)≠error
// let Z = {τ | Below(IET(g),σ), $\sigma\to\tau$ a term of Expose(IET(f))} //

then if Z = Ø then error else ∩Z

else error

UT(γ,σ,E)= if σ=σ1∩σ2 then ∪{UT(γ,σ2,E') | E'∈ UT(γ,σ1,E) }

else if σ=μx.τ then UT(γ,τ[x←σ],E)

else UW(Rename(γ,FV(σ)),σ,Ø,E∪E0)

where E0 is the finest partition of the nodes of

Rename(γ,FV(σ))

For expressions e having no explicit types and having parametric axiomatic types for all primitive components, the above algorithm reduces to the usual algorithm for deriving principal parametric types for functional applications:

IT'(e) = if e = b then Ax(b)

else // let e=fg //

if IT'(f) or IT'(g) are error then error

else // let x'→y' = Rename(x→y, FV(IT'(f)))

H = U''(x'→y',IT'(f),E0) //

if H=error then error

else // let τ = Rename(IT'(g),{x',y'}∪FV(IT'(f)))

E = U''(Sub(H)x', τ, E0) //

if E=error then error

else Sub(E) (Sub(H) y')

As an example, we will infer a type for the standard SKI factorization of the Y combinator with no explicit types: Y = S D D where D = S B (K(SII)) where B = S (KS) K. As

before, E0 will represent the appropriate finest partition of the nodes in the expressions with which UW is called.

$IT(SI) = \cap\{Sub(E)\ ((a\to b)\to a\to c) \mid E \in UW(x\to x, a\to b\to c, \emptyset, E0)\}$

Since $UW(x\to x, a\to b\to c, \emptyset, E0)$ gives $a=x=b\to c$, we have the singleton

$\{\ ((b\to c)\to b)\to(b\to c)\to c\ \}$

$IT(SII) = \cap\{Sub(E)\ ((b\to c)\to c) \mid E\in UW(x\to x, (b\to c)\to b, \emptyset,E0)\}$, and since

$UW(x\to x, (b\to c)\to b,\emptyset, E0)$ gives $b=x=b\to c$, we get the singleton

$\{\ ((\mu x.x\to c) \to c) \to c\ \}$ (or equivalently $\{\mu x.x\to c\}$)

$IT(K(SII)) = \cap\{Sub(E)\ (y\to x) \mid E \in UW(((\mu x.x\to c)\to c)\to c, x,\emptyset, E0)\ \}$

$= \{\ y \to (((\mu x.x\to c)\to c)\to c)\ \}$

$IT(B) = IT(S(KS)K)$

$IT(KS) = \{\ y \to (\ (a\to b\to c)\to(a\to b)\to a\to c)\ \}$

$IT(S(KS)) = \cap\{Sub(E)\ ((a\to b)\to a\to c) \mid$

$E\in UW(y\to((a'\to b'\to c')\to(a'\to b')\to c'), a\to b\to c, \emptyset,E0)\ \}$

$= \{\ (y\to(a'\to b'\to c'))\to y\to(a'\to b')\to a'\to c'\ \}$

$IT(S(KS)K) = \cap\{Sub(E)\ (y\to(a'\to b')\to a'\to c') \mid$

$E\in UW(x'\to y'\to x', y\to a'\to b'\to c', \emptyset, E0)\ \}$

Now $E\in UW(x'\to y'\to x', y\to a'\to b'\to c', \emptyset, E0)$ gives $x'=y=b'\to c'$, $y'=a'$, thus we get the singleton

$\{(b'\to c')\to(a'\to b')\to a'\to c'\}$

$IT(SB) = \cap\{Sub(E)\ ((a\to b)\to a\to c) \mid E\in UW((y\to z)\to(x\to y)\to x\to z, a\to b\to c,\emptyset, E0)\}$

UW((y→z)→(x→y)→x→z, a→b→c,Ø, E0) gives a=y→z, b=x→y, c=x→z,

so we get { ((y→z)→(x→y)) → (y→z) → x → z }

IT(D) = IT(SB(K(SII)))

=∩{Sub(E)((y→z)→x→z)|E∈ UW(x'→(((μx.x→c)→c)→c), (y→z)→x→y,Ø,E0)}

Now UW(x'→(((μx.x→c)→c)→c), (y→z)→x→y,Ø,E0) gives

x' = y→z, x = (μx.x→c)→c, y = c, thus we get

{ (c→z) → ((μx.x→c)→c) → z }

IT(SD) =

∩{Sub(E) ((a→b)→a→c) | E∈ UW((c'→z)→((μx.x→c')→c')→z, a→b→c,Ø, E0) }

Now UW((c'→z)→((μx.x→c')→c')→z, a→b→c,Ø, E0) gives

a = c'→z, b = (μx.x→c')→c', c = z, hence we get the singleton

{ ((c'→z) → ((μx.x→c')→c')) → (c'→z) → z }

IT(SDD) = ∩{Sub(E)((c'→z)→z) |

E∈ UW((a→b)→((μx.x→a)→a)→b, (c'→z)→(μx.x→c')→c',Ø, E0) }

Now UW gives us a = b = c' = z, hence the type inferred for Y is the singleton

{ (z→z) → z }, as expected.

The above derivation can be done completely in the parametric type system with circular types, so we could have used IT'. Although circular types add considerable power to the parametric system, many circular parametric types are not useful in all contexts, e.g., SII has type μx.x→y, which is strong enough to infer a good type for the Y combinator, but fails to infer the expected type for I = SIII. If, instead, we used SII: (a∩(a→b)) → b, then the type inferred for SIII would be ∩{Sub(E) b | E ∈ UW(x → x,a,Ø,E'),

$E' \in UW(y \rightarrow y, a \rightarrow b, \emptyset, E0)\}$, and $UW(y \rightarrow y, a \rightarrow b, \emptyset, E0)$ gives the single unification $a=b=y$, which means $UW(x \rightarrow x, a, \emptyset, E')$ is the single unification $a=b=y=x \rightarrow x$, yielding the type $x \rightarrow x$, as expected.

The type checking algorithm for claims $e:\tau$ where e may not be explicitly typed is, of course, the same as TCμ, except that IT is used in place of PTμ. In the language given in section 6.3, we allow explicit types to be given for arbitrary expressions rather than primitive combinators only. So, for example, type inference for an expression $((e_1\ e_2)::\tau)\ e_3$ would be done by inferring a type σ for $(e_1\ e_2)$, checking that σ is below τ, and then continuing with the explicit type derivation, as though $(e_1\ e_2)$ were an explicitly typed primitive combinator.

6.2 Abstract Values and Types

We extend the system so that new combinators may be defined out of old ones and given an axiomatic type by the programmer or by the type checker. For example, for appropriate type τ, we could define composition in terms of S and K:

$$comp : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c = S\ (KS)\ K$$

When comp is referenced afterwards, its computational meaning is identical to the meaning of S(KS)K, however, we are restricted to using $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$ as its axiomatic type in type derivations, just as though rule 1 of the typing rules were augmented to include $comp:(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$. This implies that an environment must associate a value *and* a type expression with an identifier; the value is used at run time, and the type expression is used at compile time as an axiomatic type.

In the same spirit, we would like to be able to define abstract types. To this end we introduce type identifiers and type parameters which may be used to define *type expression generators*. For example, to redefine the primitive $\rightarrow$ constructor as a new type expression generator (that is, as a function from pairs of type expressions to type expressions), we could define ARROW(p,q) = $p \rightarrow q$, and then later, for example, use ARROW($a \cap b$,c) $\cap$ c instead of $((a \cap b) \rightarrow c) \cap c$. The meaning of an instantiation of a type expression generator depends on the context in which it is used, and in that sense type expression generators are very much like macros in conventional languages. We require that all type variables in a type generator be parameterized, thus

$$\text{INT}(a) = (a \rightarrow a) \rightarrow a \rightarrow a$$

$$\text{BOOL}(b) = b \rightarrow b \rightarrow b$$

are the type expressions for the classical lambda calculus representations of integer and boolean values [Stoy78]. Note that we are not associating INT(a) in all contexts with the ideal $\cap\{(a \rightarrow a) \rightarrow a \rightarrow a \mid a \in T\}$, just as we are not associating $a \rightarrow a$ with $\cap\{a \rightarrow a \mid a \in T\}$ in the expression $(a \rightarrow a) \rightarrow b$.

An abstract type consists of one or more sets of abstract values, a set of function names with signatures, and a set of axioms. An implementation of an abstract data type assigns a representation to the abstract values and function names such that the set of axioms is satisfied. Having implemented an abstract type, the programmer is allowed to use the abstract values and functions freely provided no assumptions are made on their representations other than that the axioms hold (indeed, the whole purpose of an abstract type is to endow a structure with properties deemed essential to the rest of the program and to hide all other

properties). In our language, the abstract values and functions will be defined as above (e.g., INT(a) and comp), and an axiom will either be a type statement of the form "val : type," or in the form of a ≤-rule, such as "ARROW(p,q) ≤ ARROW(r,s) when r≤p & q≤s."

As a simple example, consider the following definition of a generic record called Triple which uses a representation of nested pairs. (B is the curried composition function and pair is the curried pair formation function.)

```
{
    /* Types */
        Triple(a,b,c) = (a × b) × c;
    /* Constants */
        First: Triple(a,b,c) → a  =  B P1 P1;
        Second: Triple(a,b,c) → b  =  B P2 P1;
        Third: Triple(a,b,c) → c  =  P2;
        Make3: a → b → c → Triple(a,b,c)  =  B pair pair;
    /* Rules */
        Triple(a,b,c) ≤ Triple(d,e,f) when a≤d & b≤e & c≤f
 in E   /* E is an expression which uses the constants defined above */
}
```

The type checker will maintain three environments: 1) a constant environment γ, mapping identifiers to a computational meaning and an abstract type expression, 2) a type definition environment h mapping type generators to their representations (an abstract type expression), and 3) a weaker-rule environment R mapping type generators to rules which involve them. The type definition environment is used inside a module to verify the validity of abstract

constant representations and their types (e.g., to check that the implementation of Third really has type Triple(a,b,c)→c). The axiom and weaker-rule environments are used in checking type claims made outside the module (where no assumptions are made on the implementations of the constants or the actual form of the type generators).

To encapsulate a set of definitions and hide their representations from an expression E, the programmer may use the syntax

{ abstract definitions **in** E }

Of course, the abstract definitions themselves may contain encapsulations, and so may E itself. The scope rule we wish to implement is that the representations are used to check type claims in the abstract definitions (and in other definitions nested within), but the weaker-rules declared in the abstract definition are used for checking type claims in E.

In the next section, we give a purely functional language L which accommodates all the features discussed in this dissertation, along with pairs, disjoint sums and a rich set of primitives.

6.3 The Language L

Here we add integer and boolean values as primitive constants with primitive types INT and BOOL, along with a pairing operation (with projections) and a disjoint sum operation (with injections). We also add 0, TRUE and FALSE to the language of type expressions in order to denote the singleton types (e.g., 0, as a type, denotes the ideal which is the downward closure of the set {0}), and the type constant Δ to denote the bottom of the lattice of ideals.

SYNTAX OF L Let

E∈ Exp /* Computational expressions */

N∈ Num /* Integer Numerals */

X∈ Ide /* Identifiers (pre- and user-defined) including S, K, I, B, Y */

v∈ Tvar /* Type variables */

τ∈ ATexp /* Abstract Type expressions */

Gc∈ Gcall /* Type Generator Calls */

Lt∈ Tlist /* List of abstract type expressions */

Lv∈ Vlist /* List of type variables */

Lc∈ Clist /* List of weaker conditions */

D∈ Def /* Definitions */

W∈ Wrel /* Weaker Relations */

G∈ Ghead /* Type Generator Headers */

E ::= X | (E) | E::τ | E_1 E_2 | <E_1, E_2> | { D in E } | N

N ::= 0 | 1 | -1 | 2 | -2 | ...

τ ::= v | (τ) | $\tau_1 \cap \tau_2$ | $\tau_1 \rightarrow \tau_2$ | $\tau_1 \times \tau_2$ | $\tau_1 + \tau_2$ | μv.τ | Gc | Δ

Gc ::= X | X (Lt)

Lt ::= τ | Lt_1 , Lt_2

Lv ::= v | Lv_1 , Lv_2

D ::= G = τ | X : τ = E | W | W when Lc | D_1 ; D_2

G ::= X | X (Lv)

W ::= $G_1 \leq G_2$

Lc ::= $v_1 \leq v_2$ | Lc_1 & Lc_2

As usual for functional languages, a "program" is a computational expression E.

The domain of computational values must include representations for pairs, disjoint sums, integers and boolean values, as well as the primitive functions S and K. Of course, all of these may be represented in a domain D isomorphic to D→D: Integer representations may be based on the Church numerals using any one of various natural number encodings, booleans can be represented by 0 and 1, the pair <x,y> can be represented by a function from {1,2} to {x,y}, hence the type A × B denotes the ideal ({1}→A) ∩ ({2}→B), and objects of type A + B are actually objects of ({1}×A) ∪ ({2}×B) (here "∪" denotes least upper bound in T), thus the result of injecting a∈ A into A + B gives <1,a> which is in the ideal ({1}×A) ∪ ({2}×$\perp_T$) (i.e., in A + Δ, and therefore in A+B for all B). Any primitives we wish to add which operate on these objects can be expressed as SK combinations, and therefore have meaning in D. If we choose these representations, then the following extension to our weaker rules is valid when α≤β is interpreted as ∀ρ Mμ'[[α]]ρ ⊆ Mμ'[[β]]ρ.

$\Delta \leq \alpha$ for all α

$\alpha \leq \alpha$ for all α

$N \leq$ INT for all integers N

true ≤ BOOL, false ≤ BOOL

$\alpha \leq \beta_1 \cap \beta_2$ if $\alpha \leq \beta_1$ and $\alpha \leq \beta_2$

$\alpha \leq \beta_1 \rightarrow \beta_2$ if $\cap\{\tau \mid \sigma \rightarrow \tau$ a term of $\alpha, \beta_1 \leq \sigma\} \leq \beta_2$

$\alpha \leq \beta_1 \times \beta_2$ if $\cap\{\sigma \mid \sigma \times \tau$ a term of $\alpha\} \leq \beta_1$ and $\cap\{\tau \mid \sigma \times \tau$ a term of $\alpha\} \leq \beta_2$

$\alpha \leq \beta_1 + \beta_2$ if $\cap\{\sigma \mid \sigma + \tau$ a term of $\alpha\} \leq \beta_1$ and $\cap\{\tau \mid \sigma + \tau$ a term of $\alpha\} \leq \beta_2$

$\alpha_1 \cap \alpha_2 \leq \beta$ if $\alpha_1 \leq \beta$ or $\alpha_2 \leq \beta$

By our definition of ×, we see that $\mu x.\alpha \times \beta$ has meaning in the lattice of ideals (that is, Trunc($\mu x.(1 \rightarrow \alpha) \cap (2 \rightarrow b)$, k, z) is defined for all k). It is easily shown that, like ∩, the least

upper bound operator $\cup$ on the lattice of ideals is non-expansive [MSP84], hence $\mu x.\alpha + \beta$ denotes a well defined ideal under our semantics of chapter 5.

If we restrict type generator $F(p_1,...,p_n) = \tau$ so that each occurrence of p_i in τ is contained in a subexpression formed by an $\rightarrow$, $\times$, $+$, or another type generator, and if we prohibit circularity in our definitions of type generators (that is, τ must contain only previously defined generators) then we may use the μ operator to abstract any variable x from expressions containing $F(\tau_1,...,\tau_n)$ where $x=\tau_i$, for some i. These restrictions are placed on definitions of type generators, as can be noted in the semantics below. (Actually, circularity could be allowed with a little extra work. We would have to rule out definitions such as $F(p) = G(p) \cap \tau$ and $G(p) = F(p) \cap \sigma$, while allowing $F(p) = G(p) \rightarrow \tau$ and $G(p) = F(p) \rightarrow \sigma$.)

In L, a name may identify several type generators of different arity. For example, you may define

$$List(p) = \mu s.\ NilType + (p \times s)$$

$$List = List(INT)$$

in the same module. When a generator $F(p_1,...,p_n)$ is defined, the weaker rule

$$F(p_1,...,p_n) \leq F(q_1,...,q_n) \text{ when } p_1 \leq q_1 \ \&\ q_1 \leq p_1 \ \&\ ... \ \&\ p_n \leq q_n \ \&\ q_n \leq p_n$$

is added automatically to the rule environment. This rule is overridden if the programmer adds a rule declaration involving F (of arity n) with itself, e.g.,

$$Stack(x) \leq Stack(y) \text{ when } x \leq y$$

In L, the user is not allowed to redefine a type generator of a given arity. This allows us to associate a type expression with a computational value, and use this association in all contexts, hence

{ BOOL = INT in is0 true }

would be an error.

A weaker rule declaration of the form

$$F(p_1,...,p_m) \leq G(q_1,...,q_n) \text{ when } cond_1 \ \& \ cond_2 \ \& \ ... \ \& \ cond_k$$

is allowed in L when

a) $F(p_1,...,p_m)$ and $G(q_1,...,q_n)$ have representations σ and τ, resp.

b) all p's and q's are distinct

c) each cond is either of the form $p_i \leq q_j$ or $q_j \leq p_i$ (some $i \leq m$, $j \leq n$)

d) $\sigma \leq \tau$ can be derived from the current $\leq$-rules plus the conditions (as axioms).

Such a declaration adds an inference rule to the $\leq$-rule environment, which is used in checking type claims involving generators whose representations have been hidden.

Below, we give the main semantic definitions for L, along with a brief description of the auxiliary functions needed to complete the semantics. A more detailed description of the auxiliary functions needed for a complete implementation of the type checker is given in the appendix.

Notation: The domain constructors used, in order of decreasing precedence, are

A* Flat domain of finite sequences of elements of A

$A \times B$	Ordered pairs
$A + B$	Disjoint union
$A \rightarrow B$	Functions from A to B

Finite sets of elements, such as {true, false}, denote flat domains having the listed elements plus a least element.

SEMANTIC DOMAINS Let error, undefined, unmapped and notconcrete be elements of D.

$d,e \in D = D \rightarrow D$

$n,m \in Nat = \{ 0, 1, 2, \ldots \}$

$Err = \{error\}$

$c \in Conds = (Texp^* \times Texp^*) \rightarrow (Texp \times Texp)^*$

$z \in Gen = Ide \times Nat^*$

$R \in Rules = Gen \rightarrow ((Gen \times Conds)^* + \{undefined\})$

$f \in Rep = ATexp^* \rightarrow (Atexp + Err)$

$h \in Reps = Gen \rightarrow (Rep + \{notconcrete\})$

$P \in Subst = Tvar \rightarrow ATexp$

$\gamma \in Env = Ide \rightarrow ((D \times ATexp) + \{unbound\})$

$L,null \in A^*$ (arbitrary A)

TYPES FOR MAIN SEMANTIC FUNCTIONS

M_N: Numeral $\rightarrow$ D // primitive meanings of integer numerals //

OP: {+,-,*,/,is0,eq, less, neq} $\rightarrow$ D // primitive meanings of integer functions //

M_E: Exp $\rightarrow$ Env $\rightarrow$ Reps $\rightarrow$ Rules $\rightarrow$ (D $\times$ Atexp) + Err

M_A: Atexp → Tvar* → Reps → Rules → (Atexp + Err)

M_T: Tlist → Tvar* → Reps → Rules → (Atexp* + Err)

M_{Gc}: Gcall → Rules → (Gen × Atexp* + Err)

MakeTlist: Tlist → Atexp*

M_D: Def → Env → Reps → Rules → ((Env × Reps × Rules) + Err)

M_W: Wrel → Reps → (Gen × Tvar* × Rep) × (Gen × Tvar* × Rep) + Err

M_G: Ghead → Gen × Tvar*

M_V: Vlist → (Tvar* + Err)

M_C: Clist → Tvar* → Tvar* → Conds + Err

MAIN SEMANTIC CLAUSES

For Exp

$M_E[[X]]\gamma hR$ = if $\gamma[[X]]=<d,\tau>$ then $<d,\tau>$ else error

$M_E[[E::\tau]]\gamma hR$ = if $M_E[[E]]\gamma hR=<d,\tau'>$ then

if $M_A[[\ \tau\ ]]hR = \tau''$ then

if Reduce(τ'') = τ''' then

if Below(τ', τ''', null, R) then $<d,\tau'''>$

else error

else error

else error

else error

$M_E[[<E_1,E_2>]]\gamma hR$ = if $M_E[[E_1]]\gamma hR=<d_1,\tau_1>$ and $M_E[[E_2]]\gamma hR=<d_2,\tau_2>$ then

$<<d_1,d_2>, \tau_1 \times \text{Rename}(\tau_2,FV(\tau_1))>$ else error

$M_E[[\ \{\ D \text{ in } E\ \}\ ]]\gamma hR$ = if $M_D[[D]]\gamma hR=<\gamma', h', R'>$ then $M_E[[E]]\gamma' hR'$ else error

$M_E[[N]]\gamma hR$ = if $M_N[[N]]=0$ then $<M_N[[N]], 0>$ else $<M_N[[N]], INT>$

$M_E[[E_1\ E_2]]\gamma h R$ = if $M_E[[E_1]]\gamma h R$=<e,α> and $M_E[[E_2]]\gamma h R$=<d,β> then

if Explicit[[E_1]] then

if WhichBelow(α,β,R) = τ then <e d, τ>

else error

else if Infer(α, β, R) = τ then <e d, τ>

else error

else error

For Atexp

$M_A[[\ D\ ]]L\ L'\ h\ R$ = D

$M_A[[\ (\ \tau\ )\ ]]L\ L'\ h\ R$ = $M_A[[\ \tau\ ]]L\ L'\ h\ R$

$M_A[[\ X\ ]]L\ L'\ h\ R$ =

if h<X,0> = f then f(null)

else if R<X,0>$\neq$unmapped then X

else error

$M_A[[\ X\ (\ Lt\ )\ ]]\ L\ L'\ h\ R$ =

if $M_T[[\ Lt\ ]]\ L$ null h R = L'' then

if h<X, Len(L'')> = f then f(L'')

else if R<X, Len(L'')>$\neq$unmapped then [[X(Lt)]]

else error

else error

$M_A[[\ v\ ]]L\ L'\ h\ R$ = if OnVlist(v, L') or not OnVlist(v, L) then error else v

($\forall$op$\in$ {+,$\times$,$\rightarrow$})

$M_A[[\ \tau_1\ op\ \tau_2\ ]]\ L\ L'\ h\ R$ =

if $M_A[[\ \tau_1\ ]]\ L$ null h R = σ_1 and $M_A[[\ \tau_2\ ]]\ L$ null h R = σ_2 then [[σ_1 op σ_2]]

else error

$M_A[[\ \tau_1 \cap \tau_2\ ]]\ L\ L'\ h\ R =$

if $M_A[[\ \tau_1\ ]]\ L\ L'\ h\ R = \sigma_1$ and $M_A[[\ \tau_2\ ]]\ L\ L'\ h\ R = \sigma_2$ then $[[\ \sigma_1\ op\ \sigma_2\ ]]$

else error

$M_A[[\ \mu v.\tau]]\ L\ L'\ h\ R =$ if $M_A[[\ \tau\ ]]\ L\ (cons(v,L'))\ h\ R = \sigma$ then $\mu v.\sigma$ else error

For Tlist

$M_T[[\ \tau\ ]]\ L\ L'\ h\ R\ =$ if $M_A[[\ \tau\ ]]\ L\ L'\ h\ R = \tau'$ then $<\tau'>$ else error

$M_T[[\ Lt_1, Lt_2\ ]]\ L\ L'\ h\ R\ =$

if $M_T[[\ Lt1\ ]]\ L\ L'\ h\ R = L_1$ and $M_T[[\ Lt_2\ ]]\ L\ L'\ h\ R = L_2$ then $Append(L_2, L_1)$

else error

For Gcall

$M_{Gc}[[\ X\ ]]\ R =$

if R<X,0>≠unmapped then <R<X,0>, null>

else error

$M_{Gc}[[\ X\ (\ Lt\ )\ ]]\ R =$

if R<X, Len(MakeTlist[[Lt]]))>≠unmapped then

<<X, Len(MakeTlist[[L]])>, MakeTlist[[Lt]]>

else error

$MakeTlist[[\ \tau\ ]]\ = <\tau>$

$MakeTlist[[\ Lt_1\ ,\ Lt_2\ ]]\ =\ Append(MakeTlist[[Lt_2]], MakeTlist[[Lt_1]])$

For Def

$M_D[[\ G = \tau\ ]]\gamma h R\ =$

if $M_G[[\ G\ ]] = <z, L>$ then

if Rz=unmapped then

if M_A[[τ]] L null h R = τ' then

<γ, h[ls.w (RenameList(L,τ', AppendAll(map FV s)), s) \z], R'>

where w = λ<x,t>λy. if IsNull x or IsNull y then t

else Sub(hd y, hd x, w<tl x,t> (tl y))

where R'=R[<<z,f>> \ z]

where f=λ<x,y>.Append(Merge(x,y), Merge(y,x))

else error

else error

else error

M_D[[X : τ = E]] γhR = if M_A[[τ]] null h R = τ' then

if Reduce(τ') = τ'' then

if M_E[[E]]γhR = <e, τ'''> then

if Below(τ''', τ'', R) then

<γ[<e,τ>/X], h, R>

else error

else error

else error

else error

M_D[[W]]γhR = if M_W[[W]]h = <<z_1,L_1,f_1>, <z_2,L_2,f_2>> then

if Weaker(f_1(L_1), f_2(L_2), null, R) then <γ,h,R'>

where R' = R[Cons(<z_2, λx.null>,R(z_1)) \ z_1]

else error

else error

M_D[[W when Lc]]γhR =

if M_W[[W]]h = <<z_1,L_1,f_1>, <z_2,L_2,f_2>> then

if M_C[[Lc]] L_1 L_2 = c then

if Weaker($f_1(L_1)$, $f_2(L_2)$, c <L_1,L_2>, R) then <γ,h,R'>

where R' = R[Cons(<z_1, c>, $R(z_2)) \setminus z_1$]

else error

else error

else error

M_D[[D_1 ; D_2]]γhR = if M_D[[D_1]]γhR = <γ',h',R'> then M_D[[D_2]]γ'h'R'

else error

For Ghead

M_G[[X]] = <<X, 0>, null>

M_G[[X (Lv)]] =

if M_V[[Lv]] = L then <<X, Len(L)>, L> else error

For Vlist

M_V[[v]] = < v >

M_V[[Lv_1, Lv_2]] =

if M_V[[Lv_1]] = L_1 and M_V[[Lv_2]] = L_2 then

if not HasCommon(L_1,L_2) then Append(L_2, L_1)

else error

else error

For Wrel

M_W[[$G_1 \leq G_2$]]h =

if M_G[[G_1]] = <z_1, L_1> and M_G[[G_2]] = <z_2, L_2> then

if not HasCommon(L_1, L_2) then

if $h(z_1) = f_1$ and $h(z_2) = f_2$ // f_1, f_2 in rep // then

$\langle\langle z_1, L_1, f_1\rangle, \langle z_2, L_2, f_2\rangle\rangle$

else error

else error

else error

For Clist

$M_C[[\ v_1 \leq v_2\ ]]\ L_1\ L_2 =$

if OnVlist(v_1, L_1) then // let m = FindPos(v_1, L_1) //

if OnVlist(v_2, L_2) then // let n = FindPos(v_2,L_2) //

$\lambda\langle x,y\rangle$. <Select m x, Select n y>

else error

else if OnVlist(v_2, L_1) then // let n = FindPos(v_2,L_1) //

if OnVlist(v_1, L_2) then // let m = FindPos(v_1,L_2) //

$\lambda\langle x,y\rangle$. <Select m y, Select n, x>

else error

else error

$M_C[[\ Lc_1\ ,\ Lc_2\ ]]\ L_1\ L_2\ =$

if $M_C[[\ Lc_1\ ]]\ L_1\ L_2 = c_1$ then

if $M_C[[\ Lc_2\ ]]\ L_1\ L_2 = c_2$ then

$\lambda\langle x,y\rangle$. Append($c_2\langle x,y\rangle$, $c_1\langle x,y\rangle$)

else error

else error

BRIEF DESCRIPTION OF MAIN AUXILIARY FUNCTIONS

Reduce: Atexp → (Atexp + Err)

// Reduce(α) rewrites subexpressions of α of the form σ→(τ∩ρ) as (σ→τ)∩(σ→ρ), provided μ-expressions remain reduced under unrolling, else error //

Below: Atexp × Atexp × Rules → Bool

// Below(σ,τ, R) checks if $\sigma <<_\mu \tau$, using the $\leq_\mu$-rules augmented with R //

Weaker: Atexp × Atexp × (Atexp × Atexp)* × Rules → Bool

// Computes an extension of Wμ //

Rename: Atexp × Tvar* → Atexp

// Rename(α,L) renames free variables in α to be different than those on L //

RenameList: Tvar* × Atexp × Tvar* → Tvar* × Atexp

// RenameList(L,τ,L') renames variables on L and in τ which appear on L' //

Explicit: Exp → Bool

// Explicit(e) = true if e of the form $(f::\tau)\ e_1\ e_2 \ldots e_n$, else false //

WhichBelow: Atexp × Atexp × Rules → (Atexp + Err)

// WhichBelow(α,β,R) returns the intersection of all τ such that σ→τ is a term of α, and such that β is below σ ... if no such τ exists, then error //

Infer: Atexp × Atexp × Rules → (Atexp + Err)

// Infer(α,β,R) infers a type for a non-explicitly typed expressions ef, where e:α and f:β, using an extension of UW to accommodate type generators, × and + //

OnVlist: Tvar × Tvar* → Bool

// tests if the type variable is on the list of type variables //

HasCommon: Tvar* × Tvar* → Bool

// true if there is at least one variable on both lists //

FindPos: Tvar × Tvar* → Nat

// FindPos(v,L) = first position that v occurs in L (starting with hd(L) = position 0) //

Select: $\forall A.\ Nat \rightarrow A^* \rightarrow A$

// Select n L returns element on L in the n'th position, starting with 0 //

Len: $\forall A.\ A^* \rightarrow N$

// Len(L) = the length of the sequence L //

Append: $\forall A.\ A^* \times A^* \rightarrow A^*$

// Append($\langle a_1,...,a_m\rangle$, $\langle b_1,...,b_n\rangle$) = $\langle b_1,...,\ b_n, a_1, ...,\ a_m\rangle$ //

Merge: $\forall A.\forall B.\ A^* \times B^* \rightarrow (A \times B)^*$

// Merge($\langle a_1,...,a_m\rangle$, $\langle b_1,...,b_n\rangle$) = if m=n then $\langle\langle a_1,b_1\rangle,...,\langle a_n,b_n\rangle\rangle$,

else undefined //

We add one more semantic function, M_P: Exp $\rightarrow$ (D $\times$ Atexp + Err), to denote programs. M_P simply applies M_E to its argument, while supplying the initial environment, representations and rules.

$M_P[[\ E\ ]] = M_E[[\ E\ ]]\gamma_0 h_0 R_0$ where

$$\begin{aligned}\gamma_0 = (\lambda X.\text{unbound})\ &[\langle OP[[+]],\ INT \times INT \rightarrow INT\rangle \setminus \text{add}\ ]\\ &[\langle OP[[-]],\ INT \times INT \rightarrow INT\rangle \setminus \text{sub}]\\ &[\langle OP[[*]],\ INT \times INT \rightarrow INT\rangle \setminus \text{times}]\\ &[\langle OP[[/]],\ INT \times NONZERO \rightarrow INT\rangle \setminus \text{div}]\\ &[\langle OP[[\text{is0}]],\ (0 \rightarrow TRUE) \cap (NONZERO \rightarrow FALSE)\\ &\qquad \cap (INT \rightarrow BOOL)\rangle \setminus \text{is0}]\\ &[\langle OP[[\text{eq}]],\ INT \times INT \rightarrow BOOL\rangle \setminus \text{eq}]\\ &[\langle OP[[\text{less}]],\ INT \times INT \rightarrow BOOL\rangle \setminus \text{less}]\\ &[\langle M_N[[0]],\ FALSE\rangle \setminus \text{false}]\end{aligned}$$

$[<M_N[[1]], TRUE> \setminus true]$

$[<\lambda x\lambda y\lambda z.xz(yz), (a\to b\to c)\to(a\to b)\to a\to c> \setminus S]$

$[<\lambda x\lambda y\lambda z.<xz,yz>, (a\to b) \to (a\to c)\to a\to(b \times c)> \setminus C]$

$[<\lambda x\lambda y.x, a\to b\to a> \setminus K]$

$[<\lambda x.x, a\to a> \setminus I]$

$[<\lambda x\lambda y\lambda z.x(yz), (b\to c)\to(a\to b)\to a\to c> \setminus B]$

$[<\lambda x.(\lambda y.x(yy))(\lambda y.x(yy)), (a\to a)\to a> \setminus Y]$

$[<\lambda x\lambda y.<x,y>, a\to b\to(a \times b)> \setminus pair]$

$[<\lambda<x,y>.x, (a \times b)\to a> \setminus p1]$

$[<\lambda<x,y>.y, (a \times b)\to b> \setminus p2]$

$[<\lambda x.<1,x>, a\to(a + \Delta)> \setminus inL]$

$[<\lambda x.<2,x>, a\to(\Delta + a)> \setminus inR]$

$[<\lambda<1,x>.x, (a + \Delta) \to a> \setminus outL]$

$[<\lambda l<2,x>.x, (\Delta + a) \to a> \setminus outR]$

$[<\lambda<x,y>.$if x=1 then true else false,

$((a + \Delta)\to TRUE) \cap ((\Delta + a)\to FALSE)$

$\cap ((a + b) \to BOOL) > \setminus isL]$

$[<\lambda<x,y>.$ if x=2 then true else false,

$((a + \Delta)\to FALSE) \cap ((\Delta + a)\to TRUE)$

$\cap ((a + b) \to BOOL) > \setminus isR]$

$[<\lambda x\lambda y\lambda z.$ if x=1 then y else z,

$(TRUE\to b\to c\to b) \cap (FALSE\to b\to c\to c)$

$\cap (BOOL\to a\to a\to a) > \setminus if]$

$h_0 = \lambda z.$notconcrete

R_0 = // let $c_0 = \lambda$<x,y>.null //

(λz.unmapped) [<<<0,0>, c_0>, <<INT,0>, c_0>> \ <0,0>]

[<<<NONZERO,0>, c_0>, <<INT,0>, c_0>>

\ <NONZERO,0>]

[<<<INT,0>, c_0>> \ <INT,0>]

[<<<TRUE,0>, c_0>, <<BOOL,0>, c_0>> \ <TRUE,0>]

[<<<FALSE,0>, c_0>, <<BOOL,0>, c_0>> \ <FALSE,0>]

[<<<BOOL,0>, c_0>> \ <BOOL,0>]

6.4 A sample program

In this section, we present a program written in the language L. The program illustrates the use of conjunctive types, generic lists, and the explicit typing mechanism. The main function in the program is called "DoubleSort" which takes two lists of arbitrary type and a polymorphic sorting function as arguments and returns the two lists in sorted order. The program applies DoubleSort to a list of integers and a list of integer pairs. DoubleSort is untypeable in L without the use of explicit types, and hence is untypeable using parametric types.

Before giving the program in its entirety, we will give its parts with some explanation. The declarations for abstract lists are first. (Recall that the order of precedence among the type operators is, from greatest to weakest, $\times$, +, $\rightarrow$ and $\cap$.)

```
/* Type definitions */
        NilType = BOOL ;
```

List(a) = μs. NilType + (a × s);

MTL = NilType + Δ;

NonMTL(a) = Δ + (a × List(a));

/* Constant definitions */

Nil : NilType = false;

NilList: MTL = InL Nil;

hd : NonMTL(a) → a = B P1 OutR;

tl : NonMTL(a) → List(a) = B P2 OutR;

cons : a × List(a) → NonMTL(a) = InR;

IsNil : List(a) → BOOL = IsL;

/* Rule definitions */

List(a) ≤ List(b) when a ≤ b;

MTL ≤ List(a);

NonMTL(a) ≤ List(b) when a ≤ b;

NonMTL(a) ≤ NonMTL(b) when a ≤ b

According to the semantics of L, the representations of the types List, MTL, NonMTL and NilType are used in checking the type claims of the constants. For example, the type inferred for cons is the axiomatic type of InR, namely $x \rightarrow (\Delta + x)$, which is below the claimed type of cons, using the representations for List(a) and NonMTL(a). Also, each rule definition is checked using the representations of the types involved and the "when" conditions as axioms. For example, to check the first rule, we would see if $\mu s.BOOL + (a \times s) \leq_\mu \mu s.BOOL + (b \times s)$ given that $a \leq_\mu b$ (and it is).

We intend that the representations of the List types above be hidden from the rest of the program, and that only the claimed types of the constants and the type rules be used in type

checking subsequent type claims. Hence, our program takes the form { D in E } where D denotes the declarations above.

Our program uses a generic sort function called "gsort" which performs an insertion sort given a list and a binary comparison function. Gsort calls "insert" which performs the insertion operation. We first give insert and gsort as recursive lambda expressions, and then convert them to valid expressions in L.

```
insert: (a × List(a)) × ( a × a → BOOL) → List(a)
      = λ<<x,L>,f> . if (IsNil L)  (cons <x,L>)
                                ( if (f <x, hd L>) (cons <x,L>)
                                      (cons <hd L, insert <<x, tl L>, F>>) )

gsort: List(a) × ( a × a → BOOL ) → List(a)
      = λ<L,f>.  if (IsNil L) L
                        (insert <<hd L, gsort <tl L, f>>, f>)
```

Since L has neither lambda abstraction nor recursive definitions, we must factor the functions above and eliminate recursion via the Y combinator. Factoring can be done by extending the algorithm given in chapter 3 to accommodate pairs. This is accomplished using the identity $\lambda x.\langle A,B\rangle = C\ (\lambda x.A)\ (\lambda x.B)$. Now insert becomes

```
insert =
      S  ( S  ( B if (B IsNil (B P2 P1)))  (B cons P1) )
         ( S ( S  ( S  (K if)  (S P2  (C (B P1 P1) (B hd (B P2 P1)))) )
                  (B cons P1)
```

```
        )
        ( S  (K cons)
             ( C  (B hd (B P2 P1))
                  ( S (K insert) ( C (C (B P1 P1) (B tl (B P2 P1)))  P2 ) )
    )    )      )
```

To eliminate the recursion, we convert the above definition of the form

insert = F(insert)

into Y(λx.F(x)), and then factor λx.F(x) into combinators. This gives the final form for insert:

```
insert =
  Y ( S ( K (S ( S ( B if (B IsNil (B P2 P1))) (B cons P1) ) ) )
       ( S ( K ( S (S (S (K if) (S P2 (C (B P1 P1) (B hd (B P2 P1)))) )
                    (B cons P1) ) ) )
          (S (K (S (K cons)))
             ( S (K ( C (B hd (B P2 P1))))
                  (S B (K (C (C (B P1 P1) (B tl (B P2 P1))) P2 ) ) )
      ) )  )   )
```

In a similar fashion, one obtains the expression in L for gsort:

```
gsort =
  Y( S ( K ( S ( S ( B if (B IsNil P1)) P1) ) )
      (S (K (S ( K insert) ) )
         (  S ( S (K C)  (S (K (C (B hd P1))) (S B (K ( C (B tl P1) P2 )))))
```

(K P2)

)))

It can be verified by a long calculation that the type of insert in L is below the claimed type (a × List(a)) × (a × a → BOOL) → List(a), and that the type of gsort in L is below its claimed type List(a) × (a × a → BOOL) → List(a), using the claimed type for insert. (Since insert and gsort are typeable using only parametric types, the type inference is routine and is therefore omitted.)

Our main function is DoubleSort. Here is the lambda expression for this function:

DoubleSort: $(a \times b) \times ((a \to c) \cap (b \to d)) \to c \times d$

= λx. <P2 x (P1 (P1 x)), P2 x (P2 (P1 x))>

which has factorization C (S P2 (B P1 P1)) (S P2 (B P2 P1)). The claimed conjunctive type for DoubleSort is necessary to type expressions of the form DoubleSort <e,f> where e and f do not have compatible types. The inferred type for the factored version of DoubleSort, however, has type $(a \times a) \times (a \to b) \to b \times b$, which is not below the claimed type, hence the above expression is not valid in L.

The problem above disappears if we rewrite DoubleSort with an appropriate explicit type for the C combinator. Recall that the axiomatic type for C is $(x \to y) \to (x \to z) \to x \to (y \times z)$. Let P be the substitution [x := $(a \times b) \times ((a \to c) \cap (b \to d))$, y:= c, z:= d], and let τ = P(Ax(C)), i.e.,

$$\tau = ((a \times b) \times ((a \to c) \cap (b \to d)) \to c) \to ((a \times b) \times ((a \to c) \cap (b \to d)) \to d) \to (a \times b) \times ((a \to c) \cap (b \to d)) \to c \times d$$

Certainly τ is a valid explicit type for C. The expression (S P2 (B P1 P1)) has (parametric) type $(a\times b)\times(a\to c)\to c$ which is below $(a \times b) \times ((a\to c)\cap(b\to d)) \to c$, and (S P2 (B P1 P1)) has type $(a\times b)\times(b\to d)\to d$ which is below $(a \times b) \times ((a\to c)\cap(b\to d)) \to d$. This means that $(C::\tau)$ applied to (S P2 (B P1 P1)) and then applied to (S P2 (B P1 P1)) has type $(a \times b) \times ((a\to c)\cap(b\to d)) \to c \times d$ in L, which is the claimed type. Thus, the expression for DoubleSort becomes

$$(C :: ((a \times b) \times ((a\to c)\cap(b\to d)) \to c) \to ((a \times b) \times ((a\to c)\cap(b\to d)) \to d)$$
$$\to ((a \times b) \times ((a\to c)\cap(b\to d)) \to c \times d)$$
$$(S\ P2\ (B\ P1\ P1)) \quad (S\ P2\ (B\ P2\ P1))$$

Our program calls DoubleSort to sort a list of integers and a list of pairs of integers. We define the comparison function for the list of pairs of integers in terms of the primitive functions "less" and "eq" having type $INT \times INT \to BOOL$ (the ordering is the lexicographic ordering).

```
pless: (INT × INT) × (INT × INT) → BOOL
    = λx. if (less <P1(P1 x), P1(P2 x)>) true
            (if (eq <P1(P1 x), P1(P2 x)>) (less <P2(P1 x), P2(P2 x)>) false ) )
```

Pless has the following factorization:

```
S ( S ( S (K if) (C (B P1 P1) (B P1 P2)) ) (K true) )
  ( S ( S ( S (K if) (C (B P1 P1) (B P1 P2)) )
          ( S (K less) ( C (B P2 P1) (B P2 P2)) )
      ) (K false)
  )
```

Below is the program in its entirety.

```
{  /*  Type definitions */
        NilType = BOOL ;
        List(a) = μs. NilType + (a × s);
        MTL = NilType + Δ;
        NonMTL(a) = Δ + (a × List(a));
   /* Constant definitions */
        Nil : NilType = false;
        NilList: MTL  =  InL Nil;
        hd : NonMTL(a) → a  =  B P1 OutR;
        tl : NonMTL(a) → List(a)  =  B P2 OutR;
        cons : a × List(a) → NonMTL(a)  =  InR;
        IsNil : List(a) → BOOL  =  IsL;
   /* Rule definitions */
        List(a) ≤ List(b)  when  a ≤ b;
        MTL ≤ List(a);
        NonMTL(a) ≤ List(b)  when  a ≤ b;
        NonMTL(a) ≤ NonMTL(b)  when  a ≤ b
in  {

        insert: (a × List(a)) × ( a × a → BOOL)  →  List(a)  =
        Y ( S ( K (S ( S ( B if (B IsNil (B P2 P1))) (B cons P1) ) ) )
            ( S ( K ( S (S (S (K if) (S P2 (C (B P1 P1) (B hd (B P2 P1))))) )
                        (B cons P1) ) ) )
```

```
        (S (K (S (K cons)))
           ( S (K ( C (B hd (B P2 P1))))
              (S B (K (C (C (B P1 P1) (B tl (B P2 P1))) P2 ) ) )
        ) ) ) )
in {
    gsort: List(a) × ( a × a → BOOL ) → List(a) =
    Y( S  ( K ( S ( S ( B if (B IsNil P1)) P1) ) )
          (S (K (S ( K insert) ) )
          ( S  ( S (K C) (S (K (C (B hd P1))) (S B (K ( C (B tl P1) P2 )))))
                ( K P2 )
      )  ) )
    in {
          DoubleSort: (a × b) × ((a→c)∩(b→d)) → c × d =
                (C :: ((a × b) × ((a→c)∩(b→d)) → c)
                      → ((a × b) × ((a→c)∩(b→d)) → d)
                            → ((a × b) × ((a→c)∩(b→d)) → c × d )
                      (S P2 (B P1 P1))   (S P2 (B P2 P1));
          pless: (INT × INT) × (INT × INT) → BOOL =
              S ( S ( S (K if) (C (B P1 P1) (B P1 P2)) ) (K true) )
                ( S ( S ( S (K if) (C (B P1 P1) (B P1 P2)) )
                        ( S (K less) ( C (B P2 P1) (B P2 P2)) )
                      ) (K false)
                );
          List1 : List(INT) =
                cons <8, cons <0, cons < 4, cons<150, cons <2, NilList>>>>>;
          List2 : List(INT x INT) =
```

```
                cons < <4,0>, cons < <2,155>, cons < <13,13>,
                        cons < <6,15>, cons < <0,99>, NilList>>>>>
            in
                DoubleSort <<<List1, less>, <List2, pless>>, gsort>
}   }   } }
```

7

Summary and Future Research

An expressive type language and the ability to do compile-time type inference are desirable goals in language design, but the attainment of the former may preclude the possibility of the latter. Specifically, the type conjunction operator induces a rich type language at the expense of decidability of the typeable expressions. Two extreme alternatives to this dilemma are to abandon type inference (and force the programmer to, essentially, supply a derivation for his type claims) or to abandon type conjunction in its purest form. This work presents a third alternative in which the programmer, at times, may be required to supply explicit types in order for type inference to succeed. In this way, the power of conjunctive types is preserved, yet limited type inference can be done at compile time.

We introduced a simple language, TCL, having a computational language of combinators and a type language based on type conjunction and a subtype relation, of sorts, called "weaker." The validity of the type language with respect to the usual interpretation of "type" was shown in chapter 2, and the undecidability of the type relation was shown in chapter 3. In chapter 4, the computational portion of TCL was modified to accommodate explicit type information which directed a type derivation. We showed that this new language, XTCL, has the principal type property with respect to a new relation called "below" (again, a subtype relation), but (regrettably) that deciding the "below" relation is an NP-Complete problem. A type-checking algorithm for XTCL was given and shown to be correct. In chapter 5, the type-language portion of XTCL was extended to allow certain recursive type expressions, and the algorithms from chapter 4 were extended and shown to be correct. In chapter 6, we discussed an extension which allows all expressions with parametric types to be typed

automatically, and we proposed a language which, in addition, accommodates integers, pairs, sums and abstract types in the form of type generators.

It remains to be seen whether explicit types easily accommodate extensions to the computational language, such as allowing lambda abstraction and mutually recursive function definitions, as well as extensions to the type language, such as allowing type quantification. For example, can we add Π-quantified types to the language of chapter 6 so that no expression having a *second-order* type need be explicitly typed?

It remains to be proven that type checking in TCLμ is undecidable. The method used in chapter 3, of showing that the set of typeable expressions is undecidable does not work, since the typeable expressions in TCLμ form a decidable set. Perhaps it could be shown that the set of expressions having a particular type, such as $x \rightarrow x$, is undecidable, and hence that type checking in general is undecidable.

In XTCLμ, the restriction that μ-expressions appearing in explicit types must be reducible may not be necessary. Precisely, can one show that $\alpha << _{\mu} \beta$ is decidable when β may have an infinite number of terms when $\rightarrow$ is distributed over $\cap$?

On the practical side, the language of chapter 6 could be improved in several ways:

a) Drop the restriction that type generators can not be redefined. This could be done by associating a type generator with a set of operations pertaining to it, and making sure that two types do not match unless the associated operations match as well.

b) Have the type checker transitively close weaker rules. As it is now, if the programmer gives the two rules

$$F(a) \leq G(b) \text{ when } a \leq b;$$

$$G(b) \leq H(c) \text{ when } b \leq c$$

the type checker does not infer that $F(a) \leq H(c)$ when $a \leq c$, even though the inference is valid. The general case is when there are 2 rules of the form

$$F(p_1,...,p_m) \leq G(q_1,...,q_n) \text{ when } conds_1 ;$$

$$G(q_1,...,q_n) \leq H(r_1,...,r_k) \text{ when } conds_2$$

An obvious thing to do is to produce a rule

$$F(p_1,...,p_m) \leq H(r_1,...,r_k) \text{ when } conds_3$$

where $conds_3$ is derived by transitively closing $conds_1 \cup conds_2$ and selecting the conditions relevant to the p's and r's. However, there may be conditions of the form $z_1 \leq q_i$ & $z_2 \leq q_i$ &...& $z_h \leq q_i$, for some q_i not related to any r_j, whose satisfiability must be expressed as the existence of an upper bound of the z's. Do we add an operator HasUpperBound to the syntax of the weaker conditions?

c) There is no way to specify that a given type generator distributes over intersection in a given argument position (as $\rightarrow$, + and $\times$ do), even though this may be a valid rule. It would be nice if the programmer could specify the rule

$$F(a, b \cap c) \leq F(a,b) \cap F(a,c),$$

or more generally

$$F(a, G(b,c)) \leq G(F(a,b), F(a,c)).$$

d) Allow lambda abstraction and recursive function definitions

In order to properly assess the practical value of the ideas detailed herein, an implementation of the language described in chapter 6 is currently underway.

References

[ASU85] Aho, A., Sethi, R. and Ullman, J., *Compilers, principles, techniques, and tools*, Addison-Wesley, Reading, Mass., pp 376--379 (1985).

[Back78] Backus, J., *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*, CACM, vol 21, pp 613--641 (1978).

[Bar85] Barendregt, H., *The lambda calculus: Its syntax and semantics*, Studies in Logic, North Holland, New York (Second edition, 1985).

[B&K82] Bergstra, J. and Klop, J., *Strong normalization and perpetual reductions in the lambda calculus*, Elektronische Informationsverarbeitung und Kybernetik, 18 (1982), pp 403--417.

[B&L84] Burstall, R. and Lampson, B., *A kernel language for modules and abstract data types*, Symposium on Semantics of Data Types, Sophia-Antipolis, LNCS 173, Springer-Verlag (1984).

[BMS80] Burstall, R., MacQueen, D. and Sanella, D., *Hope: an experimental applicative language*, LISP Conference, Stanford, pp 136--143 (Aug. 1980)

[Car86] Cardelli, L., *A polymorphic λ-calculus with type:type*, Report 10, Digital Corp., Systems Research Center, Palo Alto, Cal., 27 pages (1986).

[C&F58] Curry, H. and Feys, R., *Combinatory logic vol. I*, North-Holland, Amsterdam (1958).

[Chu40] Church, A., *A formulation of the simple theory of types*, J. Symb. Logic, vol. 5, pp 56--68 (1940).

[Chu41] Church, A., *The calculi of lambda conversion*, Annals of Math. Studies, No. 6, Princeton University Press, Princeton, NJ (1941).

[Cop80] Coppo, M., *An extended polymorphic type system for applicative languages*, Ninth Symposium on Mathematical Foundations of Computer Science, Rydzna, Poland, LNCS 88, Springer-Verlag, pp 194--208 (1980).

[Cop80a] Coppo, M., Denzani-Ciancaglini, M., Venneri, B., *Principal type schemes and lambda calculus semantics*, (1980) in [S&H80].

[D&D80] Demers, A. and Donahue, J., *Data types, parameters and type checking*, Seventh ACM Symp. on POPL, pp 12--23 (1980).

[Gandy80] Gandy, *An early proof of normalization*, (1980) in [S&H80].

[Gir72] Girard, J-Y, *Interpretation functionaelle et elimination des coupures dans l'arithmetique d'ordre superieur*, These de Doctorat d'etat, University of Paris (1972).

[GMetc78] Gordon, M., Milner, R., Morris, L., Newey, M and Wadsworth, C.,

A metalanguage for interactive proof in LCF, Fifth Annual ACM SIGACT-SIGPLAN Symp. on POPL, Tucson, Ariz. (1978).

[H&U79] Hopcroft, J., Ullman, J., *Introduction to automata theory, languages, and computation*, Addison-Wesley, Reading, Mass. (1979)

[Hind69] Hindley, R., *The principal type-scheme of an object in combinatory logic*, Trans. AMS, vol. 146, pp 29--60 (1969).

[Knuth69] Knuth, D., *The art of computer programming, Vol. 1*, Addison-Wesley, Reading, Mass. (1969)

[Martin86] Martin, J. J., *Data types and data structures*, C. A. R. Hoare Computer Science Series, Prentice-Hall (1986).

[McC79] McCracken, N., *An investigation of a programming language with a polymorphic type structure*, Ph.D. thesis, Computer and Inform. Science., Syracuse University (June 1979).

[McCar58] McCarthy, J., *An algebraic language for the manipulation of symbolic expressions*, MIT AI Lab, AI memo no. 1, Cambridge, Mass. (Sept 1958).

[Mee83] Meertens, L., *Incremental polymorphic type checking in B*, Tenth ACM Symp. on POPL, pp 265--275 (1983).

[Mil78] Milner, R., *A theory of type polymorphisn in programming*, JCSS, 17,

pp 348--375 (1978).

[M&O84] Mycroft, A. and O'Keefe, R., *A polymorphic type system for PROLOG*, DAI research report, Dept. of Artificial Intelligence, Edinburgh U. (1984)

[Mor68] Morris, J., *Lambda calculus model of programming languages*, Ph.D. thesis, MIT, Cambridge, Mass. (1968).

[MPS84] MacQueen, D., Plotkin, G. and Sethi, R., *An ideal model for recursive polymorphic types*, Eleventh ACM Symp. on POPL, pp 165--174 (1984).

[MS82] MacQueen, D. and Sethi, R., *A higher-order polymorphic type system for applicative languages*, Sympos. on LISP and Functional Programming, Pittsburgh, Pa., pp 243--252 (1982).

[Myc84] Mycroft, A. *Polymorphic type schemes and recursive definitions*, Intern. Symp. on Programming, Sixth Colloq., Toulouse, LNCS 167, Springer-Verlag, pp 217--228 (1984).

[Rey74] Reynolds, J., *Towards a theory of type structure*, Programming Symp., Paris, LNCS 19, Springer-Verlag (1974).

[Rob65] Robinson, J., *A machine-oriented logic based on the resolution principle*, JACM, vol 12, no. 1, pp 23--41 (1965).

[Russel] Russel, B., *Introduction to mathematical philosophy*, Simon & Schuster.

[Scott76] Scott, D., *Data types as lattices*, SIAM Journal on Computing, 5, 3, pp 522--587 (1976).

[Scott82] Scott, D., *Domains for denotational semantics*,

[S&H80] Seldin, J., Hindley, J. (editors), *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism* , Academic Press, London (1980).

[S&W77] Shamir, A., Wadge, Data types as objects, Proceedings of the Fourth ICALP, Turku, LNCS 52 (1977).

[Stoy77] Stoy, J., Denotational semantics: The Scott-Strachey approach to programming language theory, MIT Press, Cambridge, Mass. (1977).

[Str67] Strachey, C., *Fundamental concepts in programming languages*, Notes for the International Summer School in Computer Programming, Copenhagen (1967).

[Ten75] Tennent, R.D., PASQUAL: A proposed generalization of PASCAL,Tech. Report, Dept. of Computing and Information Science, Queens College, NY (1975).

[Wan84] Wand, M., *Personnal communication to D. MacQueen* (1984).

[Wan87] Wand, M., *Personnal communication* (Oct. 1987).

[Win86] Winskel, *On powerdomains and modality*, LNCS, 1986.

Appendix

A.1 Properties of $\leq$

DEFINITION Let $\alpha,\beta \in$ Texp. Then $\alpha \leq \beta$ iff one of the following:

i) β atomic, β a term of α

ii) $\beta = \beta_1 \rightarrow \beta_2$, $\exists$terms $\alpha_i = \sigma_i \rightarrow \tau_i$ of α (i in some index set I) such that

$$\beta_1 \leq \cap\{\sigma_i \mid i \in I\}, \text{ and } \cap\{\tau_i \mid i \in I\} \leq \beta_2$$

iii) $\beta = \beta_1 \cap \beta_2$, $\alpha \leq \beta_1$ and $\alpha \leq \beta_2$

PROPERTY 1 ($\leq$ is reflexive) For all $\alpha \in$ Texp, $\alpha \leq \alpha$.

proof Induct on $|\alpha|$. Base is trivial, since every atom is a term of itself. Suppose $\alpha = \alpha_1 \rightarrow \alpha_2$. Then by hypothesis, $\alpha_1 \leq \alpha_1$ and $\alpha_2 \leq \alpha_2$ implying $\alpha \leq \alpha$. Now suppose $\alpha_1, \alpha_2, \ldots, \alpha_n$ are the terms of α, $n>1$. By hypothesis, $\alpha_i \leq \alpha_i$ for each i. Certainly $\alpha_i \cap t \leq \alpha_i$ (by a quick induction) for any t, hence $\alpha \leq \alpha_i$ for each i, implying $\alpha \leq \alpha$ by part iii of the definition of $\leq$. $\Diamond$

PROPERTY 2 ($\leq$ is transitive) For all $\alpha,\beta,\gamma \in$ Texp, $\alpha \leq \beta \leq \gamma$ implies $\alpha \leq \gamma$.

proof Induct on $|\alpha| + |\beta| + |\gamma|$. Base is covered when β is atomic, in which case all terms of γ are equal to β, and β a term of α implies all terms of γ are a term of α, hence $\alpha \leq \gamma$. Suppose $\gamma = \gamma_1 \cap \gamma_2$. Then $\alpha \leq \beta \leq \gamma_1$ and $\alpha \leq \beta \leq \gamma_2$ implies (by hypothesis) $\alpha \leq \gamma_1$ and $\alpha \leq \gamma_2$, which implies $\alpha \leq \gamma_1 \cap \gamma_2$. Suppose γ=t an atom. Then t is a term of β, and (since $\alpha \leq \beta$) t is a term of α, thus $\alpha \leq \gamma$. Suppose $\gamma = \gamma_1 \rightarrow \gamma_2$. Consider when $\beta = \beta_1 \rightarrow \beta_2$. Then $\alpha \leq \beta$ implies there are terms $\sigma_i \rightarrow \tau_i$ of a ($i \in I$) such that $\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \beta_2$, and $\beta \leq \gamma$ implies $\gamma_1 \leq \beta_1$ and $\beta_2 \leq \gamma_1$. Hence (by hypothesis) $\gamma_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \gamma_2$ implies $\alpha \leq \gamma$. The remaining case is when β is an intersection (and $\gamma = \gamma_1 \rightarrow \gamma_2$).

Let J index the terms $\rho_j \to \eta_j$ of β such that $\gamma_1 \leq \cap\{\rho_j \mid j \in J\}$ and $\cap\{\eta_j \mid j \in J\} \leq \gamma_2$. For each $j \in J$, let I_j index the terms $\sigma_i \to \tau_i$ of α such that $\rho_j \leq \cap\{\sigma_i \mid i \in I_j\}$ and $\cap\{\tau_i \mid i \in I_j\} \leq \eta_j$. Certainly for all $j \in J$, $\gamma_1 \leq \cap\{\sigma_i \mid i \in I_j\}$ (by hypothesis). Let $I = \cup\{I_j \mid j \in J\}$. Then $\gamma_1 \leq \cap\{\sigma_i \mid i \in I\}$ by the definition (iii). Also $\cap\{\tau_i \mid i \in I\} \leq \cap\{\eta_j \mid j \in J\} \leq \gamma_2$, which implies (by hypothesis) $\cap\{\tau_i \mid i \in I\} \leq \gamma_2$, thus $\alpha \leq \gamma_1 \to \gamma_2$. ◊

PROPERTY 3 ($\leq$ is substitution invariant) For all $\alpha, \beta \in$ Texp, and for all substitutions P of type expressions for type variables, $\alpha \leq \beta$ implies $P\alpha \leq P\beta$.

proof Induct on $|\alpha| + |\beta|$. The base case is covered when β is atomic, then β a term of α implies $\alpha = \beta \cap \gamma$ (some γ). Certainly $P\beta \leq P\beta$ and $P\beta \cap P\gamma \leq P\beta$, thus $P\alpha \leq P\beta$. Now suppose $\beta = \beta_1 \to \beta_2$. Then let I index $\sigma_i \to \tau_i$, terms of α, such that $\beta_1 \leq \cap\{\sigma_i \mid i \in I\}$ and $\cap\{\tau_i \mid i \in I\} \leq \beta_2$. By hypothesis, $P\beta_1 \leq \cap\{P\sigma_i \mid i \in I\}$ and $\cap\{P\tau_i \mid i \in I\} \leq P\beta_2$, hence $P\alpha \leq P\beta_1 \to P\beta_2$. Suppose $\beta = \beta_1 \cap \beta_2$. Then $\alpha \leq \beta_1$ and $\alpha \leq \beta_2$ implies (by hypothesis) $P\alpha \leq P\beta_1$ and $P\alpha \leq P\beta_2$, hence $P\alpha \leq P(\beta_1 \cap \beta_2)$. ◊

A.2 Miscellaneous Proofs

LEMMA 5.2.1 (MacQueen, et. al, from page 109). Every Cauchy sequence in $\langle T', d \rangle$ converges to a unique limit.

proof Let $\langle A_i \rangle_i = A_1, \ldots$ be a Cauchy sequence. Call a finite element a *tenured* in the sequence $\langle A_i \rangle_i$ if for some k, $a \in A_i$ for all $i > k$. Let L be the set of all tenured elements in $\langle A_i \rangle_i$. Note that since each A_i is an ideal (i.e., is downward closed), so is L. Pick any $M > 0$. Let k be such that for all A_i, A_j beyond A_k, $C(A_i, A_j) > M$. Pick A_i beyond A_k, and let $a \in L \Delta A_i$. Case 1, $a \in L \setminus A_i$, then there exists h such that for all A_j beyond A_h, $a \in A_j \Rightarrow \exists i,j > k$ such that $a \in A_i \Delta A_j$, $\Rightarrow r(a) > M$. Case 2, $a \in A_i \setminus L$. Assume $r(a) \leq M$, then there is no $j > k$

such that $a \in A_i \Delta A_j$, that is, $a \in A_j$ for all $j>k$, implying $a \in L$, a contradiction, thus $r(a) > M$. In both cases, $a \in L \Delta A_i \Rightarrow r(a) > M$, hence for all $i>k$, $C(L,A_i) > M$. This shows L is a limit of $\langle A_i \rangle_i$. By the triangle property of d, it is easy to see that any other limit L' of $\langle A_i \rangle_i$ must be 0 distance from L, thus $L = L'$ $\diamond$

LEMMA 5.2.2 (MacQueen, et. al., from page 109). Let A_1, A_2, B_1 and B_2 be ideals.

i) $C(A_1 \to A_2, B_1 \to B_2) > MIN(\, C(A_1,B_1), C(A_2,B_2)\,)$ provided $A_1 \to A_2 \neq B_1 \to B_2$

ii) $C(A_1 \cap A_2, B_1 \cap B_2) \geq MIN(\, C(A_1,B_1), C(A_2,B_2)\,)$

iii) $C(\cap\{A_i \mid i \in I\}, \cap\{B_i \mid i \in I\}) \geq MIN\{\, C(A_i,B_i) \mid i \in I\,\}$

proof

i) Let $w1 = C(A_1,B_1)$, $w2 = C(A_2,B_2)$. Show $C(A_1 \to A_2, B_1 \to B_2) > MIN(w_1, w_2)$. Pick $f \in A_1 \to A_2 \,\Delta\, B_1 \to B_2$ of minimal rank. Case 1, $f \in A_1 \to A_2 \setminus B_1 \to B_2$. Note, the domain of f must intersect with B_1, else $f \in B_1 \to B_2$, so pick $e \in dom(f) \cap B_1$. If $e \notin A_1$ then $r(e) \geq w_1$, and by assumption $r(f) > r(e) > w_1$. If $e \in A_1$ then $fe \in A_2 \setminus B_2 \Rightarrow r(f) > r(fe) \geq w_2$. Thus $r(f) > MIN(w_1,w_2)$. Case 2, $f \in B_1 \to B_2 \setminus A_1 \to A_2$. This time, pick $e \in dom(f) \cap A_1$. If $e \notin B_1$ then $r(f) > r(e) \geq w_1$, and if $e \in B_1$ then $fe \in B_2 \setminus A_2 \Rightarrow r(f) > r(fe) \geq w_2$.

ii) Since $(A_1 \cap A_2) \,\Delta\, (B_1 \cap B_2) \subseteq (A_1 \Delta B_1) \cup (A_2 \Delta B_2)$, $MIN\{\, r(e) \mid e \in (A_1 \cap A_2) \Delta (B_1 \cap B_2)\,\} \geq MIN(\, MIN\{\, r(e) \mid e \in A_1 \Delta B_1\,\}, MIN\{\, r(e) \mid e \in A_2 \Delta B_2\,\}\,)$.

iii) As in ii), $\cap(\{A_i \mid i \in I\}) \,\Delta\, (\cap\{B_i \mid i \in I\}) \subseteq \cup\{A_i \setminus B_i,\, B_i \setminus A_i \mid i \in I\} = \cup\{A_i \Delta B_i \mid i \in I\}$, thus $MIN\{\, r(e) \mid e \in (\cap\{A_i \mid i \in I\}) \,\Delta\, (\cap\{B_i \mid i \in I\})\,\} \geq MIN\{\, MIN\{\, r(e) \mid e \in A_i \,\Delta\, B_i\,\} \mid i \in I\,\}$. $\diamond$

A.3 Auxiliary Functions for L

null : A* // the 0-tuple //

IsNull : A* → Bool // is the argument null? //

Member: (A × A* × (A × A → Bool)) → Bool

// checks if first argument is in the tuple given as the second argument

using the equality function given as the third argument //

hd: A* → A // returns first element of the non-null tuple given as the argument //

tl: A* → A* // removes first element in the non-null tuple given as the argument //

cons: A × A* → A* // prepends the first argument onto the second //

map: (A→B) → A* → B* // applies the function argument to each element on the tuple //

append: A* × A* → A* // attaches the first tuple on the end of the second tuple //

Merge: A* × A* → (A × A)*

= λ<x,y> . if IsNull x or IsNull y then null else cons(<hd x, hd y>, Merge(tl x, tl y))

DU: A** × A** → A*

= λ<x,y>. if IsNull x then null

else DU(tl x, map (λt.Append<(hd x),t>) y)

Common: A* × A* × (A × A → Bool) → A*

= λ<x,y,f>. if IsNull x then null

else if Member (hd x, y, f) then cons(hd x, Common(tl x, y, f))

else Common(tl x, y, f)

Subsets: A* → A** // non-null subsets of the argument list //

= λx. if IsNull x then null

else if IsNull(tl x) then <x>

else cons(<x>, Append(map (λt.cons(hd x, t)) (Subsets(tl x)), Subsets(tl x)))

AppendAll: A** → A*

= λx. if IsNull x then null else Append(AppendAll(tl x), hd x)

ApplyTuple: (A→B)* × A* → B*

= λ<f,x>. if IsNull f or IsNull x then null

else cons((hd f) (hd x), ApplyTuple(tl f, tl x))

DistributeTuples: A** → A**

= λx. if IsNull x then null

else if IsNull (tl x) then x

else if IsNull (hd x) then null

else Append(map (λt.cons(hd(hd x),t)) (DistributeTuples(tl x)),

DistributeTuples(cons(tl(hd x), tl x)))

remove: A* × (A→Bool) → A* // removes elements of the tuple for which the function argument evaluates TRUE //

Rename: Atexp × Tvar* → Atexp

// Rename(α,L) renames free variables in α to be different than those on L //

RenameList: Tvar* × Atexp × Tvar* → Tvar* × Atexp

// RenameList(L,τ,L') renames variables on L and in τ which appear on L' //

Explicit: Exp → Bool

// Explicit(e) = true if e of the form (f::τ) e_1 e_2 ... e_n , else false //

OnVlist: Tvar × Tvar* → Bool

// tests if the type variable is on the list of type variables //

= λ<x,y>. member(x,y,Eq)

HasCommon: Tvar* × Tvar* → Bool

// true if there is at least one variable on both lists //

= λ<x,y>. IsNull (Common(x,y,Eq))

FindPos: Tvar × Tvar* → (Nat + Err)

// FindPos(v,L) = first position that v occurs in L (starting with hd(L) = position 0) //

$= \lambda\langle x,y\rangle$. if IsNull y then error else if Eq(x, hd y) then 0 else 1 + FindPos(x, tl y)

Select: $\forall A.\ Nat \rightarrow A^* \rightarrow (A + Err)$

// Select n L returns element on L in the n'th position, starting with 0 //

$= \lambda x \lambda y$. if IsNull y then error else if x=0 then hd y else Select (x-1) (tl y)

Len: $\forall A.\ A^* \rightarrow N$

// Len(L) = the length of the sequence L //

$= \lambda x$. if IsNull x then 0 else 1 + Len(tl x)

Kind: Atexp $\rightarrow$ {variable, generator, +, $\times$, $\rightarrow$, $\cap$, μ}

// The kind of the root of the type expression //

Left, Right: Atexp $\rightarrow$ (Atexp + error)

// Left and right of type expressions of kind +, $\times$, $\rightarrow$, $\cap$ //

AllSameKind: Atexp* $\rightarrow$ Bool

// are all the type expressions in the tuple the same Kind? //

Eq: Atexp $\times$ Atexp $\rightarrow$ Bool

// determines if the first expression is identical to the second when μ-expressions are unrolled indefinitely--i.e., Eq(s,t) iff $\forall k \forall z$ Trunc(s,k,z) and Trunc(t,k,z) are identical //

Terms: Atexp $\rightarrow$ Atexp*

// returns a non-null list of expressions which are the terms of the argument. Each expression on the list is an $\rightarrow$, $\times$, + or type generator (i.e., μ's are unrolled). //

ArrowTerms: Atexp $\rightarrow$ Atexp*

// returns a list of all the terms of the argument of kind $\rightarrow$ //

CrossTerms, PlusTerms // analagous to ArrowTerms //

Combine: Atexp* $\rightarrow$ (Atexp $\times$ Atexp)

// splits all binary ops on the argument list into intersections of left and right trees//

= λx. // let g x = <u,v> //

if u = v = null then <D,D>

else <IntersectAll(u), IntersectAll(v)>

where g x = <u,v>

g = λx. if IsNull x then <null,null>

else if Kind(hd x) = +, ×,→,∩ then

// let <y,z> = g (tl x) //

<cons(left(hd x), y), cons(right(hd x), z)>

else g (tl x)

FV: Atexp → Tvar* // returns the list of free variables in the argument //

Rename: Atexp × Tvar* → Atexp

// returns first argument with any free variables on the given list of variables renamed to new variables //

Distinct: Tvar* → Bool // returns true if the variables in the argument tuple are distinct //

Substitute: Atexp × Tvar × Atexp → Atexp

// substitute argument 1 for free occurances of argument 2 in argument 3, renaming bound variables, if necessary, to avoid name clashes //

Valid: Atexp × Tvar* → Bool // checks for valid μ-expression formation //

= λ<a,S> . if Member(a,S,Eq) then false

else if a atomic then true

else if a = a1 op a2 where op∈ {×,→, +} then

Valid(a1,null) and Valid(a2,null)

else if a = F(a1,...,an), a type generator, then

∀i≤n. Valid(ai,null)

else if a = μx.b then Valid(b, cons(x,S))

else // a = a1∩a2 // Valid(a1,S) and Valid(a2,S)

IntersectAll: Atexp* → Atexp // forms intersection of expressions on the argument list //

Intersect: (Atexp + Err) × (Atexp + Err) → (Atexp + Err)

= λ<x,y> . if x=error then y else if y=error then x else x«y

Distribute: Atexp* × {+, ×,→} × Atexp* → (Atexp + Err)

= λ<x, op, y> . if IsNull x or IsNull y then error

else Intersect(hd(x) op hd(y),

Intersect(Distribute(<<hd(x)>, op, tl(y)>),

Distribute(<tl(x), op, tl(y)>)))

LookupConds: Gen × Gen × Rules → (Conds + Err)

= λ<x,y,R>. if Rx = undefined then error

else f (Rx) where f = λt. if IsNull t then error

else if hd(t)=<y,z> then z

else tl(t)

NewVars: N × Tvar* → Tvar // returns n-tuple of type variables different from 2nd arg. //

PartitionPairs: Tvar* × (Atexp × Atexp)* → (Atexp* × Atexp*)*

= λ<v,p>. if IsNull v then null

else cons(PP<hd v, p>, PartitionPairs(tl v, p)) where

PP: Tvar × (Atexp × Atexp)* → (Atexp* × Atexp*)

= λ<x,p>. if IsNull p then <null,null>

else // let hd p = <s,t>, PP<x, tl p> = <q,r> //

if Eq<s,x> then <q, cons(t,r)>

else if Eq<x,t> then <cons(s,q), r>

else <q,r>

FindGensAbove: Gen* × Rules → (N × Conds*)*

// returns arity and conds of generators G for which there are rules F ≤ G for each

generator F on the given list //

```
Reduce: Atexp → (Atexp + Err)
 // Reduce(α) rewrites subexpressions of α of the form σ→(τ∩ρ) as (σ→τ)∩(σ→ρ),
        provided μ-expressions remain reduced under unrolling, else error   //
 = λa . if a atomic then a
        else if a = a1∩a2 then Reduce(a1) ∩ Reduce(a2)
        else if a = a1→a2 then
                if Reduce(a1)=s and Reduce(a2)=t then
                        Distribute(<s>, →, Terms(t))
                else error
        else if a = a1 × a2 then
                if Reduce(a1)=s and Reduce(a2)=t then Distribute(Terms(s), ×, Terms(t))
                else error
        else if a = a1 + a2 then
                if Reduce(a1)=s and Reduce(a2)=t then
                        Distribute(Terms(s), +, Terms(t))
                else error
        else if a = F(a1,...,an), a type generator, then
                if t1=Reduce(a1),...,tn=Reduce(an)  then F(t1,...,tn)
                else error
        else  // let a = μx.b //
                if t=Reduce(b) then
                        if IsReduced(Substitute(t, x, t)) then μx.t
                        else error
                else error
```

```
IsReduced: Atexp → Bool
// returns true if its argument has no intersections to the right of any →, or to the left or right
of any + or ×, evenwhen μ-expressions are unrolled //

Below: Atexp × Atexp × Rules → Bool
 // Below(σ,τ, R) checks if σ <<μ τ, using the ≤μ-rules augmented with R //
 = λ<s,t,R>.
        if t=t1∩t2 then Below(s,t1,R) and Below(s,t2,R)
        else if t=μx.r then Below(s,Substitute(t,x,r),R)
        else esd(DP (<s',t>,FV(s'),null,R))
          where
                esd: (Atexp × Atexp)** → Bool
                  = λd. if IsNull d then false
                        else if esc(PartitionPairs(FV(s'),hd d)) then true
                                else esd(tl d)
                esc: (Atexp* × Atexp*)* → Bool
                  = λc. if IsNull c then true
                        else // let hd(c) = <y,z> //
                                if IsNull y and IsNull z then true
                                else if IsNull z then EUB(y, null, R)
                                else if IsNull y then true
                                else e <y, IntersectAll z>
                e: (Atexp* × Atexp) → Bool
                  = λ<x,y>. if IsNull x then true
                                else if Weaker(<hd x, y>, append(FV(hd x), FV(y)),
                                                        null, R )
```

then e <tl x, y>

else false

Weaker: Atexp × Atexp × (Atexp × Atexp)* × Rules → Bool

// Computes an extension of Wμ //

= λ<a, b, V, R>. // let V' = Append(<a,b>, V) //

if Member(D, Terms(a), Eq) then true

else if Member(<a,b>, V, (λ<<w,x>,<y,z>>.Eq<w,y> and Eq<x,z>))

then true

else if a and b Tvars then a=b

else if b = b1∩b2 then Weaker(a, b1, V', R) and Weaker(a,b2,V',R)

else if b = b1→b2 then

// let Z = {t | Member(s→t,Terms(a),Eq) and Weaker(b1,s,V',R)}//

if Z=Ø then false else Weaker(IntersectAll(Z), b2, V', R)

else if b = b1 × b2 then

// let Z1 = {s | Member(s × t, Terms(a), Eq)},

Z2 = {t | Member(s × t, Terms(a), Eq)} //

if Z1=Ø then false else Weaker(IntersectAll(Z1), b1, V', R) and

Weaker(IntersectAll(Z2), b2, V', R)

else if b = b1 + b2 then

// let Z1 = {s | Member(s + t, Terms(a), Eq)},

Z2 = {t | Member(s + t, Terms(a), Eq)} //

if Z1=Ø then false else Weaker(IntersectAll(Z1), b1, V', R) and

Weaker(IntersectAll(Z2), b2, V', R)

else if b = F(Lt) then

if a = a1∩a2 then Weaker(a1, b, V', R) or Weaker(a2, b, V', R)

```
            else if a = G(Lt') then
                    if R <G,Len(Lt')> = undefined then false
                    else if Lookup(<F,Len(Lt)>, R <G,Len(Lt')>) = f then
                            ApplyWeaker(f <Lt', Lt>, V', R)  else false
        else false

Lookup: Gen × (Gen × Conds)* → (Conds + Err)
 = λ<<x,n>,y>. if IsNull y then Err
                    else // let hd y = <<g,m>,f>  //
                            if g=x and n=m then f  else Lookup(<x,n>, tl y)

ApplyWeaker: (Atexp × Atexp)* × (ATexp × Atexp)* × Rules  →  (Bool + Err)
ApplyWeaker(V, V', R) =
        if IsNull(V) then true
        else // let hd(V) = <a,b> //
                if Weaker(a,b,V',R) then ApplyWeaker(tl(V), V', R) else false

DP: (Atexp × Atexp) × Tvar* × (Atexp × Atexp)* × Rules → (Atexp × Atexp)**
 = λ<<a,b>, fv, V, R>.  // let V' = cons(<a,b>, V)  //
        if Member(<a,b>, V, λ<<w,x>,<y,z>>.Eq<w,y> and Eq<x,z>)
                or Member(D,Terms(a),Eq) then <null>
        else if b or a of the form μx.t then DP(<Expose(a),Expose(b)>,fv,V',R)
        else if b=b1∩b2 then DU(DP(<a,b1>, fv, V', R), DP(<a,b2>, fv, V', R))
        else if b in Tvar and not Member(b, fv,Eq) then
                if a=a1∩a2
                  then Append(DP(<a1,b>, fv, V', R), DP(<a2,b>, fv, V', R))
```

```
        else <<a,b>>
else if b=b1→b2 then
        if a=a1→a2 then DU(DP(<b1,a1>, fv, V', R), DP(<a2,b2>, fv, V', R))
        else if a=a1∩a2 and Common(FV(b),fv,Eq)=null and IsReduced(b) then
                        Append(DP(<a1,b>, fv, V', R), DP(<a2,b>, fv, V', R))
        else AppendAll( Map f (ApplyCombine(Subsets(Arrowterms(a))))
                where f = λ<x,y>.DU(DP(<b1,x>,fv,V',R),
                                        DP(<y,b2>,fv,V',R)  )
else if b=b1 × b2 then  // let Z = CrossTerms(a)  //
        if Z=null then null
        else if a = a1∩a2 and Common(FV(b),fv)=null and IsReduced(b) then
                Append(DP(<a1,b>, fv, V', R), DP(<a2,b>, fv, V', R))
        else  DU(DP(<s,b1>,fv,V',R),  DP(<t,b2>,fv,V',R))
                where <s,t> = Combine(Z)
else if b=b1 + b2 then  // let Z = PlusTerms(a)  //
        if Z=null then null
        else if a = a1∩a2 and Common(FV(b),fv)=null and IsReduced(b) then
                Append(DP(<a1,b>, fv, V', R), DP(<a2,b>, fv, V', R))
        else  DU(DP(<s,b1>,fv,V',R),  DP(<t,b2>,fv,V',R))
                where <s,t> = Combine(Z)
else  // b = F(L), a type generator //
        if a = a1∩a2 then Append(DP(<a1,b>, fv, V', R), DP(<a2,b>, fv, V', R))
        else if a = G(M) and LookupConds(<G,Len(M)>,<F,Len(L)>,R)=g then
                ApplyDU( map (λx.DP(x,fv,V',R) (g <M,L>))
        else null
else null
```

```
EUB: Atexp* × Atexp** × Rules → Bool
 = λ<A,W,R> .    // Let Z=DistributeTuples(map Terms A'),
                      A' = remove<A, lx.Member<D,Terms x>>
                      W' = cons(A,W),
                      f = λ<x,y>. if Len(x)=Len(y) then if IsNull(x) then true
                                                       else Eq<x,y> and f<tl x, tl y>
                                  else false                          //
        if Member(A,W,f) or IsNull(A') then true
        else g Z, where
              g = λz. if IsNull z then false
                      else if EUBT(hd z, W', R) then true
                      else g (tl z)

EUBT: Atexp* × Atexp** × Rules → Bool
 = λ<B,W,R> . // let W' = append(B,W) //
        if IsNull B then false
        else if not AllSameKind(B) then false
        else if hd(B) a variable then true
        else if hd(B)=a→b then EUB(map (λ<x,y>.y) (Combine B), W', R)
        else if hd(B)=a × b or hd(B)=a + b then
                              EUB(map (λ<x,y>.x) (Combine B), W', R)  and
                              EUB(map (λ<x,y>.y) (Combine B), W', R)
        else  // all expressions on B are type generators
              Let G = map MakeGen B       //
              g (FindGensAbove(G,R)), where
```

```
g = λx. if IsNull x then false
        else if h(hd x) then true
        else g(tl x)
h = λ<n,z>. // let n = NewVars(n, AppendAll(map FV B)) //
              let K=AppendAll(ApplyTuple z (map (λt.<t,H>) G))
              let M=PartitionPairs(H,K) //
        f(M)
f = λx. if IsNull x then true
        else // let hd(x) = <y,z> //
              if IsNull y and IsNull z then true
              else if IsNull z then EUB(y, W', R)
              else if IsNull y then true
              else e <y, IntersectAll z>
e = λ<x,y>. if IsNull x then true
              else if Weaker(<hd x, y>,
                             append(FV(hd x), FV(y)),
                             null, R  )
                   then e <tl x, y>
              else false
```

```
UW: Atexp × Atexp × (Atexp × Atexp)* × Atexp** → Atexp***
  = λ<a,b,V,z>.  // let a' = Find(a, z),  b' = Find(b, z), V'=cons(<a,b>, V)  //
      if Eq<a', b'> or OnVlist(<a,b>,V) then < z >
      else if b' = b1 ∩ b2 then
            AppendAll(map (λe.UW(a,b2,V',e)) (map (λe.UW(a,b1,V',e)) z))
      else if a' = a1 ∩ a2 then  append(UW(a1,b,V',z), UW(a2,b,V',z))
```

```
        if a' or b' a variable then MergeClasses(a',b',z)
    else if a'=μx.s then UW(Substitute(a',x,s),b',V,z)
    else if b'=μx.s then UW(a',Substitute(b',x,s),V,z)
    else if Kind(a')≠Kind(b') then null
    else
       if a' = a1 → a2 and b' = b1 → b2 then
            AppendAll(map (λe.UW(a2,b2,V',e)) (map (λe.UW(b1,a1,V',e)) z))
    else if a' = a1 × a2 and b' = b1 × b2 then
            AppendAll(map (λe.UW(a2,b2,V',e)) (map (λe.UW(a1,b1,V',e)) z))
    else if a' = a1 + a2 and b' = b1 + b2 then
            AppendAll(map (λe.UW(a2,b2,V',e)) (map (λe.UW(a1,b1,V',e)) z))
    else if a' = F(L) and b' = G(M)  // type generators // then
            if LookupConds(MakeGen a', MakeGen b', R) = f ≠ error then
               g (f <L,M>) z where
                     g : (Atexp × Atexp)* → Atexp** → Atexp***
                       = λxλy. if IsNull x then <y>
                                    else // let <s,t> = hd x  //
                                          AppendAll(map (g (tl x)) (UW(s,t,y)) )
            else null
    else null

MergeClasses: Atexp × Atexp × Atexp** → Atexp**
 = λ<x,y,z>.  cons(g(AppendAll(remove z f)), remove z (λt. not (f t)))
       where f = λw. if IsNull w then true
                        else if Eq <x,hd w> or Eq<y,hd w> then true else false
             g = λv. if IsNull v then v
```

```
                else if Kind(hd v)≠variable then v
                else Append(hd v, g(tl v))
Find: Atexp × Atexp** → Atexp
 = λ<x,y>. if IsNull y then x
                else if member<x, hd y, Eq> then hd(hd y)
                else Find<x, tl y>

WhichBelow: Atexp × Atexp × Rules → (Atexp × Err)
 = λ<a,b,R>. if a=a1∩a2 then Intersect(WhichBelow(a1,b,R),WhichBelow(a2,b,R))
                else if a=μx.t then WhichBelow(Substitute(a, x, t), b, R)
                else if a = a1→a2 then
                        if Below(b,a1,R) then a2 else error
                else error

Infer: Atexp × Atexp × Rules → (Atexp + Err)
  = λ<a,b,R>. if a=a1∩a2 then Intersect(Infer(a1,b,R), Infer(a2,b,R))
                else if a=μx.t then Infer(Substitute(a,x,t),b,R)
                else if Kind(a)=variable then a
                else if a=a1→a2 then
                        if a1=μx.t then Infer((Substitute(a1,x,t)→a2), b, R)
                        else ApplySubstitutions(UT(b,a1,g(Subexpressions(a1)),R), a2)
                                where g=λx. if IsNull x then x
                                                else cons(<hd x>, g (tl x))
                else error

Bipartite: (Tvar × Tvar)* × Tvar* × Tvar* → Bool
```

```
= λ<<w,x>,y,z> . if Member(w,y,Eq) then Member(x,z,Eq)
                 else if Member(w,z,Eq) then Member(x,y,Eq)
                 else false
```

SubstituteList: Atexp* × Tvars* × (Tvar × Tvar)* → (Atexp × Atexp)*

```
= λ<x,y,z> . if IsNull x  or  IsNull y  then z
             else if IsNull z then z
             else  // let hd z = <z1, z2> //
                   cons(<Substitute(hd x, hd y, z1), Substitute(hd x, hd y, z2)>,
                        SubstituteList(tl x, tl y, tl z) )
```

The vita has been removed from the scanned document

Conjunctive Polymorphic Type Checking with Explicit Types

by

Kevin E. Flannery

(ABSTRACT)

An expressive type language and the ability to do compile-time type inference are desirable goals in language design, but the attainment of the former may preclude the possibility of the latter. Specifically, the type conjunction operator (type intersection) induces a rich type language at the expense of decidability of the typeable expressions. Two extreme alternatives to this dilemma are to abandon type inference (and force the programmer to, essentially, supply a derivation for his type claims) or to abandon (or restrict) type conjunction. This work presents a third alternative in which the programmer, at times, may be required to supply explicit types in order for type inference to succeed. In this way, the power of conjunctive types is preserved, yet compile-time type inference can be done for a large class of polymorphic functions, including those typeable with parametric types.

To this end, we introduce a simple combinator based language with typing rules based on type conjunction and a subtype relation, of sorts, called "weaker." The validity of the type rules with respect to the usual interpretation of "type" is shown, along with the undecidability of the type relation. It is shown how the computational portion of the language can be modified to accommodate explicit type information which may direct an automatic type derivation. This new language has the principal type property with respect to a decidable relation, although deciding this relation is shown to be an NP-Complete problem. The language is extended to accommodate type fixedpoints, and extended further to allow all expressions with parametric types to be typed automatically, and to accommodate integers, pairs, sums and abstract types in the form of type generators.