ELSEVIER

Artificial Intelligence 75 (1995) 161-194

Artificial Intelligence

A typed resolution principle for deduction with conditional typing theory

Tie-Cheng Wang [*]

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, USA

Received October 1992; revised March 1994

Abstract

The formal reasoning involved in solving a real world problem usually consists of two parts: type reasoning associated with the type structure of the problem domain, and general reasoning associated with the kernel formulation. This paper introduces a typed resolution principle for a typed predicate calculus. This principle permits a separation of type reasoning from general reasoning even when the background typing theory shares the same language constructs with the kernel formulation. Such a typing theory is required for an accurate formulation of the type structure of a computer program which contains partial functions and predicate subtypes, and also is useful for efficiently proving certain theorems from mathematics and logic by typed (sorted) formulation and deduction. The paper presents a typed deduction procedure which employs type reasoning as a form of constraints to general reasoning for speeding up the proof discovery. The paper also discusses further refinements of the procedure by incorporating existing refinements of untyped resolution.

1. Introduction

The theory of data types often plays a fundamental role in the formal reasoning involved in solving a real world problem. Type specification is indispensable both for restricting the categories, or *data types*, of individuals involved in the kernel specification of the problem, and for formulating the data type theories about the problem domain. Consequently, the formal reasoning usually consists of two parts: one associated with type relations, called *type reasoning,* and the other associated with general relations, called *general reasoning.*

[*] E-mail: wang@kestrel.edu.

0004-3702/95/$09.50 © 1995 Elsevier Science B.V. All rights reserved
SSDI 0004-3702(94)00027-X

The simplest approach to handling type reasoning is to use predicate calculus: type relations are specified by standard first-order formulae. This approach has the advantage that it permits a direct application of all existing proof techniques for first-order theory. The disadvantage is that it is inefficient: it uses a general method to solve a special problem for which more efficient methods can be applied. In addition, it is inelegant: it mixes the type calculus into the deductive inference of general reasoning.

Another approach is hybrid knowledge representation and reasoning. This approach uses specialized algorithms to carry out type reasoning, and uses type reasoning as a form of constraints to the general reasoning for speeding up the proof discovery. This approach includes sorted (many-sorted or order-sorted) logic, constraint logic programming, constrained resolution, and theory resolution, etc. The typed predicate calculus reported here belongs to this approach. However, motivated by reasoning about computer programs and efficiently proving certain theorems from mathematics by typed (sorted) formulation and deduction, we shall emphasize the capability of handling type constructors and relations associated with polymorphism, partial functions, and subtypes. Particularly, we need to deal with conditional typing rules.

A conditional typing rule is a typing rule having a hypothetical condition on general relations. Conditional typing rule is needed in a number of important cases.

(1) It is needed for accurately typing partial functions. For example, $div(x,y){:}\,int/y \neq 0/x{:}\,int \wedge y{:}\,int$. (The symbols "/" and ":" will be introduced formally as a part of our typed language in the next section.)
(2) It is needed for specifying subranges and enumeration types occurring in a conventional programming language. For example, $i{:}\,dbid/lb \leqslant i \wedge i \leqslant ub/i{:}\,integer$, and $x{:}\,weekend/x \in \{saturday, sunday\}$.
(3) It is needed for specifying predicate subtypes. For example, $x{:}\,odd/\neg even(x)/x{:}\,integer$.
(4) It may be needed for some typing rules created naturally from normalization of a typed formula. For example, $x{:}\,string/vkd(x)$ (see Example 2.3).
(5) It may be needed for typing some Skolem functions. For example, $q(x){:}\,xpt/x \not\leqslant l/x{:}\,dpt$ (see Example 2.4).

Due to the existence of conditional typing rules, existing techniques on many-sorted or order-sorted predicate calculus may be inefficient, or difficult to apply directly. With few exceptions (to be addressed later), most of previous studies in sorted logic assume that the underlying sorting (typing) theory is nonconditional in the sense of having no hypothetical condition on general relations. Thus, a crucial problem that must be solved by our typed approach is how to separate type reasoning from general reasoning when the background typing theory and the kernel logical formulation share the same language components for general relations. The typed resolution principle is introduced for solving this problem.

The typed resolution principle permits a normalization of the denial of a theorem being proved into two subsets of clauses. One is a set of typing rules, called a typing theory, and the other is a set of typed clauses. The principle consists of two special forms of binary resolution. One is the binary resolution of two typed clauses upon their kernels, called kernel resolution. The other is the binary resolution upon the type restriction of a typed clause and the head of a typing rule, called TP-resolution. Kernel resolution is used to deduce consequences from the general relations contained in the kernel formulation. TP-resolution is used to carry out type reasoning: namely, type-checking and TP-deduction. Type-checking detects those consequences of kernel resolution whose type restrictions cannot be satisfied in the given typing theory. TP-deduction deduces consequences related to the general relations contained in the typing theory.

We shall prove a typed version of the Herbrand theorem, and based on this theorem, prove the completeness of the typed resolution principle. The typed Herbrand theorem permits one to reduce the proof of the unsatisfiability of a set of typed clauses in a typing theory to the search for an unsatisfiable subset of untyped logical consequences of these clauses in the typing theory. It thus provides a basis for developing typed deduction procedures from untyped deduction.

Despite the gain in the efficiency by specializing type reasoning, in many cases the main cost of proving a theorem by a typed (sorted) deduction is still due to general reasoning. By noting this problem, we shall devote a good deal of discussion to the design of typed deduction procedures which can also be efficient in general reasoning. In particular, we shall define a TP-second deduction procedure, and investigate its further improvements by incorporating the existing refinements of untyped resolution.

Besides theorems derived from program analysis, many theorems from general problem solving can be formulated in our typed language in a nontrivial manner, such that the typed deduction techniques can be effectively applied. The paper will present some examples which apply typed deduction to general problem solving, and demonstrate improvements both in the clarity of representation and in the efficiency of reasoning.

The basic idea of specializing type reasoning in deductive systems dates back to Bledsoe's famous paper on non-resolution theorem proving [3] and Reiter's work on typed deductive databases [24]. Since then, much research in this direction has been focused on the sorted predicate calculus (Walther [31], Cohn [11], Frisch[13], and Schmidt-Schauss [27], etc.). The sorted predicate calculus employs sorted languages to formulate taxonomic relations, and uses special algorithms, mainly sorted unification, to handle deduction on sort relations. This approach has demonstrated a great advantage over unsorted approach in solving certain problems from logic and artificial intelligence.

The typed predicate calculus presented here extends the existing approach of sorted deduction by allowing general predicates to appear in the underlying typing theory. Regarding this extension, we must mention that the logic of Weidenbach and Ohlbach [38] also has the ability to include general predicates in typing theory. The sorted prover LLAMA [12] of Cohn allows certain limited use of general predicates in the sort theory. However, most of these systems were produced from scratch in the sense that they give no answer to the question of how existing refinements of unsorted resolution can be incorporated, whereas our typed approach emphasizes the incorporation of existing refinements of untyped resolution. Thus, our typed approach is in the direction of Frisch's substitution framework [13,14]. Frisch's framework provides a method of systematically transforming an unsorted deduction system into a sorted one. It also provides a method for systematically transforming a completeness proof for an unsorted deduction into one of the corresponding sorted deduction.

Besides sorted deduction, our typed approach is closely related to Bürckert's quantification system (RQS) [7,8], constraint logic programming (CLP) [17,16], and Stickel's theory resolution [28]. All of these systems improve on automated deduction by separating out the reasoning associated with those parts of problem solving for which more efficient knowledge representation and reasoning methods are known, or can be developed.

However, both Frisch's substitution framework and Bürckert's RQS require the background theory to use a set of predicates disjoint from the set of predicates for general relations.[1] CLP has the additional restriction that the kernel logical formulation must be a Horn set. In contrast, Stickel's theory resolution has no general restriction on the language of background theory; and there exist certain similarities between Stickel's total wide theory resolution ([28, Lemma 7]) and a special version of our typed deduction (TP-first deduction). The difference is that theory resolution requires an immediate solution of the unsatisfiability of some subclauses (constraints) in the entire background theory in each theory resolution step. For our typed deduction, such a solution can be postponed.

The remainder of this paper is organized as follows. Section 2 introduces the typed first-order language. Section 3 presents a least Herbrand model for a typing theory, and based on this model proves the typed Herbrand theorem. Section 4 presents the typed resolution principle. Section 5 presents TP-second deduction and proves its completeness. Section 6 discusses issues about the refinements of typed resolution. Section 7 presents a summary of some results obtained by a computer implementation of a typed theorem prover. Section 8 concludes this paper.

2. The typed first-order language

2.1. Basic definition

The typed language introduced here is an ordinary first-order language expanded with a set of special constructs associated with type relations. The language contains the usual function and predicate symbols, variables, and logic connectives of a first-order language. The set of special constructs consists of type descriptions (TD), typed quantifiers, type restrictions (TR), typing rules, and typed clauses. A TD (type description) is an atom of form $t: \tau$, where ":" is a special symbol of the language, and t and τ are both terms of the language. A TR (type restriction) is a conjunction of TDs. Typing rules and typed clauses are normal forms of the language to be defined soon. There are two kinds of type quantifiers $\forall(x: \tau)$ and $\exists(x: \tau)$, which are used as abbreviations of certain language components according to the following rules: for every formula ϕ of the language,

$$\forall(x: \tau)(\phi) = \forall(x)(x: \tau \Rightarrow \phi).$$

$$\exists(x: \tau)(\phi) = \exists(x)(x: \tau \wedge \phi).$$

Semantically, a form $t: \tau$ usually has an intuitive meaning that t is of type τ. (The word "type" in this paper is a synonym of the word "sort" of the sorted predicate

[1] The equality predicate is an exception for RQS, where it is used as a constraint predicate. See [7] for detail.

calculus.) But syntactically, t and τ can be any terms; there is no restriction that t and τ are terms that must be interpreted by such semantics.

2.2. *Typing rules and typed clauses*

Similar to the design of efficient proof procedures for untyped predicate calculus, clausal normal forms will play a fundamental role in developing efficient proof procedures for typed predicate calculus. For our typed language, we define two kinds of normal forms. One is called a typing rule, and the other is called a typed clause.

Definition 2.1. A *typing rule* is a formula of form $D/H/R$, where D is a TD, R is a TR (a conjunction of TDs), and H is a conjunction of literals on general relations (i.e. literals whose atoms are not TDs). H is called the general condition of the typing rule.

Semantically, a typing rule $D/H/R$ is equivalent to the universal closure of the first-order formula, $R \Rightarrow (H \Rightarrow D)$. The following are some examples of typing rules. (We use *true* for the empty conjunction, $D//R$ for $D/true/R$, D/H for $D/H/true$, and D for $D/true/true$)

1. $x + y\colon int//x\colon int \wedge y\colon int$
2. $x + y\colon real//(x\colon real \wedge y\colon int) \vee (x\colon int \wedge y\colon real) \vee (x\colon real \wedge y\colon real)$.
3. $concat(x, y)\colon seq(\alpha)//x\colon seq(\alpha) \wedge y\colon seq(\alpha)$
4. $size(x)\colon int//x\colon seq(\alpha)$
5. $size(x)\colon int//x\colon set(\alpha)$
6. $x\colon string//x\colon seq(char)$
7. $x\colon seq(char)//x\colon string$
8. $x\colon \text{small-}int/-100 \leqslant x \wedge x \leqslant 100/x\colon int$
9. $x\colon weekend/x \in \{saturday, sunday\}$
10. $first(x)\colon \alpha/x \neq [\,]/x\colon seq(\alpha)$
11. $M(x)\colon real/0 \leqslant x \wedge x \leqslant 9/x\colon int$
12. $div(x, y)\colon int/y \neq 0/x\colon int \wedge y\colon int$
13. $div(x, y)\colon string/y = 0/x\colon int \wedge y\colon int$
14. $reduce(op, S)\colon \alpha//op\colon \alpha \times \alpha \rightarrow \alpha \wedge S\colon set(\alpha)$
15. $image(f, S)\colon set(\beta)//f\colon \alpha \rightarrow \beta \wedge S\colon set(\alpha)$
16. $*\colon int \times int \rightarrow int$

Rules 1 and 2 indicate that + is overloaded. Rule 2 is used to represent three individual typing rules (i.e., a conjunction of three typing rules). Rule 3 indicates that *concat* is a polymorphic function [21]. Rules 4 and 5 indicate that *size* is both overloaded and (parametric) polymorphic [9]. Rules 6 and 7 specify a type equality relation between *string* and *seq*(*char*). Rule 8 is an axiomization of a subtype of integer. Rule 9 is given for an enumeration type. Rule 10 is a typing rule for the partial function *first*. Rule 11 is a type specification for an array M of reals. If division by zero is handled by an exception mechanism that returns an error message string, then the typing for the division function can be formulated by rules 12 and 13. A meta-function is a function that takes functors as arguments. The functions *reduce* and *image* given in Rules 14 and 15 are meta-functions. The typing rules for meta-functions make use of typing rules for functors such as rule 16.

A collection of typing rules is called a *typing theory*. In the practical application to program analysis, a typing theory may consist of three sources of typing rules. One is a set of basic typing rules, say, a basic typing theory, which is an axiomatization of the type system of the programming language. The second is a set of typing rules for the user-defined data types and functions. The third is a set of typing rules for Skolem functions created by normalization of the input formulae.

For a given typing theory T, we shall use $(T)_{gr}$ to denote the entire set of ground instances of elements of T, $T|_{TD}$ to denote the restriction of T in the entire set of TDs, and $\|T\|$ to denote the set of nonconditional typing rules of T. Symbolically, $T|_{TD} = \{d//r \mid d/h/r \in T\}$, and $\|T\| = \{d//r \mid d//r \in T\}$. Clearly, T is nonconditional iff $T = \|T\| = T|_{TD}$.

Definition 2.2. A *typed clause* is a formula of form K/R, where K is a disjunction of literals on general relations, and R is a TR (a conjunction of TDs). K and R are called the kernel and the type restriction of the clause, respectively.

To transform a typed first-order formula F into normal form, we first transform F into an ordinary formula F' by replacing every $\forall(x{:}\,\tau)(\phi)$ and $\exists(x{:}\,\tau)(\phi)$ of F with $\forall(x)(x{:}\,\tau \Rightarrow \phi)$ and $\exists(x)(x{:}\,\tau \wedge \phi)$, respectively. Then we use the standard Skolemization and normalization procedures to transform F' into a set S of ordinary clauses. Finally, we transform each clause of S into a typing rule or a typed clause according to its syntax. There may be some cases in which a clause cannot be transformed into a typing rule, nor a typed clause (e.g. $a{:}\,\tau_1 \vee b{:}\,\tau_2 \vee a = b$). How to deal with such cases is outside of the scope of this paper. Frisch presents in [14] a normal form transformation system, which can avoid the introduction of conditional typing rules for Skolem functions under certain assumptions, and may be useful for transforming some of those typed formulae into normal form which cannot be directly converted into a set of typing rules and typed clauses.

2.3. Relations to some other languages

The typed language, as described above, is close to Frisch's sorted language (SFOPC) [13,14]. SFOPC is a first-order language with sorted syntax sugar. However, SFOPC is more specific in the structure of the sorted sublanguage: it distinguishes sort symbols from ordinary non-logical symbols. For the typed language of this paper, such a distinction is unimportant. This is because here we are concerned mainly with the schema of typed deduction procedures, but not with concrete algorithms for type reasoning; only the restriction on predicate symbols used for specifying type relations is essential.

We wish to point out that in order to have efficient algorithms for type reasoning, typed sublanguages must be studied specifically. For this study, it is possible to incorporate existing results from previous research in sorted deduction, such as Walther's sorted unification [31] for a forest sort structure, Cohn's sorted logic and sort array computation [11], and Frisch's sorted unification algorithm [14] for monomorphic tree structure, etc.

Our typed language is a form of constraint first-order language, which is close to the language of Bürckert's restricted quantification system (RQS) [7,8]. In particular, we have borrowed some notions (i.e. $C//R$) from RQS. Actually, a typing theory can be viewed as a restriction theory of RQS expanded with general relations. Without considering the exclusion of general predicates, the approach to constrained resolution made by RQS is more general: RQS permits any theory of constraints; for example, the set of models can be the models of a non-Horn theory.

2.4. Examples

We now present a number of examples to demonstrate the use of our typed language. These examples will show that conditional typing rules play a key role in transforming a typed formula into normal form. A summary of computer solutions for these examples will be presented later.

Example 2.3 (*VKD-T*). This example is a typed theorem obtained from the formal verification of a computer program. The theorem states that, given two strings, if both of them are valid key data (*vkd*), then they can be concatenated together and the result must also be a valid key data. With our typed language, the theorem is formulated as

$$\forall(x{:}\, string, y{:}\, string)(vkd(x) \wedge vkd(y) \Rightarrow vkd(concat(x,y))),$$

where *vkd* is defined by

$$\forall(st)(vkd(st) \equiv (st{:}\, string) \wedge \forall(ch{:}\, char)(ch \in st \Rightarrow (\#2 \leqslant ch \wedge ch \leqslant \#7))).$$

The entire set of clauses obtained from the denial of the theorem and from the the domain theory[2] of the program language consists of 5 typing rules and 10 typed clauses (a, b and k are Skolem symbols).

1. $concat(x,y){:}\, string//x{:}\, string \wedge y{:}\, string$
2. $z{:}\, string/vkd(z)$
3. $k(z){:}\, char/\neg vkd(z)/z{:}\, string$
4. $a{:}\, string$
5. $b{:}\, string$

[2] The domain theory contains actually several hundreds of rules (typed clauses).

6. $\neg vkd(z) \vee \neg(x \in z) \vee (x <= \#7)/x{:}\,char$
7. $\neg vkd(z) \vee \neg(x \in z) \vee (\#2 <= x)/x{:}\,char$
8. $vkd(z) \vee \neg(\#2 <= k(z)) \vee \neg(k(z) <= \#7)/z{:}\,string$
9. $vkd(z) \vee (k(z) \in z)/z{:}\,string$
10. $(x \in concat(u,v)) \vee \neg(x \in u)/x{:}\,char \wedge u{:}\,string \wedge v{:}\,string$
11. $(x \in concat(u,v)) \vee \neg(x \in v)/x{:}\,char \wedge u{:}\,string \wedge v{:}\,string$
12. $(x \in u) \vee (x \in v) \vee (\neg x \in concat(u,v))/x{:}\,char \wedge u{:}\,string \wedge v{:}\,string$
13. $vkd(a)/true$
14. $vkd(b)/true$
15. $\neg vkd(concat(a,b))/true$

Clause 1 is a typing rule for the operator *concat*. Clauses 2–3 and 6–9 are obtained by normalizing the definition of *vkd*. Among them, clauses 2 and 3 are typing rules. Clauses 10–12 are typed clauses transformed from axioms about *concat*. Since both $\leqslant$ and *concat* can be overloaded operators, adding type restrictions is important to the soundness of these rules. Note that clause 2 is a conditional typing rule. This example demonstrates that conditional typing rules may be produced for proving a theorem, even if the background typing theory is nonconditional.

Example 2.4 (*AM8-T*). The original example is the well-known theorem AM8 [34] formulated by Bledsoe for the minimum (maximum) value theorem from real analysis. AM8-T given here is a direct translation of AM8 to a typed language. This language contains a set of TDs for specifying the boundary conditions stated in the theorem. Among them, $x{:}\,xpt$ means that x is a point in the domain of the function f, $y{:}\,fpt$ means that y is a point in the range of f, and $x{:}\,dpt$ means that x is a point of the closed interval $[a,b]$ in the domain of f. The reader may refer to [34]] for an intuitive model of this translation. AM8-T consists of a basic typing theory of five typing rules k_1–k_5, two typed clauses b_1, b_2, and a typed formula F_0.

$$\begin{aligned}
F_0{:}\ & \exists(l{:}\,dpt)\\
&\quad (\forall(x{:}\,xpt)(a \leqslant x \wedge x \leqslant l \Rightarrow f(l) \leqslant f(x)))\\
&\quad \wedge \forall(x{:}\,dpt)(\forall(y{:}\,xpt)((a \leqslant y \wedge y \leqslant x \Rightarrow f(x) \leqslant f(y)) \Rightarrow x \leqslant l))\\
&\quad \wedge \forall(w{:}\,dpt)(\exists(y{:}\,dpt)(\ f(y) \leqslant f(w)\\
&\qquad\qquad\qquad\qquad \wedge (\forall(x{:}\,dpt)(f(x) \leqslant f(w) \Rightarrow y \leqslant x))))\\
&\quad \Rightarrow \exists(u{:}\,dpt)(\forall(t{:}\,dpt)(f(u) \leqslant f(t))).
\end{aligned}$$

The set of clauses obtained by normalizing F_0 consists of four typing rules, k_6–k_9, and six typed clauses b_3–b_8. Each of k_6–k_9 corresponds to a Skolem function created during the normalization.

k_1.	$a{:}\,xpt$	k_2.	$b{:}\,xpt$
k_3.	$x{:}\,dpt/a \leqslant x \wedge x \leqslant b/x{:}\,xpt$	k_4.	$x{:}\,xpt//x{:}\,dpt$
k_5.	$f(x){:}\,fpt//x{:}\,dpt$		
k_6.	$l{:}\,dpt$	k_7.	$k(x){:}\,dpt//x{:}\,dpt$
k_8.	$h(x){:}\,dpt//x{:}\,dpt$	k_9.	$q(x){:}\,xpt/x \nleqslant l/x{:}\,dpt$

b_1.	$a \leqslant x/x{:}\,dpt$	b_2.	$x \leqslant b/x{:}\,dpt$
b_3.	$f(l) \leqslant f(x) \vee a \nleqslant x \vee x \nleqslant l/x{:}\,xpt$	b_4.	$x \leqslant l \vee q(x) \leqslant x/x{:}\,dpt$
b_5.	$x \leqslant l \vee a \leqslant q(x)/x{:}\,dpt$		
b_6.	$x \leqslant l \vee f(x) \nleqslant f(q(x))/x{:}\,dpt$	b_7.	$f(h(w)) \leqslant f(w)/w{:}\,dpt$
b_8.	$h(w) \leqslant x \vee f(x) \nleqslant f(w)/x{:}\,dpt \wedge w{:}\,dpt$	b_9.	$f(u) \nleqslant f(k(u))/u{:}\,dpt$

With the typed language, the set of inequality axioms for AM8-T is also typed, which are given as follows. (Note that a clause of form $K/R_1 \vee \cdots \vee R_n$ is treated as n's (standard) typed clauses, $K/R_1, \ldots, K/R_n$.)

a_1. $x \leqslant x/x{:}\,xpt \vee x{:}\,fpt$

a_2. $x \leqslant y \vee y \leqslant x/(x{:}\,xpt \wedge y{:}\,xpt) \vee (x{:}\,fpt \wedge y{:}\,fpt)$

a_3. $x \nleqslant y \vee y \nleqslant z \vee x \leqslant z/(x{:}\,xpt \wedge y{:}\,xpt \wedge z{:}\,xpt) \vee (x{:}\,fpt \wedge y{:}\,fpt \wedge z{:}\,fpt)$

a_4. $x \nleqslant y \vee y \nleqslant x \vee f(x) \leqslant f(y)/(x{:}\,xpt \wedge y{:}\,xpt)$

Example 2.5 (*PLUS-T*). The set of clauses given below is unsatisfiable. With the typed language, we have divided the set into a typing theory and a set of typed clauses. Clearly, using the typed language adds to the clarity of the representation. As it will be shown later, it also improves the efficiency for deriving a refutation.

1. $a{:}\,fixnum$
2. $b{:}\,fixnum$
3. $x{:}\,int//x{:}\,nat$
4. $x{:}\,nat/0 \leqslant x/x{:}\,int$
5. $x{:}\,nat//x{:}\,fixnum$
6. $x{:}\,fixnum/x \leqslant 100/x{:}\,nat$
7. $x + y{:}\,int//x{:}\,int \wedge y{:}\,int$
8. $x - y{:}\,int//x{:}\,int \wedge y{:}\,int$
9. $plus(x,y){:}\,fixnum//x{:}\,fixnum \wedge y{:}\,fixnum \wedge x + y{:}\,fixnum$
10. $minus(x,y){:}\,fixnum//x{:}\,fixnum \wedge y{:}\,fixnum \wedge x - y{:}\,fixnum$

11. $x \leqslant y \wedge y \leqslant z$
    $\Rightarrow x \leqslant z/(x{:}\,int \wedge y{:}\,int \wedge z{:}\,int) \vee (x{:}\,real \wedge y{:}\,real \wedge z{:}\,real)$
12. $plus(x,y) = x + y/x{:}\,fixnum \wedge y{:}\,fixnum \wedge x + y{:}\,fixnum$
13. $minus(x,y) = x - y/x{:}\,fixnum \wedge y{:}\,fixnum \wedge x - y{:}\,fixnum$
14. $x + (y - x) = y/(x{:}\,int \vee x{:}\,real) \wedge (y{:}\,int \vee y{:}\,real)$

15. $y \leqslant x \Rightarrow 0 \leqslant (x - y)/(x{:}\, int \vee x{:}\, real) \wedge (y{:}\, int \vee x{:}\, real)$
16. $x \leqslant x/(x{:}\, int) \vee (x{:}\, real)$
17. $0 \leqslant x/x{:}\, nat$
18. $x \leqslant 100/x{:}\, fixnum$
19. $0 \leqslant a$
20. $0 \leqslant b$
21. $a \leqslant b$
22. $plus(a, minus(b, a)) \neq b/true$

3. The typed Herbrand theorem

Herbrand theorem is the basis of resolution-based proof methods. This theorem relates the satisfiability of a set of clauses to the satisfiability of the set of ground instances of these clauses, thus permitting one to reduce the proof of unsatisfiability of the set of clauses to the search of an unsatisfiable subset of these ground instances. Our purpose is to develop efficient deductive procedures for typed predicate calculus. However, instead of starting from scratch, we will develop them by extending existing deductive procedures of predicate calculus. For this extension, we need a typed version of the Herbrand theorem which relates the satisfiability of a set of typed clauses in the given typing theory to the satisfiability of untyped logical consequences of these clauses in the theory. This section is devoted to developing such a theorem.

We introduce two notations for the following discussion:

- $U[TD]$: the entire set of ground TDs of the language,
- $U[KA]$: the entire set of ground (kernel) atoms on general relations of the language.

Note that $U[TD] \cup U[KA]$ is the Herbrand base of the language.

3.1. The least Herbrand model of a typing theory

We start from the study of a special Herbrand model, called the least Herbrand model of a typing theory.

Definition 3.1 (*Ground typing precondition—gtp*). Let T be a typing theory, and let r_0 be a ground TR (type restriction). The set of ground typing preconditions (gtp) of r_0 in T is denoted by $gtp(T, r_0)$, where

$$\begin{aligned} gtp(T, r_0) = \{\, h_1 \wedge \cdots \wedge h_n \mid \\ & \{d_1/h_1/r_1, \ldots, d_n/h_n/r_n\} \subset (T)_{gr} \text{ and} \\ & \{d_1//r_1, \ldots, d_n//r_n, \neg r_0\} \text{ is minimally unsatisfiable}\}. \end{aligned}$$

Definition 3.2 (*Weakest typing precondition—wtp*). Let T be a typing theory, and let r_0 be a ground TR. The expression $wtp(T, r_0)$ defined below is called a weakest typing precondition (wtp) of r_0 in T:

$$wtp(T, r_0) = \bigvee_{W \in gtp(T, r_0)} (W).$$

Intuitively, a gtp of r_0 in T is a precondition under which r_0 is implied by T, and the wtp of r_0 in T is the weakest precondition under which r_0 is implied by T.

Example 3.3. Let $d_0, d_1, d_2, \ldots$ be each a ground TD (i.e. a member of $U[TD]$), and let $\{p_0, p_1, p_2, \ldots\}$ be a subset of $U[KA]$ (i.e. a subset of ground atoms on general relations). Let

$$T_1 = \{d_1, d_2/p_1, d_3/p_1 \wedge p_2/d_1, d_4/\neg p_1 \wedge \neg p_2, d_4/p_3/d_1, d_6//d_5\}.$$

(We use $\square$ for empty disjunction, and use *true* for empty conjunction.) Then we have the following results.

$$\begin{array}{ll} gtp(T_1, d_1) = \{true\}, & wtp(T_1, d_1) = true, \\ gtp(T_1, d_2) = \{p_1\}, & wtp(T_1, d_2) = p_1, \\ gtp(T_1, d_3) = \{p_1 \wedge p_2\}, & wtp(T_1, d_3) = p_1 \wedge p_2, \\ gtp(T_1, d_4) = \{\neg p_1 \wedge \neg p_2, p_3\}, & wtp(T_1, d_4) = (\neg p_1 \wedge \neg p_2) \vee p_3, \\ gtp(T_1, d_6) = \phi, & wtp(T_1, d_6) = \square. \end{array}$$

Proposition 3.4. *Let L be a typed language, and let T be a typing theory. For every $e_1, \ldots, e_n$ of ground TDs, $wtp(T, e_1 \wedge \cdots \wedge e_n)$ is logically equivalent to $wtp(T, e_1) \wedge \cdots \wedge wtp(T, e_n)$.*

Proof. Let M be an interpretation, such that $[wtp(T, e_1 \wedge \cdots \wedge e_n)]^M = true$.[3] From Definition 3.2, there exists a member g of $gtp(T, e_1 \wedge \cdots \wedge e_n)$, such that $[g]^M = true$. From Definition 3.1, there exists $T_0 = \{d_1/h_1/r_1, \ldots, d_k/h_k/r_k\}$, such that $T_0 \subset (T)_{gr}$ and $T_0 \cup \{\neg e_1 \vee \cdots \vee \neg e_n\}$ is minimally unsatisfiable. Then for each j, $1 \leqslant j \leqslant k$, there exists $T_j = \{d_{j1}/h_{j1}/r_{j1}, \ldots, d_{jk}/h_{jk}/r_{jk}\}$, such that $T_j \subset T_0$ and $T_j \cup \{\neg e_j\}$ is minimally unsatisfiable. Let $g_j = h_{j1} \wedge \cdots \wedge h_{jk}$. The $g_j \subset gtp(T, e_j)$. From $[g]^M = true$, and the fact that g_j contains only conjuncts of g, we conclude that $[g_j]^M = true$, and consequently $[wtp(T, e_j)]^M = true$. Hence $wtp(T, e_1 \wedge \cdots \wedge e_n)$ entails $wtp(T, e_1) \wedge \cdots \wedge wtp(T, e_n)$. The proof for other direction of the equivalence is similar. We leave it to the reader. $\square$

For every typing theory T and every Herbrand interpretation B on $U[KA]$, we define a special Herbrand model for T, called a B-based least Herbrand model of T.

Definition 3.5 (*Least Herbrand model*). Let T be a typing theory. For every subset B of $U[KA]$, let

$$M_T^B = \{d \in U[TD] \mid [wtp(T, d)]^B = true\}.$$

Then $B \cup M_T^B$ is called the B-based *least Herbrand model of* T.

[3] In this paper, we use $[e]^M$ to denote the truth value of e in M for every formula e and every interpretation M.

Example 3.6. For the set T_1 given in Example 3.3, we present the B-based least Herbrand model of T_1 for different values of B in the following table.

	B	$M_{T_1}^B$	$B \cup M_{T_1}^B$	$[T_1]^{B\cup M_{T_1}^B}$
1.	$\{\}$	$\{d_1, d_4\}$	$\{d_1, d_4\}$	*true*
2.	$\{p_1\}$	$\{d_1, d_2\}$	$\{p_1, d_1, d_2\}$	*true*
3.	$\{p_1, p_2\}$	$\{d_1, d_2, d_3\}$	$\{p_1, p_2, d_1, d_2, d_3\}$	*true*
4.	$\{p_3\}$	$\{d_1, d_4\}$	$\{p_3, d_1, d_4\}$	*true*
5.	$\{p_1, p_2, p_3\}$	$\{d_1, d_2, d_3, d_4\}$	$\{p_1, p_2, p_3, d_1, d_2, d_3, d_4\}$	*true*

The significance of Definition 3.5 is justified by the following theorem.

Theorem 3.7 (Least Herbrand model theorem). *Let T be a typing theory. For every subset B of $U[KA]$, the B-based least Herbrand model of T, $B \cup M_T^B$, defined in Definition 3.5, is indeed a model of T.*

Proof. Assume that $I = B \cup M_T^B$ is not a model of T. We are going to deduce a contradiction from this assumption. Since I is not a model of T, then there exists at least one member $d_0/h/d_1 \wedge \cdots \wedge d_n$ of $(T)_{gr}$ which is false in I. Then we must have $[d_0]^I = \mathit{false}$, $[h]^I = \mathit{true}$, and for every j, $1 \leqslant j \leqslant n$, $[d_j]^I = \mathit{true}$. Since $d_j \notin B$, $[d_j]^I = \mathit{true}$ implies that $[d_j]^{M_T^B} = \mathit{true}$. Then from the definition of M_T^B, $[\mathit{wtp}(T, d_j)]^B = \mathit{true}$. From Definition 3.1, there exists at least one element g_j of $\mathit{gtp}(T, d_j)$ which is true in B. Consequently, there exists a set $T_j \subset (T)_{gr}$, $T_j = \{d_{j_1}/h_{j_1}/r_{j_1}, \ldots, d_{j_k}/h_{j_k}/r_{j_k}\}$, such that $g_j = h_{j_1} \wedge \cdots \wedge h_{j_k}$, and $T_j|_{TD} \cup \{\neg d_j\}$ is minimally unsatisfiable. Let $T_0 = T_1 \cup \cdots \cup T_n \cup \{d_0/h/d_1 \wedge \cdots \wedge d_n\}$. $T_0|_{TD} \cup \{\neg d_0\}$ must be unsatisfiable. Let $T_0' = \{d^1/h^1/r^1, \ldots, d^s/h^s/r^s\}$ be a subset of T_0, such that $T_0'|_{TD} \cup \{\neg d_0\}$ is minimally unsatisfiable. Then $g' = h^1 \wedge \cdots \wedge h^s$ must be a member of $\mathit{gtp}(T, d_0)$, and hence a disjunct of $\mathit{wtp}(T, d_0)$. However, each conjunct h^l of g' must be either h (if h^l comes from $d_0/h/d_1 \wedge \cdots \wedge d_n$) or a conjunct h_{j_i} of g_j for some j, $1 \leqslant j \leqslant n$, $j_1 \leqslant j_i \leqslant j_k$ (if h^l comes from an element of S_j). For each case, it is easy to determine $[h]^B = \mathit{true}$, and so we conclude that $[g']^B = \mathit{true}$. Since g' is a disjunct of $\mathit{wtp}(T, d_0)$, $[\mathit{wtp}(T, d_0)]^B = \mathit{true}$. Consequently, $d_0 \in M_T^B$, and $[d_0]^I = \mathit{true}$. This result contradicts the earlier assertion that $[d_0]^I = \mathit{false}$. □

3.2. The typed Herbrand theorem

Lemma 3.8. *Let T be a typing theory, and let S_0 be a set of ground typed clauses. S_0 is T-unsatisfiable (i.e., $T \cup S_0$ is unsatisfiable) iff the set $S_w = \{K \vee \neg \mathit{wtp}(T, R) \mid K/R \in S_0$, $\mathit{wtp}(T, R) \neq \Box\}$ is unsatisfiable.*

Proof. Assume that S_w is satisfiable. Let B be a Herbrand model of S_w. Since S_w contains no TDs, we can assume that B is a subset of $U[KA]$. Let $I = B \cup M_T^B$, where $M_T^B = \{d \in U[TD] \mid [\mathit{wtp}(T, d)]^B = \mathit{true}\}$. By Theorem 3.7, I is a model of T. Let K'/R' be an arbitrary element of S_0. We now show that K'/R' is true in I. Assuming

$[K'/R']^I = \mathit{false}$, we must have $[K']^I = \mathit{false}$, and $[R']^I = \mathit{true}$. Since B is a model of S_w, $[K' \vee \neg wtp(T,R')]^B = \mathit{true}$. Consequently, $[K' \vee \neg wtp(T,R')]^I = \mathit{true}$. Then from $[K']^I = \mathit{false}$, we conclude that $[wtp(T,R')]^I = [wtp(T,R')]^B = \mathit{false}$. Since R' is a ground TR, then $R' = d_1 \wedge \cdots \wedge d_n$, for some TDs $d_1, \ldots, d_n$. By Proposition 3.4, $wtp(T,R') = wtp(T,d_1) \wedge \cdots \wedge wtp(T,d_n)$. Then there must exist a d_j of R', $[wtp(T,d_j)]^B = \mathit{false}$. Hence, from the definition of M_T^B, $d_j \notin M_T^B$. Since the set of TDs of R' is disjoint from B, $d_j \notin M_T^B$ implies that $[d_j]^I = \mathit{false}$. Consequently, $[R']^I = \mathit{false}$, which contradicts the assertion that $[R']^I = \mathit{true}$ derived earlier. Therefore K'/R' must be true in I. Note that K'/R' is chosen arbitrarily from S_0. Then I must be a model of S_0. We have noted earlier that I is also a model of T. Therefore I is a model of $T \cup S_0$. This result contradicts the hypothesis that S_0 is T-unsatisfiable.

On the other hand, since each element of S_w is a logical consequence of $T \cup S_0$, if S_w is unsatisfiable, then S_0 must be T-unsatisfiable. □

Definition 3.9 (*Gtp-descendant*). Let T be a typing theory, and let K/R be a typed clause. A ground clause C' is called a gtp-descendant of K/R in T iff there exists a ground instance K'/R' of K/R and a gtp g of R' in T, such that $C' = K' \vee \neg g$. If $g = \mathit{true}$, then C' is called a *proper gtp-descendant* of K/R in T.

Example 3.10. Consider the typing theory T_1 given in Example 3.3 and a clause $K/d_1 \wedge d_2 \wedge d_3$. Note that $p_1 \wedge p_2$ is a gtp of $d_1 \wedge d_2 \wedge d_3$ in T_1. Then for any ground instance K' of K, $K' \vee \neg p_1 \vee \neg p_2$ is a gtp-descendant of $K/d_1 \wedge d_2 \wedge d_5$. Now consider the clause K/d_1. Since $wtp(T', d_1) = \mathit{true}$, K' is a proper gtp-descendant of K/d_1.

Theorem 3.11 (Typed Herbrand theorem). *Let T be a conditional typing theory, and let S be a set of typed clauses. S is T-unsatisfiable iff there exists a finite unsatisfiable set S' of gtp-descendants of clauses of S in T.*

Proof. According to the Herbrand theorem, S is T-unsatisfiable iff there exists a finite set T' of ground instances of clauses of T and a finite set S' of ground instances of clauses of S such that S' is T'-unsatisfiable. By Lemma 3.8, S' is T'-unsatisfiable iff the set $S_w = \{K' \vee \neg wtp(T', R') \mid K'/R' \in S'\}$ is unsatisfiable. To complete the proof of the theorem, we now need only to show that S_w is a finite set of gtp-descendants of clauses of S in T. Note that T' is finite. Then for each ground TR R', $gtp(T', R')$ is finite, and then $wtp(T', R')$ must be a disjunction of a finite number of gtps. Then, by Definition 3.9, each $K' \vee \neg wtp(T', R')$ of S_w stands for a finite set (or say, a finite conjunction) of gtp-descendants of K/R in T. Therefore, given that S' is finite, S_w must be a finite set of gtp-descendants of clauses of S in T. □

3.3. Remarks

Remark 1. The set of clauses contained in a typing theory forms a special class, call *conditional Horn set* (cHs). Let $\mathcal{L}$ be a first-order language. Let R be a set of atoms of $\mathcal{L}$, which is closed under substitutions. A set S of clauses is called a cHs with respect to R iff the restriction of S on R, $S|_R$, is a Horn set. A typing theory is a special form of cHs, which is a cHs with respect to the entire set of TDs. The least Herbrand model theorem, Theorem 3.7, can be generalized to any cHs [35].

It is well known that each Horn set S has a least Herbrand model, which equals the set of ground atoms that can be deduced from S [30]. We claim that Theorem 3.7 is consistent with this classical result in the following sense:

Proposition 3.12. *If T is nonconditional, then for every Herbrand interpretation B, the set $M_T^B = \{d \in U[TD] \mid [wtp(T,d)]^B = true\}$ given in Theorem 3.7 is exactly the least Herbrand model of T: $\{d \in U[TD] \mid T \models d\}$.*

Proof. It follows immediately from the following proposition. $\square$

Proposition 3.13. *If T is a nonconditional typing theory, then for every d of $U[TD]$, $wtp(T,d) = true$ iff $T \models d$; and $wtp(T,d) = \square$ iff $T \not\models d$. Consequently, for every Herbrand interpretation B on $U[KA]$, $[wtp(T,d)]^B = true$ iff $T \models d$, and $[wtp(T,d)]^B = false$ iff $T \not\models d$.*

Proof. If T is nonconditional, then if $T \models d$, $gtp(T,d)$ must contain the empty clause *true*, and so $wtp(T,d) = true$. If $T \not\models d$, then $gtp(T,d)$ must be empty, and so $wtp(T,d) = \square$. On the other hand, if $[wtp(T,d)]^B = true$, then since T is nonconditional, then $wtp(T,d) = true$, and so $T \models d$. If $[wtp(T,d)]^B = false$, then $wtp(T,d) = \square$, and $gtp(T,d) = \phi$. So $T \not\models d$. $\square$

Remark 2. For nonconditional typing theory, our typed Herbrand theorem is similar to Frisch's sorted Herbrand theorem given in [13]. The sorted Herbrand theorem states that for every set Σ of nonconditional typing rules (i.e., S-sentences of [13]) and every set α of typed clauses (i.e., R-sentences), $\alpha \cup \Sigma$ is unsatisfiable iff $\alpha_{\Sigma_{gr}}$ is, where $\alpha_{\Sigma_{gr}}$ is the set of those ground instances of members of α that are well sorted with respect to Σ. According to our Lemma 3.8, $\alpha \cup \Sigma$ is unsatisfiable iff $S_w = \{K \vee \neg wtp(\Sigma, R) \mid K/R \in (\alpha)_{gr} \wedge wtp(\Sigma, R) \neq \square\}$ is. We now show $S_w = \alpha_{\Sigma_{gr}}$.

Since Σ is nonconditional, by Proposition 3.13, $wtp(\Sigma, R) \neq \square$, iff $\Sigma \models R$, and iff $wtp(\Sigma, R) = true$. Then for each element of S_w, we have $K \vee \neg wtp(\Sigma, R) = K$. Hence $S_w = \{K \mid K/R \in (\alpha)_{gr} \wedge \Sigma \models R\}$. Then by the definition of $\alpha_{\Sigma_{gr}}$, S_w equals $\alpha_{\Sigma_{gr}}$.

A general version of the Herbrand theorem with constraint theory is given by Bürckert in [7,8]. This theorem does not require that the underlying constraint theory must be a Horn set. The sorted Herbrand theorem described by Frisch in [14] also releases the restriction on Horn sets to a certain extent. However, these Herbrand theorems are limited in the applicability to those constraint theories that contain no general predicates. It may be an interesting topic to develop a super Herbrand theorem of which each of these existing Herbrand theorems becomes a special case.

4. The typed resolution principle

Given a typing theory T and a T-unsatisfiable set S of typed clauses, a refutation proof of $T \cup S$ by applying the classical resolution principle may require all of the following five kinds of binary resolution (recall that a formula of form K/R represents a typed clause, and a formula of form $D/H/R$ represents a typing rule):

(1) binary resolution of K_1/R_1 and K_2/R_2 upon literals of K_1 and K_2;
(2) binary resolution of K_1/R_1 and $D/H/R$ upon D and a literal of $\neg R_1$;
(3) binary resolution of K_1/R_1 and $D/H/R$ upon literals of K_1 and $\neg H$;
(4) binary resolution of $D_1/H_1/R_1$ and $D_2/H_2/R_2$ upon D_2 and a literal of $\neg R_1$;
(5) binary resolution of $D_1/H_1/R_1$ and $D_2/H_2/R_2$ upon literals of $\neg H_1$ and $\neg H_2$.

Clearly, for our typed calculus, some of these are not good to use. Not only may they contribute too much redundancy, but they may also create some problems which are difficult to handle. For instance, since the third and fourth kinds of resolution may produce a resolvent which is a typing rule, they will force us to deal with a changeable typing theory. The use of the fifth kind of resolution is more troublesome: it may produce a resolvent of the form $D'_1 \vee D'_2/H'/R'$, where D'_1 and D'_2 are both TDs, which is neither a typed clause nor a typing rule. In order to avoid using these kinds of unwanted resolution, we introduce in this section a special version of the resolution principle, called the *typed resolution principle.* This principle will use only the first and second kinds of binary resolution.

Definition 4.1. Let T be a typing theory, R a type restriction, and K/R a typed clause. R is called *T-acceptable* iff $T|_{TD} \models \exists(R)$; R is called *T-provable* iff $\|T\| \models \exists(R)$. K/R is called *T-acceptable* (*T-provable*) iff R is *T-acceptable* (*T-provable*).

The kernel resolution defined below is close to the RQ-resolution rule of [7].

Definition 4.2 (*Kernel resolution*). Let K_1/R_1 and K_2/R_2 be two typed clauses containing no variables in common. Let L_1 be a literal of K_1, and L_2 a literal of K_2. If L_1 and $\neg L_2$ are unifiable, and θ is a most general unifier (mgu) of L_1 and $\neg L_2$, then $(K_1 \backslash L_1)\theta \vee (K_2 \backslash L_2)\theta/(R_1 \wedge R_2)\theta$ is called a *binary kernel resolvent* (BKR) of K_1/R_1 and K_2/R_2.

Similar to the factoring rule for an ordinary clause, *factoring* for a typed clause K/R is an inference rule which deduces factors from the clause. If two or more literals of K, or two or more TDs of R have a mgu σ, then $K\sigma/R\sigma$ obtained by applying σ to K/R is called a *factor* of K/R. As was mentioned in [40], factoring is often not used in actual problem solving, but it needs to be incorporated for establishing the completeness of a proof procedure. This remark applies also to our typed resolution.

Let C_1 and C_2 be typed clauses. A *kernel resolvent* of C_1 and C_2 is a BKR of C_1 and C_2, a BKR of C_1 and a factor of C_2, a BKR of a factor of C_1 and C_2, or a BKR of a factor of C_1 and a factor of C_2.

Definition 4.3 (*TP-resolution*). Let K_1/R_1 be a typed clause and let d_1 be a conjunct of R_1. Let $D/H/R$ be a typing rule that contains no variables in common with K_1/R_1. If d_1 and D are unifiable, and θ is a mgu of d_1 and D, then $K_1\theta \vee \neg H\theta/(R_1 \backslash d_1)\theta \wedge R\theta$ is called a TP-resolvent of K_1/R_1 and $D/H/R$ upon d_1.

A sequence of TP-resolution steps starting from a typed clause will be called a TP-deduction.

Definition 4.4 (*TP-descendant, TP-deduction*). Let T be a typing theory, and let K_0/R_0 be a typed clause. A typed clause K_n/R_n is called a TP-descendant of K_0/R_0 in T iff there exists a sequence D: $K_0/R_0, K_1/R_1, \ldots, K_n/R_n$, such that for each i, $1 \leqslant i \leqslant n$, K_i/R_i is a TP-resolvent of K_{i-1}/R_{i-1} in T. D is called a *TP-deduction* of K_n/R_n from K_0/R_0 in T.

Our typed resolution principle consists of two deduction rules: kernel resolution and TP-resolution. With this principle, we can propose a general version of typed deduction, TP0 deduction, by the following definition.

Definition 4.5 (*TP0 deduction*). Let T be a typing theory, and let S be a set of typed clauses. A finite sequence D: $K_1/R_1, K_2/R_2, \ldots, K_n/R_n$ $(= K/R)$ is called a TP0 deduction of K/R from S in T iff each K_i/R_i is
(1) a clause of S, or
(2) a T-acceptable kernel resolvent of two typed clauses of D preceding K_i/R_i, or
(3) a T-provable TP-descendant of a clause of D preceding K_i/R_i.
A TP0 deduction of $\square/\mathit{true}$ from S in T is called a TP0 refutation of S in T.

The process of verifying if a kernel resolvent is T-acceptable or T-provable is called *type-checking*. We need to point out that the requirement that a kernel resolvent be T-acceptable and a TP-descendant be T-provable in our typed deduction procedures (TP0 deduction and those to be defined later) is optional. Type-checking can be implemented flexibly. In one extreme, it can be abandoned completely. In the other extreme, it may be applied to each resolvent. In between, it may be applied to a subset of resolvents, and implemented flexibly by choosing different levels of reasoning and resource bounds. This argument has been made in the constrained resolution literature (see [8,16]).

Example 4.6. Let $T_{4.1} = \{t_1, t_2, t_3, t_4\}$ be a typing theory, and let $S_{4.1} = \{h_1, h_2, h_3, h_4\}$ be a set of typed clauses. Prove that $S_{4.1}$ is $T_{4.1}$-unsatisfiable.

t_1.	$a: E$	h_1.	$s(x, f(x)) \vee v(x)/x: E$
t_2.	$a: P$	h_2.	$c(f(x)) \vee v(x)/x: E$
t_3.	$f(x): Q//x: E$	h_3.	$\neg v(x)/x: P$
t_4.	$x: P/s(a, x)/x: Q$	h_4.	$\neg c(x)/x: P$

First, we deduce a TP-descendant c_1 of h_4 and a TP-descendant c_2 of h_3. (We shall use TP and KR to stand for TP-deduction and kernel resolution, respectively.)

c_1. $\neg c(f(x)) \vee \neg s(a, f(x))/x{:}\, E$ (TP. h_4, t_4, t_3)
c_2. $\neg v(f(x)) \vee \neg s(a, f(x))/x{:}\, E$ (TP. h_3, t_4, t_3)

Then, we deduce four kernel resolvents, c_3–c_6, by kernel resolution, and finally, deduce the empty clause from c_6 by TP-deduction.

c_3.	$s(x, f(x))/x{:}\, E \wedge x{:}\, P$	(KR. h_1, h_3)
c_4.	$c(f(x))/x{:}\, E \wedge x{:}\, P$	(KR. h_2, h_3)
c_5.	$\neg s(a, f(x))/x{:}\, E \wedge x{:}\, P$	(KR. c_4, c_1)
c_6.	$\Box/x{:}\, E \wedge x{:}\, P$	(KR. c_5, c_3)
c_7.	$\Box/true$	(TP. c_6, t_1, t_2).

Note that for nonconditional typing theory T, a typed clause is T-acceptable iff it is T-provable. This means that TP-deduction can be replaced completely by type-checking, and a refutation is found once a T-provable kernel-empty resolvent is deduced. (A kernel-empty resolvent is a typed clause of form $\Box/R$.) In such a case, TP0 will be similar to Bürckert's RQ-resolution (when the restriction theory is given by a definite clause theory) [7], and Frisch's sorted deduction [14]. Thus the following theorem is essentially a special version of the completeness theorem of RQS deduction and a special version of the completeness theorem of Frisch's sorted deduction.

Theorem 4.7. *Let T be a typing theory, and let S be a set of typed clauses. If T is nonconditional (i.e., $T = \|T\|$), and if S is T-unsatisfiable, then there must exist a TP0 refutation of S in T.*

The theorem can be proved by using the typed Herbrand theorem (Theorem 3.11) and by lifting the refutation for a ground case in the conventional way. We leave the formal proof to the reader.

For conditional typing theory, TP0 deduction is also complete. This completeness can be established on the basis of the typed Herbrand theorem, Theorem 4.7, and a lemma given below.

Lemma 4.8. *Let T be a typing theory, and let K/R be a typed clause. If C' is a gtp-descendant of K/R in T, then there exists a TP-deduction of K_n/R_n from K/R such that K_n/R_n is T-provable and C' is a proper gtp-descendant of K_n/R_n.*

Proof. This lemma can be viewed as a special case of Lemma 5.8 to be proved in the next section. $\Box$

Theorem 4.9 (Completeness theorem of TP0). *If S is a T-unsatisfiable set of typed clauses, then there must exist a TP0 refutation of S in T.*

Proof. By the typed Herbrand theorem, there exists a finite unsatisfiable set S_{gtp} of gtp-descendants of clauses of S in T. Let $C' \in S_{gtp}$ be a gtp-descendant of a clause K/R of S. By Lemma 4.8, there exists a TP-deduction of K_j/R_j from K/R in T, such that

C' is a proper gtp-descendant of K_j/R_j in T, and K_j/R_j is T-provable. Since each such TP-deduction consists of only a finite number of steps of TP-resolution, there must exist a deduction D_1 of a set S_k from S by TP-deduction, such that each element of S_{gtp} is a proper gtp-descendant of a clause of $S \cup S_k$ in T. Then $S \cup S_k$ is $\|T\|$-unsatisfiable. By Theorem 4.7, there exists a TP0 refutation D_2 of $S \cup S_k$ in $\|T\|$. Then the entire deduction formed by appending D_2 onto the end of D_1 is a TP0 refutation of S in T. □

We point out that this completeness theorem is not strong enough to support the practical use of our typed resolution principle. This is because the proof requires that one must apply TP-deduction first to derive a set of consequences that is unsatisfiable in the nonconditional subset of the given typing theory. This process is called weakening. (However, our use of "weakening" may be in a different sense from the use in a sorted calculus literature, such as [33].) A TP0 deduction with weakening is called a *TP-first deduction*. It has been known that weakening may be very inefficient because, in general, there exist no criteria, except the final derivation of the empty clause, by which one can decide when the TP-deduction phase should be terminated; but during weakening, many useless TP-resolvents may have to be deduced. The TP-descendant c_2 of h_2 in Example 4.6 is such a redundant TP-resolvent. Assuming there exist in $T_{4.1}$ additional N conditional typing rules,

$$\{d_1: P/h_1/r_1, \ldots, d_n: P/h_n/r_n\},$$

since each of them can be used to contribute two TP-descendants (one with h_3 and the other with h_4), there will possibly be additional 2N TP-resolvents generated. It turns out that the general TP-deduction may be inefficient when it is used in the earlier stages of a proof; we should delay the application of the general TP-deduction if this is possible. The TP-second deduction to be presented next is motivated by this consideration.

5. TP-second deduction

5.1. Basic definitions

We first introduce a special form $K|P/R$ for a typed clause, called a divided typed clause. $K|P$ is called a divided clause; and K and P are called the *left-kernel* and the *right-kernel*, respectively. Semantically, $K|P$ is the same as a standard clause $K \vee P$. However, for a standard clause, we assume that it contains no duplicated literal occurrences. That is, $K \vee P$ indicates a clause obtained by removing each literal of P which has an occurrence in K (i.e. merging right). For a divided typed clause $K|P/R$, we assume only that there exist no duplicated literal occurrences within the subclauses K and P, respectively. But K and P may share some literal occurrences. Note that, because R is a type restriction, $\neg R$ cannot share a literal occurrence with K or P. Correspondingly, a factor of a divided clause $K|P/R$ is obtained from an instantiation $K\theta|P\theta/R\theta$ of the clause by merging the same literal occurrences in $K\theta$ and in $P\theta$, respectively. Such a factor is called a *divided factor*.

Normally, the *left-kernel* (i.e. K of $K|P/R$) is used to hold relations contained in, or inherited from, a kernel formulation, whereas the *right-kernel* (i.e., P of $K|P/R$) is used to hold (general) relations inherited from the typing theory. Thus, a divided typed clause from an input set will usually be in the form $K|\square/R$ (written by K/R for simplicity). A kernel-empty clause and its TP-descendants will be in the form $\square|P/R$.

Why do we use divided typed clauses? Note that the general relations contained in the underlying typing theory will be hidden from kernel resolution until they are inherited by the kernel of a TP-descendant through TP-deduction. Restricting the use of TP-deduction will have the side effect of restricting the reasoning associated with these general relations. In order to avoid the proof failure possibly caused by this side effect, we need to keep track of these relations in the course of searching for a proof, and use special inference rules or control strategies to deal with the deduction on these relations. Divided clause is introduced here partly for this purpose.

Why are the subclauses K and P of a divided typed clause $K|P/R$ allowed to share the same literal occurrences? This is because a literal L of the clause P may stand for a literal to be inherited from the typing theory by TP-deduction. Before the inheritance, L is implicit, so cannot be merged into K even if K contains an occurrence L. Of course, when L is explicitly presented, L can be removed by merging. But this merging needs to be considered as a refinement, instead of a default operation.

Definition 5.1. Let T be a typing theory, and let $K_1|P_1/R_1$ and $K_2|P_2/R_2$ be two divided typed clauses that contain no variables in common. A *binary kernel resolvent* (BKR) of $K_1|P_1/R_1$ and $K_2|P_2/R_2$ in T is a form $K|P/R$ defined by considering the following three cases.

(1) $L_1 \in K_1$, $L_2 \in K_2$, θ is a mgu of L_1 and $\neg L_2$, $K = (K_1\backslash L_1)\theta \vee (K_2\backslash L_2)\theta$, $P = (P_1 \vee P_2)\theta$, and $R = R_1\theta \wedge R_2\theta$;

(2) $L_1 \in K_1$, $K_2 = \square$, $L_2 \in P_2$, θ is a mgu of L_1 and $\neg L_2$, $K = (K_1\backslash L_1)\theta$, $P = P_1\theta \vee (P_2\backslash L_2)\theta$, and $R = R_1\theta \wedge R_2\theta$;

(3) $K_1 = \square$, $K_2 = \square$, $L_1 \in P_1$, $L_2 \in P_2$, θ is a mgu of L_1 and $\neg L_2$, $K = \square$, $P = (P_1\backslash L_1)\theta \vee (P_2\backslash L_2)\theta$, and $R = R_1\theta \wedge R_2\theta$.

Let C_1 and C_2 be divided typed clauses. A *kernel resolvent* of C_1 and C_2 is a BKR of C_1 and C_2, a BKR of C_1 and a divided factor of C_2, a BKR of a divided factor of C_1 and C_2, or a BKR of a divided factor of C_1 and a divided factor of C_2.

We now extend the concepts of gtp-descendant, TP-resolvent, and TP-descendant introduced earlier to divided clauses.

Definition 5.2. Let T be a typing theory, and let $K|P/R$ be a divided typed clause. A ground clause $K'|W'$ is called a *gtp-descendant* of $K|P/R$ iff there exist a ground instance $K'|P'/R'$ of $K|P/R$ and a gtp g of R' in T such that $W' = P' \vee \neg g$. If $g = true$, then $K'|W'$ is called a *proper divided gtp-descendant* of $K|P/R$ in T.

Definition 5.3. Let $K_1|P_1/R_1$ be a typed clause, and let d_1 be a conjunct of R_1. Let $D/H/R$ be a typing rule that contains no variables in common with $K_1|P_1/R_1$. If d_1 and

D are unifiable and θ is a mgu of d_1 and D, then $K_1\theta|(P_1 \vee \neg H)\theta/(R_1\backslash d_1)\theta \wedge R\theta$ is called a divided *TP-resolvent* of $K_1|P_1/R_1$ and $D/H/R$ upon d_1.

Definition 5.4. Let T be a typing theory, and let $K_0|P_0/R_0$ be a typed clause. A typed clause $K_n|P_n/R_n$ is called a divided *TP-descendant* of $K_0|P_0/R_0$ in T iff there exists a sequence

$$D\colon K_0|P_0/R_0, K_1|P_1/R_1, \ldots, K_n|P_n/R_n$$

such that for each i, $1 \leqslant i \leqslant n$, $K_i|P_i/R_i$ is a divided TP-resolvent of $K_{i-1}|P_{i-1}/R_{i-1}$ in T. D is called a *TP-deduction* of $K_n|P_n/R_n$ from $K_0|P_0/R_0$ in T.

Definition 5.5 (*TP-second deduction*). Let T be a typing theory, and let S be a set of typed divided clauses. A finite sequence $D\colon K_1|P_1/R_1, K_2|P_2/R_2, \ldots, K_n|P_n/R_n$ ($= K|P/R$) is called a TP-second deduction of $K|P/R$ from S in T iff each $K_i|P_i/R_i$ is

(1) a clause of S, or

(2) a T-acceptable kernel resolvent of two typed clauses of D preceding $K_i|P_i/R_i$, or

(3) a T-provable TP-descendant of a kernel-empty clause of D preceding $K_i|P_i/R_i$.

A TP-second deduction of $\Box|\Box/\mathit{true}$ from S in T is called a TP-second refutation of S in T.

TP-second deduction is complete and that is to be proved in Section 5.3. As was mentioned earlier, the requirement that a kernel resolvent be T-acceptable and a TP-descendant be T-provable is optional. The purpose of including them in the definition is to obtain a stronger version of completeness theorem.

5.2. Examples

Example 5.6. Use TP-second deduction to solve Example 4.6.

Since $S_{4.1}$ contains no kernel-empty clause, only kernel resolution (KR) is applicable at the beginning of the deduction. We have the following kernel resolvents (note that $K|\Box/R$ is written as K/R for the simplicity):

p_1.	$s(x, f(x))/x\colon E \wedge x\colon P$	(KR. h_1, h_3)
p_2.	$c(f(x))/x\colon E \wedge x\colon P$	(KR. h_2, h_3)
p_3.	$\Box/x\colon E \wedge x\colon P \wedge f(x)\colon P$	(KR. p_2, h_4)

Note that p_3 is a kernel-empty clause, to which TP-deduction can be applied. This will produce a TP-descendant p_4, and eventually empty clause p_6.

p_4.	$\Box|\neg s(a, f(x))/x\colon E \wedge x\colon P$	(TP. p_3, t_4, t_3)
p_5.	$\Box/a\colon E \wedge a\colon P$	(KR. p_4, p_1)
p_6.	$\Box/\mathit{true}$	(TP. p_5, t_1, t_2)

In comparing the above derivation with that given in Example 4.6, the search space for a TP-second refutation is reduced, because only kernel resolution can be applied before

p_3 is deduced. There are only three kernel resolvents that can possibly be produced. All of them are useful. Note that the need of deducing a kernel-empty resolvent before obtaining finally a refutation can serve as a strong heuristic. According to this heuristic, a kernel-empty resolvent (e.g., P_3 in this example) should be given higher priority to develop.

Example 5.7. Use TP-second deduction to prove AM8-T given in Example 2.4.

The TP-second deduction proof-tree presented below was found automatically by our typed prover in proving this theorem. We shall give an analysis of this proof-tree in Section 7.

g_1.	$f(u) \not\leqslant f(k(u))/u{:}dpt$	Given (b_9)
g_2.	$v \not\leqslant f(k(u)) \vee f(u) \not\leqslant v/r_2$	KR. g_1, a_3
g_3.	$f(u) \not\leqslant f(h(k(u)))/r_3$	KR. g_2, b_7
g_4.	$a \not\leqslant h(k(l)) \vee h(k(l)) \not\leqslant l/r_4$	KR. g_3, b_3
g_5.	$h(k(l)) \not\leqslant l/r_5$	KR. g_4, b_1
g_6.	$a \leqslant q(h(k(l)))/r_6$	KR. g_5, b_5
g_7.	$q(h(k(l))) \leqslant h(k(l))/r_7$	KR. g_5, b_4
g_8.	$f(h(k(l))) \not\leqslant f(q(h(k(l))))/r_8$	KR. g_5, b_6
g_9.	$q(h(k(l))) \not\leqslant h(k(l)) \vee h(k(l)) \not\leqslant q(h(k(l)))/r_9$	KR. g_8, a_4
g_{10}.	$v \not\leqslant f(q(h(k(l)))) \vee f(h(k(l))) \not\leqslant v/r_{10}$	KR. g_8, a_3
g_{11}.	$h(k(l)) \not\leqslant q(h(k(l)))/r_{11}$	KR. g_9, g_7
g_{12}.	$v \not\leqslant q(h(k(l))) \vee h(k(l)) \not\leqslant v/r_{12}$	KR. g_{11}, a_3
g_{13}.	$f(q(h(k(l)))) \not\leqslant f(k(l))/r_{13}$	KR. g_{11}, b_8
g_{14}.	$f(k(l)) \leqslant f(q(h(k(l))))/r_{14}$	KR. g_{13}, a_2
g_{15}.	$f(h(k(l))) \not\leqslant f(k(l))/r_{15}$	KR. g_{14}, g_{10}
g_{16}.	$\square/r_{16}$	KR. g_{15}, b_7
g_{17}.	$\square \vert a \not\leqslant q(h(k(l))) \vee h(k(l)) \leqslant l \vee (h(k(l))) \not\leqslant b/r_{17}$	TP. g_{16}, k_3–k_9
g_{18}.	$\square \vert q(h(k(l))) \not\leqslant b \vee h(k(l)) \leqslant l/r_{18}$	KR. g_{17}, g_6
g_{19}.	$b \leqslant q(h(k(l))) \vert h(k(l)) \leqslant l/r_{19}$	KR. g_{18}, a_2, k_2, k_6–k_9
g_{20}.	$h(k(l)) \not\leqslant b \vert h(k(l)) \leqslant l/r_{20}$	KR. $g_{19}, g_{12}, k_2, k_4, k_6$–$k_9$
g_{21}.	$\square \vert h(k(l)) \leqslant l/r_{21}$	KR. g_{20}, b_2, k_6–k_8
g_{22}.	$\square/r_{22}$	KR. g_{21}, g_5
g_{23}.	$\square \vert h(k(l)) \leqslant l/true$	TP. g_{22}, k_2, k_4, k_6–k_9
g_{24}.	$\square/true$	KR. g_{23}, g_5

The type restrictions r_2–r_{22} are given bellow. For simplicity, only r_2–r_4 are given with their full expressions. The rest are given only with their simplified results (those on the right hand of $\rightarrow$), produced by a simplification procedure TR-simplifier (to be described briefly later).

$$\begin{array}{ll}
r_2\colon\ f(u){:}fpt \wedge v{:}fpt \wedge f(k(u)){:}fpt \wedge u{:}dpt & \rightarrow v{:}fpt \wedge u{:}dpt \\
r_3\colon\ k(u){:}dpt \wedge f(u){:}fpt \wedge f(h(k(u))){:}fpt \wedge f(k(u)){:}fpt \wedge u{:}dpt & \rightarrow u{:}dpt \\
r_4\colon\ h(k(l)){:}xpt \wedge k(l){:}dpt \wedge f(l){:}fpt \wedge f(h(k(l))){:}fpt \wedge f(k(l)){:}fpt \wedge l{:}dpt & \\
 & \rightarrow true \\
r_5, r_6, r_7, r_8\colon & \rightarrow true \\
r_9\colon & \rightarrow h(k(l)){:}xpt \wedge q(h(k(l))){:}xpt \\
r_{10}\colon & \rightarrow f(h(k(l))){:}fpt \wedge v{:}fpt \wedge f(q(h(k(l)))){:}fpt \\
r_{11}\colon & \rightarrow h(k(l)){:}xpt \wedge q(h(k(l))){:}xpt \\
r_{12}\colon & \rightarrow v{:}xpt \wedge h(k(l)){:}xpt \wedge q(h(k(l))){:}xpt \\
r_{13}\colon & \rightarrow h(k(l)){:}xpt \wedge q(h(k(l))){:}dpt \\
r_{14}, r_{15}, r_{16}\colon & \rightarrow f(q(h(k(l)))){:}fpt \wedge q(h(k(l))){:}dpt \wedge h(k(l)){:}xpt \\
r_{17}, r_{18}, r_{19}\colon & \rightarrow true \\
r_{20}, r_{21}, r_{22}\colon & \rightarrow (b{:}xpt) \wedge (h(k(l)){:}xpt) \wedge q(h(k(l))){:}xpt
\end{array}$$

5.3. Completeness theorem

We now give a formal proof of the completeness of TP-second deduction. Besides confirming the completeness, this formal discussion provides a theoretical base for the further refinement of TP-second deduction.

Lemma 5.8 (Lifting lemma for TP-deduction). *Let T be a typing theory, and let $K|P/R$ be a divided typed clause. If $K'|W'$ is a gtp-descendant of $K|P/R$ in T, then there exists a TP-descendant $K_l|P_l/R_l$ of $K|P/R$ such that $K'|W'$ is a proper gtp-descendant of $K_l|P_l/R_l$.*

Proof (*Outline*). By Definitions 3.1 and 5.2, and well-known completeness property of input resolution on Horn sets, there exist a ground instance $K'|P'/R'$ of $K|P/R$ and a multiset $T_0 = \{d_1/h_1/r_1, \ldots, d_n/h_n/r_n\}$, where for each i, $1 \leqslant i \leqslant n$, $d_i/h_i/r_i \in (T)_{gr}$, such that $T_0|_{TD} \cup \{\neg R'\}$ is unsatisfiable, $W' = P' \vee \neg h_1 \vee \cdots \vee \neg h_n$, and there exists an input refutation $D'\colon \neg R'_0\ (= \neg R'), \neg R'_1, \ldots, \neg R'_n\ (= \Box)$, where for each i, $\neg R'_i$ is a resolvent of $\neg R'_{i-1}$ and an element $d_i//r_i$ of $T_0|_{TD}$. Let $W'_i = P' \vee \neg h_1 \vee \cdots \vee \neg h_i$, and let D'_w be the deduction $K'_0|W'_0/R'_0$ $(= K'|P'/R')$, $K'_1|W'_1/R'_1, \ldots, K'_n|W'_n/R'_n$ $(= K'|W'/true)$. Clearly, each $K'_i|W'_i/R'_i$ is a TP-resolvent of $K'_{i-1}|W'_{i-1}/R'_{i-1}$ and $d_i/r_i/h_i$. By treating each typed clause or typing rule as an ordinary clause, and by the lifting lemma [10] for ordinary clause, D'_w can be "lifted" into a deduction D_p of $K_n|P_n/R_n$ from $K|P/R$ and T, such that $K'_i|W'_i/R'_i$ is an instance of $K_i|P_i/R_i$ (i.e., for some θ_i, $K'_i = K_i\theta_i$, $W'_i = P_i\theta_i$, $R'_i = R_i\theta_i$). By Definition 5.4, D_p is a TP-deduction of $K_n|P_n/R_n$ from $K|P/R$ in T. Let j be n or the smallest index for which $h_{j+1} = \cdots = h_n = true$. Then we must have $W'_j = W'_{j+1} = \cdots = W'_n = W'$ and $\|T_0\| \models R'_j, \ldots, \|T_0\| \models R'_n$. Thus, for every l, $j \leqslant l \leqslant n$, $K'|W' = K'_l|W'_l$, and *true* is a gtp of R'_l in T. By Definition 5.2, $K'|W'$ is a proper gtp-descendant of $K_l|P_l/R_l$ in T. □

Corollary 5.9. *If $K'|W'$ is a proper gtp-descendant of $K|P/R$ in T, then there exists a TP-deduction of $K_l|P_l/true$ from $K|P/R$ in $\|T\|$ such that $K'|W'$ is a ground instance of $K_l|P_l$. If $\Box|\Box$ is a proper gtp-descendant of $\Box|\Box/R$, then $\Box|\Box$ can be deduced from $\Box|\Box/R$ by a TP-deduction in T.*

To prove that TP-second deduction is complete, we need a lemma for the completeness of a *divided ground resolution.* The divided ground resolution is a special TP-second deduction, for which the typing theory is assumed to be empty, and each typed clause is a ground clause whose type restriction is *true* (so it can be written in a simpler form $K|W$). Consider the following example.

Example 5.10. Let S'_0 consist of four divided ground clauses d_1–d_4 given below. It is easy to check that S'_0 is unsatisfiable. The following is a refutation of S'_0 obtained by divided ground resolution.

d_1. $p \vee q\|p$	[given]		d_8. $q\|\neg p$	$[d_7(\|p), d_3(\neg p\|)]$
d_2. $p \vee \neg q\|true$	[given]		d_9. $\neg q\|\neg p$	$[d_7(\|p), d_4(\neg p\|)]$
d_3. $\neg p \vee q\|\neg p$	[given]		d_{10}. $\Box\|\neg p$	$[d_8(q\|), d_9(\neg q\|)]$
d_4. $\neg p \vee \neg q\|true$	[given]		d_{11}. $q\|p$	$[d_{10}(\|\neg p), d_1(p\|)]$
d_5. $p\|p$	$[d_1(q\|), d_2(\neg q\|)]$		d_{12}. $\neg q\|true$	$[d_{10}(\|\neg p), d_2(p\|)]$
d_6. $\neg p\|\neg p$	$[d_3(q\|), d_4(\neg q\|)]$		d_{13}. $\Box\|p$	$[d_{11}(q\|), d_{12}(\neg q\|)]$
d_7. $\Box\|p \vee \neg p$	$[d_5(p\|), d_6(\neg p\|)]$		d_{14}. $\Box\|\Box$	$[d_{10}(\|\neg p), d_{13}(\|p)]$

For divided ground resolution, we do not allow that the duplicated literal occurrences between left-kernels and right-kernels be merged. This restriction is useful later for lifting a divided ground refutation to a divided typed refutation by means of a lifting lemma for divided kernel resolution (Lemma 5.12 to be given soon).

Lemma 5.11. *Let S be a set of ground divided clauses. If S is unsatisfiable, then there exists a deduction of empty clause $\Box|\Box$ from S by divided ground resolution.*[4]

Proof. If $\Box|\Box$ is in S, then the lemma holds trivially. Otherwise, let us proceed by induction on the number of excess literal occurrences of S [1], $EX(S) = \sum_{c\in S}(size(c) - 1)$, where $size(K|W) = size(K) + size(W)$ and $size(\Box) = 0$.

[4] According to the talk *"Resolution Completeness Proofs"* given by Ken Kunen at The 4th Annual Frontiers in Computing Symposium Honoring Professor W. W. Bledsoe, November 1991, Austin, Texas, this lemma appears to be a reinvention of Ken Kunen's research results reported in the Symposium. Lemma 5.11 was originally reported by the author in a manuscript (a preliminary version of this paper) submitted to CADE-11 in October 1991.

Case 1: $EX(S) = 0$. Then S contains only unit clauses. Since S is unsatisfiable, there must exist two complementary units $K_i|W_i$, for $i = 1,2$, in S. From Definition 5.1, $\Box|\Box$ must be a divided resolvent of $K_1|W_1$ and $K_2|W_2$.

Case 2: $EX(S) > 0$. Assume that for all S', $EX(S') < N$, the lemma is true.

Case 2.1. For each $K|W$ of S, $W = \Box$. Let $S_k = \{K \mid K|W \in S\}$. Then S_K must be unsatisfiable. Since S_K is a set of ordinary clauses, there exists a deduction D_K of the empty clause from S_K by any complete resolution procedure for propositional logic. Let D be obtained from D_K by replacing each clause C of D_K by a divided clause $C|\Box$. Then each clause of D is a divided clause of S, or a divided resolvent of D. Since $\Box|\Box$ must be in D, then D is a refutation of S by divided resolution.

Case 2.2. For every $K|W$ of S, $K = \Box$. Let $S_W = \{W \mid K|W \in S\}$. Then S_W is unsatisfiable. Thus, there exists a deduction D_W of the empty clause from S_W by any complete resolution procedure for propositional logic. Let D be obtained from D_W by replacing each clause C of D_W by a divided clause $\Box|C$. Then each clause of D is a divided clause of S, or a divided resolvent of D. Since $\Box|\Box$ must be in D, D is a refutation of S by divided resolution.

Case 2.3. S contains a divided clause $K_0|W_0$ such that neither $K_0 = \Box$ nor $W_0 = \Box$. Let $S_{K_0} = (S - \{K_0|W_0\}) \cup \{K_0|\Box\}$, and let $S_{W_0} = (S - \{K_0|W_0\}) \cup \{\Box|W_0\}$. Then both S_{W_0} and S_{K_0} are unsatisfiable. Since each of $EX(S_{K_0})$ and $EX(S_{W_0})$ is less than N, by the induction hypothesis, there exists a refutation D_{K_0} of S_{K_0} by divided resolution and a refutation D_{W_0} of S_{W_0} by divided resolution. Let D_1 be the deduction obtained from D_{K_0} by putting W_0 back to the right-kernel of $K_0|\Box$ and to the right-kernel of each of its descendants in D_{K_0}. Then D_1 must be a deduction by divided resolution with S, and D_1 contains either a node $\Box|\Box$ or a node $\Box|W_0$. If it is the former, the lemma has been proved. If it is the latter, the deduction D obtained by appending D_{W_0} onto the end of D_1 must form a refutation of S by a divided resolution. $\Box$

Lemma 5.12 (Lifting lemma for divided kernel resolution). *Let T be a typing theory, and let $K_1|P_1/R_1$ and $K_2|P_2/R_2$ be typed clauses containing no variables in common. Let $K_1'|W_1'$ and $K_2'|W_2'$ be gtp-descendants of $K_1|P_1/R_1$ and $K_2|P_2/R_2$, respectively. If $K'|W'$ is a kernel resolvent of $K_1'|W_1'$ and $K_2'|W_2'$, then there exists a kernel resolvent $K|P/R$ of $K_{11}|P_{11}/R_{11}$ and $K_{22}|P_{22}/R_{22}$ such that $K'|W'$ is a gtp-descendant of $K|P/R$ in T, where for $i = 1,2$, if $K_i \neq \Box$ then $K_{ii}|P_{ii}/R_{ii}$ is identical to $K_i|P_i/R_i$, otherwise $K_{ii}|P_{ii}/R_{ii}$ is a T-provable TP-descendant of $K_i|P_i/R_i$.*

Proof. From the definition of gtp-descendant (Definition 5.2) and the hypothesis of the lemma that $K_i'|W_i'$ is a gtp-descendant of $K_i|P_i/R_i$ $(i = 1,2)$, there exists a ground instance $K_i'|P_i'/R_i'$ of $K_i|P_i/R_i$ and a gtp g_i of R_i' such that $W_i' = P_i' \vee \neg g_i'$. By Definition 5.1, we need only to consider the following three cases:

Case 1: $K_1' \neq \Box$, $K_2' \neq \Box$. Then by the definition, K' is a resolvent of K_1' and K_2', and $W' = W_1' \vee W_2'$. Consider the clause $C' = K'|(P_1' \vee P_2')/R_1' \wedge R_2'$. Clearly, C' is a kernel resolvent of $K_1'|P_1'/R_1'$ and $K_2'|P_2'/R_2'$. Then if we treat divided clauses as ordinary clauses and apply the standard lifting lemma [10], there must exist a kernel resolvent C of $K_1|P_1/R_1$ (or a divided factor of it) and $K_2|P_2/R_2$ (or a divided factor of it) such that C' is an instance of C. However, because no merging exists between K_i', P_i', and R'_i in producing C', then C must be a form $K|P/R$, such that for some substitution $\sigma, K' = K\sigma$, $P'_1 \vee P'_2 = P\sigma$, and $R'_1 \wedge R'_2 = R\sigma$. Since g'_i is assumed to be a gtp of R'_i in T, by Definition 3.1 $g'_1 \wedge g'_2$ is a gtp of $R'_1 \wedge R'_2$ in T, and hence a gtp of $R\sigma$. Also note that $W'_1 \vee W'_2 = P'_1 \vee P'_2 \vee \neg g'_1 \vee \neg g'_2$. Then by Definition 5.2, $K'|(W'_1 \vee W'_2)$ is a gtp-descendant of $K|P/R$.

Case 2: $K'_1 \neq \square$, $K'_2 = \square$, L'_1 of K'_1 equals $\neg L'_2$ of W'_2. Then by the definition, $K' = (K'_1 \backslash L'_1)$, and $W' = W'_1 \vee (W'_2 \backslash L'_2)$. By Lemma 5.8, there exists a TP-deduction of $K_{22}|P_{22}/R_{22}$ from $K_2|P_2/R_2$ such that $\square|W'_2$ is a proper gtp-descendant of $K_{22}|P_{22}/R_{22}$ in T. That is, for some substitution λ, $\square = K_{22}\lambda$, $W'_2 = P_{22}\lambda$ and $\|T\| \models R_{22}\lambda$. Thus $\square|P_{22}/R_{22}$ must be T-provable. Note that since $K'_2 = \square$, K_2 must be the empty clause $\square$. Then the TP-deduction of $\square|P_{22}/R_{22}$ from $K_2|P_2/R_2$ obeys the definition of TP-second deduction. Now consider the clause $C' = (K'_1 \backslash L'_1)|P'_1 \vee (P_{22}\lambda \backslash L'_2)/R'_1 \wedge R_{22}\lambda$. Clearly, C' is a kernel resolvent of $K'_1|P'_1/R'_1$ and $\square|P_{22}\lambda/R_{22}\lambda$ upon a literal of K_1 and a literal of $P_{22}\lambda$. Similar to the discussion of Case 1 above, there must exist a resolvent $C = K|P/R$ of $K_1|P_1/R_1$ (or a divided factor of it) and $\square|P_{22}/R_{22}$ (or a divided factor of it) upon a literal L_1 of K_1 and a literal L_2 of P_{22}, such that C' is an instance of C. That is, for some substitution σ, $K'_1 \backslash L'_1 = K\sigma$, $P'_1 \vee (P_{22}\lambda \backslash L'_2) = P\sigma$, and $R'_1 \wedge R_{22}\lambda = R\sigma$. From the facts $W'_1 = P'_1 \vee \neg g'_1$, $W'_2 = P_{22}\lambda$, and $W' = W'_1 \vee (W'_2 \backslash L'_2)$, we have $W' = P'_1 \vee (P_{22}\lambda \backslash L'_2) \vee \neg g'_1$, and hence $W' = P\sigma \vee \neg g_1$. Thus, by Definition 5.2, in order to finally prove that $K'|W'$ is a gtp-descendant of $K|P/R$, we need only to show that g'_1 is a gtp of $R\sigma$. But this is true by the facts that $R\sigma = R'_1 \wedge R_{22}\lambda$, g'_1 is a gtp of R'_1, and *true* is a gtp of $R_{22}\lambda$.

Case 3: $K'_1 = \square$, $K'_2 = \square$. Then by the definition, $K' = \square$, and W' is a resolvent of W'_1 and W'_2. By Lemma 5.8, for $i = 1, 2$, there exists a TP-deduction of $K_{ii}|P_{ii}/R_{ii}$ from $K_i|P_i/R_i$ such that $\square|W'_i$ is a proper gtp-descendant of $K_{ii}|P_{ii}/R_{ii}$. That is, for some λ_i, $\square = K_{ii}\lambda_i$, $W'_i = P_{ii}\lambda_i$, and $\|T\| \models R_{ii}\lambda_i$. Thus, $\square|P_{ii}/R_{ii}$ must be T-provable. Note that since $K'_i = \square$, K_i must be the empty clause $\square$. Then the TP-deduction of $\square|P_{ii}/R_{ii}$ from $K_i|P_i/R_i$ obeys the definition of TP-second deduction. Now consider the clause $C' = \square|(P_{11}\lambda_1 \backslash L'_1) \vee (P_{22}\lambda_2 \backslash L'_2)/R_{11}\lambda_1 \wedge R_{22}\lambda_2$. Clearly, C' is a kernel resolvent of $\square|P_{11}\lambda_1/R_{11}\lambda_1$ and $\square|P_{22}\lambda_2/R_{22}\lambda_2$. Then similar to the discussion of Cases 1 and 2 above, there must exist a kernel resolvent $C = K|P/R$ of $\square|P_{11}/R_{11}$ (or a divided factor of it) and $\square|P_{22}/R_{22}$ (or a divided factor of it) such that C' is an instance of C. That is, for some substitution σ, $(P_{11}\lambda_1 \backslash L'_1) \vee (P_{22}\lambda_2 \backslash L'_2) = P\sigma$, and $R_{11}\lambda_1 \wedge R_{22}\lambda_2 = R\sigma$. Note that from the facts $W'_i = P_{ii}\lambda_i$, and $W' = (W'_1 \backslash L'_1) \vee (W'_2 \backslash L'_2)$, we have $W' = (P_{11}\lambda_1 \backslash L'_1) \vee (P_{22}\lambda \backslash L'_2) = P\sigma$. Then by Definition 5.2, in order to finally prove that $K'|W'$ is a gtp-descendant of $K|P/R$, we need only to show that *true* is a gtp of $R\sigma$. But this is true by the facts that $R\sigma = R_{11}\lambda_1 \wedge R_{22}\lambda_2$, and *true* is a gtp of $R_{11}\lambda_1$ and a gtp of $R_{22}\lambda_2$. $\square$

Theorem 5.13 (Completeness of TP-second deduction). *Let T be a typing theory, and let S be a set of divided typed clauses. If S is T-unsatisfiable, then there exists a TP-second refutation of S in T.*

Proof. Since S is T-unsatisfiable, by the typed Herbrand theorem, there exists a finite set S_{gtp} of gtp-descendants of clauses of S in T such that S_{gtp} is unsatisfiable. Assume that each member of S_{gtp} is written in the form of a divided gtp-descendant as defined by Definition 5.1. Then by Lemma 5.11, there exists a refutation D' of S_{gtp} by divided resolution. We now show that we can transform D' into a TP-second refutation D of S in T. First, for each leaf node of D', replace the divided clause by the clause in S of which the divided clause is a gtp-descendant. Then for each non-leaf node, if its two parent nodes have been replaced by clauses of S or typed clauses deduced from S by TP-second deduction, replace it with the typed clause that can be deduced from these parent clauses by TP-second deduction in T confirmed by Lemma 5.12. Finally, replace the empty clause $\square\|\square$ of D' by a typed clause, of which $\square\|\square$ is a gtp-descendant. This clause must be of form $\square\|\square/R$, and $\square\|\square$ is a proper gtp-descendant of $\square\|\square/R$. Then by Corollary 5.9, $\square\|\square$ can be deduced from $\square\|\square/R$ by a TP-deduction in T. $\square$

6. On the refinements of typed binary resolution

We have described the typed resolution principle for providing the basic inference rules for the typed predicate calculus. However, unlimited application of the principle may produce many redundant resolvents. Although there exists a TP-second deduction for which the redundancy from TP-resolution can be reduced significantly, many redundant resolvents may be produced by unconstrained kernel resolution. By noting that the kernel resolution is close to the classical binary resolution, it is natural to consider the improvement of the kernel resolution by extending existing refinements of binary resolution. A formal study of this extension is outside of the scope of this paper. Here we only present an overview of the results from our preliminary study in this extension.

6.1. On the refinements with nonconditional typing theory

If only nonconditional typing theory is considered, then a refinement of resolution can be extended to typed deduction by a direct transformation similar to Frisch's framework [14] for transforming an unsorted deduction procedure to a sorted one. Actually, each sort in the sorted logic corresponds to an atomic type in our typed language; and a nonconditional typing theory can be viewed as a kind of sorting theory which has a least Herbrand model. Frisch indicated that each of those refinements of resolution that can be proved to be complete by usual method based on Herbrand theorem (i.e. first proving the ground completeness, then lifting the ground proof) can be extended to be a completeness-preserving refinement of sorted deduction by a direct transformation. This result holds in our typed domain.

Thus, following Frisch's framework, we outline the transformation by the following steps. Let P be a resolution-based deduction procedure.

(1) Transform each clause of P into a typed clause.
(2) Transform each resolution rule of P into a typed resolution rule in a similar manner to transforming binary resolution into the kernel binary resolution stated in Definition 4.2.
(3) Extend all constraints attached to the clauses and resolvents of P into the corresponding restrictions to the typed clauses and typed resolvents.

(4) Apply type-checking to kernel resolvents (this is optional to non kernel-empty resolvents).

6.2. *On the refinements with conditional typing theory*

In dealing with conditional typing theory, a simple approach is to use the direct transformation as outlined above, plus a use of TP-deduction. However, such a simple approach may result in a deduction system that is incomplete. Consider the problem of Example 4.6. By treating TDs and TRs as ordinary literals and conjunctions of ordinary literals, respectively, and by using an untyped linear deduction procedure starting from $h_1 = s(x, f(x)) \vee v(x)/x{:}\,E$, we can obtain a (ordered) linear refutation of the set of clauses, $T_{4.1} \cup S_{4.1}$. Assuming that the left-most literal of the goal, i.e., $s(x, f(x))$ of h_1, was the literal resolved upon, then the clause $t_4 = x{:}\,P/s(a,x)/x{:}\,Q$ must be used as a side clause of the goal h_1 for this refutation. Now let us use a typed (ordered) linear deduction, directly transformed from the linear deduction to solve the problem, and again use h_1 as the starting goal. Note that, now t_4 becomes a part of the underlying typing theory, which can no longer be a side clause for the goal h_1. Then the proof will fail immediately.

It turns out that in dealing with conditional typing theory, an important question is how to make a typed refinement complete. A simple approach is to use weakening, that is, to apply TP-deduction first for deriving a set of consequences which is unsatisfiable in the nonconditional subset of the given typing theory. After that, because only the nonconditional portion of the typing theory is used, the direct transformation mentioned earlier can be used for incorporating refinements of untyped resolution. However, as we have mentioned earlier, weakening is unsatisfactory.

6.3. *On the completeness-preserving refinements of TP-second deduction*

A more interesting question is how to transform a refinement of resolution into TP-second deduction without introducing incompleteness. Note that for TP-second deduction, the left-kernel of a typed clause can be freely used in the deduction. But a literal of the right-kernel of a typed clause is usually inherited from a general condition of the given typing theory, which has been prevented to be resolved upon before the inheritance. Then, a strategy for avoiding incompleteness is to attach the additional constraints to the left-kernel only. For example, we can require that the left-kernel of each kernel resolvent produced by TP-second deduction not be a tautology. One can prove with a proof similar to the proof of the completeness of TP-second deduction that this strategy is complete. However, if we require that the entire kernel not be a tautology, then the procedure will no longer be complete. For a counterexample, note that a tautology d_7 in Example 5.10 is needed for the proof.

Our preliminary study indicates that, with this strategy, it is possible to extend unit resolution [10], lock resolution [6], and semantic resolution [10], into TP-second deduction without introducing incompleteness. The extension can be done by essentially the same steps described earlier for handling nonconditional typing theory. However, in order to avoid introducing incompleteness, we need to extend it in such a manner, that the constraints associated with the refinement will be attached to the left-kernel of a divided typed clause only. For instance, for extending the lock deduction into TP-second deduction, we attach an index only to each literal of left-kernels of input divided typed clauses and their divided kernel resolvents. Thus, the locked kernel resolution will be the same as binary kernel resolution (Definition 5.1), except

(1) only the left-kernels of two given clauses ($K_1|P_1/R_1$ and $K_2|P_2/R_2$) are indexed;
(2) if the literal L_i ($i = 1,2$) resolved upon is from K_i, then the index of L_i must be the "lowest" among the indices of K_i;
(3) if $K_i = \Box$, and so L_i is from P_i, then L_i can be any literal of P_i.

In a similar manner, one can extend the untyped semantic resolution to a typed one, by extending PI-clash [10] to a typed PI-clash. The typed PI-clash is similarly to PI-clash, except all constraints associated with PI-clash are only applicable to the left-kernels of the clauses.

6.4. *On other refinements of TP-second deduction*

For some other refinements of resolution, such as the set of support strategy [39], the linear-like refinements [10], and hierarchical deduction [36], there may exist no general methods that can be used for extending them into TP-second deduction without causing incompleteness. In the following, we give a brief discussion about some possible extensions on a case by case basis.

First consider how to extend the set of support strategy into TP-second deduction. Clearly, an unrestricted use of this strategy in a typed deduction may cause proof failure. Consider Example 4.6. If the singleton set $\{h_1 = s(x, f(x)) \vee v(x)/x{:}\,E\}$ is chosen as the set of support, then there will be only one resolvent $s(x, f(x))/x{:}\,E \wedge x{:}\,P$ that can be deduced (from h_1 and h_3). Then the proof will fail. In order to preserve the completeness, we need to add certain restrictions to the selection of the set of support. For example, we can require that the set of support must include all those typed clauses, whose type restriction is associated with a conditional type (τ is a conditional type if there is a rule of form $x{:}\,\tau/H/R$ in T, such that $H \neq$ *true*). For example, since in Example 4.6, P is a conditional type, but E and Q are not, then a set of support chosen for Example 4.6 should include both h_3 and h_4. It is easy to check that with such a set of support, there does exist a TP-second refutation of $S_{4.1}$ in $T_{4.1}$.

Unfortunately, for a linear-like refinement of resolution, such as linear resolution, ME [20], or SL-resolution [18], there may exist no typed version of it which is complete in dealing with conditional typing theory. However, since the proof failure is mainly caused by the restriction that a typing rule cannot be used as a side clause for the goal clauses, a proof failure may be avoided by giving certain restriction to the selection of the starting goal. For example, we may require that an input clause G cannot be used as the starting goal unless there exists a model for the set of kernels of all other clauses of S. The reader may verify that, if S is T-unsatisfiable, then for such a goal G, there must exist at least one member of $S - \{G\}$ which can be used as a side clause for G. Applying this requirement to the goal selection for Example 4.6, the previous selection of h_1 as the starting goal is illegal, since the set of kernels of h_2, h_3, h_4 is unsatisfiable. But under such a requirement, each of h_2, h_3, h_4 can be chosen as the starting goal, and each of them can lead to a typed linear refutation.

The typed hierarchical deduction obtained by extending hierarchical deduction [36] into TP-second deduction is also incomplete. However the failure cases should be fewer than those of a typed linear-like refinement. Unlike linear deduction, hierarchical deduction searches a proof along with a tree of nodes, each of which contains all legal resolvents that can be produced from the parent goal. In light of this feature, the hierarchical deduction permits one to lock some literals of the input clauses (i.e. disallow them to be resolved upon) for reducing the redundancy. This restriction is the so-called partial set of support strategy (PSS) defined in [36]. For typed hierarchical deduction, the constraints associated with the general relations contained in the given typing theory are quite similar to the application of PSS, that is, locking the literals associated with these relations. Although PSS is not complete, it has been used to help efficiently prove many theorems, including some challenge theorems.

It has been noted that the success of a goal-oriented refinement of TP-second deduction in proving a theorem is closely related to the possibility of incorporating semantic guidance in the refinement. Associated with this relation, one interesting subject for the further research is how to extend the semantic guidance into the typed domain. It is also very interesting to see how this typed approach can be used to enhance other refinements of resolution, such as Bibel's connection graph deduction [5], Plaisted's problem reduction format [23], Nie and Plaisted's semantic back chaining proof system [22], Sandford's hereditary lock resolution [26], Sutcliffe's semantically guided linear deduction system [29], and Lee and Plaisted's hyper-linking strategy [19].

6.5. *On the procedures for type reasoning*

How to carrying out type-checking and TP-deduction is an important question not answered in this paper. The answer to this question depends on the properties of the type theories considered by a particular implementation of the typed deduction procedure. In many practical applications, especially for reasoning about computer programs, a typing theory often possesses certain additional constraints. For instance, it may contain only typing rules of certain structures; it may contain no empty types, and the entire set of types may have a tree-like structure, etc. By taking these constraints into account, efficient type reasoning procedures can be designed. For this design, the existing results in sorted unification may be incorporated.

6.6. *About equality reasoning*

The typed logic considered here does not have equality as a logical symbol. In other words, the logic treats equality relations as ordinary (general) relations, and in particular, uses the symbol "=" for specifying a general relation. In order to use "=" for conveying the meaning of equality to our typed deduction procedures, one needs to explicitly supply the set of equality axioms (typed or untyped), or enrich the logic with a logical symbol "=" and the corresponding equality inference rules (e.g. paramodulation). There is no essential difference from the way of handling equality in an untyped predicate calculus.

7. Some implementation results

A typed theorem prover [37] has been implemented by extending a resolution-based theorem prover into TP-second deduction. The type reasoner of the prover consists of a number of special purpose procedures based on certain assumption about the underlying typing theory. The assumption about the nonconditional portion of a typing theory T (i.e. $T|_{TD}$) is close to Walther's forest structure of sorts [31] and Frisch's tree-structure restriction [13]. Besides carrying out the normal TP-deduction and type-checking, this reasoner also makes simplification of some type restrictions by means of a TR-simplifier. The TR-simplifier is a special TP-deduction procedure, which is used to simplify the type restriction of each newly generated kernel resolvent. For example, it can reduce $k(a)$: *dpt* to *true* in proving AM8-T of Example 2.4, reduce K/x: *int* $\wedge$ K/x: *nat* to K/x: *nat* in proving PLUS-T of Example 2.5, and detect ill-typed resolvents, e.g., K/x:*fox* $\wedge$ x: *wolf* in proving SSR-T (see Table 1 and Remarks 1–4). The simplification is also done by deleting (merging) a portion of the typing precondition associated with the type restriction under certain conditions.

This typed prover has been used to efficiently prove many (typed) theorems, including VKD-T, AM8-T, and PLUS-T given in Examples 2.3, 2.4, and 2.5. These experiments indicate that the typed prover is better than the untyped prover both in the efficiency of proving a theorem, and in the clarity of knowledge representation. A summary of some implementation results is given in Table 1. After a brief explanation of the content of this table, we shall give an analysis of the computer proof of AM8-T.

Table 1
A summary of the implementation results

Theorem	AM8-T		VKD-T		PLUS-T		SSR-T	
Prover	typed	untyped	typed	untyped	typed	untyped	typed	untyped
Input typing rules	9	0	5	0	12	0	15	0
Input kernel clauses	13	26	10	15	11	23	8	27
Input kernel literals	25	73	22	44	19	63	12	66
Accepted kernel resolvents	238	2495	24	50	29	64	52	1085
Kernel-empty clauses	4	1	2	1	3	1	1	1
Ill-typed resolvents	3	0	0	0	0	0	85	0
Useful kernel resolvents	23	42	17	22	10	36	14	64
Useful goal resolvents	19	37	14	17	10	36	14	64
Total CPU seconds	12.5	96.4	1.32	2.58	1.80	3.18	4.98	32.5
CPU seconds for typing	4.57	0	0.07	0	0.60	0	2.08	0

Remark 1. The untyped prover is simply the typed prover running in an untyped mode. In the untyped mode, the prover used the same input data as that for the typed prover, except it treated an expression t: τ as an ordinary literal instead of a TD (type descriptor), and treat both typing rules and typed clauses as ordinary clauses.

Remark 2. The number of *input kernel literals* is counted as the total number of literals of the input data on general relations, including those contained in the kernels of typed clauses and those contained in typing rules. *CPU seconds for typing* is the total CPU seconds spent for type reasoning, including type-checking, TP-deduction, and simplification. *Total CPU seconds* is the total CPU seconds used for the entire proof.

Remark 3. *SSR-T* is a standard example (Schubert's Steamroller) in the literature of sorted calculus. The proof for SSR-T found by our typed prover is longer than some other published proofs. This proof consists of 14 steps (i.e. 14 useful goal resolvents). Whereas the published proof in [32] consists of 10 steps. This is because our prover did not do factoring during the deduction. But for the shorter proof, a factoring resolvent is needed (see [11] and [32]).

Remark 4. The prover merged automatically a list of typed clauses $K/R_1, , \ldots , K/R_n$ into one formula $K/R_1 \vee \cdots \vee R_n$, and treated it as one clause. This caused the total number of input clauses (typing rules and kernel clauses) for the typed prover to be less than the total number of clauses for the untyped prover for some problems, such as AM8-T and SSR-T.

An analysis of the computer proof of AM8-T

AM8 mentioned in Example 2.4 was acknowledged a challenge theorem for automated theorem prover. Up to now, very few automated proofs of it were reported [22,36]. For untyped provers, AM8-T should be a little harder than AM8, because it uses typed inequality axioms, which contain more literals than the untyped ones. (Please consult Example 2.4 for the set of clauses, and Example 5.7 for the proof-tree.) In comparison with the untyped proof, which took 96.4 seconds with the acceptance of 2495 resolvents,[5] the typed prover can prove AM8-T fair efficiently: it took only 12.5 seconds with the acceptance of 238 kernel resolvents. Why can the typed proof be much more efficient than an untyped proof? In answering this question, we shall give an illustration of the general feature and advantage of the typed predicate calculus based on TP-second deduction.

- For TP-second deduction, the branching factor of the search tree may be reduced. This is because, by classifying many clauses (i.e. $k_1, \ldots, k_9$) into typing rules, and many literals (i.e. $a \leqslant x \wedge x \leqslant b$) of the clauses into TDs or the general relations of the typing theory, the number of clauses (kernel clauses) and literals (kernel literals) that must be considered by kernel resolution is reduced. Note that, for TP-second deduction, kernel resolution is the main inference rule in generating new resolvents. Another inference rule, TP-resolution, is used mainly as a constraint to kernel resolution. Specifically, TP-resolvents produced by type-checking do not

[5] This typed prover (in untyped mode) performs worse than the HD-prover described in [36] for solving certain problems. For example, it created 1478 resolvents to prove AM8, and 577 resolvents to prove IMV, but the HD-prover created only 204 resolvents to prove AM8, and only 149 resolvents to prove IMV. This is partly because that the HD-prover employs special heuristics, based on twin-symbol matching (see [36]), which are not fully incorporated by this typed prover.

contribute to the search space; they are intermediate results produced only for checking the type consistency of kernel resolvents. The role of TR-simplifier is to transform a kernel resolvent into a simpler form. Although TP-descendents produced by TP-deduction contribute to the space of search, their number is usually small, because, as shown in Table 1, there are usually few kernel-empty clauses that can be generated.

- The need for deducing a kernel-empty resolvent provides a direction to the search for a refutation. While for untyped deduction, usually the only determined goal is deducing the empty clause, for TP-second deduction, there is a determined intermediate goal—deducing the kernel-empty clauses. Once a kernel-empty clause is deduced (such as g_{16} of the proof-tree of Example 5.7), it should be given a very high priority to develop. In comparison with the deduction of the empty clause by untyped resolution, the number of steps (i.e., useful goal resolvents) for deducing a kernel-empty clause must be smaller than or equal to the number of steps for deducing the empty clause. This reduction, plus the reduction in the branch factor, leads to a reduction in the search space. As was shown in Table 1, the untyped prover took 37 steps to deduce the empty clause, for which it has to generate and accept 2495 resolvents. Whereas, the typed prover took only 10 steps (goals) to deduce the kernel-empty clause g_{16} with 186 resolvents accepted.
- The use of specialized procedures and implementation strategy (e.g., caching) for typing reasoning also contributes to the gain in the efficiency. If we count the total number of TP-resolvents generated during the deduction, then, strictly speaking, there may be no savings in the total number of steps (resolvents) needed for a typed proof. However, except TP-descendants (i.e. those produced by applying TP-deduction to kernel empty clauses), TP-resolvents are used only as intermediate results. They need not be produced, evaluated, and maintained as ordinary kernel resolvents. As was shown in Table 1, for proving AM8-T by the typed prover, a total of 4.57 seconds was spent on type reasoning. We may assume that, corresponding to the generation of each kernel resolvent, there were, in a *virtually* sense, at least 5 TP-resolvents produced for type-checking and simplification. Since there were a total of 238 resolvents generated, each TP-resolvent took $4.57/(238 \times 5)$ seconds $= 0.00384$ seconds. This is about 10 time faster as the $(12.5 - 4.57)/238$ seconds $= 0.033$ seconds spent for generating a kernel resolvent.

8. Conclusion

We introduced a typed resolution principle by extending existing results in sorted deduction and constrained resolution. This principle is capable of separating type reasoning from general reasoning even when the background typing theory shares general relations with the kernel formulation. We proved a typed Herbrand theorem, and based on this theorem, established the completeness of the typed resolution principle. In order to improve the efficiency in proving theorems by using the typed resolution principle, we introduced a TP-second deduction procedure which allows one to restrict the application of general TP-deduction to kernel-empty resolvents. We discussed the refinements of the typed deduction by extending existing refinements of untyped resolution, and identified a number of interesting subjects for future research. We demonstrated and analyzed the advantage of our typed approach using the results obtained by a computer implementation of a typed prover. We compared our approach with the existing work in the field, and referred to the existing results, based on which, our conclusion can be made.

Acknowledgment

The author would like to thank the anonymous referees whose comments and suggestions have helped in improving and clarifying the presentation, thank A. Goldberg and D. Smith at the Kestrel Institute for fruitful discussion on the subject, and thank R. Furst and M. Pryce for proofreading the drafts.

References

[1] R. Anderson and W.W. Bledsoe, A linear format for resolution with merging and a new technique for establishing completeness, *J. ACM* **17** (1970) 525-534.
[2] H.P. Barendregt, *The Lambda Calculus* (Elsevier Science Publishers, B.V. (North-Holland), Amsterdam, 1985).
[3] W.W. Bledsoe, Non-resolution theorem proving, *Artif. Intell.* **9** (1977) 1-35.
[4] W.W. Bledsoe, The UT interactive prover, University of Texas at Austin, Mathematics Department Memo ATP-17b, Austin, TX (1983).
[5] W. Bibel, On matrices with connections, *J. ACM* **28** (4) (1981) 633-645.
[6] R.S. Boyer, Locking: a restriction of resolution, Ph.D. Thesis, University of Texas at Austin, TX (1971).
[7] H.-J. Bürckert, A resolution principle for clauses with constraints, in: *Proceedings 10th International Conference on Automated Deduction*, Kaiserslautern, Germany (1989) 178-202.
[8] H.-J. Bürckert, *A Resolution Principle for a Logic with Restricted Quantifiers*, Lecture Notes in Artificial Intelligence **568** (Springer-Verlag, Berlin, 1991).
[9] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism, *Comput. Surv.* **17** (4) (1985) 472-522.
[10] C.L. Chang and R.C. Lee, *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, New York, 1973).
[11] A.G. Cohn, A more expressive formulation of many sorted logic, *J. Automated Reasoning* **3** (2) (1987) 113-200.
[12] A.G. Cohn, On the appearance of sorted literals: a nonsubstitutional framework for hybrid reasoning, in: H.J. Levesque and R. Reiter, eds., *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ont. (1989) 55-66.
[13] A.M. Frisch, A general framework for sorted deduction: fundamental results on hybrid reasoning, in: H.J. Levesque and R. Reiter, eds., *Proceedings First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ont. (1989) 126-136.
[14] A.M. Frisch, The substitutional framework for sorted deduction: fundamental results on hybrid reasoning, *Artif. Intell.* **49** (1991) 161-198.
[15] A.M. Frisch and A.G. Cohn, An abstract view of sorted unification, in: D. Kapur, ed., *Proceedings 11th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence **607** (1992) 178-192
[16] M. Höfeld and G. Smolka, Define relations over constraint languages, LILOG-Report 53, IBM Deutschland (1988).
[17] J. Jaffar and J.-L. Lassez, Constraint logic programming, in: *Proceedings 14th ACM Symposium on Principles of Programming Languages*, Munich, Germany (1987) 111-120.

[18] R. Kowalski and D. Kuehner, Linear resolution with selection function, *Artif. Intell.* **2** (1971) 227–260.
[19] S.-J. Lee and D.A. Plaisted, Eliminating duplication with the hyper-linking strategy, *J. Automated Reasoning* **9** (3) (1992) 25–42
[20] D.W. Loveland, Mechanical theorem-proving by model-elimination, *J. ACM* **15** (1968) 236–251.
[21] R. Milner, A theory of type polymorphism in programming, *J. Comput. Syst. Sci.* **17** (1978) 348–375.
[22] X. Nie and D.A. Plaisted, A semantic backward chaining proof system, *Artif. Intell.* **55** (1992) 109–128.
[23] D.A. Plaisted, Non-Horn clause logic programming without contrapositives, *J. Automated Reasoning* **4** (3) (1988) 287–325.
[24] R. Reiter, On the integrity of typed first-order data bases, in: H. Gallaire, J. Minker and J. Nicolas, eds., *Advances in Data Base Theory, Vol. 1* (Plenum, New York, 1981) 137–157.
[25] J.A. Robinson, A machine oriented logic based on the resolution principle, *J. ACM* **12** (1) (1965) 23–41.
[26] D.M. Sandford, *Using Sophisticated Models in Resolution Theorem Proving* (Springer-Verlag, New York, 1980).
[27] M.S. Schmidt-Schauss, *Computational Aspects of an Order-Sorted Logic with Term Declarations*, Lecture Notes in Computer Science **395** (Springer-Verlag, Berlin, 1989).
[28] M.E. Stickel, Automated deduction by theory resolution, *J. Automated Reasoning* **1** (4) (1985) 333–355.
[29] G. Sutcliffe, The semantically guided linear deduction system, in: D. Kapur, ed., *Proceedings 11th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence **607** (1992) 677–680.
[30] M.H. van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, *J. ACM* **23** (4) (1976) 733–742.
[31] C. Walther, A many-sorted calculus based on resolution and paramodulation, in: *Proceedings IJCAI-83*, Karlsruhe, Germany (1983) 882–891.
[32] C. Walther, A mechanical solution of Schubert's Steamroller by many-sorted resolution, *Artif. Intell.* **26** (1985) 217–224.
[33] C. Walther, Many-sorted unification, *J. ACM* **35** (1) (1988) 1–17.
[34] T.C. Wang, Designing examples for semantically guided hierarchical deduction, *Proceedings IJCAI-85*, Los Angeles, CA (1985) 1201–1207.
[35] T.C. Wang, On the least Herbrand model for conditional Horn sets, *Assoc. Automated Reasoning Newslett.* **24** (1993).
[36] T.C. Wang and W.W. Bledsoe, Hierarchical deduction, *J. Automated Reasoning* **3** (1) (1987) 35–71.
[37] T.C. Wang and A. Goldberg, KITP-93: an automated inference system for program analysis, in: A. Bundy, ed., *Proceedings 12th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence (Springer-Verlag, Berlin, 1994).
[38] C. Weidenbach and H.J. Ohlbach, A resolution calculus with dynamic sort structures and partial functions, in: *Proceedings ECAI-90*, Stockholm, Sweden (1990) 6–10.
[39] L. Wos, G.A. Robinson and D.F. Carson, Efficiency and completeness of the set of support strategy in theorem proving, *J. ACM* **12** (4) (1965) 484–489.
[40] L. Wos, R. Overbeek, E. Lusk and J. Boyle, *Automated Reasoning: Introduction and Applications* (Prentice Hall, Englewood Cliffs, NJ, 1992).