Static Types As Search Heuristics

Hao Xu

xuh@cs.unc.edu
Department of Computer Science,
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA

Abstract. Static analysis techniques have been studied for many years in the functional programming community, most prominently the use of inferred types to eliminate run-time checks. In analogy, if we regard checking whether an instance is useful for a proof as run-time checks and we can find types that eliminate irrelevant instances, we may also be able to prevent proof searches from checking those irrelevant instances, thereby improving the performance. This paper introduces a method that employs types as heuristics in proof search.

1 Introduction

Instance-based theorem proving techniques have been implemented in many theorem provers: [1], [2], [3], [4], and [5], among others. Some of these provers have been shown to prove problems in some categories faster than or comparable to resolution-based theorem provers[6]. On the other hand, resolution-based theorem provers continue to lead on other problem categories. While instance-based theorem provers are believed to have a number of advantages, such as being capable of generating models for satisfiable problems and making use of the highly efficient SAT solvers, their run-time performance is suboptimal without guidance from instance generation heuristics such as resolution, semantics, etc..

Static analysis techniques in compilers, such as using types to eliminate run-time checks, have been studied for many years. In analogy, if we regard checking whether an instance is useful for a proof as run-time checks, then we may find types that prevent proof searches from checking some irrelevant instances, thereby improving the performance. In general, static analysis techniques generate heuristics from the input, in contrast to various static transformation techniques which preprocess the input. Following is a (incomplete) list of the techniques used in static analysis:

Strategy Selection chooses a strategy that is believed to best suit the problem. Implementation: E[7], Vampire[8], and DCTP[5], among others.

Restriction finds instances that may be pruned from the search space. Implementation: FM-Darwin[9], Paradox[10], and OSHL-S, among others.

Ordering determines what kind of ordering can lead to contradiction faster. Implementation: E(literal ordering), among others.

This paper introduces a method that employs inferred types from resolution as heuristics in first-order logic proof search. The general idea is deriving a type inference algorithm from a complete calculus, and using types to filter out a subset of instances that are not generated by the calculus by searching only well-typed instances. Properly designed types should bring some of the advantages from one method to another.

2 Notations

A vector is written as $[a_1, \ldots, a_n]$, or vertically as in (5). The ith component of a vector $\mathbf{v}$ is v_i. We write $\mathbf{v}^n$ to explicitly mark that $\mathbf{v}$ has n components. A vector is implicitly converted to a set. For example, in $\mathbf{v} \subset \mathbf{x}$. Given a signature Σ and a set of variables Var, Term is the set of terms over $\Sigma \cup \mathsf{Var}$; GrTerm the set of terms over Σ. An expression is a term or a literal. Expr is the set of expressions over $\Sigma \cup \mathsf{Var}$; GrExpr the set of expressions over Σ. By default, $\Sigma = FS(S)$ and $\mathsf{Var} = VS(S)$, given a clause set S. Constants are nullary function symbols. The empty clause is denoted by $\emptyset$. The function fv maps an expression to a set of free variables occurring in that expression. $\mathbf{fv}(x)$ is the vector constructed from $fv(x)$ by sorting the element lexicographically (other total orderings should also work). A substitution is a partial function $\theta : \mathsf{Var} \to \mathsf{Term}$. $fv(\theta) = \bigcup_{v \in dom(\theta)} fv(\theta(v))$. A substitution of variables $\mathbf{v}$ with terms $\mathbf{t}$ is denoted by $[\mathbf{t}/\mathbf{v}]$. If $dom(\theta) \subset D$, the domain extension $\theta|_D$ of θ to D is defined by $\theta|_D(v) = \theta(v)$, if $v \in dom(\theta)$ and $\theta|_D(v) = v$, if $v \in D \backslash dom(\theta)$. If $fv(e) \not\subset dom(\theta)$, $e\theta = e\theta|_{dom(\theta) \cup fv(e)}$. The well-formedness requirement for a substitution θ is that $\theta(v) = v$, for all $v \in fv(\theta) \cap dom(\theta)$, namely, the substitution is idempotent. The well-formedness requirement for an mgu σ of literals L, N is that $fv(\sigma) \subset dom(\sigma) = fv(L) \cup fv(N)$. The complement of a literal L is denoted by $\overline{L}$.

3 A Theory

3.1 An Example

Example 1. $S = \{\{P(X), Q(f(X)\}, \{\neg P(f(Y)), \neg Q(Y)\}, \{Q(a)\}, \{\neg Q(f(f(a))\}\}$.

Since S is unsatisfiable, there exists a minimal (but not necessarily minimum) set S_1 of ground instances of S that is unsatisfiable. By minimality of the instances in S_1, for all ground literals $N \in \bigcup S_1$, $\overline{N} \in \bigcup S_1$. For example, $S_1 = \{\{P(f(a)), Q(f(f(a))\}, \{\neg P(f(a)), \neg Q(a)\}, \{Q(a)\}, \{\neg Q(f(f(a)))\}\}$. If we use a brute force method to find S_1, then both X and Y are initiated to terms in $x = \{a, f(a), f(f(a)), \ldots\}$. Our goal is to restrict the instantiation of variables to proper subsets of x given by a instantiation function $I : \mathsf{Term} \to \mathcal{P}(\mathsf{GrTerm})$. We establish a similar minimality requirement for I: for any literal $N \in S$ and any ground instance N_1 in $I(N)$, there exists L s.t. $\overline{N_1} \in I(L)$. Rewriting this requirement as an inequality of sets, we obtain:

$$\bigcup_{L \in \bigcup S} I(L) \supset \bigcup_{L \in \bigcup S} \overline{I(L)} \tag{1}$$

Informally speaking, all instances of literals in S should be resolvable. In the example, the only literal in S that is unifiable with $\overline{P(X)}$ is $\neg P(f(Y))$. Therefore, any instance of $P(X)$ must have a complement, which must be an instance of $\neg P(f(Y))$. Hence $I(\neg P(f(Y))) \supset \overline{I(P(X))}$. If $I(e) = \{e[\mathbf{t}/\mathbf{v}]|t_i \in I(v_i), \mathbf{v} = \mathbf{fv}(e)\}$, then $\{f(t)|t \in I(Y)\} \supset I(X)$. We restrict the proof search by instantiating any variable v to elements of $I(v)$, which we refer to as the semantics of the type of the variable v.

3.2 Algebra $\mathcal{V}$

In this section, we introduce an algebra $\mathcal{V}$. The motivation of $\mathcal{V}$ is to enable simultaneously manipulating functions and sets, as in linear algebra. There are several groups of notations used in this paper. Algebra $\mathcal{V}$: ρ, τ type; v variable; $\mathbf{X}$ term of the form $[v_i]_{i=1}^n$; $A, \mathbf{A}, \mathbf{B}, F, G, H, T, \mathbf{T}$ term. Mathematics: D domain; x, y set; f, g function; $\mathbf{a}, \mathbf{b}, \mathbf{v}$ vector; l, m, n, o, p natural number. Logic: u, v, X, Y variable; t term; e expression; L, N literal; C, D clause; S clause set; a, c, f function symbol.

Definition 1. *The types are: set of vectors of length n $\{n\}$; function $\tau \to \tau'$.*

Types are mainly used to define what terms are meaningful in $\mathcal{V}$. To avoid the complexity of recursive domain equations, we define families of constructors indexed by natural numbers. The indices do not affect the mapping of the function that a term denotes, but the domain and range of that function.

Definition 2. *$T : \tau$ denotes that T is a term of type τ. $\tau_{m,n} = \{m\} \to \{n\}$.*

$\emptyset_n : \{n\}$	*SumUnit*	$\lambda \mathbf{v}^n.\emptyset_m : \{n\} \to \{m\}$	*FSumUnit*
$\cup_n : \{n\} \to \{n\} \to \{n\}$	*Sum*	$\sqcup_{m,n} : \tau_{m,n} \to \tau_{m,n} \to \tau_{m,n}$	*FSum*
$\cap_n : \{n\} \to \{n\} \to \{n\}$	*Inter*	$\sqcap_{m,n} : \tau_{m,n} \to \tau_{m,n} \to \tau_{m,n}$	*FInter*
$[] : \{0\}$	*ProdUnit*	$\lambda \mathbf{v}^n.[] : \{n\} \to \{0\}$	*FProdUnit*
$\dfrac{p = m + n}{\bullet_{m,n} : \{m\} \to \{n\} \to \{p\}}$	*Prod*	$\dfrac{p = m + n}{\blacksquare_{l,m,n} : \tau_{l,m} \to \tau_{l,n} \to \tau_{l,p}}$	*FProd*
$p_{i/n} : \{n\} \to \{1\}$	*Proj*	$\circ_{m,n,p} : \tau_{m,n} \to \tau_{n,p} \to \tau_{m,p}$	*Comp*
$\dfrac{\mathbf{v} \supset fv(e)}{\Lambda \mathbf{v}^n.e : \{n\} \to \{1\}}$	*TFunc*	$v : \{1\}$	*Var*
$\dfrac{\mathbf{v} \supset fv(e)}{\Lambda^{-1} \mathbf{v}^n.e : \{1\} \to \{n\}}$	*RTFunc*	$\dfrac{T : \rho \to \tau \quad T' : \rho}{TT' : \tau}$	*App*

We use parentheses to delimit terms when ambiguous. We omit subscripts and superscripts indicating vector length and type indices when not ambiguous. There are two kinds of terms used in the paper: a term in the set Term and a term in $\mathcal{V}$. We refer to the latter as a term of $\mathcal{V}$ to disambiguate only when necessary. $TFunc$ is called a term function; $RTFunc$ a reverse term function.

Let us now switch context to mathematical objects. As in domain theory, we say that a function f is continuous if $x_0 \subset x_1 \subset \ldots$ implies $\bigcup_{i=0}^{\infty} f(x_i) = f(\bigcup_{i=0}^{\infty} x_i)$. Define $\mathbf{a}^m \mathbf{b}^n = [a_1, \ldots, a_m, b_1, \ldots, b_n]$, $x \times y = \{\mathbf{a}^m \mathbf{b}^n | \mathbf{a}^m \in x, \mathbf{b}^n \in$

$y\}$, $\prod_{i=1}^n x_i = x_1 \times (\ldots \times x_n \times \{[]\})$, $\bigcup_{i=1}^n x_i = x_1 \cup \ldots \cup x_n$, $\bigcap_{i=1}^n x_i = x_1 \cap \ldots \cap x_n$, and $[x_i]_{i=1}^n = [x_1, \ldots, x_n]$. For example, $[a][f(a)] = [a, f(a)]$, $\{[a], [a']\} \times \{[f(a)]\} = \{[a, f(a)], [a', f(a)]\}$. If we define $D_n = \{[x_i]_{i=1}^n | x_i \in \mathsf{GrExpr}\}$, then the domains D_τ are defined as follows, where $[D_\rho \to D_\tau]$ is the set of functions from D_ρ to D_τ.

$$D_{\{n\}} = \mathcal{P}(D_n) \quad D_{\rho\to\tau} = [D_\rho \to D_\tau]$$

Definition 3. *An interpretation is a partial function $I : \mathsf{Var} \to \mathcal{P}(\mathsf{GrTerm})$. An interpretation is trivial if $I(v) = \emptyset$ for some variable v.*

Given an interpretation I, the semantic function $()_I^*$ maps a term $T : \tau$ of $\mathcal{V}$ to an element of D_τ.

Definition 4. *The semantic function $()_I^*$ is defined as follows. The semantics of function terms are given by partial evaluation.*

$$\begin{array}{rlrl}
(\emptyset_n)_I^* &= \emptyset & (\lambda\mathbf{v}.\emptyset_n)_I^*(x) &= \emptyset \\
(\cup_n)_I^*(x)(y) &= x \cup y & (\sqcup_{m,n})_I^*(f)(g)(x) &= f(x) \cup g(x) \\
(\cap_n)_I^*(x)(y) &= x \cap y & (\sqcap_{m,n})_I^*(f)(g)(x) &= f(x) \cap g(x) \\
([])_I^* &= \{[]\} & (\lambda\mathbf{v}.[])_I^*(x) &= \{[]\} \\
(\bullet_{m,n})_I^*(x)(y) &= x \times y & (\blacksquare_{l,m,n})_I^*(f)(g)(x) &= f(x) \times g(x) \\
(p_{i/n})_I^*(\mathbf{x}^n) &= x_i & (\circ_{m,n,p})_I^*(f)(g)(x) &= f(g(x)) \\
(\Lambda\mathbf{v}.e)_I^*(x) &= \{e[\mathbf{t}/\mathbf{v}] | \mathbf{t} \in x\} & (v)_I^* &= I(v) \\
(\Lambda^{-1}\mathbf{v}.e)_I^*(x) &= \{\mathbf{t} | e[\mathbf{t}/\mathbf{v}] \in x\} & (FT)_I^* &= (F)_I^*((T)_I^*)
\end{array}$$

Notations such as $\emptyset, \cup, \cap$ are reused as both set-theoretical notations and terms of $\mathcal{V}$. A term is variable-free if there is no subterm that is a variable. Since the denotations of *variable-free* terms are independent of I, we usually make I implicit when the term is variable-free. For example, $(\Lambda[v_1].f(v_1))^*\{[a]\} = \{[f(a)]\}$, $(\Lambda^{-1}[v_1].f(v_1))^*\{[f(a)]\} = \{[a]\}$, $(p_{1/2})^*\{[a, f(a)]\} = \{[f(a)]\}$.

$I \models T = T'$ if and only if $(T)_I^* = (T')_I^*$. $T = T'$ if and only if it holds for all I. A subset order is defined on set terms: $I \models T \sqsubset T'$ if and only if $(T)_I^* \subset (T')_I^*$. The order is extended to functions $I \models F \sqsubset G$ if and only if for all terms T, $I \models FT \sqsubset GT$. $T \sqsubset T'$ if and only if it holds for all I. The reverse is denoted by $\sqsupset$. F is continuous if and only if $(F)_I^*$ is continuous for all I.

We write $[T_i]_{i=1}^n = [T_1, \ldots, T_n] = T_1 \bullet \ldots \bullet T_n \bullet []$, $[F_i]_{i=1}^n = [F_1, \ldots, F_n] = T_1 \blacksquare \ldots \blacksquare T_n \blacksquare \lambda\mathbf{v}.[]$. We usually want to convert a vector to a term of the form $[v_i]_{i=1}^n$ and conversely, which we call algebraify and dealgebraify. If $\mathbf{v}$ is a vector of variables, then $\mathcal{A}\mathbf{v}$ is a term $v_1 \bullet \ldots \bullet v_n \bullet []$; if $\mathbf{X}$ is a term of the form $[v_i]_{i=1}^n$, then $\mathcal{A}^{-1}\mathbf{X} = \mathbf{v}$. As a convention, we have $\mathbf{X} = \mathcal{A}\mathbf{v}$ and $\mathbf{v} = \mathcal{A}^{-1}\mathbf{X}$.

We make the following auxiliary definitions, where $\mathbf{v}_e = \mathbf{fv}(e)$, f is a function symbol of arity o, $\mathbf{v}_f = [v_1, \ldots, v_o]$, and $F_n : \{n\} \to \{n\}$ for some natural number n. $F_n^0 \triangleq \cup_n \emptyset_n$. $F_n^{k+1} \triangleq F_n F_n^k$. $f^{-1} \triangleq (\Lambda^{-1}\mathbf{v}_f.f(v_1, \ldots, v_o))$. $e \triangleq (\Lambda\mathbf{v}_e.e)\mathbf{X}_e$. $I(e) \triangleq (\Lambda\mathbf{v}_e.e)^* I(\mathbf{v}_e)$. $I(\mathbf{v}^n) \triangleq \prod_{i=1}^n I(v_i)$. For example, if f is binary, then $f(X, Y) = (\Lambda[X, Y].f(X, Y))[X, Y]$, $f^{-1} = \Lambda^{-1}[X, Y].f(X, Y)$. We write binary functions as infix, for example, $\sqcup FG$ is written as $F \sqcup G$ except for composition where we write $\circ FG$ as FG.

It can be proved that (a) if $F \sqsupset G$ and $F \sqsupset H$, then $F \sqsupset G \sqcup H$; (b) $F \sqcup G \sqsupset F$ and $F \sqcup G \sqsupset G$; (c) $(F \sqcup G)H = FH \sqcup GH$; (d) $F(G \sqcup H) = FG \sqcup FH$; (e) $[F_i]_{i=1}^n T = [F_i T]_{i=1}^n$ (f) $(\lambda \mathbf{v}.\emptyset)F = \lambda \mathbf{v}.\emptyset$; (g) $p_i[T_k]_{k=1}^n = T_i$, where $i \in \{1, \ldots, n\}$.

3.3 Constraints

Definition 5. *(a) An* instantiation set *of an expression e is a set of ground substitutions Θ s.t. for every $\theta \in \Theta$,$dom(\theta) \supset fv(e)$. The* instance set *of e w.r.t. Θ is $\{x\theta | \theta \in \Theta\}$. (b) An instantiation set of $\mathbf{v}$ induced by an interpretation I s.t. $dom(I) \supset \mathbf{v}$ is the set $\{[\mathbf{t}/\mathbf{v}] | \mathbf{t} \in I(\mathbf{v})\}$. (c) An instantiation set is complete for a clause set S if the instances obtained from it are inconsistent. An interpretation is complete if the induced instantiation set is complete.*

In this section we formalize the ideas discussed in Section 3.1 in terms of $\mathcal{V}$. In the following discussion, S is a set of clauses s.t. $\emptyset \notin S$ and for any $C, D \in S$, $fv(C) \cap fv(D) = \emptyset$.

Suppose that $L, N \in \bigcup S$ and that σ is a well-formed mgu of $L, \overline{N}$. We require that for every $v \in fv(L), \sigma(v) \neq v, t = \sigma(v), \mathbf{v} = \mathbf{fv}(t)$

$$I \models v \sqsupset (\Lambda \mathbf{v}.t)\mathbf{X} \tag{2}$$

and for every $u \in fv(L), \sigma(u) \neq u, t = \sigma(u), \mathbf{v} = \mathbf{fv}(t), v = v_i$,

$$I \models v \sqsupset p_i(\Lambda^{-1}\mathbf{v}.t)u \tag{3}$$

The apparent minimum solution to (1) is the constant function $I(v) = \emptyset$ for any variable v. Therefore, we add the following constraints to ensure that the solution is complete for S. Given an arbitrary partial function $g : \mathsf{Var} \to \mathsf{GrTerm}$ s.t. $dom(g) \supset fv(S)$, for all $v \in fv(S)$, $c = g(v)$,

$$I \models v \sqsupset c \tag{4}$$

In effect, (4) pins down a specific class of inconsistent sets of instances which are generated by instantiating a resolution proof to a ground resolution proof.

The algorithm for constraints generation (2), (3), and (4) is shown as follows.

Algorithm 1. *(Constraints Generation)*

1. *Given a set of nonempty clauses S, rename so that any two clauses have no common variables.*
2. *For every clause C, every literal L in C, every clause D in S, and every literal N in D such that $\overline{N}$ and L are unifiable by mgu σ, generate constraint (2) and (3).*
3. *For all variable X generate constraint (4).*

Denote the constraints generated by a clause set S by $cons(S)$. If I makes a constraint true, then we say that I is a solution to the constraint. If any solution to constraints K is also a solution to constraints K' and vice versa, then we say that $K \equiv K'$.

Next, we look at an example.

Example 2. We choose $g(X) = f(a), g(Y) = a$ and generate constraints for Example 1. Because $P(X)$ is unifiable with $\overline{\neg P(f(Y))}$ with mgu $[f(Y)/X]$, the algorithm generates the following constraints $I \models X \sqsupset f(Y), I \models Y \sqsupset p_1 f^{-1} X$. Other constraints are generated similarly. We usually write the constraints in an equivalent but more compact form. $I \models X \sqsupset f(Y) \cup p_1 f^{-1}(Y) \cup f(a), I \models Y \sqsupset f(X) \cup p_1 f^{-1}(X) \cup a$. We also make I implicit when not ambiguous.

Sometimes it is more compact to present a theory if we write inequality constraints in a vector normal form. For example, the constraints in Example 2 can be rewritten in the vector form:

$$I \models \mathbf{X} \sqsupset \begin{bmatrix} \Lambda \mathbf{v}.f(Y) \sqcup p_1 \Lambda^{-1} \mathbf{v}.f(X) p_2 \sqcup \Lambda \mathbf{v}.f(a) \\ \Lambda \mathbf{v}.f(X) \sqcup p_2 \Lambda^{-1} \mathbf{v}.f(Y) p_1 \sqcup \Lambda \mathbf{v}.a \end{bmatrix} \mathbf{X}, \text{ where } \mathbf{X} = \begin{bmatrix} X \\ Y \end{bmatrix} \quad (5)$$

Definition 6. *A normal form of constraints is* $I \models \mathbf{X} \sqsupset \mathbf{AX}$*, where* $\mathbf{X}$ *is of the form* $[v_i]_{i=1}^n$ *and* $\mathbf{A}$ *is of the form* $[A_i]_{i=1}^n$*, where* A_i *is produced by the following grammar* $\mathsf{A} \to \lambda \mathbf{v}.\emptyset | \left(\Lambda \mathbf{v}.t | p(\Lambda^{-1}\mathbf{v}.t)p'\right)^{*,\sqcup}$ *where* $t \in \mathsf{Term}$*,* $\mathbf{v} \subset \mathsf{Var}$*,* p *and* p' *are Proj, there is no repeated terms in* A*, and the terms in* A *are ordered in a certain total order.*

3.4 Theorems[1]

In this section, we look at a few theorems. Theorem 1 shows that $\mathbf{X} \sqsupset \mathbf{AX}$ is solvable. Theorem 2 shows that a solution to constraints $cons(S)$ is complete for S, which is the main result of this section. Lemma 1 shows that resolution does not affect the solution of the constraints. If we assume that the semantics of S is given by resolution, then Lemma 1 is analogous to preservation (or subject reduction) of a type system.

Theorem 1. *The minimum solution* I *to an inequality* $\mathbf{X} \sqsupset \mathbf{AX}$ *is given by* $I(\mathbf{v}) = \bigcup_{i=0}^{\infty} (\mathbf{A}^i)^* \emptyset$*. The rhs is a fixpoint of the function* $f(x) = (\mathbf{A})^*(x)$*.*

Lemma 1. *Given a set of clauses* S *s.t. for any clauses* $C, D \in S$*,* $fv(C) \cap fv(D) = \emptyset$*, the binary resolvent* D *of two clauses* C_1 *and* C_2 *in* S *by some well-formed mgu,* $cons(S) \equiv cons(S \cup \{D\})$*.*

Theorem 2. *Given a set of clauses* S *s.t. for any clauses* $C, D \in S$*,* $fv(C) \cap fv(D) = \emptyset$*, any solution* I *to* $cons(S)$ *is complete for* S*.*

4 Context Free Type

Generally speaking, I may have complicated structures. We need to find a finite representation for these sets that facilitates enumeration of terms, as one of the purposes of types is generating terms. By Theorem 1, $\mathbf{A}$ is a finite representation of the minimum solution of the constraint, but it is not very efficient as it includes

[1] A long version of this paper can be found at http://cs.unc.edu/~xuh/oshls.

$RTFunc$s which require pattern matching. Besides, it is also hard to generate terms in certain order, say, the size-lexicographic order. The approach introduced in this section converts the constraints into a grammar that produces a solution without $RTFunc$s.

Given a $FProd$ $\mathbf{A} = [A_i]_{i=1}^n$ in the normal form, for any component A of $\mathbf{A}$, we denote the $FSum$ of $TFunc$ or $FSumUnit$ by $A^{\rightarrow}$, and the $FSum$ of other terms (which contain $RTFunc$s) by $A^{\leftarrow}$, for example, $A^{\rightarrow} = \bigcup_{i=1}^n (\Lambda \mathbf{v}.t_i)$ and $A^{\leftarrow} = \bigcup_{i=1}^n p_i(\Lambda^{-1}\mathbf{v}.t_i)p_i$. Using this notation, an inequality can be written in the following form.

$$\mathbf{X} \sqsupset (\mathbf{A}^{\rightarrow} \sqcup \mathbf{A}^{\leftarrow})\mathbf{X} \tag{6}$$

Example 3. Suppose (5). $\mathbf{A}^{\rightarrow} = \begin{bmatrix} \Lambda \mathbf{v}.f(Y) \sqcup \Lambda \mathbf{v}.f(a) \\ \Lambda \mathbf{v}.f(X) \sqcup \Lambda \mathbf{v}.a \end{bmatrix}$. $\mathbf{A}^{\leftarrow} = \begin{bmatrix} p_1\Lambda^{-1}\mathbf{v}.f(X)p_2 \\ p_2\Lambda^{-1}\mathbf{v}.f(Y)p_1 \end{bmatrix}$.

$$\mathbf{A}^{\leftarrow}\mathbf{A}^{\rightarrow} = \begin{bmatrix} p_1\Lambda^{-1}\mathbf{v}.f(X)\Lambda\mathbf{v}.f(X) \sqcup p_1\Lambda^{-1}\mathbf{v}.f(X)\Lambda\mathbf{v}.a \\ p_2\Lambda^{-1}\mathbf{v}.f(Y)\Lambda\mathbf{v}.f(Y) \sqcup p_2\Lambda^{-1}\mathbf{v}.f(Y)\Lambda\mathbf{v}.f(a) \end{bmatrix} = \begin{bmatrix} \Lambda\mathbf{v}.X \\ \Lambda\mathbf{v}.Y \sqcup \Lambda\mathbf{v}.a \end{bmatrix},$$

by equations $\begin{matrix} p_i(\Lambda^{-1}\mathbf{v}.f(v_i))\Lambda\mathbf{v}.f(t) = \Lambda\mathbf{v}.t \\ p_i\Lambda^{-1}\mathbf{v}.f(v_i)\Lambda\mathbf{v}.a \quad = \lambda\mathbf{v}.\emptyset \end{matrix}$.

We generalize the idea illustrated in Example 3. $\mathbf{A}^{\leftarrow}\mathbf{A}^{\rightarrow}$ can be expanded to an $FProd$ of $FSum$ of terms of the form $p_k(\Lambda^{-1}\mathbf{v}.t)(\Lambda\mathbf{v}.t')$ or $\lambda\mathbf{v}.\emptyset$. In general, not all terms of the form $p_k(\Lambda^{-1}\mathbf{v}.t)(\Lambda\mathbf{v}.t')$ can be converted to a simpler, equal term. The inequalities shown as follows, where $ll' \in \mathsf{Pos}, l \in \mathsf{Pos}$, can be used to simplify them to simpler terms.

$$p_k(\Lambda^{-1}\mathbf{v}.t)(\Lambda\mathbf{v}.t') \sqsubset \begin{cases} p_k(\Lambda^{-1}\mathbf{v}.t'')p_{k'} & \sigma = mgu(t',t), \sigma(v_k) = v_k, \\ & v_k \in fv(t''), t'' = \sigma(v_{k'}) \notin \mathsf{Var} \\ \Lambda\mathbf{v}.t'' & \sigma = mgu(t',t), \sigma(v_k) = t'' \\ \lambda\mathbf{v}.\emptyset & t', t \text{ not unifiable} \end{cases}$$

For example, if $\mathbf{v} = [X, Y]$, then $p_1(\Lambda^{-1}\mathbf{v}.f(f(X)))(\Lambda\mathbf{v}.f(Y)) \sqsubset p_1(\Lambda^{-1}\mathbf{v}.f(X))p_2$, $p_1(\Lambda^{-1}\mathbf{v}.f(X))(\Lambda\mathbf{v}.f(f(Y))) \sqsubset \Lambda\mathbf{v}.f(Y)$, $p_1(\Lambda^{-1}\mathbf{v}.f(f(X)))(\Lambda\mathbf{v}.a) \sqsubset \lambda\mathbf{v}.\emptyset$. If two simplifications are applicable simultaneously, then we choose one arbitrarily. The reduced term can be rearranged into an equal normal form. If the original term is $\mathbf{T}$, then we denote the normal form of the reduced term by $s(\mathbf{T})$. $s(\mathbf{T}) \sqsupset \mathbf{T}$ for any $\mathbf{T}$.

Algorithm 2. *(CFT) Given an inequality* $\mathbf{X} \sqsupset \mathbf{AX}$*, compute* $\mathbf{B}_k$ *as follows.*

$$\begin{aligned} \mathbf{B}_0 &= \mathbf{A} \\ \mathbf{B}_{k+1} &= \mathbf{B}_k \sqcup s(\mathbf{B}_k^{\leftarrow}\mathbf{B}_k^{\rightarrow}) \end{aligned}$$

Theorem 3. *(Termination) For any constraint* $\mathbf{X} \sqsupset \mathbf{AX}$*, there is an integer* N *s.t.* $\mathbf{B}_N = \mathbf{B}_{N+1}$.

Lemma 2. $\bigcup_{i=0}^{\infty}(\mathbf{B}_N^i)^*\emptyset = \bigcup_{i=0}^{\infty}((\mathbf{B}_N^{\rightarrow})^i)^*\emptyset$.

Theorem 4. *(Soundness) If* $I(\mathbf{v}) = \bigcup_{i=0}^{\infty}((\mathbf{B}_N^{\rightarrow})^i)^*\emptyset$*, then* $I \models \mathbf{X} \sqsupset \mathbf{AX}$.

Unlike $\mathbf{A}$, $\mathbf{B}_N^{\rightarrow}$ does not contain $RTFunc$s. Constraint $\mathbf{X} \sqsupset \mathbf{B}_N^{\rightarrow}\mathbf{X}$ can be straightforwardly converted to BNF productions by syntactically converting "$\cup$" to "$|$" and "$\sqsupset$" to "$\rightarrow$". The resulting grammar generates the solution I.

Example 4. Suppose (5). $\mathbf{B}_0 = \mathbf{A}$. $s(\mathbf{A}^{\leftarrow}\mathbf{A}^{\rightarrow})$ is shown in (b). $\mathbf{B}_1$ is shown in (a). $s(\mathbf{B}_1^{\leftarrow}\mathbf{B}_1^{\rightarrow}) \sqsubset \mathbf{B}_1$. Therefore, we may choose $N = 1$. $\mathbf{X} \sqsupset \mathbf{B}_1^{\rightarrow}\mathbf{X}$ is shown in (c). A grammar is generated as shown in (d). The solution generated by this grammar is $I(\mathbf{v}) = \{[f(a)], [f(f(f(a)))], \ldots\} \times \{[a], [f(f(a))], \ldots\}$.

$$\begin{bmatrix} p_1\Lambda^{-1}\mathbf{v}.f(X)p_2 \sqcup \Lambda\mathbf{v}.f(Y) \sqcup \Lambda\mathbf{v}.f(a) \sqcup \Lambda\mathbf{v}.X \\ p_2\Lambda^{-1}\mathbf{v}.f(Y)p_1 \sqcup \Lambda\mathbf{v}.f(X) \sqcup \Lambda\mathbf{v}.a \sqcup \Lambda\mathbf{v}.Y \end{bmatrix}$$

(a)

$$\begin{bmatrix} \Lambda\mathbf{v}.X \\ \Lambda\mathbf{v}.Y \sqcup \Lambda\mathbf{v}.a \end{bmatrix} \quad \begin{bmatrix} X \\ Y \end{bmatrix} \sqsupset \begin{bmatrix} f(Y) \cup f(a) \cup X \\ f(X) \cup a \cup Y \end{bmatrix} \quad \begin{matrix} X \rightarrow f(Y)|f(a)|X \\ Y \rightarrow f(X)|a|Y \end{matrix}$$

(b) (c) (d)

5 Related Work

The (Inst-Gen) rule introduced in [11] uses unification as guide for instance generation. The algorithms presented in [12] also make use of unification to find blockages. [13] shows that using a proper semantics for OSHL is implicitly performing unification. One of the differences between our approach and others is that we infer types before proof search, which divides a theorem proving algorithm into the static analysis stage and the run-time stage. About clause linking, we have the following observation(probably others also have the same observation): in order for any instance $C\sigma$ of C to be useful, each literal $L\sigma$ in $C\sigma$ should be resolvable with some other literals. Given a set of clauses S, a clause C, and a literal L in C, we may *decompose* C on L w.r.t. S to a set of instances of C, $\{C\sigma_i | i \in \{1, 2, \ldots, n\}\}$, s.t. for any literal N in S, σ_i is an mgu of L and $\overline{N}$ for some $i \in \{1, 2, \ldots, n\}$ without affecting the completeness. Our approach generates criteria inspired by this observation. For a survey of instance-base methods, refer to [14].

Sort inference algorithms are implemented in some finite model finders such as Paradox[10] and FM-Darwin[9]. Sorted unification[15] uses sets of unary predicates as sorts. A key difference between our approach and multi-sorted logic is that the types are inferred from S, instead of being part of the logic itself. Therefore, we need to guarantee that the types inferred are backward compatible with the untyped version S. Theorems in this paper is essential for our approach but irrelevant to a multi-sorted logic without inferred sort. Our approach can be viewed as a type system[16] that is equipped with a type inference algorithm. The difference between our type system and those that are usually found in programming languages is that our type system is used to restrict instance generation while type systems in programming languages are usually used to eliminate expressions whose reduction leads to errors. In advanced compilers, some type systems are also used to guide optimization, in which sense it is similar to our type system. The presentation of this paper makes (very primitive)

use of domains[17] and is loosely related to a type theory[18] in the sense that a type system is related to a type theory.

A closely related concept to grammar is tree grammar, where the rhs is composed of trees instead of sequences. The choice of grammar reflects the fact that terms can be internally represented by sequences instead of trees.

6 Discussion

It is possible to refine the constraints (2) and (3) by transforming them into $FInter$s. We denote the $FSum$ of rhs of all constraints (2) and (3) generated by L by $A^L\mathbf{X}$. With this notation, we can combine multiple constraints (2) and (3) generated by L into one constraint: for every $L \in C, C \in S, v \in fv(L)$, $I \models v_L \sqsupset \sqcap_{L \in C} A^L\mathbf{X}$.

The granularity of types is the key to the efficacy of these types. CFT is useful for some problems as shown in Table 1.[2] Finer granularity may be achieved by using enhanced grammars. These grammars are useful when a variable occurs twice in a literal, for example, $P(f(X, X))$. If we have a literal $\neg P(Y)$, then productions produced by CFT include $Y \to f(X, X)$ which does not reflect the correlation between the two occurrences of X. If we make use of an enhanced grammar to mark that X should always be instantiated to the same term, then we have better approximation to the minimum solution. A possible solution is adding correlation markers to correlated variables.

Table 1. Comparison of Performance with and without CFT in OSHL-S 0.5.0

Problem Set	Total	CFT on	CFT off	Difference
SYN	902	487	454	7.3%
PUZ	70	54	49	10.2%
SWV	479	136	123	10.5%

The constraints generated by CFT may have redundancy which affects the time and space complexity of the term generator. We implemented the following algorithms to reduce redundancy. Suppose that $\mathbf{X} \sqsupset \mathbf{T}$, where $\mathbf{X} = [v_k]_{k=1}^n, \mathbf{T} = [T_k]_{k=1}^n, T_i = \bigcup_{s=1}^m T_{i,s}, i \in \{1, 2, \ldots, n\}$. In each iteration of Algorithm 2, if $T_{i,o} = v_j$ and $T_{j,p} = v_i$, then put v_i, v_j into an equivalence class; choose a representative for each equivalence class; for every representative v_i and $v_j \neq v_i$ in the equivalence class, set T_i to $T_i \cup T_j$ and T_j to v_i and replace v_j by v_i in $\mathbf{T}$; and for every T_i s.t. $T_{i,o} = v_i$ for some o, remove v_i from T_i. A variable is *nontrivial* if the rhs of its production contains a term containing only variables that are nontrivial. We slightly modify constraint (4): for some $\mathbf{T} = [T_i]_{i=1}^n$,

[2] Problem sets: TPTP 3.2.0, unsatisfiable or theorem only. Time limit: 30s. Running environment: P4 2.4GHz, 512M, Windows XP, Sun JDK 6.0, OSHL-S 0.5.0, ProverTools. Website for the prover and tools: http://cs.unc.edu/~xuh/oshls/.

where $T_i = c$ or $\emptyset$, for $i \in \{1, 2, \ldots, n\}$ so that there is no trivial variable, $\mathbf{X} \sqsupset \mathbf{T}$. To find a minimal $\mathbf{T}$, first mark all trivial variables; then, break a chain of trivial variables by adding a constant to one of them; repeat the process until there is no trivial variable. Other optimizations include inlining productions that contain no variable and removing redundant terms.

References

1. Plaisted, D.A., Zhu, Y.: Ordered semantic hyper linking. In: J. Autom. Reason. (2000)
2. Baumgartner, P., Fuchs, A., Tinelli, C.: Darwin: A theorem prover for the model evolution calculus. In Schulz, S., Sutcliffe, G., Tammet, T., eds.: IJCAR ESFOR (aka S4). Electronic Notes in Theoretical Computer Science (2004)
3. Claessen, K.: Equinox, a new theorem prover for full first-order logic with equality. Presentation at Dagstuhl Seminar 05431 on Deduction and Applications (October 2005)
4. Korovin, K.: iProver — an instantiation-based theorem prover for first-order logic (system description). In: Proc. IJCAR '08, Berlin, Heidelberg, Springer-Verlag (2008)
5. Letz, R., Stenz, G.: DCTP: A disconnection calculus theorem prover. In Goré, R., Leitsch, A., Nipkow, T., eds.: Proc. IJCAR-2001, Siena, Italy. Volume 2083 of LNAI., Springer, Berlin (June 2001)
6. Sutcliffe, G.: CASC-J4 the 4th IJCAR ATP system competition. In: Proc. IJCAR '08, Berlin, Heidelberg, Springer-Verlag (2008)
7. Schulz, S.: System abstract: E 0.81. In: Proc. 2nd IJCAR. Volume 3097 of LNAI., Springer (2004)
8. Riazanov, A., Voronkov, A.: The design and implementation of vampire. AI Commun. **15**(2-3) (2002)
9. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. Journal of Applied Logic **7** (2009)
10. Claessen, K.: New techniques that improve mace-style finite model finding. In: Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications. (2003)
11. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. LICS (2003)
12. Hooker, J.N., Rago, G., Chandru, V., Shrivastava, A.: Partial instantiation methods for inference in first-order logic. J. Autom. Reason. **28**(4) (2002)
13. Plaisted, D.A., Miller, S.: The relative power of semantics and unification. In Andreas Podelski, A.V., Wilhelm, R., eds.: Workshop on Programming Logics in memory of Harald Ganzinger, Saarbruecken, Germany (2005)
14. Jacobs, S., Waldmann, U.: Comparing instance generation methods for automated reasoning. J. Autom. Reason. **38**(1-3) (2007)
15. Weidenbach, C.: First-order tableaux with sorts. Logic Journal of the IGPL **3**(6) (1995) 887–906
16. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)
17. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science, Clarendon Press (1994) 1–168
18. Coquand, T.: Type theory. In Zalta, E.N., ed.: The Stanford Encyclopedia of Philosophy. (Spring 2009)