使用显式策略进行程序构造

谭庆平 陈火旺

(国防科技大学计算机系，长沙 410073)

USING EXPLICIT STRATEGIES TO GUIDE PROGRAM CONSTRUCTION

Tan Qingping and Chen Huowang

(*Department of Computer Science, Changsha Institute of Technology, Changsha 410073*)

Abstract In this paper we propose an interactive approach to program synthesis: Explicit strategies are used to guide program construction. Our approach regards program synthesis as a constructive theorem proving task. By specifying proof strategies in a higher order functional meta-language, we are able to represent the programmer's advice, mathematical inductions, program transformation rules/strategies and resolution methods in a uniform manner. The implementation of strategies depends on higher order unification. Our approach can be automatized easily.

摘要 本文提出一种交互式的程序(半)自动综合方法：使用显式策略引导系统进行程序构造．这些策略包括程序员提供的问题求解策略和系统内部的标准策略，如数学归纳法、归结方法和程序变换规则/策略．策略都用高阶函数式元语言 TSL/ML 统一地描述，它们的施用则通过高阶一致化完成．因此，我们的程序综合方法可以在统一的框架下使用多种软件自动构造技术，且易于自动实现．

§1. 引 言

程序综合方法试图从描述性的软件规范出发，系统地派生出目标程序并保证其正确性．由于软件设计是一个需要大量知识和高度智能的问题求解过程，至少在相当长一段时间内，人的作用不可能完全被机器所取代．因此，应该允许程序员在规范描述和程序综合过程中显式说明问题求解策略，以引导系统构造目标程序．为减轻交互方式下程序员的负担，程序综合系统自身应具备足够多的程序设计知识(规则和策略)，并提供友好的人机界面．

本文 1990 年 4 月 28 日收到，1990 年 9 月 19 日定稿．本项研究得到“八六三”计划“软件生产自动化”项目和国家自然科学基金的部分资助．作者 **谭庆平**，1988 年硕士毕业于国防科技大学，现为在读博士生，目前主要从事软件工具与环境，计算机科学理论方面的研究工作．**陈火旺**，教授，国防科技大学研究生院副院长，主要从事计算机科学理论，软件工程方面的研究工作．

基于上述认识，本文提出一个策略制导的交互式程序综合方法．我们将程序综合视为构造性的定理证明过程，并使用高阶函数式元语言统一地描述各种证明策略，包括程序员提供的问题求解策略和系统内部的标准策略，如数学归纳法，归结方法和程序变换规则 / 策略．策略的施用则通过高阶一致化完成．

§ 2. 一个简单的规范语言 TSL

为了讨论程序的 (半) 自动综合过程，首先引进规范语言 TSL.

2.1 TSL 类型

设 T_0 为基本类型符号的有穷集． T_0 中的所有类型均在抽象数据类型中说明.

TSL 的类型集 T 递归定义如下：

· $T_0 \subseteq T$;

· 若 α 是类型变元，则 $\alpha \in T$;

· 若 $S_1, \cdots, S_n \in T$, $S_0 \in T_0$, 则 $(S_1, \cdots, S_n \to S_0) \in T$.

2.2 TSL 项

首先定义原子集 $A = C \oplus V$, 其中：

· C 是带类型的常元符号集． C 中包括条件算符 'if_then_else', 'if_then' 和不动点算符 'fixpt', 它们的类型依次是 Bool, α, $\alpha \to \alpha$, Bool, $\alpha \to \alpha$ 和 $(\alpha \to \alpha) \to \alpha$;

· $V = \bigcup_{s \in T} V_s$, V_s 是类型为 s 的变元符号集.

TSL 的项集 $Term$ 由以下规则递归定义：

· $A \subseteq Term$;

· 若 $f : s_1, \cdots, s_n \to s$ 是 C 中常元或 (用来定义基本类型的) 抽象数据类型中的函数符号， t_i 是 $Term$ 中类型为 s_i 的项，则 $f(t_1, \cdots, t_n)$ 是 $Term$ 中类型为 s 的项；

· 若 $t \in Term$, 类型为 s, $u_i \in V_{s_i}$, 则 $\lambda u_1, \cdots, u_n.t$ 是 $Term$ 中类型为 $s_1, \cdots, s_n \to s$ 的项；

· 若 t 是 $Term$ 中类型为 s 的项， α 是类型变元， s_0 是 TSL 类型，则 $t[s_0/\alpha]$ 是 $Term$ 中类型为 $s[s_0/\alpha]$ 的项.

今后不再显式说明 TSL 项的类型，因为它们均可由上下文确定.

2.3 功能模块

定义 2.1： TSL 中的功能模块形如

$$FM \equiv < x_1 : s_1, \cdots, x_m : s_m; z_1 : s'_1, \cdots, z_n : s'_n;$$
$$pre(x_1, \cdots, x_m); post(x_1, \cdots, x_m, z_1, \cdots, z_n);\ advice >,$$

其中 x_i 为输入参数， z_j 为输出参数． $pre(\bar{x})$ 和 $post(\bar{x}, \bar{z})$ 是类型为 Bool(通常的布尔类型) 的 TSL 项，它们分别表示 FM 的输入、输出性质的前后置断言． advice 用来描述程序员对 FM 的实现策略． § 4 将讨论它的语法表示和语义.

非形式地，定义 2.1 中的功能模块对应下述带类型的一阶断言：

$$Th \vdash \forall \bar{x} : \bar{s}. \exists \bar{z} : \bar{s}'. (pre(\bar{x}) \supset post(\bar{x}, \bar{z})) \qquad (**)$$

这里 Th 是相关抽象数据类型中所有公理的集合. 对 FM 进行程序综合就是构造函数 $\bar{f} =< f_1, \cdots, f_n >$ 使得 $Th \vdash \forall \bar{x} : \bar{s}.(pre(\bar{x}) \supset post(\bar{x}, \bar{f}(\bar{x})))$ 成立.

§ 3. 高阶函数式元语言 TSL/ML

TSL/ML 用来表示目标公式 (**) 及各种证明策略.

3.1 元类型

定义 3.1: TSL/ML 的元类型定义如下:

· 若 s 是 TSL 类型, 则 $TERM_s$ 是元类型. FORMULA 是 $TERM_{Bool}$ 的缩写;

· 若 MS 是元类型, 则 $MS_{sequence}$ 是元类型, 这里 $sequence$ 为抽象表类型;

· 若 $MS_1, \cdots, MS_n$ 和 MS 是元类型, 则 $MS_1, \cdots, MS_n \to MS$ 是元类型;

· $SEQUENT$ 是 TSL/ML 的特殊元类型, 它的元级项是逻辑演算中的后继式, 形如 $\Gamma \vdash A$, 其中 Γ, A 的元类型分别为 $\text{FORMULA}_{sequence}$ 和 FORMULA.

类型为 TERM_s 的元级项类似于 TSL 中类型为 s 的项, 但元项中可能出现 "逻辑变元"(该术语出自 Isabelle[9]). 逻辑变元与普通变元的区别在于: 前者在高阶一致化过程中可能被实例化, 而后者则形如常元, 不能被高阶一致化子替换. 引进逻辑变元是为了表示目标公式的 Skolem 标准形以及关于公式模式的证明策略. 本文以前缀 "?" 标记逻辑变元.

3.2 元项

定义 3.2: TSL/ML 的元项定义如下:

· TSL 中类型为 s 的变元或常元是 TSL/ML 中类型为 TERM_s 的元项;

· (带元类型的) 元级常元是元项. 对任意 TSL 类型 s, 类型为 TERM_s 的逻辑变元 $?f, ?g, \cdots$ 是元项;

· 若 t 是类型为 $MS_1, \cdots, MS_n \to MS$ 的元项, u_i 是类型为 MS_i 的元项, 则 $t(u_1, \cdots, u_n)$ 是类型为 MS 的元项;

· 若 t 是类型为 TERM_s 的元项, x_i 是类型为 S_i 的变元, 则 $\lambda x_1, \cdots, x_n.t$ 是类型为 $\text{TERM}_{s_1, \cdots, s_n \to s}$ 的元项;

· 若 t, u_i 分别是类型为 $\text{TERM}_{s_1, \cdots, s_n \to s}$ 和 TERM_{s_i} 的元项, 则 $t(u_1, \cdots, u_n)$ 是类型为 TERM_s 的元项;

· 空序列 [ ] 是类型为 $MS_{sequence}$ 的元项, 这里 MS 为任意元类型; 若 Γ, A 分别是类型为 $MS_{sequence}$ 和 MS 的元项, 则 $[\Gamma, A]$ 是类型为 $MS_{sequence}$ 的元项;

· 若 Γ, A 分别是类型为 $\text{FORMULA}_{sequence}$ 和 FORMULA 的元项, 则 $\Gamma \vdash A$ 是类型为 SEQUENT 的元项.

§ 4. 证明规则 / 策略在 TSL/ML 中的表示

4.1 目标公式的表示

按定义 3.2, 任何类型为 s 的 TSL 项均可视为元类型为 TERM_s 的元级项. (**) 式右边的 $\forall\exists_-$ 型公式可表示成类型 FORMULA 的元项: 首先, 将公式 Skolem 化, 每个 Skolem 函数被表成相应元类型的逻辑变元; 然后, 去掉所有的全称量词就得到了 $\forall\exists_-$ 型公式的

元级表示. (**) 式左边的 Th 可表示成类型 $\text{FORMULA}_{Sequence}$ 的元项. 因此 TSL/ML 将目标公式 (**) 表达为类型 SEQUENT 的元项： $Th \vdash pre(\bar{x}) \supset post(\bar{x}, ?\bar{f}(\bar{x}))$.

4.2 证明规则 / 策略的表示

类似于 LCF[3] 和 Isabelle[9], TSL/ML 使用元函数描述证明策略. 所有策略均具有以下元类型：

$$TYPE \quad strat \equiv SEQUENT \to SEQUENT_{sequence}$$

定义 4.1: 策略有以下两种形式：

i) 简单策略： $ST \equiv < \rho, In, Out >$, (记为 $\rho \to ST(In) = Out$ 或 $ST(In) = Out$, 当 $\rho \equiv true$), 其中 ρ 是 ST 的可应用性条件, 类型为 FORMULA; In, Out 分别为 ST 的输入模式与输出模式, 它们都是 TSL/ML 的元项, 类型分别是 SEQUENT 与 $\text{SEQUENT}_{sequence}$;

ii) 复合策略： $ST \equiv < \rho, In, Body >$, 其中 $Body$ 是策略 ST 的函数体, 它将 ST 说明为 TSL/ML 中某些策略的组合. 策略组合运算类似于 Isabelle[9].

以下列出几个典型的策略组合运算：

· TYPE All_ST: strat
VAL All_ ST (ξ) = [ξ];

· TYPE THEN: strat, strat → strat

THEN (ST_1, ST_2) (ξ) 首先对后继式 ξ 使用 ST_1 产生子目标序列, 然后对这些子目标依次使用策略 ST_2;

· TYPE ORELSE: strat, strat → strat

$$\text{ORELSE } (ST_1, ST_2)\ (\xi) = \begin{cases} ST_1(\xi) & \text{若} ST_1 \text{对} \xi \text{可应用} \\ ST_2(\xi) & \text{否则} \end{cases}$$

· TYPE REPEAT: strat → strat

REPEAT (ST)=(ST THEN REPEAT (ST)) ORELSE ALL_ST.

为了对后继式 ξ 使用策略 ST, 首先必须在 ξ 和 ST 的输入模式 In 之间进行高阶一致化. 如果对 ξ 和 In 中的逻辑变元存在一致化子 σ 满足：

$\sigma(\xi) = \sigma(In)$ 且 $\sigma(\rho)$ 成立, 那么：

i) 当 ST 为简单策略：不妨设 $ST =< \rho, In, [\eta_1, \cdots, \eta_k] >$, 此时 $ST(\xi) = [\sigma(\eta_1), \cdots, \sigma(\eta_k)]$;

ii) 当 ST 为复合策略：设 ST=$< \rho, In, Body >$, 则 $ST(\xi) = Body(\xi)$.

综上, 系统对策略的使用主要依赖于高阶一致化算法 [6][4].

以下依次说明各种证明策略在 TSL/ML 中的表示形式.

4.2.1 程序员策略

目前, TSL/ML 允许程序员在软件规范及程序构造过程中说明情形策略、归纳策略、分而治之 (divide_and_conquer) 策略以及通过建立子目标或引理指导系统进行程序综合. 这里仅介绍前两种.

为叙述简单起见, 我们只考虑单输入参数, 单输出参数的功能模块.

情形策略的一般形式是：

```
ST1:: Case_strategy  bool_exp1;
                        ⋮
                     bool_expk
      End_strategy
```

其中 bool_exp_i 是类型为 Bool 的 TSL 项.

系统将 ST_1 编译成下述形式的 TSL/ML 表示:

$$\begin{aligned}\bigvee_{i=1}^{k} \text{bool\_exp}_i \rightarrow \text{ST}_1\ (?\ \Gamma \vdash ?\text{R}\ (&\underline{\text{if}}\ \text{bool\_exp}_1\ \underline{\text{then}}\ ?\ \text{f}_1\\ &\underline{\text{else if}}\ \text{bool\_exp}_2\ \underline{\text{then}}\ ?\ \text{f}_2\\ &\quad\vdots\\ &\underline{\text{else if}}\ \text{bool\_exp}_k\ \underline{\text{then}}\ ?\ \text{f}_k))\\ = [?\Gamma,\ \text{bool\_exp}_1 \vdash ?\text{R}\ (?\ \text{f}_1);&\\ \vdots&\\ ?\Gamma,\ \text{bool\_exp}_k \vdash ?\ \text{R}\ (?\ \text{f}_k)]&\end{aligned}$$

显然, 这一编译过程是可以机械完成的.

归纳策略的一般形式是:

ST_2:: Induction_strategy

　　Ind_Base: $x = \text{exp}_b$

　　Ind_step: $x = \text{exp}_h \Rightarrow x = \text{exp}_i$

End_strategy

系统在完成对上述归纳框架的有效性验证 (略) 之后, 生成:

$$\begin{aligned}&\text{ST}_2\ (?\ \Gamma \vdash ?\ \text{R}\ (\text{fixpt}\ (\lambda \text{F}.\ \lambda x.\ \underline{\text{if}}\ (x == \text{exp}_b)\underline{\text{then}}\ ?\ \text{f}_0\\ &\qquad\qquad \underline{\text{else if}}(x == \text{exp}_i)\underline{\text{then}}?\text{g})))\\ &= [?\Gamma \vdash ?\text{R}\ (?\ \text{f}_0)[\text{exp}_b/x];\\ &\quad ?\Gamma, ?\text{R}\ (\text{F}) \vdash ?\text{R}\ (?\text{g})\ [\text{exp}_i/x]]\end{aligned}$$

这里 $t[\text{exp}/x]$ 表示将项 t 中 x 的所有出现替换成 exp 所得到的项.

4.2.2 系统内部的标准策略.

4.2.2.1 数学归纳法: 在程序构造过程中, 如果当前欲证明的后继式中含有递归类型的输入参数, 系统首先对该式进行递归分析 [10], 产生归纳框架 (induction scheme), 并生成归纳策略的 TSL/ML 表示.

4.2.2.2 归结方法: 简单的归结规则可表示成:

$$\text{Resolution\_strategy}\ (?\ \Gamma \vdash ?\ \text{R}\ ((\neg(?A_1) \vee ?A_2) \& ?A_1)) = [?\Gamma \vdash ?R(?A_2)]$$

4.2.2.3 分而治之方法: TSL/ML 可表达 [11] 中提出的分而治之策略.

4.2.2.4 程序变换规则 / 策略: 程序变换规则本质上是关于程序 / 规范的重写规则, 所以 TSL/ML 中的简单策略可以很直接地表达这些规则. 程序变换策略则由 TSL/ML 的复合策略表示.

§ 5. 策略制导的交互式程序综合

我们的程序综合系统采用面向目标、策略制导的交互式定理证明技术. 在程序综合过程中, 系统维持一棵不断生长的证明树. 初始时, 该树仅含一个根节点, 它以欲证明的后继式为标记. 在证明过程中的任意时刻, 系统对当前子目标寻找可使用的策略. 如果有, 证明树按通常方式生长. 否则, 系统进行自动回溯. 因为最一般的高阶一致化子不一定存在, 所以系统可尝试父节点与相应策略的输入模式的下一个一致化子, 派生出新的子目标, 然后对这些子目标继续证明; 或者, 对父节点寻找其它可用的策略. 考虑

到系统的实际运行速度，回溯不可能完全．如果系统决定在某失败节点不再回溯，它将向程序员报告失败原因，请求提供策略．

为减轻交互方式下程序员的负担，系统应具有友好的人机界面、充分的形式推理及算法构造能力，因此，我们要求：系统拥有策略库，其中包括软件开发和一些重要应用领域内的问题求解策略和典型的算法构造方法；系统具有自动进化能力，它可以将程序员提供的某些策略纳入策略库，并且，它应该从经验中学习各种证明策略．

TSL/ML 为各种证明策略的表示和实现提供了统一的机制，这便为策略库的建立和扩充打下了极好的基础；TSL/ML 的高阶机制为系统的自动进化提供了现实可能性 [1].

§ 6. 实例：一个分类算法的自动综合过程

本节通过实例说明策略的施用过程．

6.1 记号说明

List 是通常的整数表类型．$<>$ 表示空表，$a.x$ 表示整数 a 与表 x 连接之后形成的新表．$ord(x)$ 和 $perm(x,y)$ 分别表示 x 是有序的 (自小至大)，x,y 互为排列．$|x|$ 表示表 x 的长度．$min(x)$ 表示 x 的最小元素，$del(x,a)$ 表示从表 x 中删除元素 a 之后得到的表．

欲证明的目标后继式是 $Th \vdash ord(?f(x))\&perm(x,?f(x))$,
这里 Th 是抽象数据类型 List 中所有公理的集合．为简便记，今后略去 Th.
对上式进行递归分析后，系统生成归纳策略：

$$Induction-ST(\vdash ord(FIX(x))\&perm(x,FIX(x))) = [\vdash ord(?g)\&perm(<>,?g); \quad (**1)$$

$$|?u|<|a.x'| \supset ord(F(?u))\&perm(?u,F(?u)) \vdash ord(?b.?z)\&perm(a.x',?b.?z)] \quad (**2)$$

对原后继式使用该策略后得到子目标 $(**1)$ 和 $(**2)$, 高阶一致化子是：

$$?f \Rightarrow FIX \quad \text{(U1)}$$

其中

$$FIX \equiv (\underline{\text{fixpt}}\ \lambda F.\lambda x.\ \underline{\text{if}}(x == <>)\underline{\text{then}}?g$$
$$\underline{\text{else}}\ \underline{\text{if}}(x == a.x')\underline{\text{then}}?b.?z)$$

对 $(**1)$ 的证明过程略．显然有 $?g \Rightarrow <>$ (U2)
设有策略：

$$ST_1(?\Gamma \vdash ord(\min(?y).?w)\&perm(?y,min(?y).?w))$$
$$= [?\Gamma \vdash ord(?w)\&perm(del(?y,min(?y)),?w)]$$

施用于 $(**2)$ 后得到
$|?u|<|a.x'| \supset ord(F(?u))\&perm(?u,F(?u)) \vdash ord(?w)\&perm(del(a.x',min(a.x')),?w)$ $(**3)$
使用的一致化子是：

$$\{?y \Rightarrow a.x', ?b \Rightarrow \min(a.x'), ?z \Rightarrow ?w\} \quad \text{(U3)}$$

再对 $(**3)$ 使用标准策略：$ST_2(?A \supset ?B \vdash ?B) = [?A]$ 得到 $|DEL|<|a.x'|$ $(**4)$

使用的一致化子是：

$$\{?u \Rightarrow DEL, ?w \Rightarrow F(DEL), ?A \Rightarrow |DEL| < |a.x'|, \\ ?B \Rightarrow ord(F(DEL)) \& perm(DEL, F(DEL))\} \qquad (U4)$$

其中 $DEL \equiv del(a.x', min(a.x'))$

对 $(**4)$ 的证明略．最后，合成 $(U1) \sim (U4)$ 得：

$$?f \Rightarrow (\underline{\text{fixpt}}\ \lambda F.\lambda x.\underline{\text{if}}(x == <>)\underline{\text{then}}\ <> \\ \underline{\text{else}}\ \underline{\text{if}}\ (x == a.x')\underline{\text{then}}\ \min(a.x').F(del(a.x', \underline{\min\ (\text{a.x}')})))$$

§ 7. 结论及有关工作

本文提出的策略制导的程序综合方法试图利用程序员提供的策略引导系统进行程序构造．各种证明策略均具有相同的表示形式和实现机制，因此该方法在统一的框架下结合了多种软件自动构造技术，并且易于自动实现．

LCF[3] 是基于策略的定理证明器的主要代表，近年来 Isabelle[9] 和 λ_Prolog[2] 开始采用高阶逻辑描述和实现证明策略．利用高阶一致化进行程序变换的思想源于 [5] 和 [7]．我们的研究还受到了演绎综合方法 [8] 的影响．进一步的研究及机器实现目前正在进行中．

参考文献

[1] M.R.Donat & L.A.Wallen: "Learning and Applying Generalized Solutions Using Higher-Order Resolution", Proc. of 9th CADE, (1988), 41–60.

[2] A.Felty and D.Miller: "Specifying Theorem Provers in a Higher-Order Logic Programming Language", Proc. of 9th CADE, (1988), 61–80.

[3] M.J.Gordon & A.J.Milner & C.P.Wadsworth: "Edinburgh LCF: A Mechanized Logic of Computation", LNCS, Vol. 78, (1979).

[4] J.H.Gallier & W.Snyder: "Design Unification Procedures Using Transformations: A Survey", Bulletin of EATCS, Vol. 40, Feb. 1990, 273–326.

[5] G.P.Huet and B.Lang: "Proving and Applying Program Transformations Expressed with Second-Order Patterns", Acta Informatica 11, (1978), 31–55.

[6] G.P.Huet: "A Unification Algorithm for Typed λ-Calculus", Theoretical Computer Science 1, (1975), 27–57.

[7] D.Miller & G. Nadathur: "A Logic Programming Approach to Manipulating Formulas and Programs", IEEE Sym. Logic Programming, 1987, 379–388.

[8] Z.Manna and R.Waldinger: "A Deductive Approach to Program Synthesis", ACM Trans. Programming Languages and Systems 2 (1), (1980), 91–121.

[9] T.Nipkow: "Equational Reasioning in Isabelle", Science of Computer Programming 12, (1989), 123–149.

[10] A.Stevens: "A Rational Reconstruction of Boyer and Moore's Technique for Constructing Induction Formulas", Proc. of ECAI' 88, 565–570.

[11] D.R.Smith: "Top-Down Synthesis of Divide-and-Conquer Algorithms", Artificial Intelligence, Vol. 27, No. 1, Sep. 1985.