KITP-93: An Automated Inference System for Program Analysis

T. C. Wang and Allen Goldberg
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304

KITP is an automated inference system developed for supporting the formal design, verification, and validation of computer programs. It has evolved from an automated verification system, RVF [5]. The latest version of KITP, KITP-93, features a typed formulation and deduction, meta-level reasoning, and resolution-based proving enhanced by conditional term-rewriting. This report describes KITP-93 from a user's perspective.

1 Logical Framework

KITP-93 employs a classical higher-order language for a convenient interaction with the user, and a typed clausal language for efficiently carrying out the inference. It accepts, in general, any well-formed higher-order formula. Internally, all input axioms and lemmas will be transformed into clausal normal form, and stored as rules in the knowledge-base (KB) of the system. Only these KB rules will be directly accessed by the deductive components of the system. For example, the user is allowed to input the following statement into the KB,

1. $\forall(s)(stringp(s) \wedge \forall(k\!: char)(k \ in \ s \Rightarrow (k \geq \#2 \wedge k \leq \#7)) \Rightarrow valid\text{-}key(s))$

The system will transform the statement into the following three KB rules (k! is a Skolem symbol).

1a. $(k!(s) : char)/\neg valid\text{-}key(s)/(s : seq(char))$
1b. $valid\text{-}key(s) \vee \neg(\#2 \leq k!(s)) \vee \neg(k!(s) \leq \#7)/(s : seq(char))$
1c. $valid\text{-}key(s) \vee (k!(st) \ in \ s)/(s : seq(char))$

The set of clausal normal forms is divided into two classes: typing rule and typed clause. A typing rule has a form $t\!: \tau/H/R$, which specifies that a term t has a type τ if H and R holds. $t\!: \tau$ is called a type description (TD). R is a conjunction of TDs. H is a conjunction of ordinary literals. For example, the above 1a is a typing rule, which states that $k!(s)$ is of type *char* if s is of type $seq(char)$ and $\neg valid\text{-}key(s)$ holds. A *typed clause* has a form K/R, where K is an ordinary clause and R is a conjunction of TDs, which means that K holds under R. The above 1b and 1c are typed clauses.

The inference supported by this framework is primarily predicate calculus with equality and a restricted form of higher-order reasoning. It employs two kinds of resolution (and paramodulation). One is kernel resolution, which is a resolution made by unifying two typed clauses $K1/R1$ and $K2/R2$ upon literals of $K1$ and $K2$. Another is a type resolution, which is a resolution made by unifying a TD of a typed clause and the head of a typing rule. As will be described later, while kernel resolution is used to replace the ordinary resolution in transforming an untyped proof procedure into a typed one, type resolution is used for type-checking and type-deduction.

The ability to perform higher-order reasoning is due to a special feature of the clausal language: predicate and function symbols can be denoted by higher-order logical variables. This feature is similar to the use of meta variables in Bundy's meta-level inference system [1]. For instance, the following KB rule contains a variable f which denotes an arbitrary one-arity function.

$$image(f, s\ with\ e) = image(f, s)\ with\ f(e)/(f{:}\,map(\alpha, \beta)) \wedge (s{:}\,set(\alpha)) \wedge (e{:}\,\alpha)$$

Despite the use of higher-order and typed variables, KITP-93 still employs an ordinary Skolemization algorithm for the normalization and a form of first-order unification in the deduction. It treats $\forall(x{:}\,\tau)\phi$ and $\exists(x{:}\,\tau)\phi$ as abbreviations of $\forall(x)(x{:}\,\tau \Rightarrow \phi)$ and $\exists(x)(x{:}\,\tau \wedge \phi)$, respectively. Its ability in higher-order theorem proving has not been well developed. Its logical framework is essentially a predicate calculus expanded with specialized notions and inference rules for type relations. There is no requirement that every logical variable must be typed. In particular, this framework still assumes the set of standard (untyped) equality axioms, and uses (untyped) paramodulation for equality-oriented deduction.

2 Knowledge Base

KITP-93 contains a large KB (300^+ rules), which is built on the data type theory for integers, reals, characters, strings, sets, sequences, tuples and maps. Natural number is introduced as a subtype (*nat*) of integer (*int*). A difficult problem in proving theorems in a real programming environment is the possible search explosion caused by reasoning about large KBs. To attack this problem, we have developed a KB management mechanism for a controlled use of KB rules. KB rules are classified into different rule types with specific constraints. Among them, a *typing-rule* rule will be used only for type-checking and type-deduction, a *reduction* rule only for term-rewriting, a *forward-implication* rule only by the forward inference procedure (FIP), and a *backward-implication* rule only by the backward inference procedure (BIP). While an *any-rule* rule can be used by both FIP and BIP, a *manual-rule* rule will be used nowhere unless it is explicitly claimed by a particular inference task. In addition, the syntactical form of a rule may also add some constraints to its use. For example, for a forward-implication rule, only the first conjunct of its hypothesis is resolve or paramodulated. For a backward-implication, only literals from the conclusion are resolved or paramodulated (this restriction is based on a partial set of support strategy of hierarchical deduction [4]). Many KB rules have been chosen as reduction rules in order to promote the use of term-rewriting. The user may modify the KB or define their own KBs. The following are some examples of KB rules.

2. $(x{:}\,nat)/(x \geq 0)/(x{:}\,int)$ [**typing-rule**]
3. $(x{:}\,int)/true/(x{:}\,nat)$ [**typing-rule**]
4. $(x \geq 0)/(x{:}\,nat)$ [**any-rule**]
5. $stringp(x) = (x{:}\,seq(char))$ [**reduction**]

6. $(x: string) = (x: seq(char))$ **[reduction]**
7. $(concat(x,y): string)/true/(x: string) \wedge (y: string)$ **[typing-rule]**
8. $(x\ in\ concat(p1,p2))or\neg(x\ in\ p1)/(x: char) \wedge (p1: seq(char)) \wedge (p2: seq(char))$ **[any-rule]**
9. $(x\ in\ concat(p1,p2))or\neg(x\ in\ p2)/(x: char) \wedge (p1: seq(char)) \wedge (p2: seq(char))$ **[any-rule]**
10. $(x\ in\ p1)or(x\ in\ p2)or\neg(x\ in\ concat(P1,P2))/(x: char) \wedge (p1: seq(char)) \wedge (p2: seq(char))$ **[any-rule]**
11. $(y: \alpha)/x = y/(x: \alpha)$ **[manual-rule]**

3 Proof Objects

KITP-93 provides inference service through a language construct called a proof-object. A proof-object is a record of classified information about an inference task. The basic usage of a proof-object is to introduce a proof-obligation into the prover. But, it can also be used to introduce user directions and to help incremental development of proofs. A proof-object produced by a batch-mode verification/analysis procedure usually contains no other information except a conjecture to be proved. However, the proof-obligation given in a simple proof-object may not always be discharged automatically and efficiently. Sometimes, a more complex proof-object needs to be used for helping the prover. Such a proof-object can be created by the user from a scratch or by editing a proof-object originally produced by an application procedure. For instance, in order to help prove a conjecture $h_1 \wedge ... \wedge h_n \Rightarrow C$, one can decompose the hypothesis into a set of proof-rules, $h_1, ..., h_n$, and annotate each of them with a specific rule type for the intended use. Other informations that can be provided by proof-object include local bindings (constans with a lexical scope of the object body), KB rules to be disabled, KB rules to be enabled, additional lemmas, hypotheses, and additional conjectures to be proved, specific values of control parameters, special proof-strategies, etc. The following table contains an example of proof-object, which contains a conjecture, and two axioms about *valid-key* mentioned in the conjecture, as well as some informations for helping the proof.

Proof valid-key-prop
local-bindings {valid-key}
proof-rules

valid-key-def-1 : **forward-implication** (definition)
$\forall(s)(valid\text{-}key(s) \Rightarrow stringp(s) \wedge \forall(k : char)(k\ in\ s \Rightarrow (k \geq \#2 \wedge k \leq \#7)))$
valid-key-def-2 : **backward-implication** (definition)
$\forall(s)(stringp(s) \wedge \forall(k: char)(k\ in\ s \Rightarrow (k \geq \#2 \wedge ch \leq \#7)) \Rightarrow valid\text{-}key(s))$

proof-conjectures
conjecture valid-key-prop **[any-rule]**
$\forall(a: string) \wedge \forall(b: string)(valid\text{-}key(a) \wedge valid\text{-}key(b)$
$\Rightarrow valid\text{-}key(concat(a,b)))$
forward-step-limit {1}
backward-step-limit {8}
max-induced-vars {1}
end-conjecture

Normally, all KB rules (except manual-rule rules) together with proof-rules given in a proof-object will be used by the prover in proving a proof-conjecture of the object. To direct the prover to use only a subset of KB rules, one can use the directives **disabled** and **enabled**. For example, to prevent the prover from using other KB rules, except rules 2 – 10 given earlier, one can insert in the above proof-object a line **disabled {all}**, and following it, a line **enabled** {2 – 10 }. However, one can not disable the linear arithmetic theory that has been built in by KITP-93. This theory will be implicitly used in proving all theorems (for example, it will be used to simplify $lb + t_1 - lb$ to t_1).

4 Typed Deduction with Conditional Term Rewriting

The prover of KITP-93 is constructed by incorporating natural deduction, term-rewriting, partial evaluation, unit resolution and paramodulation, set of support strategy, and hierarchical deduction. Except the use of typed deduction, the basic architecture of this inference engine is similar to the prover documented in [5]. Here we discuss some of the advanced features recently added to the prover, namely typed deduction enhanced by conditional term rewriting.

Term rewriting is done in two distinct levels. The basic level uses existing facts to verify the conditions of rewriting rules. It is applied exhaustively to each expression input and generated by other inference procedures of the prover. The advanced level of term rewriting is employed by BIP. For BIP, if the (instantiated) condition of a rewriting rule, which is applied to the current goal G, can not be established immediately, then a resolvent will be produced by combining the instantiated condition and the result of rewriting G. Thus, the condition of a rewriting rule can be handled similarly as the subgoals (literals) inherited from a backward-implication rule. Consider to apply a rewriting rule $p \Rightarrow if(p, q, r) = q/(p : boolean)$ to a goal, $\neg evenp(if(n = 0, ans, foo(n-1, n * ans))) \vee G_1$. If the (instantiated) condition p (i.e., $n = 0$) is not contained in the current KB, then a candidate goal $\neg evenp(ans) \vee n \neq 0 \vee G_1$ will be produced. Thus the verification of the condition can be handled by BIP by including $n \neq 0$ as a subgoal.

The type restriction (TR) attached to a rewriting rule is verified by a type-checking procedure. Given a TR R, the procedure will try to prove $T \models \exists(R)$, where T is the entire set of typing rules. For example, to apply a rewrite rule $(x < y) = (x + 1 \leq y)/(x : int) \wedge (y : int)$ to an expression $t_1 < t_2$, the expression will not be rewritten into $t_1 + 1 \leq t_2$ unless $T \models (t_1 : int) \wedge (t_2 : int)$ has been proved (This proof will fail if $<$ is overloaded and t_1 and/or t_2 is of type $real$).

BIP is based on a typed hierarchical deduction, which differs from the untyped one described in [4] mainly in two aspects. First, the typed deduction uses kernel resolution to carry out the hierarchical deduction. If the current goal K/R is not a kernel-empty clause (i.e. $K \neq box$), then the subgoal to be developed must be chosen from K, and the rules to be used must chosen from the set of typed clauses, but not from T. Moreover, for each resolvent K'/R' produced by a kernel resolution, type-checking will be applied immediately to the type restriction R' in order to determine if R' is satisfiable in T (in the sense that the result of the type-deduction procedure described below is non-empty); and discard the resolvent if it is not.

Second, if a goal is a kernel-empty clause box/R, then a type-deduction procedure will be applied. The type-deduction procedure will deduce from R and T a set of preconditions by which R is implied by T. It thus plays a role similar to Milner's type inference algorithm [3]. Each precondition must be a form $H \wedge Q$, where H is a conjunction of ordinary literals and Q is a conjunction of variable TDs (such as $(x\colon nat) \wedge (y\colon \alpha)$). For example, assuming R is $((a + x)\colon nat)$, and T contains $(a\colon int)$, $(x + y\colon int)/true/(x\colon int) \wedge (y\colon int)$, and all typing rules given earlier, then H will be $a + x \geq 0$ and Q will be $(x\colon int)$, and the resolvent obtained from box/R will be $\neg(a + x \geq 0)/(x\colon int)$.

Type-deduction is implemented by a varient of SLD-resolution extended to abductive reasoning. Type-checking for rewriting and for BIP (and FIP) both is implemented by a simplified version of type-deduction. A reasonable assumption to the structure of the underlying typing theory has been made, which guarantees the finite termination and an efficient implementation of these procedures. The results are cached for reuse. The deterministic nature (and the efficiency) of the type-checking and type-deduction procedures helps improve the performance of the prover. For example, in proving the conjecture of the *valid-key* proof-object, the prover produced a kernel-empty resolvent (named by) -23. With this resolvent, the type-deduction procedure deduced immediately one and only one resolvent (named by) -24. However, if using the untyped deduction, the prover must produce at least four resolvents (i.e. candidate goals) before this useful resolvent can be deduced, because it needs to be deduced from the goal -23 by using four distinct (typing) rules: 1a, 7, $(a! : string)$, and $(b! : string)$.

$-23.\ box/(concat(a!, b!) : seq(char))(k!(concat(a!, b!)) : char)(a! : seq(char))(b! : seq(char))$
$-24 : valid\text{-}key(concat(a!, b!))/true$

BIP can run both automatically and interactively. In the interactive mode, the user can investigate the status of the procedure, trace the history of a derivation, select a goal for the next deduction step, etc.

5 Conclusion

Our goal is to produce a powerful inference system which is capable of dealing with large number of KB rules, and large number of conjectures with diverse features. To achieve this goal, we have built KITP-93 on a logical framework which allows a convenient user interaction and an easy incorporation of existing inference techniques. We have developed a management mechanism for a controlled use of KB rules, and a proof manager for supporting a high-level interaction and incremental development of proofs. We have designed an inference engine by incorporating a variety of efficient inference techniques for raising the degree of automation. In conducting deeper reasoning, we emphasize the role of term-rewriting, goal-oriented deduction, and decision procedures, as well as interactive proof utilities

KITP-93 has been incorporated as an inference server by a number of formal development/analysis environments. Significantly, it has been used successfully by a large industrial user in control flow analysis of Ada procedures. A review of the use of KITP for solving real world problem is included in [Jül93]. Besides proving theorems, other inference services that KITP-93 can provide include disproving a non-theorem, simplifying program fragmentss, answering some questions, and deducing antecedents.

References

1. Bundy, A and Sterlin, L. "Meta-level inference: Two applications," *J. Automatic Reasoning,* 4(1), 15-28 (1988).
2. Jüllig, R. K. "Applying formal software synthesis," *IEEE Software,* 10(3), 11-22 (1993).
3. Milner, R. "A theory of type polymorphism in programming," *J. Comput. System Science,* 17, 348-375 (1978).
4. Wang, T. C., and Bledsoe, W. W. 'Hierarchical Deduction', *J. Automatic Reasoning,* 3(1), 35-71 (1987).
5. Wang, T. C., and Goldberg, A. "RVF: an automated formal verification system," *Proceedings CADE-11* (ed. D. Kapur), LNCS 607, 735-739 (1992).