Smashing Isabelle: Flex-Flex Pairs in 2016

Simon Wimmer

Institut für Informatik, Technische Universität München

Abstract

Flex-flex pairs are an old asset of Isabelle that exists due to the undecidable nature of higher-order unification. While they surface to users of Isabelle very rarely, Isabelle's inference kernel has to go through some effort to accommodate for the existence of flex-flex pairs. This paper presents an experiment that suggests that flex-flex pairs are a feature that is hardly ever used by today's Isabelle users, and discusses some benefits that could be gained from eliminating flex-flex pairs from Isabelle.

1 Introduction

From Huet's seminal paper [3] it is well-known that higher-order unification for typed λ-calculus is undecidable. Many users of Isabelle are aware that Isabelle's answer to this problem is actually two-fold. In order to compute a unifier of two terms, the inference kernel first tries an algorithm for the easier problem of *pattern unification* [4]. If this algorithms fails to find a unifier, a more powerful algorithm, which is basically Huet's semi-decision procedure for higher-order unification [2], is used as a fallback. To ensure termination, this algorithm uses a maximal search bound, which can be freely configured by users of Isabelle.

An important property of Huet's algorithm [1] is that it divides terms into two types, *rigid* and *flexible*. A term is called flexible if its head symbol is a free variable (a *schematic* variable in Isabelle-speak), and it is called rigid if its head symbol is of any other type. Then, equations between terms take one out of three different shapes:

(Rigid-rigid) $\lambda\ x_1\ \dots\ x_m.\ f\ t_1\ \dots\ t_p \equiv \lambda\ x_1\ \dots\ x_n.\ g\ u_1\ \dots\ u_q$

(Flexible-rigid) $\lambda\ x_1\ \dots\ x_m.\ ?X\ t_1\ \dots\ t_p \equiv \lambda\ x_1\ \dots\ x_n.\ f\ u_1\ \dots\ u_q$

(Flexible-flexible) $\lambda\ x_1\ \dots\ x_m.\ ?X\ t_1\ \dots\ t_p \equiv \lambda\ x_1\ \dots\ x_n.\ ?Y\ u_1\ \dots\ u_q$

[1] A good presentation can be found in [1]

Huet's algorithms tries to resolve the first two types of equations until only equations of the last type are left. In this case it is easy to solve the remaining equations to find a unifier. For the equation from above this trivial unifier would look like

$$< ?X := \lambda\ x_1\ \ldots\ x_p.\ t,\ ?Y := \lambda\ x_1\ \ldots\ x_q.\ t >$$

for any term t of suitable type.

It is an old design decision of Isabelle not resolve flexible-flexible equations at the end of higher-order unification but instead to attach them to theorems in order to let the user decide for an apt instantiation or leave them to further resolution steps to resolve them. This leads to the existence of the infamous *flex-flex pairs* in Isabelle. Usually flex-flex pairs are of no concern to most users of Isabelle as they are resolved by most automatic proof tactics and final proof steps such as *by*. The most common sources for flex-flex pairs to surface on the top-level are manual rule combinations via the *THEN* and *OF* attributes (or ML combinators), and the manual application of rules (often in longer apply-style proof scripts). Given that most users can live happily without making explicit use of this old feature of Isabelle, one could wonder wether flex-flex pairs are still essential in what Isabelle has evolved into to today. This paper studies this question by presenting what would happen if flex-flex pairs were eliminated by resolving them after every invocation of higher-order unification.

The rest of this paper is structured as follows. Section 2 presents our method. Section 3 gives a detailed account of the changes that would have to be made to existing theories if flex-flex pairs were eliminated from Isabelle. Finally, Section 4 argues that removing flex-flex pairs would indeed be feasible, and outlines some potential merits that could be gained from freeing Isabelle from flex-flex pairs.

2 Method

At the heart of Isabelle's inference kernel lies the method `bicompose_aux` [2], which is the main participant in the process of resolving two theorems. The first step of `bicompose_aux` is to identify a set of *disagreement pairs*. This set consists of the parts of the theorems that need to be unified and the flex-flex pairs that are already attached to the theorems. For finding unifiers of such disagreeing terms, `bicompose_aux` makes use of the functionality of `Unify.unifiers` [3]. Given a list of of disagreement pairs, this function produces a sequence of pairs of a substitution and a list of new flex-flex pairs, which in combination form a unifier. In Isabelle, these substitutions consist of instantiations for schematic term and type variables.

[2] `src/Pure/thm.ML`

[3] `src/HOL/unify.ML`

The idea of our experiment is simply to turn this sequence of unifiers into one where all flex-flex pairs are resolved before making further use of the unifiers in `bicompose_aux`. For this purpose, the unification module already provides the method `Unify.smash_unifiers`, which first computes the same unifiers as `Unify.unifiers` and then *smashes* unifiers in the style of Huet described above. More precisely, flex-flex pairs are eliminated one-by-one, resolving pairs of the form

$$\lambda\ x_1\ \ldots\ x_m.\ ?X\ t_1\ \ldots\ t_p = \lambda\ x_1\ \ldots\ x_n.\ ?Y\ u_1\ \ldots\ u_q$$

with

$$<?X := \lambda\ x_1\ \ldots\ x_p.\ ?Z,\ ?Y := \lambda\ x_1\ \ldots\ x_q.\ ?Z>$$

for some fresh variable $?Z$.

The Isabelle source comments already state that unfortunately this unifier is sometimes less general than it would have to be. For instance, consider the following flex-flex pair:

$$\lambda\ x\ y.\ ?f\ x\ y = \lambda\ x\ y.\ ?g\ x\ y\ .$$

The implementation would eliminate the flex-flex pairs with an instantiation of the form

$$<?f := \lambda\ x\ y.\ ?a,\ ?g := \lambda\ x\ y.\ ?a>\ ,$$

while we could simply set $?f := ?g$. We have sometimes found this method too aggressive to produce the desired resolvents of two theorems and thus have also experimented with two more general variants of smashing flex-flex pairs. In the following, we will call them type two and three, while the already existing variant of Isabelle will be called type one. The two new types are:

(Type two) Pairs of the form

$$\lambda\ x_1\ \ldots\ x_n.\ ?X\ t_1\ \ldots\ t_p = \lambda\ x_1\ \ldots\ x_n.\ ?Y\ t_1\ \ldots\ t_p$$

are resolved by setting $?Y := ?X$.

(Type three) Pairs of the form

$$\lambda\ x_1\ \ldots\ x_m.\ ?X\ t_1\ \ldots\ t_p = \lambda\ x_1\ \ldots\ x_n.\ ?Y\ u_1\ \ldots\ u_q$$

are resolved with the substitution

$$<?X := \lambda\ x_1\ \ldots\ x_p.\ ?Z\ x_1\ \ldots\ x_k,\ ?Y := \lambda\ y_1\ \ldots\ y_q.\ ?Z\ y_1\ \ldots\ y_k>$$

for some fresh variable $?Z$ where the arguments to $?Z$ are the bound variables that already appear as both, a term t_i and a term u_j.

The source code for both types can be found in the appendix.

We introduced two configuration options, one to toggle smashing of flex-flex pairs, and the other one two switch between type one and type two smashing. Where necessary, we applied type three smashing on the user level. We then simply switched on type one smashing by default and tried to rebuild the Isabelle distribution and the Archive of Formal Proofs (AFP) [4] to see what would happen to existing proofs when flex-flex pairs were disabled. All findings are based on a development version of the AFP and the distribution from June 8, 2016.

3 Results

3.1 Smashing the Isabelle Distribution

In the distribution, problems in 40 files were discovered. The changes that had to be made to repair those proofs fall into five different categories:

- turning on type two smashing,
- applying type three smashing,
- trivial proof restructurings,
- substantial changes,
- and unresolved issues.

The first two type of problems do not need any further explanation. A total of ten instances of broken proofs required type two smashing, while only one instance required to be resolved by type three smashing. It is worthy to note that each of these ten instances occurred in apply-style proof scripts.

Trivial adjustments include one explicit instantiation of a variable in a theorem and one instance where a combined rule that would would usually carry a flex-flex pair could not be applied to the goal any more because smashing had produced a too specific instantiation. In the latter case the single rule application could be split in two by sequentially applying the two rules that were previously combined. A more frequent adjustment (five occurrences) that had to be made made was rewriting applications of *rule* into a combination of '$-$' and *rule* or alternatively an application of *rule-tac*. In Isar proof scripts, this phenomenon can appear when a fact of the form $P\ ?x$ is chained into an application of *rule* with a rule of the form $(\bigwedge\ x.\ P\ x) \Longrightarrow Q$. Due to the way *rule* handles these facts, the chained fact $P\ ?x$ is resolved against the rule first, sometimes resulting in a flex-flex pair. After smashing, the resulting theorem may not fit to be resolved against the

[4] `isa-afp.org`

goal anymore. Turning the chained fact into an assumption (i.e. $\bigwedge x.\ P\ x$) by applying '−' first then resolves the problem.

Most of the broken proofs were caused by a single tactic in the HOL formalization of non-standard analysis (18 files). A custom tactic *transfer* (not to be confused with *transfer* from the Transfer package) is used to implement a transfer principle. It first rewrites a goal into a standard format using simplification w.r.t. to a set of rewrite rules, next applies a special rule to start the transfer process, and then exhaustively applies a number of introduction rules. The problem that was produced by smashing flex-flex pairs could be tracked back to a single one of these introduction rules (`transfer_Ifun`):

$$\frac{?f \equiv \textit{star-n}\ ?F \qquad ?x \equiv \textit{star-n}\ ?X}{?f \star ?x \equiv \textit{star-n}\ (\lambda n.\ ?F\ n\ (?X\ n))}$$

The cause of the issue are the nested schematic variables on the right-hand side of the conclusion. Usually the rule is applied to goals of the form

$$\ldots \Longrightarrow \bigwedge X.\ \textit{star-of}\ f \star \textit{star-of}\ g \star \textit{star-n}\ X \equiv \textit{star-n}\ (\lambda\ n.\ ?X\ (X\ n))$$

where the left-hand side can contain arbitrary nesting of *star-of*, *star-n* and $\star$, and more nested bound and schematic variables can appear on the right-hand side of the meta equality. To prove this type of goal, the definition *star-of* has to be used additionally:

$$\textit{star-of}\ ?x \equiv \textit{star-n}\ (\lambda n.\ ?x)\ .$$

We resolved the issue by adding a simple modification to *transfer* that applies `transfer_Ifun` more deterministically. The modification removes `transfer_Ifun` from the set of introduction rules and only tries to apply it when no other rules can be applied anymore. In this case, simple recursion over the structure of the left-hand side of the goal is used to find to find the right instantiation of the rule. The implementation of this reasoning step can be found in the appendix. The changes means that the formalization of non-standard analysis moves from being the largest producer of flex-flex pairs to produce zero flex-flex pairs. In addition, the change clarifies the reasoning process of *transfer*.

The two rather old logic sessions `CTT` and `Sequents` each exhibited two problems that we left unresolved for now. In `Sequents` they both arise in the proof of a schematic goal statement where a combined rule that already carries a complex flex-flex pair is applied. Similarly, both problems in `CTT` appear in proofs of schematic goal statements. One of them is caused by the custom *rew* tactic of this session. These proofs are very hard to follow, and the correct instantiation of the rules is not obvious. Our conjecture is that in a modern apply-style proof script or Isar proof of the corresponding lemmas, these problems would disappear due to a more explicit reasoning style.

More significant unresolved issues are caused by the BNF package and *metis*. The distribution contains nine mutually recursive *datatype* definitions and two definitions via *primcorec* that only go through when smashing is disabled. A single invocation of *metis* with the *lifted* attribute in `src/HOL/Transcedental.thy` also breaks for at least type one and two smashing. As the author is not an expert for either one of these packages, these problems have not been investigated further.

3.2 Smashing the Archive of Formal Proofs

Fortunately, fixing the whole AFP for smashing did not require much more work than the Isabelle distribution: only 39 files needed to be changed. This, however, does not include the two 'slow' sessions `ConcurrentGC` and `Flyspeck-Tame`. Possible repercussions on these have not been analyzed yet.

The required changes did not uncover much different problems than for the distribution. Type two smashing had to be used 24 times, while type three smashing could remedy four more problems. On 14 instances applications of *rule* had to be preceded by an application of '−'. Proofs had to be slightly restructured due to rule composition three times. Some other restructurings required a bit more creativity: three times an automatic proof method had to be swapped with another one, and once a metis proof was replaced with a different one found by sledgehammer. A short apply-script could be replaced by a simple one-line proof at four instances. Again, most of these restructurings were made to apply-style proof scripts, not Isar proof scripts.

To the category of unresolved issues the AFP added four more applications of *metis* and nine datatype definitions. Interestingly, the many AFP sessions that are concerned with program verification and program synthesis, and include long apply-style proofs, complex tactics, and schematic goal statements, were not harmed significantly by the change. Most of the fixes with type two and three smashing fall into this category. This shows that, compared to problems discovered in the old logic sessions of the distribution, modern well-behaved tactics and apply-style proofs do not rely on flex-flex pairs to get their job done.

3.3 Summary

The following table summarizes the changes that had to be made to the Isabelle distribution and the AFP.

Issue/Modification	Description	Number of Occurrences		
		Distribution	AFP	Total
Type 2 Smashing		9	24	33
Type 3 Smashing		1	4	5
Trivial Restructuring	Added ‘-’-proof step	5	14	19
	Other	2	16	18
Unresolved	BNF package	11	9	20
	`metis`	1	4	5
	Other	4	0	4
Total		33	71	104

4 Discussion

Our findings show that today's Isabelle users do not explicitly rely on flex-flex pairs as a feature of Isabelle in their applications. What is more, they do not even seem to do so implicitly. The only real occurrence of this seems to be the *transfer* tactic in the HOL formalization of non-standard analysis. However, we have demonstrated that the tactic can be made slightly more well-behaved, while at the same time completely eliminating the need for flex-flex pairs. The only places where flex-flex pairs seem to serve a real purpose are the *CTT* and *Sequents* logic sessions from nearly ancient times of Isabelle. And yet, there are only four of these places! Admittedly, this bright picture is slightly clouded by the fact that the BNF package and *metis* sometimes break when introducing smashing. However, as exemplified for the *transfer* tactic, resolving these issues might even be a chance to clarify a slightly odd situation in the codebase of these packages. In particular, it is somewhat surprising that removing a feature of higher-order unification breaks the first-order reasoning of *metis*.

Summarizing, we believe that if the issues with the BNF package and *metis* could be resolved without much trouble, there would be a strong point for removing flex-flex pairs entirely. In this case, one would have to decide for the mode of smashing that should be used to eliminate flex-flex pairs produced by higher-order unification. Given that there were 33 occurrences where type two smashing had to be used and given that some of these included applications of custom proof tactics, our experiments seem to indicate that at least type two smashing should be used to not limit current users. Moreover, the ML implementation of type two smashing is hardly more complicated than the existing implementation of type one smashing. For type three smashing the situation is less clear. Only five potential use cases in the current Isabelle universe possibly do not make a sufficiently strong point for adding a trusted piece of code to Isabelle's inference kernel that is more complicated than for the other variants of smashing.

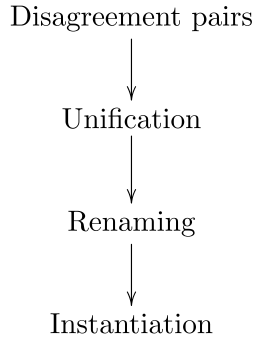

Figure 1: Structure of `bicompose_aux`.

What could be gained from removing flex-flex pairs from Isabelle? First and foremost, this would keep a feature that might be slightly dubious and disturbing from surfacing to novice users. Moreover, it might even force users to sometimes write more well-behaved proofs and tactics in a positive way. However, our main prospect would tend to the direction of clarifying Isabelle's inference kernel with respect to higher-order unification. On a high level, the current functionality of `bicompose_aux` could be described as being divided into four phases: identification of disagreement pairs (from the rules that are ought to be resolved with each other), unification, renaming, and instantiation. The situation is depicted in Figure 1. Currently, higher-order unification is inherently used by `bicompose_aux`, and thus a part of Isabelle's trusted code base. As this code is highly complex and was previously found to have issues, the trustworthiness of the inference kernel could be improved by moving higher-order unification out of the trusted code base. A potential way would be to add a *certifier* at the interface of `bicompose_aux` and the unifier. This certifier would check that the substitution environment which is computed by the unifier really unifies the disagreement pairs under question. The revised schema could be depicted as in Figure 2 (the dotted line separates trusted from untrusted code). Without flex-flex pairs, this check would simply amount to applying the given substitutions and checking the resulting terms for $\alpha\beta\eta$-equivalence. Results of pattern-unification could optionally go unchecked for performance reasons.

Another route to go down would be to restructure `bicompose_aux` more radically. Only two of its work steps seem to necessarily be part of the trusted code base (in one or the other form): renaming and instantiation. Instead of combining all these functionalities in one elementary inference step, the resolution aspect of `bicompose_aux` could be reduced to a more primitive inference step that would only allow to resolve rules that are already correctly instantiated. After eliminating flex-flex pairs, this would again simply amount to checking for $\alpha\beta\eta$-equivalence. The restructured version of resolution could be depicted as in Figure 3. Renaming and instantiation

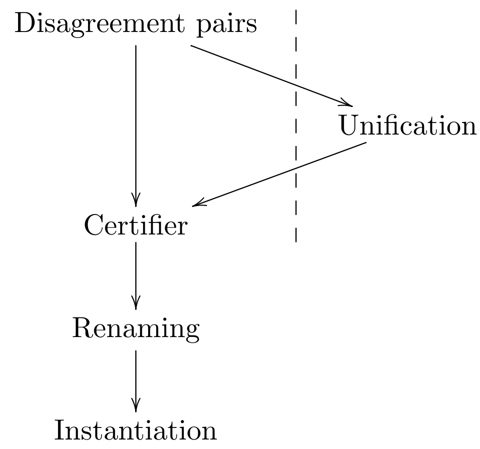

Figure 2: Structure when using a certifier.

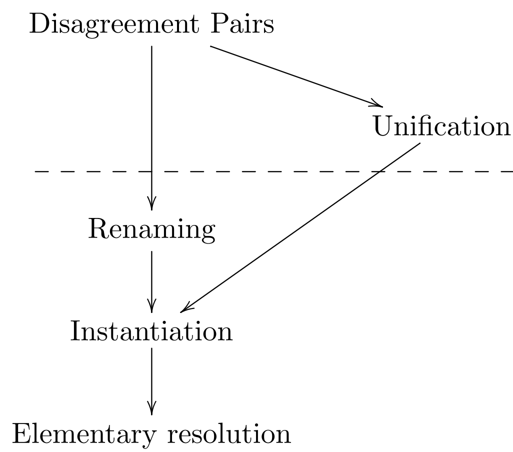

Figure 3: Structure when breaking up `bicompose_aux`.

are already available as exported functionalities of the kernel, however additional primitives for these steps might have to be introduced for performance reasons. Our initial experimentation has shown that a tricky part of this endeavor would be to apply renaming in a way to retain full compatibility with the present naming scheme (which is necessary to avoid work on fixing proofs that rely on names produced by the inference kernel).

In addition to minimizing the trusted code base, both approaches would share the positive traits that one could freely add more complicated types of smashing and even allow the user to configure resolution steps with a unification procedure of their liking (e.g., as a context option). The downside would be a nearly unavoidable performance degradation as checking steps would be introduced and optimizations for lifting would be lost at least in the case of the second approach. The challenge is to find a good tradeoff between performance and size of the trusted code base.

References

[1] G. Dowek. Higher-order unification and matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 1009 – 1062. North-Holland, Amsterdam, 2001.

[2] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27 – 57, 1975.

[3] G. P. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257 – 267, 1973.

[4] T. Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.

Appendix

Listing 1: ML implementation of type 2 smashing with type 1 as fallback

```
fun smash_flexflex2 (t, u) env : Envir.env =
  let
    fun strip_eq (t1 $ t2, u1 $ u2) flag =
      strip_eq (t1, u1) (t2 = u2 andalso flag)
      | strip_eq (Var lhs, Var rhs) flag = (lhs, rhs, flag)
      | strip_eq (t, u) _ = (
        case head_of t of
          Var (v, T) => (
          case head_of u of
            Var (u, U) => ((v, T), (u, U), false)
          | _ => raise CANTUNIFY)
        | _ => raise CANTUNIFY)
    val (vT as (v, T), wU as (_, U), flag) =
      strip_eq (strip_abs_body (Envir.norm_term env t),
                strip_abs_body (Envir.norm_term env u)) true;
    fun triv () =
      let
        val (env', var) =
          Envir.genvar (#1 v) (env, Envir.body_type env T);
        val env'' =
          Envir.vupdate (wU, type_abs (env', U, var)) env';
      in
        if vT = wU then env''
        else Envir.vupdate (vT, type_abs (env', T, var)) env''
      end;
    fun non_triv () =
      Envir.vupdate (wU, Var vT) env
  in
    if flag then non_triv () else triv ()
  end;
```

Listing 2: ML implementation of type 3 smashing on the user level

```
fun enumerate xs = fold (fn x =>
  fn (i, xs) => (i + 1, (x, i) :: xs)) xs (0, []) |> snd
fun dummy_abs _ [] t = t
  | dummy_abs n (T :: Ts) t =
    Abs ("x" ^ Int.toString n, T, dummy_abs (n + 1) Ts t)
fun common_prefix Ts
    (t1 as Abs (_, T, t)) (u1 as Abs (_, U, u)) =
  if U = T then common_prefix (T :: Ts) t u else ([], t1, u1)
```

```
  | common_prefix Ts t u = (Ts, t, u);
fun dest_app acc (t $ u) = dest_app (u :: acc) t
  | dest_app acc t = (t, acc);
fun add_bound (Bound i, n) bs = (i, n) :: bs
  | add_bound _ bs = bs;
fun smash_flexflex3 ctxt thm (t, u) =
  let
    val idx = Thm.maxidx_of thm + 1;
    val (Ts, t1, _) = common_prefix [] t u;
    val (tas, t2) = Term.strip_abs t;
    val (uas, u2) = Term.strip_abs u;
    val (tx as Var (_, T1), ts) = Term.strip_comb t2;
    val (ux as Var (_, U1), us) = Term.strip_comb u2;
    val Ts1 = Term.binder_types T1;
    val Us1 = Term.binder_types U1;
    val T = Term.fastype_of1 (Ts, t1);
    val tshift = length tas - length Ts;
    val ushift = length uas - length Ts;
    val tbs = fold add_bound (enumerate (rev ts)) []
      |> map (apfst (fn i => i - tshift));
    val ubs = fold add_bound (enumerate (rev us)) []
      |> map (apfst (fn i => i - ushift));
    val bounds = inter (op =) (map fst tbs) (map fst ubs)
      |> distinct (op =);
    val T' = map (nth Ts) bounds ---> T;
    val v = Var (("dummy_var", idx), T');
    val tbs' = map (fn i =>
         find_first (fn (j, _) => i = j) tbs
       |> the |> snd |> Bound)
      bounds;
    val t' = list_comb (v, tbs') |> dummy_abs 0 Ts1;
    val ubs' = map (fn i =>
        find_first (fn (j, _) => i = j) ubs
          |> the |> snd |> Bound)
      bounds;
    val u' = list_comb (v, ubs') |> dummy_abs 0 Us1;
    val subst = [(Term.dest_Var tx, Thm.cterm_of ctxt t'),
      (Term.dest_Var ux, Thm.cterm_of ctxt u')];
  in
    instantiate_normalize ([], subst) thm
  end;
```

Listing 3: Resolving with the right instantiation of `transfer_Ifun`

```
local exception MATCH
```

```
in
fun transfer_star_tac ctxt =
  let
    fun thm_of (Const (@{const_name Ifun}, _) $ t $ u) =
      @{thm transfer_Ifun} OF [thm_of t, thm_of u]
    | thm_of (Const (@{const_name star_of}, _) $ _) =
      @{thm star_of_def}
    | thm_of (Const (@{const_name star_n}, _) $ _) =
      @{thm Pure.reflexive}
    | thm_of _ = raise MATCH;

    fun thm_of_goal (Const (@{const_name Pure.eq}, _) $ t $
        (Const (@{const_name star_n}, _) $ _)) =
      thm_of t
    | thm_of_goal _ = raise MATCH;
  in
    SUBGOAL (fn (t, i) =>
      resolve_tac ctxt [thm_of_goal
        (strip_all_body t |> Logic.strip_imp_concl)] i
      handle MATCH => no_tac)
  end;
end;
```