MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

A General Technique for Automatically Optimizing Programs Through the Use of Proof Plans

Peter Madden and Ian Green

MPI–I–94–239 August 1994

Im Stadtwald
66123 Saarbrücken
Germany

A General Technique for Automatically Optimizing Programs Through the Use of Proof Plans

Peter Madden and Ian Green

MPI–I–94–239 August 1994

Authors' Addresses

Peter Madden
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany
Tel: + 49 681 302 5434
Email: madden@mpi-sb.mpg.de

Ian Green
Department of AI
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland (UK)
Tel: + 44 31 650 2722
Email: img@uk.ac.ed.aisb

Publication Notes

To appear in the Proceedings of the Second International Workshop/Conference on Artificial Intelligence And Symbolic Mathematical Computing (AISMC-2)

Acknowledgements

Some of the research reported in this paper was carried out when the author was an SERC Post-Doctoral Research Fellow within the *Mathematical Reasoning Group* at the Department of Artificial Intelligence, Edinburgh University. Acknowledgements are due to Jane Hesketh, Alan Smaill and, in particular, to Alan Bundy.

1 Synopsis

In this paper we investigate how *proof plans* – formal patterns of reasoning for theorem proving – can be used for controlling the synthesis of *efficient* functional programs from standard sets of equational definitions. By exploiting meta-level control strategies, a general framework for *automatically* synthesizing efficient programs has been developed. A key meta-level strategy is called *middle-out reasoning*, henceforth MOR, which involves the controlled use of higher-order meta-variables at the meta-level planning phase. This allows the planning to proceed even though certain object-level objects are (partially) unknown. Subsequent planning provides the necessary information which, together with the original definitional equations, will allow us to instantiate such meta-variables through higher-order unification (HOU) procedures. MOR allows for the circumvention of eureka steps during the optimization process concerning, amongst other things, the identification of recursive data-types, and unknown constraint functions. Such steps have typically required user-intervention in more traditional ("pure") transformational systems such as unfold/fold [BD77]. The control provided by proof planning allows us to view such syntheses as verification together with MOR.

The proof planning approach to controlling the synthesis of efficient programs was originally investigated within the context of synthesizing tail-recursive programs from naive definitions by using a tail-recursive generalization strategy [HBS92]. In this paper we present a *general* framework for automatically synthesizing efficient programs through the use of proof planning and MOR. We illustrate the methodology by describing a novel form of generalization strategy which, together with an induction strategy and MOR, automatically affects *constraint-based* optimizations: the *constraint-based generalization* proof plan is used for generating families of efficient programs from definitions which include expensive expressions.

In previous generalization proof plans, such as that described in [HBS92], MOR has been limited to introducing (higher-order) meta-variables into goal statements according to the pre-conditions of the generalization proof plan. We refere to this kind of MOR as *generalization*-MOR, or simply gen-MOR. We significantly extend the mechanism by which MOR operates by allowing for the use of higher-order meta-variables in rewrite rules *in addition* to those introduced via the proof plan application. Such meta-variables are introduced via the exploitation of higher-order recursive definition schemas. These can be viewed as higher-order schematic rule templates. This significantly increases the scope for delaying proof commitments until subsequent theorem proving provides the requisite information to identify the relevant data structures. We shall refer to this usage of MOR as template MOR

Different characterizations of proofs can be formalized as proof plan pre-conditions that subsequently effect the kind of optimization exhibited by the synthesized functions. In particular, the way in which meta-variables are introduced, via gen-MOR, into the proof of the goal statement(s) specifying the program being synthesized. Constraint-based optimizations are, for example, characterized differently from tail-recursive optimizations. This basically accounts for the new form of generalization systemized in the constraint-based generalization. However, so as to illustrate many features of the general framework we shall, in this paper, choose a running example which consists of synthesizing an efficient program, from standard equational definitions, which is *both* tail-recursive *and* constraint-based. The example will illustrate both the usage of gen-MOR and the new template MOR

We believe that a large class of otherwise diverse, and often ad hoc, transformation strategies can be encompassed within this uniform proof plan framework. We show how the proof planning framework provides the necessary meta-level control over HOU and proof structure. Furthermore, the underlying logic we use guarantees the total correctness of the synthesized function with respect to the specification.

Toward the end of this paper we compare the proof planning approach to synthesizing efficient programs with existing optimization strategies and discuss its advantages.

Contents

In the remainder of this section we explain what precisely constraint-based optimization is with the introduction of our running example. In §2 we briefly describe the proofs as programs paradigm and illustrate, in §2.0.2, an (interactive) synthesis of the example in §1.1. In §2.0.1 and §2.1 we describe, respectively, the object level OYSTER proof refinement system and the meta-level CLAM proof planner. We provide an outline of the planning strategies employed and present the notation used to illustrate the rewriting process. In §3 we describe the *general* framework for optimization by proof planning. §4 addresses one kind of optimization encapsulated by the general framework described in §3: the use of proof plans for the purposes of constraint-based optimization. We provide pre- and post-conditions for the application of a constraint-based transformation proof plan, and, in §4.2 we describe the use of higher-order schematic rule templates. In §4.3 we revisit the example of §1.1 and show how, using proof planning together with MOR, the optimization process is automated. In §5 we address the benefits of our approach as compared with standard transformational approaches such as unfold/fold.

1.1 An Example Optimization

Hesketh et al. considers the automatic synthesis of *tail-recursive* programs from inefficient non-tail-recursive programs using standard sets of equational definitions [HBS92]. However, tail-recursive programs may, in turn, present scope for further optimization. Consider the following two *tail-recursive* definitions of procedures for simultaneously producing both the sum and reverse of the input list:[1]

$$rev\_sm(nil, w) = (sum(w), w); \quad (1)$$
$$rev\_sm(hd :: tl, w) = rev\_sm(tl, hd :: w) \quad (2)$$

and where:

$$sum(nil) = 0; \quad (3)$$
$$sum(a :: x) = a + sum(x) \quad (4)$$

$$rev\_sm_{C\_\perp}(l, x) = rev\_sm_C(l, x, 0) \quad (5)$$
$$rev\_sm_C(nil, w, sum(w)) = (sum(w), w); \quad (6)$$
$$rev\_sm_C(hd :: tl, w, sum(w)) = rev\_sm(tl, hd :: w, hd + sum(w)). \quad (7)$$

The *rev_sm* procedure has an expensive expression, $sum(w)$, in its base-case equation (1). The program's inefficiency stems from the double traversal of its input, which builds up a reversed list in the second parameter, w, and then finally performs a summation of this reversed list. The inefficiency may be removed by introducing the new constrained tail-recursive function definition, rev_sm_C, which uses only a single traversal of its input list in order to obtain both the summation together with its reversed list.

The removal of such expensive expressions has been investigated within the context of program transformation and is called *constraint-based transformation* [Chi90]

[1] Tail recursive definitions have the feature that recursive calls occur as the outermost function of the procedure body, and an *accumulator*, w in the examples, is used to construct the output as the recursion is entered.

(or sometimes *finite differencing* [RP82]). In general, it involves the replacement of expensive expressions in program loops, or recursion, by equivalent expressions which are incrementally maintained. As indicated by the above example, this is done through the introduction of new parameters – *constraint parameters* – whose values are specified using constraints (thus, regarding rev_sm_C, the constraint parameter is $sum(w)$). We shall call functions such as *rev_sm unconstrained*, and their optimal counterparts, such as rev_sm_C, *constrained*.

[Chi90] outlines how functional programs containing expensive expressions can be optimized using standard *unfold/fold* type rewrites to obtain constrained function definitions such as that for rev_sm_C. The unfold/fold strategy was pioneered by Darlington, and its most influential implementation has been within the NPL program transformation system [BD77, Dar89]. Unfold/fold transformation typically involves the sequential application of rewrites which use definitions (specifically using instantiated definitions to replace terms) and known properties of functions in order to derived a target program which is independent of the source definition. However, the identification of the new constraint parameters, as $sum(w)$ in the rev_sm_C example, constitute *eureka steps*, thus presenting obstacles for providing a general automatic procedure for constraint-base optimization. Regarding program optimization in general, such *eureka steps* correspond to the problems of identifying *explicit definitions* for target programs: that is, new definitions where the target program is defined explicitly in terms of the source. Indeed, within the context of unfold/fold transformations, providing explicit definitions is the key to the optimization process: by subsequently folding the explicit definition (or derivations thereof) with the original source equations recursion is introduced into the target program. Further difficulties include the search involved with identifying and applying rewrites to explicit definitions in order to derive the recursive target definitions. For example, unfold/fold transformations are motivated by the desire to find a successful fold. This involves extensive search and the somewhat arbitrary application of laws thus presents difficulties regarding automation. We discuss these difficulties in more depth in §5.

In this paper we consider a general technique for circumventing the aforementioned eureka steps, and for reducing the search control problems in the rewriting process, by exploiting proof plans.

2 Proofs as Programs

Constructive logic allows us to correlate computation with logical inference. This is because proofs of propositions in such a logic require us to construct objects, such as functions and sets, in a similar way that programs require that actual objects are constructed in the course of computing a procedure.[2] Historically, this duality is accounted for by the *Curry-Howard isomorphism* which draws a duality between the inference rules and the functional terms of the λ-calculus [CF58, How80].

Such considerations allow us to correlate each proof of a proposition with a specific λ-term, λ-terms with programs, and the proposition with a specification of the program. Hence the task of generating a program is treated as the task of proving a theorem: by performing a proof of a formal specification expressed in constructive logic, stating the *input-output* conditions of the desired program, an algorithm can be routinely extracted from the proof. A program specification can

[2] Thus we can not, for example, compute (or constructively prove) that there are an infinity of prime numbers by assuming the converse and deriving a contradiction, rather we must produce a program that computes them (or a proof that we can always construct another one greater than the ones known so far).

be schematically represented thus:

$$\forall inputs,\ \exists output.\ spec(inputs, output)$$

Proofs of such specifications must establish (constructively) how, for any input vector, an output can be constructed that satisfies the specification.[3] Thus any synthesized program is guaranteed correct with respect to the specification. Different constructive proofs of the same proposition correspond to different ways of computing that output. By placing certain restrictions on the nature of a synthesis proof we are able to control the efficiency of the target procedure. Thus by controlling the form of the proof we can control the efficiency with which the constructed program computes the specified goal. Here in lies the key to synthesizing efficient programs. For example, we can synthesize constrained functions from unconstrained ones by placing the restriction that the new constraint parameter is some function on the non-recursive (non-inductive) parameter (we illustrate this in §2.0.2, and in more detail in §4.1). We can also guarantee that a synthesized program is tail recursive by ensuring that the witnesses of the two existential quantifiers, one in the induction hypothesis and one in the induction conclusion, are identical [Wai89].[4] By making these witnesses identical we ensure that the function does not change value as the recursion is exited. Alternatively, we can use special schematic rules to affect the nature of the recursion exhibited by the program under construction. In §4.2 we illustrate how tail-recursive behaviour can be ensured through the application of such rules.

2.0.1 The OYSTER System

The OYSTER system is an implementation of a constructive type theory which is based on Martin-Löf type theory, [ML79].[5] OYSTER is written in Quintus Prolog, and run at the Prolog prompt level, so it is controlled by using Prolog predicates as commands. Proof tactics can be built as Prolog programs, incorporating OYSTER commands. The language uniformity of the logic programming environment allows for the construction of *meta-theorems* which express more general principles, concerning the object level theorem proving. So, for example, we are able to construct *tactics* which combine the object-level rules of the system in various ways and apply them to proof (sub)goals.

At any stage during the development of a proof it is possible to access the *extract term* of the proof constructed so far. Each construct in the extract term corresponds to a proof construct. As such, the extract term reflects the computational content of the proof of the theorem. The extract programs consist of λ-calculus function terms, $\lambda(x, f_x)$ where f is the computed function and f_x the output when f is applied to input x.

For the purposes of illustrating our methodology we do not need to make the type information contained in the proofs explicit. Indeed, it is adequate and aids clarity to present, in this paper, our proofs in a classical framework.

2.0.2 Example: Synthesis of Constraint Function rev_sm_C

To illustrate the synthesis process we shall outline the synthesis of rev_sm_C using the definition of the unconstrained function *rev_sm*. We indicate those proof steps

[3] Thus constructive logic *excludes* pure existence proofs where the existence of *output* is proved but not identified.

[4] A witness constitutes an instantiation of an existential quantifier thus providing evidence of the existence asserted.

[5] OYSTER is the Edinburgh Prolog implementation of NuPRL; version "nu" of the *Proof Refinement Logic* system originally developed at Cornell [Horn 88, Constable *et al* 86].

which, regarding traditional program transformation re-writing systems, correspond to *eureka* steps.

A key feature of the proof plan approach to automatically synthesizing efficient programs is to use the inefficient program definition to specify the required procedure. We specify the output for the constrained function, rev_sm_C, in terms of the unconstrained rev_sm thus:

$$\forall x, \forall w, \exists z.\ z \quad = \quad rev\_sm(x, w) \tag{8}$$

We then introduce a new sub-goal, providing an explicit definition for rev_sm_C which includes a constraint parameter instantiated to $sum(w)$:

$$\forall x, \forall w. rev\_sm(x, w) \quad = \quad rev\_sm_C(x, w, sum(w)) \tag{9}$$

The identification of $sum(w)$ as the constraint parameter (required for the full identification of the explicit definition) is the first eureka step. Note that the proof satisfies the *constraint-based restriction* that the constraint parameter is some function on the non-recursive (non-inductive) parameter.

Also Note that since we have provided the identity of the constraint parameter that the proof will in fact be a verification proof. In §4.3 we illustrate how (higher-order) meta-variables are used to partially identify such constraint parameters and there by avoid such eurekas. That is, synthesis without eureka steps can be affected through higher-order verification proofs.

In §4.3 we show how such eureka steps can be automated through the use of MOR. In general terms, MOR allows us to delay choice commitments by introducing (higher-order) meta-variables at the meta-level application of rules of inference. Subsequent planning provides the requisite information to instantiate the meta-variables by (higher-order) unification. Thus, the main difference between verification and synthesis proofs is precisely the identification of those structures for which MOR is used. This idea is captured by our slogan that *synthesis is equivalent to verification plus meta-variables.*

The introduction of the new goal (9) also leaves us with the trivial proof obligation that the new goal entails the original one (i.e. (9) $\vdash$ (8)):

$$\forall x, \forall w. rev\_sm(x, w) \ = \ rev\_sm_C(x, w, sum(w)) \ \vdash \ \forall x, \forall w, \exists z.\ z \ = \ rev\_sm(x, w)$$

To prove (9) standard stepwise induction on x is used:

The Base Case: The base case is as follows:

$$\vdash \forall w. rev\_sm(nil, w) \quad = \quad rev\_sm_C(nil, w, sum(w)) \tag{10}$$

Using the definition of rev_sm, the left hand side of (10) rewrites to $(sum(w), w)$:

$$\vdash (sum(w), w) \quad = \quad rev\_sm_C(nil, w, sum(w))$$

The Step Case: The step case of the induction is:

$$\begin{aligned} \forall x, \forall w. rev\_sm(tl, w) \quad &= \quad rev\_sm_C(tl, w, sum(w)) \ \vdash \\ \forall x, \forall w. rev\_sm(hd :: tl, w) \quad &= \quad rev\_sm_C(hd :: tl, w, sum(w)) \end{aligned} \tag{11}$$

We can use the definition of rev_sm to rewrite the left hand side of (11) to $rev\_sm(tl, hd :: w)$. However the re-writing process is blocked on the right-hand side: the available recursive definitions provide no suitable re-writes to unfold the rev_sm_C term any further. Hence to arrive at the following equation:

$$\ldots \vdash \forall x, \forall w. rev\_sm(tl, hd :: w) \quad = \quad rev\_sm_C(tl, hd :: w, hd + sum(w))$$

a second eureka step is required to allow for the re-writing of $rev\_sm_C(hd :: tl, w, sum(w))$ to $rev\_sm_C(tl, hd :: w, hd + sum(w))$. In §4 we illustrate how a further new form of MOR allows us to avoid the eureka step.

The step case is completed by stripping of the universal quantifiers and instantiating the w in the induction hypothesis to $hd :: w$ in the induction conclusion (reducing the induction step to true since both hypothesis and conclusion are identical). Analyses of the base and step cases of the proof provide the base and recursive branch for the rev_sm_C procedure. In the next section we discuss the program extraction process.

2.1 Proof Plans – Automating the Proof Process

The induction strategy, together with other commonly used proof tactics, has been systematized in a metalogic in the automatic plan formation program CLAM [BvHHS90]. This consists of formal *proof plans* where each tactic is specified by a *method* which includes pre- and post-conditions. By using *meta-level reasoning*, CLAM executes the individual proof plans to obtain a combination of tactics customized to the particular theorem at hand. Execution of this tactic combination, at the object level, will then produce a proof of that theorem.

A key strategy of the CLAM proof planner is *rippling*. Our explanation of rippling, and the corresponding notation, will be necessarily simplified. For a fully comprehensive account the reader should consult [BSvH+93]. In an inductive proof, the goal of the rippling proof plan is to reduce the induction step case to terms which can be unified with those in the induction hypothesis. This unification is called *fertilization*, and is facilitated by the fact that the induction conclusion is structurally very similar to the induction hypothesis except for those function symbols which surround the induction variable in the conclusion. These points of difference are called *wave-fronts*. Thus, the remainder of the induction conclusion – the *skeleton* – is an exact copy of the hypothesis. Wave fronts consist of expressions with holes – *wave holes* – in them corresponding to sub-terms in the skeleton. Wave-fronts are indicated by placing them in boxes, and the wave-holes are underlined, e.g.

$$\begin{aligned} & \forall x, \forall w.\, rev\_sm(tl, w) &=& \; rev\_sm_C(tl, w, sum(w)) \\ \vdash\; & \forall x, \forall w.\, rev\_sm(\boxed{hd :: \underline{tl}}^{\uparrow}, \lfloor w \rfloor) &=& \; rev\_sm_C(\boxed{hd :: \underline{tl}}^{\uparrow}, \lfloor w \rfloor, sum(\lfloor w \rfloor)) \end{aligned}$$

To understand the additional notation, the arrows and terms surrounded by $\lfloor\ \rfloor$, we must explain the function of the structural rewrite rules, or *wave-rules*. Rippling applies wave-rules so as to remove the difference (wave-fronts) from the conclusion, thus leaving behind the skeleton and allowing fertilization to take place. For the purposes of this paper we need to identify three kinds of wave-rule distinguished by the direction in which they move wave-fronts in the conclusion. Wave-fronts may be moved outwards, *rippling-out*, such that they surround the entire induction conclusion thus allowing a match between everything in the wave-front with the induction hypothesis. Wave-rules for rippling-out are called *longitudinal* wave-rules and an upward arrow signals the outward direction of movement. Examples of longitudinal wave rules are:

$$\boxed{s(\underline{U})}^{\uparrow} + V \;\Rightarrow\; \boxed{s(\underline{U + V})}^{\uparrow} \tag{12}$$

$$sum(\boxed{Hd :: \underline{Tl}}^{\uparrow}) \;\Rightarrow\; \boxed{Hd + \underline{sum(Tl)}}^{\uparrow} \tag{13}$$

$$U + (\boxed{\underline{V} + W}^{\uparrow}) \;\Rightarrow\; \boxed{(\underline{U + V}) + W}^{\uparrow} \tag{14}$$

Throughout this paper upper-case variables denote *meta-variables* such that the above rules are best understood as *rule schemata*. Note that we include (14) to

show that wave rules need not only be formed from the step cases of inductive definitions ((14) is formed from the associative law of +).[6] Wave-rules may also move wave-fronts sideways, *rippling sideways*, such that they surround non-induction universal quantified variables (such as accumulators). The sub-terms which sideways rippled wave-fronts surround are called *sinks* and are demarcated by ⌊ ⌋. This allows the wave fronts, and the universally quantified variable that they surround, to be identified with the corresponding universally quantified variable in the hypothesis. Thus again fertilization can take place. Such a wave-rule is called a *transverse* wave-rule, e.g.

$$rev\_sm(\boxed{Hd :: \underline{Tl}}^{\uparrow}, W) \Rightarrow rev\_sm(Tl, \boxed{Hd :: \underline{W}}^{\downarrow}) \quad (15)$$

$$\boxed{s(\underline{X})}^{\uparrow} + Y \Rightarrow X + \boxed{s(\underline{Y})}^{\downarrow}, \quad (16)$$

Rippling into sinks typically involves an application of a longitudinal wave-rule followed by a transverse wave rule. It may also, however, require *rippling-in*: a *reverse* application of a longitudinal wave-rule. A downward arrow signals this inward direction of movement. In §4.3 we provide a worked example that illustrates both kinds of wave rule, and all three directions of rippling.

Rippling has numerous desirable properties. A high degree of control is achieved for applying the rewrites since the wave-fronts in the rule schemas must correspond to those in the instance. This leads to a very low search branching rate. Rippling is guaranteed to terminate since wave-front movement is always propagated in a desired direction toward some end state (a formal proof of this property is presented in [BSvH+93]).

Other strategies formalized in proof plans, in addition to induction, rippling, MOR and fertilization, include symbolic evaluation and tautology.

3 General Technique for Optimization by Proof Plans

Constraint-based optimization is representative of only one of the kinds of optimization possible by using our general technique. The synthesis of tail-recursive programs from naive definitional equations has also been implemented as a *tail-recursive* proof plan [HBS92]. Other kinds of optimization that we are now investigating include deforestation transformations, fusion transformations and tupling transformations [Wad88, Chi90, Pet84]. In this section we describe the *general* technique for controlling the syntheses of efficient programs from the definitions of inefficient programs. The technique encapsulates all the aforementioned kinds of optimizations. We intend to continue expanding this range of optimizations.

Proof plans are used to control the (automatic) synthesis of functional programs, specified in a standard equational form, $\mathcal{E}$, by using the proofs as programs principle. The goal is that the program extracted from a constructive proof of the specification is an optimization of that defined solely by $\mathcal{E}$. Thus the theorem proving process is a form of program optimization allowing for the construction of an efficient, *target*, program from the definition of an inefficient, *source*, program.

The proof planning approach to optimization is depicted by fig.1 where we show the general form of the inductive generalization proof. The strategy can involve four main steps. Firstly, a target program specification, $\mathcal{S}$, is formed from the source program's equational definitions $\mathcal{E}$. $\mathcal{S}$ is then set up as the conjecture to prove.

[6] Indeed, rippling is not restricted to being employed solely within *inductive* proof plans. Other forms of mathematical proof may also be controlled using the rippling technique [WNB92].

Secondly, the technique involves *sequencing* into the proof of $\mathcal{S}$ a new sub-goal, $\mathcal{G}$. The sub-goal $\mathcal{G}$ is produced as an output of the constraint-based generalization proof-plan. $\mathcal{G}$ is partially identified by the use of higher-order (HO) meta-variables, thus initiating the gen-MOR process. The application of the sequencing rule produces two subgoals: the first being the original goal $\mathcal{S}$ with $\mathcal{G}$ as an additional hypothesis (the so-called *justification* goal), and the second being $\mathcal{G}$ itself.[7] The inductive proof, $P1$, of (sub)goal $\mathcal{G}$ is then responsible for synthesizing the more efficient computation of the input-output relation specified in $\mathcal{S}$. $\mathcal{G}$ will be some form of generalization on $\mathcal{S}$, such as in tail-recursive generalization [HBS92], or, as in our example, it may place additional constraints on $\mathcal{S}$ so as to affect constraint based synthesis. Whatever the relation between $\mathcal{S}$ and $\mathcal{G}$, the main requirements are that a more efficient procedure can be synthesized through proving $\mathcal{G}$ than through proving $\mathcal{S}$ *and* that $\mathcal{G}$ entails $\mathcal{S}$.

The application of the constraint-based generalization proof plan, the induction proof plan and any subsequent proof plans (such as symbolic evaluation) is automatically co-ordinated by the CLAM proof planner according to the which proof plan has it's pre-conditions satisfied by the current goal statement. So, for example, the pre-conditions of a generalization proof plan will be satisfied by $\mathcal{S}$, the proof plan will be applied producing the corresponding post-conditions. These post-conditions will satisfy the pre-conditions of the induction proof plan which will then be subsequently applied. Proof plans may also be applied as sub-plans. For example the ripple proof plan is called within the application of the induction proof plan.

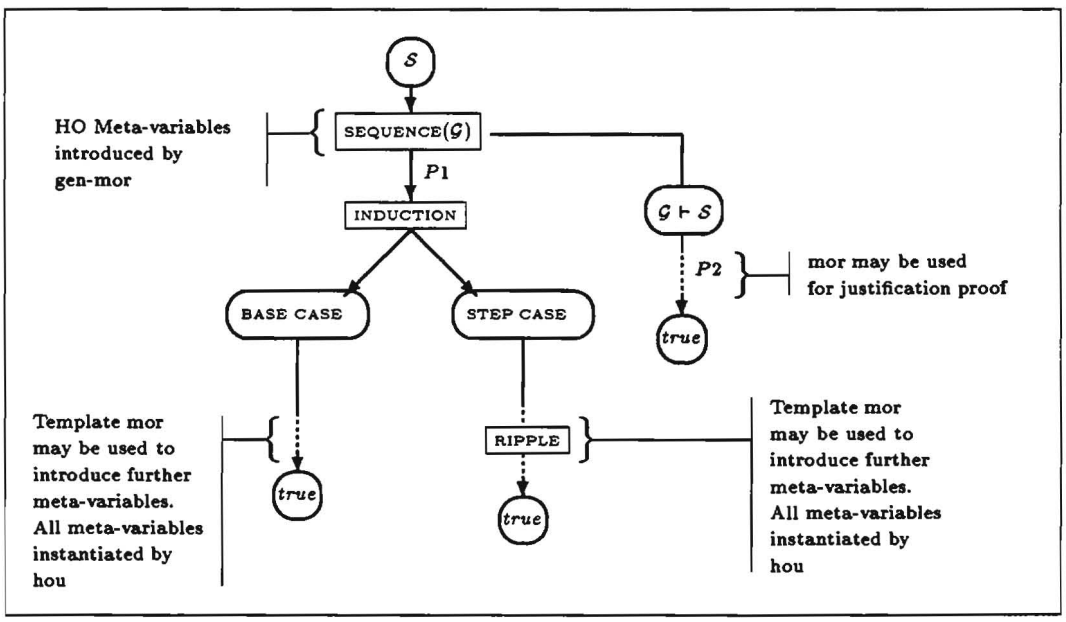

Figure 1: Form of Generalization Synthesis Proof

The third step of the strategy consists of using gen-MOR to fully identify $\mathcal{G}$ through higher-order unification (HOU). Recall that this is the case if only the form, but not the precise content of $\mathcal{G}$, is known at the sequencing step. HOU instantiates the meta-variables by matching subsequent proof derivations with the available rewrite rules (which will always include those formed from $\mathcal{E}$).[8] In this way, the need to

[7] In some case, the proof of the justification goal may also require MOR. This is not the case with the example we shall use where the justification goal is trivial.

[8] Many efficient functions can be synthesized without the use of higher-order meta-variables. In such cases MOR is not required. and fig.1 would simply resemble a standard existential inductive proof (i.e. we would omit the sequencing step and the justification proof $P2$).

treat the identification of $\mathcal{G}$ as a eureka step is removed by higher-order unification using rewrite rules formed from the available definitional equations.

In traditional program transformation, eureka steps may also occur during the identification of the recursive branche(s) of the target program definition. Within the context of proof synthesis, this would correspond to providing the requisite recursive date-types at the induction step case of the proof. The application of template MOR, the fourth step of the strategy, allows us to circumvent such eureka steps by introducing meta-variables during the rippling stages of the induction step. Template MOR applies higher-order schematic rewrite-rules which have the effect of introducing new HO meta-variables. These allow for the delaying of proof commitments concerning the identity of recursive data-types until further proof development enables the meta-variables to be instantiated (more detail is provided in §4.2).

The precise nature of $\mathcal{G}$, the induction rule applied to $\mathcal{G}$, the kind of higher-order rewrite-rule employed by template MOR, and any restrictions on how the meta-variables are instantiated, fully determine the type of recursion constructed in the target proof (and thereby the efficiency of the extract algorithm). The complete synthesis process is mechanical and the resulting program is guaranteed to satisfy the program specification ($\mathcal{S}$).

4 Proof Plans for Controlling Constrained (Tail) Recursion

Recall that the main objective of constraint based transformations is the replacement of expensive expressions in programs. Constraint-based transformations cover a wide variety of function types. Three kinds of constraint-based transformation may be distinguished according to the properties of the expensive expression(s): the expensive expression(s) may occur either at the repetitive places of recursive definitions, or in the terminating branches of a procedure, or the expensive expression is based on the fixed, or constant, parameter of a recursive function. We shall for now only consider an example of the latter kind – e.g. the *rev_sm* example – although what is said here concerning proof restrictions and pre- and post-conditions for the constraint based generalization proof plan applies to the other kinds of constraint-based transformation.

4.1 Pre- and Post-Conditions For Constraint-Based Generalization

Recall that the identification of the new constraint parameter in rev_sm_C, as $sum(w)$ is a *eureka step*. The *eureka step* can be circumvented by MOR: introducing a new parameter identified in terms of meta-variables (which will then be subsequently instantiated through HOU). To ensure that we obtain the efficient constrained function definition we place the restriction that the new constraint parameter will be some function, represented by a meta-variable M, on the non-recursive (non-inductive) parameter, i.e. w, of the original *rev_sm* program.

In general, this *constraint based restriction* ensures the removal of the expensive expression, and is implemented as a CLAM proof plan method. The input goal for the constraint based proof plan is of the following form:

$$\forall x, \forall \vec{y}, \exists z.\ z \quad = \quad f_n(x, \vec{y}) \tag{17}$$

where $\vec{y}$ is a vector that denotes 0 or more additional parameters. The effects will consist of a synthesis and a justification goal where the former is of the following

general form (where the constraint parameter is partially identified by $M(\vec{y})$, and where M is a meta-variable):

$$\forall x, \forall \vec{y}.\ f_n(x, \vec{y}) \quad = \quad f_m(x, \vec{y}, M(\vec{y})) \tag{18}$$

and where the justification goal requires a trivial proof that (18) $\vdash$ (17), viz:

$$\forall x, \forall \vec{y}. f_n(x, \vec{y}) \ = \ f_m(x, \vec{y}, M(\vec{y})) \quad \vdash \quad \forall x, \forall \vec{y}, \exists z. z \ = \ f_n(x, \vec{y}) \tag{19}$$

We also require that the target of the constraint-based synthesis is to tail-recursive. Unlike the tail-recursive syntheses reported in [HBS92], proofs of goals such as (18) are *equality proofs* as opposed to existential proofs (a direct proof of (17) would constitute an existential proof, but we require the introduction of the meta-variable in the manner described). Thus, we cannot use the tail-recursive restrictions on the identity of existential quantifiers described in [HBS92].[9] Instead, for such equality proofs, we achieve the desired tail-recursive form through the application of schematic, or higher-order, rule templates mentioned in §3.

4.2 Higher-order Rule Templates

The constraint-based proof plan has access to higher-order rule templates. These are higher-order rule schemas which, upon application, provide partially identified recursive definitions. The templates are designed to provide definitions of the desired form from which new *higher-order* wave-rules may be formed. These wave-rules facilitate the rippling process and the higher-order meta-variables introduced into the proof, as a result of their application, become instantiated through subsequent theorem proving.

In our example we desire, in addition to containing constraint parameters, a tail-recursive program which takes three arguments. Thus, the corresponding template is as follows:

$$F(nil, W, D) \ \Rightarrow \ G_3(W, D) \tag{20}$$

$$F(\boxed{Hd :: \underline{Tl}}^{\uparrow}, W, D) \ \Rightarrow \ F(Tl, \boxed{G_1(Hd, \underline{W})}^{\downarrow}, \boxed{G_2(Hd, \underline{D})}^{\downarrow}) \tag{21}$$

where G_1, G_2 and G_3 are second-order meta-variables. In general, for a function of n arguments there will be n meta-variables $A_1, ..., A_n$:

$$F_n(nil, A_1, ..., A_n) \ \Rightarrow \ G_n(A_1, ..., A_n)$$

$$F_n(\boxed{Hd :: \underline{Tl}}^{\uparrow}, A_1, ..., A_n) \ \Rightarrow \ F_n(Tl, \boxed{G_1(Hd, \underline{A_1})}^{\downarrow}, ..., \boxed{G_{n-1}(Hd, \underline{A_n})}^{\downarrow})$$

Such a use of higher-order variables within rule schemas is a new kind of MOR. It is not the same as the MOR which introduces meta-variables in the constraint-based proof plan preconditions (nor as MOR is identified in previous publications such as [HBS92]). Although both uses employ the meta-variables to stand in for "unknown constructs", the pre-condition usage delays proof commitments by partially identifying goal statements, whereas the above usage constructs partially identified (wave) rule templates. The above example corresponds to a tail-recursive template. This allows us to introduce further meta-variables during re-writing (i.e. to delay further proof commitments), in addition to those introduced by the proof plan preconditions, whilst ensuring that the function being constructed adheres to a tail-recursive form.

[9] That is, that the witnesses of the two existential quantifiers, one in the induction hypothesis and one in the induction conclusion, should be identical. This would ensure that the value of the function before the recursion is entered (determined by the induction hypothesis) to be the same as the value as the recursive call is exited (determined by the induction conclusion).

4.3 The Synthesis of rev_sm_C Revisited

We now repeat the rev_sm_C example except this time we include the rippling annotations so as to illustrate how the *eureka steps* are circumvented, and how search is tamed, through the use MOR.[10] HO meta-variables are used to postpone the commitments to existential witnesses and the identification of the new constrained goal. As in §2.0.2, the specification goal is,

$$\forall x, \forall w, \exists z.\, z \;=\; rev\_sm(x, w) \tag{22}$$

This forms the input to the constraint-based proof plan (the pre-conditions of which ensure that this is the first successfully applied proof plan). The output consists of the synthesis goal (23):

$$\vdash\ \forall x, \forall w. rev\_sm(x, w) \;=\; rev\_sm_C(x, w, M(w)) \tag{23}$$

and the trivial justification goal:

$$\forall x, \forall w. rev\_sm(x, w) \;=\; rev\_sm_C(x, w, M(w)) \;\vdash\; \forall x, \forall w, \exists d.\, d \;=\; rev\_sm(x, w)$$

where M is a (higher-order) meta-variable, and $M(w)$ the partially identified constraint parameter. (23) is set up as the conjecture to prove (the specification goal).

Amongst the rewrite rules are those formed from the available recursive definitions given in §1.1. (25) and (27) are wave rules formed from the recursive branches (2) and (4). . The non-wave rules (24) and (26) are formed from the corresponding terminating branches (1) and (3).

$$rev\_sm(nil, w) \;\Rightarrow\; (sum(w), w); \tag{24}$$

$$rev\_sm(\boxed{Hd :: \underline{Tl}}^{\uparrow}, W) \;\Rightarrow\; rev\_sm(Tl, \boxed{Hd :: \underline{W}}^{\downarrow}) \tag{25}$$

$$sum(nil) \;\Rightarrow\; 0; \tag{26}$$

$$sum(\boxed{Hd :: \underline{Tl}}^{\uparrow}) \;\Rightarrow\; \boxed{Hd + \underline{sum(Tl)}}^{\uparrow} \tag{27}$$

As yet, rev_sm_C, is undefined, but this is precisely where the use of the transformation templates comes into play: by instantiating F in the higher-order rewrites (20) and (21), we introduce second-order meta-variables G_1, G_2 and G_3, and provide a partially identified *tail recursive* definition for rev_sm_C which yields the following schematic rewrites:

$$rev\_sm_C(nil, W, D) \;\Rightarrow\; G_3(W, D) \tag{28}$$

$$rev\_sm_C(\boxed{Hd :: \underline{Tl}}^{\uparrow}, W, D) \;\Rightarrow\; rev\_sm_C(Tl, \boxed{G_1(Hd, \underline{W})}^{\downarrow}, \boxed{G_2(Hd, \underline{D})}^{\downarrow}) \tag{29}$$

Note that (29) can now be employed as a *higher-order wave-rule.* All we have assumed here is that rev_sm_C is a tail-recursive program with three arguments, we have not begged the question concerning the identity of the recursive data-types for the rev_sm_C definition.

Following ∀-introduction, induction is performed on (23) yielding the following induction cases:[11]

[10] It should become clear that the example is an instance of the general framework represented by fig.1.

[11] A feature of the goal-directed proofs is that introduction rules have the effect of eliminating existential quantifiers in the consequents of sequents. Conversly, elimination rules have the effect of introducing an existential instantiation in the hypotheses.

Proof Base: The base case is as follows:

$$\vdash \forall w.\, rev\_sm(nil, w) \;=\; rev\_sm_C(nil, w, M(w)) \tag{30}$$

By symbolic evaluation (using the rewrite yielded by (1) and (28):

$$\vdash \forall w.\, (sum(w), w) \;=\; G_3(w, M(w)) \tag{31}$$

(31) reduces to true by tautology, and HOU instantiates $G_3(w, d)$ to (d, w) and M to $\lambda x.sum(x)$.

Proof Step: The identification of $M(w)$, the constraint-parameter, as $sum(w)$ during the base case proof enables us to instantiate M to sum in the step case of the proof. Thus, at the induction step we have the following sequent to prove (where (32) is the induction hypothesis, and (33) the induction conclusion):

$$\forall w.\, rev\_sm(tl, w) \;=\; rev\_sm_C(tl, w, sum(w)) \tag{32}$$

$$\vdash \forall w.\, rev\_sm(\boxed{hd :: \underline{tl}}^{\uparrow}, \lfloor w \rfloor) \;=\; rev\_sm_C(\boxed{hd :: \underline{tl}}^{\uparrow}, \lfloor w \rfloor, sum(\lfloor w \rfloor)) \tag{33}$$

Rippling sideways using, on the l.h.s., (25) and, on the r.h.s.,(29):

$$\vdash \forall w.\, rev\_sm(tl, \left\lfloor \boxed{hd :: w}^{\downarrow} \right\rfloor) \;=\; rev\_sm_C(tl, \left\lfloor \boxed{G_1(hd, \underline{w})}^{\downarrow} \right\rfloor, \boxed{G_2(hd, \underline{sum(\lfloor w \rfloor)})}^{\downarrow})$$

We then ripple-in using (27) and in the process instantiates G_2 to $\lambda x, y.x+y$ through HOU:

$$\vdash \forall w.\, rev\_sm(tl, \left\lfloor \boxed{hd :: w}^{\downarrow} \right\rfloor) \;=\; rev\_sm_C(tl, \left\lfloor \boxed{G_1(hd, \underline{w})}^{\downarrow} \right\rfloor, \left\lfloor sum(\boxed{hd :: \underline{w}}^{\downarrow}) \right\rfloor)$$

Fertilization with the induction hypothesis, (32), now applies: w in the induction hypothesis is instantiated to $hd :: w$ from the induction conclusion. In the process, G_1 is instantiated to $\lambda x, y.x :: y$.

Analysis of the proof yields the desired tail-recursive program with the constraint-parameter $sum(w)$:

$$\begin{aligned} rev\_sm_C([\,], w, sum(w)) &= (sum(w), w); \\ rev\_sm_C(h :: t, w, sum(w)) &= rev\_sm_C(t, h :: w, h + sum(w)) \end{aligned}$$

4.4 Fixed expensive expressions

In the case of functions where the expensive expression is based on a fixed, or constant, parameter of a recursive function no MOR is required. For example, consider the following function:

$$\begin{aligned} g(nil, \epsilon) &= nil; \\ g(h :: t, \epsilon) &= sqr(\epsilon) \times h :: g(t, \epsilon). \end{aligned}$$

Here the second parameter, ϵ, remains constant throughout the recursion (and hence so will the expensive expression, based on the constant, $sqr(\epsilon)$).

Again, the procedure we follow to optimize this function, by lifting the expensive expression out of the recursion, is to introduce a new (generalized) function with an additional parameter, d, which will be a function on the non-recursive parameter, ϵ. However, in the case of fixed parameters, there is no need for meta-variables since in such cases we know that d is to be identified with the constant expensive expression $sqr(\epsilon)$.

Hence the new (generalized) function, g_new, is introduced as follows:

$$\forall x, \exists d.g(x,\epsilon) \;=\; g\_new(x,\epsilon,d) \;\; where \;\; d \;=\; sqr(\epsilon) \qquad (34)$$

After applying induction, followed by rippling, an analysis of the resulting proof yields the following optimization:

$$\begin{aligned} g\_new(nil,\epsilon,d) &= nil; \\ g\_new(h :: t,\epsilon,d) &= d \times h :: g\_new(t,\epsilon,d). \end{aligned}$$

5 Benefits and Comparisons

Using proof plans to synthesize efficient algorithms presents search and control advantages over the *unfold/fold* approach to transformation [BD77]. Unfold/fold transformations are motivated by the desire to find recursive terms which can be used for *folding* with definitional equations. This involves quite extensive search in order to find a successful fold.

The proof plan analysis, on the other hand, is motivated by the desire to find witnesses at the induction step of a synthesis *proof*. Once this has been achieved, through rippling, which may include MOR, then the proof is completed in much the same way as any inductive synthesis proof: by a process of *unfolding* until all terms in the conclusion match terms in the proof hypotheses. The fact that in rippling, the wave-fronts in the proof must correspond to those in the wave-rule schemas provides considerable control with a low branching rate. The unfold/fold transformations on the other hand require numerous applications of laws for which any overall strategy is difficult to characterize. Thus rippling is far easier to automate. The most persuasive empirical evidence for this being within the context of automatic proof plan application through the automation of the rippling out process (*cf.* [BSvH+93]).

A further benefit of the rippling strategy is that it is guaranteed to terminate: wave-fronts are always propagated in a direction toward achieving a match between conclusion and hypothesis.

Rippling, incorporated with MOR, allows us to circumvent eureka steps required by sequential rewriting transformation strategies such as unfold/fold. This clearly aids automation since key decisions regarding the identity of recursive terms in the target program can be delayed, rather than user-supplied, until subsequent planning provides the requisite information.

By providing the characteristics of the various generalizations, in the form of restrictions on the proof, we can ensure that the desired optimization is built into the algorithm being synthesized.

Since any of the synthesis proofs *must* satisfy the specification formed from the standard equational form, $\mathcal{E}$, of the function being synthesized then the proof extract program is guaranteed to satisfy the specification, and hence to compute the function defined by $\mathcal{E}$.

By proving that the synthesized program satisfies the original specification, we avoid the need to establish that any rewrite rules used are in themselves correctness (equivalence) preserving. This will, as a general rule, require as much effort as providing an explicit proof of correctness for the source to target transformations. For example, many of the systems that employ the *unfold/fold* strategy rewrite the recursive step(s) of a source program through the application of various *equality* lemmas, each of which needs to be proved (by induction) if the source to target transformation is to preserve equivalence [TT84, Dar89]

6 Conclusion

We described a general technique for controlling the synthesis of efficient programs using automatic proof planning. The technique encapsulates a diverse range of program optimizations, and benefits from the principled search and control strategies of proof plans. In particular, the syntactic pattern matching properties of rippling mean that we avoid the control problems encountered by the arbitrary application of rules and laws in program transformation systems. The optimization process is automatic and correctness is guaranteed. The technique circumvents eureka steps which have prevented total automation in program transformation systems. This has been achieved by incorporating MOR into proof plans.

References

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. Research Paper 507, Dept. of Artificial Intelligence, Edinburgh, 1990. Appeared in the proceedings of CADE-10.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic.* North-Holland, 1958.

[Chi90] W. N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, University of London (Imperial College), 1990.

[Dar89] J. Darlington. A functional programming environment supporting execution, partial evaluation and transformation. In *PARLE 1989*, pages 286–305, Eindhoven, Netherlands, 1989.

[HBS92] J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 310–324, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[ML79] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[Pet84] A. Pettorossi. A powerfull strategy for deriving programs by transformation. In *ACM Lisp and Functional Programming Conference*, pages 405–426, 1984.

[RP82] S. Koenig R. Paige. Finite Differencing of Computable Expressions. *ACM Transformation on Functional Programming Languages and Systems*, 4:pp. 405–454, 1982.

[TT84] H. Tamaki and T.Sato. Transformational logic program synthesis. In *Proceedings of the International Conference on Fifth Generation Computer Systems*. ICOT, 1984.

[Wad88] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *Proceedings of European Symposium on Programming*, pages 344–358. Nancy, France, 1988.

[Wai89] S.S. Wainer. Computability - logical and recursive complexity, July 1989.

[WNB92] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 325–339. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.

Below you find a list of the most recent technical reports of the research group *Logic of Programming* at the Max-Planck-Institut für Informatik. They are available by anonymous ftp from our ftp server `ftp.mpi-sb.mpg.de` under the directory pub/papers/reports. If you have any questions concerning ftp access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Regina Kraemer
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `kraemer@mpi-sb.mpg.de`

MPI-I-94-241	J. Hopf		Genetic Algorithms within the Framework of Evolutionary Computation: Proceedings of the KI-94 Workshop
MPI-I-94-235	D. A. Plaisted		Ordered Semantic Hyper-Linking
MPI-I-94-234	S. Matthews, A. K. Simpson		Reflection using the derivability conditions
MPI-I-94-233	D. A. Plaisted		The Search Efficiency of Theorem Proving Strategies: An Analytical Comparison
MPI-I-94-232	D. A. Plaisted		An Abstract Program Generation Logic
MPI-I-94-230	H. J. Ohlbach		Temporal Logic: Proceedings of the ICTL Workshop
MPI-I-94-228	H. J. Ohlbach		Computer Support for the Development and Investigation of Logics
MPI-I-94-226	H. J. Ohlbach, D. Gabbay, D. Plaisted		Killer Transformations
MPI-I-94-225	H. J. Ohlbach		Synthesizing Semantics for Extensions of Propositional Logic
MPI-I-94-224	H. Aït-Kaci, M. Hanus, J. J. M. Navarro		Integration of Declarative Paradigms: Proceedings of the ICLP'94 Post-Conference Workshop Santa Margherita Ligure, Italy
MPI-I-94-223	D. M. Gabbay		LDS – Labelled Deductive Systems: Volume 1 — Foundations
MPI-I-94-218	D. A. Basin		Logic Frameworks for Logic Programs
MPI-I-94-216	P. Barth		Linear 0-1 Inequalities and Extended Clauses
MPI-I-94-209	D. A. Basin, T. Walsh		Termination Orderings for Rippling
MPI-I-94-208	M. J{ä	ger	A probabilistic extension of terminological logics
MPI-I-94-207	A. Bockmayr		Cutting planes in constraint logic programming
MPI-I-94-201	M. Hanus		The Integration of Functions into Logic Programming: A Survey
MPI-I-93-267	L. Bachmair, H. Ganzinger		Associative-Commutative Superposition
MPI-I-93-265	W. Charatonik, L. Pacholski		Negativ set constraints: an easy proof of decidability