Correct and User-Friendly Implementations of Transformation Systems[1]

Kolyang[†], T. Santen[‡], B. Wolff[†]

[†]Universität Bremen , FB3
P.O. Box 330440
D-28334 Bremen
{bu,kol}@informatik.uni-bremen.de

[‡]GMD FIRST Berlin
Rudower Chaussee 5
D-12489 Berlin
santen@first.gmd.de

Abstract. We present an approach to integrate several existing tools and methods to a technical framework for correctly developing and executing program transformations. The resulting systems enable program derivations in a user-friendly way.
We illustrate the approach by proving and implementing the transformation Global Search on the basis of the tactical theorem prover Isabelle. A graphical user-interface based on the X-Window toolkit Tk provides user friendly access to the underlying machinery.

1 Introduction

Development by transformation is a prominent approach in formal program development (CIP [Bau+85], PROSPECTRA [HK 93], KIDS [Smi 90]). Many case studies have proven its feasibility and demonstrated how much more abstract and user-oriented developments could be achieved than using usual post-verification approaches (fundamental for systems like PVS [OSR 93]). One recent case study is [KW 95]; and a prominent one is [SPW 95] where a strategic transportation scheduling algorithm is developed which is 200 times faster than the ones in practical use today. Unfortunately, implementations of transformation systems tend to be complicated and insecure. The correctness issue of transformation rules is usually not treated at the implementation level of existing systems.

In contrast to this, there is a family of "tactical theorem provers" in the tradition of LCF available with systems like *HOL* [GM 93] and *Isabelle* [Pau 94a], that are both well-designed and powerful. Coming with an open system-design going back to Milner, they allow for user-programmed extensions in a logically sound way. But there is recent prominent criticism that these provers, because of their "academic (ivory tower) origins", have "historically placed more emphasis on logical foundations and less on usability" [Gor 95]. This is clearly one of the reasons for the small acceptance of these provers in industry up to now.

In this paper, we demonstrate a technique to combine these two approaches. It results in a *simple* implementation design, in proven correct transformations which are easy to extend and to modify, and in a graphical user-interface that allows developers to profit from the abstraction of the transformational approach. We claim that our

---

[1] This work has been supported by the BMBF projects **UniForM** [Kri+95] and **ESPRESS**.

technique is applicable in a fairly wide range of problems, simply by modifying and extending our prototype implementation.

Our work integrates three existing and well documented public domain tools — the tactical theorem prover Isabelle based on *Standard ML* [HMM 86] and the X-Window toolkit *Tk* [Ous 94]. As object-language, we chose higher-order logic (HOL) which is one instantiation of the generic system Isabelle with an object logic and is delivered with the standard package. A subset of HOL formulas can easily be translated into functional programs (e.g. ML).

The basic idea of our approach is to separate the *logical core* of a transformation from the pragmatics of its application. As a *synthesis theorem* it can be proven correct independently in the logics of the object language, while the *tactical sugar*, which often highly system dependent, is concerned with the concrete application in a development context, i.e. the construction of suitable substitutions, "hard-wired" quantifier eliminations and standard simplifications, together with user interaction to control this process. The distinction between synthesis theorem and tactical sugar establishes an important separation of concerns.

We illustrate our approach by the transformation *Global Search* [Smi 87] that converts a non-constructive specification into a constructive one. This complex transformation has the character of a "design tactic" [Smi 90] or "design method" [HK 93]. Other transformations like *Divide-and-Conquer* or *Split of Postcondition* and elementary transformations like *recursion removal* or *fusion* also fit into our framework.

We proceed as follows: after introducing the idea of synthesis theorems and a brief presentation of Isabelle, we present Global Search as a synthesis theorem and prove it correct within Isabelle/HOL. We sketch several possibilities of tactical programs for our synthesis theorem. The resulting system is embedded into a user interface. Lastly, a small application example demonstrates the use of the resulting prototype system.

2 Transformations as Synthesis Theorems

2.1 The Concept

The core of our presentation is a general scheme of synthesis theorems, i.e. of logical formulas. The automatic construction of substitutions and other deduction-technical machinery, for short — the tactical sugar — is discussed in section 3.5.

Our intuition of "performing a program transformation" motivates several key notions (cf. [HK 93]). A transformation is composed of an *input pattern I* which is matched against an application context of a specification. This pattern is designed to be as general as possible and at the same time to be best supportable by automatic matching procedures (which belong to the tactical sugar). From the result of matching *I* against the specification at a specific position, an instance of the *output pattern O* is constructed automatically. All side conditions that can not be treated by automatic procedures and require theorem proving are collected in the *applicability condition V*. Usually, the output pattern contains function symbols that are introduced by the application of the transformation. They represent the design decisions of the whole transformation step, i.e. auxiliary functions whose definitions have to be provided by the user. These items are called the *parameters* $P_1 \ldots P_n$ of the transformation.

On the logical side, these items can be organised as a conditional equation:

$$\forall P_1 .. P_n .\, V \Rightarrow I \rightsquigarrow O$$

where $\rightsquigarrow$ is a transitive binary operator that typically stands for

- logical equality or equivalence in case of symmetric transformations or
- the implication $\Leftarrow$ in case of classical refinement (the input pattern has to follow from the output; in algebraic jargon: the model class of the output specification is included in the model class of the input specification) or
- the Scott-definedness ordering $\sqsubseteq$ in case of "robust implementations" using object-logics like LCF (see [Pau 94a]).

This scheme is strong enough to capture a large variety of transformations — from "Filter Fusion" [BM 93] to "Split of Postcondition" [HK 93]. These have been presented in [KW 95]. The synthesis theorem for Global Search is discussed later.

The separation into synthesis theorem and tactical sugar has the following consequence for the soundness of a transformation: The logical core can be proven *within* the logic in which it is represented. This can be done by showing that the synthesis theorem follows from the basic axioms of the logic — or, in other words, the synthesis theorem follows from a *conservative extension of the core logic* (see below).

2.2 Introduction to Isabelle

Isabelle is a *generic* theorem prover that supports a family of logics, e.g. first-order logic (FOL), Zermelo-Fränkel set theory (ZF), constructive type theory (CTT), the Logic of Computable Functions (LCF), and others. We only use its set-up for higher-order logic (HOL). Isabelle supports natural deduction style. Its principal inference techniques are resolution (based on higher-order unification) and term-rewriting. Isabelle provides syntax for hierarchical theories (containing signatures and axioms).

As an example, let us create the theory List0 from the theory HOL that contains the basic rules of the logic. All input in the form of UNIX files or user input will be denoted with this FONT — enriched by the usual mathematical notation for $\forall$, $\exists$,... instead of ASCII-transcriptions.[2] We define the unary type constructor list, its constructors ([], #) and the concatenation @:

```
List0 = HOL +
      types     list        1
      arities   list        :: (term)term
      consts    "[]"        :: "α list"                    ("[]")
                "#"         :: "[α, α list] → α list"      (infixr 70)
                "@"         :: "[α list, α list] → α list" (infixl 60)
rules
      app_mt          "[]@m = m"
      app_cons        "(a#n)@m = a#(n@m)"
end
```

Here, list belongs to the type universe term of HOL and accepts types from term. This construct is a tribute to the genericity of Isabelle. "→" is the function space constructor, and the brackets denote curried functions: [α, α list] → α list is equivalent to α→α list→α list. The equality "=" stems from HOL, while "≡" is

---

2 We do not distinguish quantifications and implications at the different logical levels throughout this paper; see [Pau 94a].

used to denote the definitional equality. The comments ("[]") and (infixl 70) are pragmas setting up the parsing and pretty-printing machinery of Isabelle.

2.3 The Logic HOL

In this section, we will give a short overview of the concepts and the syntax. Our object-language HOL goes back to [Chu 40]; a more recent presentation is [And 86]. In the formal methods community, it has achieved some acceptance, especially in hardware-verification. HOL is a classical logic with equality. It is based on total functions denoted by λ-abstractions like "$\lambda x.x$". Function application is denoted by $f\ a$. Although its type discipline incorporates polymorphism with type-classes (as in Haskell), in this paper we only use Milner-Polymorphism (as in ML).

Logical rules of HOL like:

$$\frac{P \quad Q}{P \wedge Q}\ (\text{conjI}) \qquad \frac{P \wedge Q}{P}\ (\text{conjunct1})$$

will be represented in Isabelle by

```
[| ?P ; ?Q |] ⟹ ?P ∧ ?Q        ?P ∧ ?Q ⟹ ?P
```

where variables prefixed by a question mark are called *meta-variables*. Their exact meaning in the deduction process is discussed later.

2.4 Proving in Isabelle

Isabelle as a system is a set of function definitions in the ML-environment (or: "data-base"). They represent a collection of function- and data type definitions. Most notable are the three mutually dependent data types: tactic, thm and (internal) proof_state.

Isabelle supports two styles of theorem proving: forward proof and backward proof.

Backward proving in Isabelle. The general scheme of a backward proof consists of three steps:

(1) The initialisation of the internal "proof-state" with the formula to be proven (the "goal"). It is done by the operation:

```
goal: theory -> string -> thm list
```

where the string contains the textual representation of the formula to be type-checked and proved within theory.

(2) A refinement of the proof-state. It is performed with the operation:

```
by : tactic -> unit
```

This refinement can be seen as a transformation of the proof-state by means of tactics and already proven theorems. Two of these are the following pivotal *built-in* tactics

```
atac: int -> tactic        rtac: thm -> int -> tactic
```

The integer parameters specify the subgoal to which the tactic is applied. They encapsulate the Isabelle meta-inferences *assumption* (basically "A" implies "A" modulo unification) and *resolution* (essentially "A ⇒ B" and "B ⇒ C" implies "A ⇒ C" modulo unification)

(3) The extraction of a theorem produced out of a proof-state with no subgoals:

```
result: unit -> thm;
```

returns a value that can be bound to an arbitrary ML-identifier.

By composition of these operations on the proof-state, large proof-scripts can be organised in the *.ML files that are executed automatically when loading a theory.

Forward proving in Isabelle. Forward reasoning mimics the classical way of constructing proof trees. The combination of two rules, say conjI and conjunct1 given above, can be done by using the resolution combinator (involving unification):

```
RS : thm * thm -> thm
```

in the form:

```
conjI RS conjunct1
```

which is evaluated by Isabelle to the derived rule:

```
[| ?P; ?Q |] ⟹ ?P
```

As simple example for higher-order unification, we consider the specialisation rule:

```
∀ x. ?P x ⟹ ?P ?x                                    (spec)
```

With this HOL rule it is possible via forward proving in a theorem, say ∀ y. y = a, to eliminate the quantification and to replace the bound variable y by the *meta-variable* ?x. The resulting formula would be ?x = a. We can interpret the meta-variable ?x as a "hole" in the formula that can be filled later by substitutions (usually produced as a consequence of unification inside atac and rtac). This possibility of "postponing substitutions" and of transforming theorems in a programmed, but logically sound way, is important for our approach.

Isabelle provides substantially more machinery, especially for people who want to set-up their own logic or who yearn for a higher degree of automatisation. However, with the subset presented here, extended by some variants, it is already possible to perform substantial proofs and to describe the relevant operations in this paper.

2.5 Conservative Extension in HOL

The introduction of new axioms ("rules" in case of List0) while building a new theory is an extremely dangerous method, since the resulting theory may easily be inconsistent. Hence it is necessary to recall that there is a number of syntactic schemes for specification-extension that maintain the consistency of the extended one. (For a more formal and very readable account on "conservative extensions schemes" the reader is referred to [GM 93]). Some schemes are:

- the *constant definition* "$c \equiv t$" or "$c\ x \equiv t\ x$" of a fresh constant symbol c by a closed expression t not containing c,
- the *type definition* (a set of axioms stating an isomorphism between a non-empty subset of a base-type and the new type to be defined),
- a set of equations forming a *primitive recursive scheme* over a fresh constant symbol f,
- a set of equations forming a *well-founded recursive scheme* over a fresh symbol f.

The basic idea of these extension schemes is to avoid general recursion. Instead, they introduce axioms only in a controlled way. The desired properties have to be derived from these. Building up large theories by methodically using conservative extensions used to be a quite tedious enterprise, but recent advances in the Isabelle implementation have substantially improved the support for this approach [Pau 94b].

3 Global Search Transformation

We use the approach sketched in section 2 to implement a transformation based on the theory of *Global Search* algorithms that has been developed at Kestrel Institute and implemented in the *Kestrel Interactive Development System* (KIDS) [Smi 87, Smi 90]. After presenting the basic idea of global search, we show how the theory can be formalised in Isabelle/HOL. We prove a synthesis theorem under the resulting theory, and finally provide a tactic program (the sugar) converting the synthesis theorem into an executable transformation.

3.1 The Algorithm Design Theory *Global Search*

The global search theory characterises a large class of algorithmic problems that are solvable by search or optimisation algorithms. It covers problems typically solved, e.g., by backtracking, branch-and-bound, or simplex algorithms.

For the purpose of this paper, we closely stick to the notation used in [Smi 87]. There, algorithms are described by input / output predicates. A *problem specification* is a quadruple $P = \langle D,R,I,O\rangle$ where D is the input domain and R is the output range of the function f to synthesise. The predicate I describes the admissible inputs, and O describes the input / output behaviour of f. Hence, f is a solution to P if

$$\forall\, x{:}D.\;\; I(x) \wedge y = f(x) \Rightarrow O(x, f(x))$$

A *design theory* extends a problem specification by additional functions. It states sufficient properties of these functions to formulate a schematic algorithm that solves the problem correctly. The basic idea of global search is to associate inputs x with *search spaces* that initially contain all solutions z with $O(x,z)$. Search is then performed by splitting search spaces into "smaller" ones until solutions are directly extractable. This idea is captured in the design theory of Figure 3.1.1.

sorts D, R, R'

operations

$I : D \to \text{Bool}$	$Satisfies : R \times R' \to \text{Bool}$
$O : D \times R \to \text{Bool}$	$Split : D \times R' \times R' \to \text{Bool}$
$I' : D \times R' \to \text{Bool}$	$Extract : R \times R' \to \text{Bool}$
$r'_0 : D \to R'$	$< : R' \times R' \to \text{Bool}$

axioms

GS0 $I(x) \Rightarrow I'(x,(r'_0(x))$

GS1 $I(x) \wedge I'(x,r') \wedge Split(x,r',s') \Rightarrow I'(x,s') \wedge s' < r'$

GS2 $I(x) \wedge O(x,z) \Rightarrow Satisfies(z,r'_0(x))$

GS3 $I(x) \wedge I'(x,r') \Rightarrow Satisfies(z,r') = (\exists\, s'.\; Split^*(x,r',s') \wedge Extract(z,s'))$

GS5 $<$ is a well-founded ordering on R'

Figure 3.1.1: Global Search Theory.[3]

The sort R' is the type of search space descriptors, I' defines legal descriptors. For an input x, r'_0 and *Split* describe the search tree for solutions z with $O(x,z)$: its root is

---

[3] In [Smi 87], axiom GS4 deals with necessary filters, which we do not consider in this paper. Still, we call our last axiom GS5 to stay consistent with the literature.

$r'_0(x)$, the initial search space; a descendant relation on nodes is given by *Split*: *Split*(x,r',s') is true if s' is a (direct) subspace of r' for an input x. *Split** is defined by

$$Split^*(x,r',s') = (\exists\, k{:}\mathbb{N}.\ Split^k(x,r',s'))$$
$$Split^0(x,r',s') = (r' = s')$$
$$Split^{k+1}(x,r',s') = (\exists\, t'.\ Split(x,r',t') \wedge Split^k(x,t',s'))$$

The possible solutions that can be extracted from a node r' are *Extract*(z,r').

Axioms GS0 and GS1 ensure that all considered search spaces are legal. Axiom GS1 additionally ensures that search spaces can be split only finitely often, i.e. that the search tree has a finite depth. GS2 requires the initial search space to contain all feasible solutions.

By axiom GS3, *Satisfies*(z,r') describes the solutions z contained in a search space r' that can be found with finite effort: there must exist a finite path in the search tree from r' to a search space s' from which z can be extracted.

Under the global search theory, we can use *Split* and *Extract* to get an algorithm schema satisfying the problem specification $\langle D,R,I,O\rangle$. Following [Smi 87], we express the algorithm by input / output predicates.

$$I(x) \wedge I'(x,r') \Rightarrow Fgs(x,r',z) = (Extract(z,r') \wedge O(x,z) \vee (\exists\, s'.\ Split(x,r',s') \wedge Fgs(x,s',z)))$$
$$I(x) \Rightarrow F(x,z) = Fgs(x,r'_0(x),z)$$

F computes a solution z for some admissible input x by searching in the initial search space $r'_0(x)$. Searching is performed by the auxiliary function *Fgs*: if z is not directly extractable from r' this search space is split and its subspaces are searched.

The global search theory described above is relatively simple. More refined ones incorporate filters to prune search spaces. The most elaborate one stated in [SPW 95] uses a refinement relation on search spaces and cutting constraints to profoundly exploit the problem domain and synthesise highly efficient search algorithms.

3.2 Formalisation in Isabelle/HOL

How do we know that a particular application of Global Search is correct, i.e. how can we be sure that we get a correct implementation of our problem specification when we instantiate the abstract global search algorithm on the basis of particular I', r'_0, *Satisfies*, *Split*, *Extract* and < defined in our problem domain? There are two reasons why correctness might be spoiled: we may make a mistake in the particular application, e.g. choosing components that do not fulfil the global search axioms, or, more fundamentally, the implementation of the transformation may be faulty, i.e. the actually implemented transformation may be unsound. It is not in question here that "something" like the theory presented in section 3.1 describes a mathematically sound transformation. But it is a long way from a paper-and-pencil proven "idea" of a transformation to its actual implementation and application. We must make sure that the transition from idea to implementation is traceable and based on well-understood principles, and that it leads to a *soundly implemented* transformation.

In contrast to the KIDS system which is not based on a general logical framework but implements transformations like Global Search directly, we have chosen higher-order logic as implemented in Isabelle. Implementing the transformation in our system first of all means formalising the description of global search given in

section 3.1 in a Isabelle/HOL theory. The Isabelle theory is sketched in the following. It is based on the HOL theories of natural numbers and sets.

```
GlobalSearch = Nat + Set +
consts
    ...
    GS3 :: "[δ → bool,[δ, ρ'] → bool,
             [δ, ρ', ρ'] → bool,
             [ρ, ρ'] → bool,[ρ, ρ'] → bool] → bool"
    ...
defs
...
REC_def "REC Fgs I I' Extract Out Split ≡
             ∀ x r z . I x ∧ I' x r ⇒
             Fgs x r z = (Extract z r ∧ Out x z ∨
                          (∃ s'.Split x r s' ∧ Fgs x s' z))"

GS3_def "GS3 I I' Split Satisfies Extract ≡
             ∀ x z r'. I x ∧ I' x r' ⇒
               Satisfies z r' = (∃ s'. rep_s Split x r' s' ∧ Extract z s')"
...
GSA_def "GSTHEORY I Out I' r0 Split Extract subspace Satisfies ≡
             GS0 I I' r0                          ∧
             GS1 I I' Split subspace              ∧
             GS2 I Out Satisfies r0               ∧
             GS3 I I' Split Satisfies Extract     ∧
             GS5 subspace"
```

Figure 3.2.1: Isabelle theory of Global Search.

Using the definitional equality ≡, we define higher-order predicates GS1 through GS5 for the global search axioms. Their conjunction GSTHEORY gives us a predicate that represents the global search axiomatization. We chose to formalise the parameter sorts *D*, *R* and *R'* of Figure 3.1.1 by making GS1 through GS5 polymorphic and use the type variables δ, ρ and ρ', respectively. In this way, we need not explicitly instantiate parameter sorts when applying the Global Search transformation: Isabelle's type inference system will find suitable sorts for us. The predicates rep_n and rep_s are defined as primitive recursors on Nat, which construct $Split^k$ and $Split^*$ from a given *Split*. REC provides an abbreviation of the characteristic equation for *Fgs*.

Building the theory in this way ensures consistency since each axiom is formalised as a conservative constant definition. We indicate this by the keyword defs, and the system checks for conservativity of these axioms as sketched in section 2.5.

3.3 The Global Search Synthesis Theorem

The following synthesis theorem for global search is based on theory GlobalSearch:

```
        ∀ I' r0 Split Extract subspace Satisfies.
          GSTHEORY I Out I' r0 Split Extract subspace Satisfies ⇒
            (I x ⇒ F x z = Out x z)
(GS)        =
            (I x ⇒ (∃ Fgs. REC Fgs I I' Extract Out Split ∧ F x z = Fgs x (r0 x) z ))
```

Assuming a global search theory, (GS) relates the problem specification to the schematic search algorithm. The function Fgs we get when composing I, I', Extract, Out, and Split according to REC finds exactly the solutions z that are specified by Out if the search starts with the initial search space r0(x) for some legal input x.

Note that (GS) has the form of synthesis theorems introduced in section 2.1. The components of the problem specification are free variables, while the bound variables I' through Satisfies serve as parameters to the transformation obtained from (GS).

What are axioms in the theory of Figure 3.1.1 appears as the premise (GSTHEORY …) in the implication of (GS). Therefore, an inconsistency in Figure 3.1.1 can not affect the "global" consistency of the Isabelle theory we are working in. If the theory of global search algorithms were inconsistent, this would only affect the applicability of the global search transformation: the synthesis theorem would trivially hold but the corresponding transformation could never be applied.

3.4 Mechanical Proof of the Synthesis Theorem

Figure 3.4.1 shows the structure of the proof of (GS) that we have carried out in Isabelle. To keep the picture readable, we omit most of the functions' parameters.

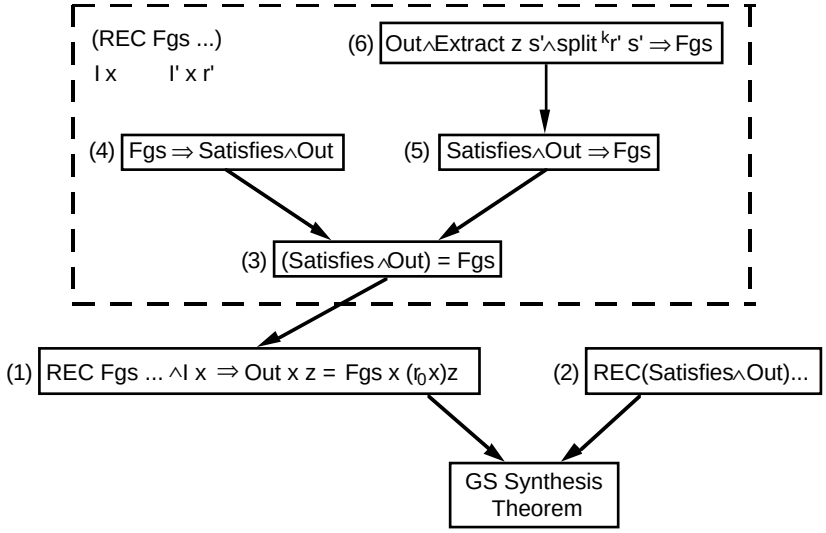

Figure 3.4.1: Proof Structure

The proof proceeds backwards in a goal-directed fashion. The first steps are to apply the introduction rules for universal quantification and implication and exhibit the equality

$$(I\ x \Longrightarrow F\ x\ z = Out\ x\ z)$$
$$=$$
$$(I\ x \Longrightarrow (\exists\ Fgs.\ REC\ Fgs\ I\ I'\ Extract\ Out\ Split \wedge F\ x\ z = Fgs\ x\ (r0\ x)\ z\ ))$$

We prove this equality by mutual implication. The "right-to-left" direction is the hard one, which after some simplification reduces to Lemma (1):

(1) $I\ x \wedge REC\ Fgs\ I\ I'\ Extract\ Out\ Split \Longrightarrow Out\ x\ z = Fgs\ x\ (r0\ x)\ z$

The central proof-idea is to find a closed form for the recursively defined Fgs.

(3) REC Fgs ... ∧ ... ⇒ (Satisfies z r' ∧ Out x z) = Fgs x r' z

This lemma says that any function Fgs satisfying the recursive equation REC behaves like the conjunction of Satisfies and Out. By GS3, Satisfies z r' means that we only need finitely many applications of Split to find a subspace of r' where we can extract z from. On the other hand, Fgs x r' z is defined by recursively splitting r' and extracting solutions z that additionally fulfil Out x z — the latter condition being the only intuitive difference between the two predicates. Once we have proved (3), we can use GS2 to specialise it and show (1).

With Lemma (3) in mind, it is easy to prove the "left-to-right" direction of (GS). Here, we basically have to show that there exists a function Fgs fulfilling REC. From (3) we know that *if* a function Fgs satisfies REC then it behaves like the conjunction of Satisfies and Out. So we use this conjunction — suitably abstracted — as a witness for the existential quantification and show that it indeed satisfies REC.

The proof of (3) does the real work. Here, we generally assume that Fgs satisfies REC, and that the input x and search space r' are admissible, which is indicated by the dashed frame in Figure 3.4.1. Again, we prove the equality by mutual implication and reduce (3) to (4) and (5). Both can be interpreted computationally. Lemma (4) deals with termination and correctness of solutions produced by Fgs.

(4) Fgs x r' z ⇒ (Satisfies z r' ∧ Out x z)

We not only have to show that all output z produced by Fgs is a feasible solution, i.e. Out x z holds, but also that it can be extracted from the input search space r' by finitely many Split's, i.e. Satisfies z r' holds. Here it is crucial that Split produces a decreasing chain of search spaces with respect to a well-founded ordering (cf. GS1 and GS5). Only this requirement allows us to interpret REC as a definition of a recursive function. Otherwise predicates that are true on cycles of Split's where Extract is false would satisfy REC. GS5 allows us to use a theory of well-founded sets that comes with Isabelle/HOL: we prove (4) by well-founded induction on r'.

Lemma (5) deals with completeness of the set of solutions produced by Fgs: all feasible solutions are indeed found by Fgs.

(5) (Satisfies z r' ∧ Out x z) ⇒ Fgs x r' z

The proof idea for (5) is induction on the length of search paths, i.e. the number *k* of Split's leading to the search space from which the solution z can directly be extracted. Lemma (6) formally captures this idea. It is gained from (5) by unfolding the definitions of Satisfies and rep_s, i.e. $Split^*$.

Global search is an example of a non-trivial transformation. The entire proof script for (GS) consists of about 140 tactics' applications. Isabelle under Standard ML of New Jersey takes about 60 CPU seconds to execute it on a Sun Sparc 5 workstation. We needed several attempts to develop the global search theory and the proof of the synthesis theorem in Isabelle. The first version of the theory was non-conservative and explicitly introduced parameter sorts *D*, *R* and *R'*. We then abolished these sorts and used polymorphism. The next stage in the theory development was to come to the conservative theory sketched in Figure 3.2.1.

The proofs had to be adapted to each rephrasal of the theory. The structure of the proofs also changed several times due to new proof ideas — the latest being to intro-

duce Lemma (3) – and due to changes in the formulation of the synthesis theorem: the first version only was an implication from the algorithm schema to the problem specification. In this version, we also left out the precondition $I\ x \wedge I'\ x\ r$ in the definition of REC. This formulation of the theorem was still correct but its premises would have been too strong to be practically useful. Only after we introduced Lemma (3) and tried to prove equality instead of implication, we became aware of the missing precondition.

Despite of all these changes to the theory, it was relatively easy to adapt the proof scripts. Simple "replay until failure" was usually sufficient to find the points were changes had to be made, and these were mostly local ones like inserting a tactic to re-establish some syntactic structure, that the next tactic depended on.

3.5 Tactical Sugar for Global Search

Global Search is used in algorithm construction by providing a mapping from GlobalSearch to an extension of the concrete problem theory such that the global search axioms are theorems under the extended problem theory. We can apply the same mapping to the schematic algorithm and get a solution for our problem, i.e. we have transformed the non-constructive problem specification into a constructive form. This algorithm is usually inefficient and has to be optimised by further transformations.

In [Smi 90, SPW 95], elaborate techniques to find a global search algorithm for a given problem specification are described. They are based on a library of global search theories that basically describe the structures of search trees for various data structures.

While it is certainly possible to implement these techniques in our framework, we focus on the description of the basics of our approach and restrain ourselves to much simpler tactical sugar: the proven synthesis theorem is used to define an ML function

```
fun GLOBAL_SEARCH : nat * string list → tactic
```

that takes the subgoal number and the list of parameters to produce a tactic. This function successively removes the universal quantification via forward proof and rule spec (see section 2.4). Similarly, the implication and the equality are converted by application of the modus ponens and substitutivity rule. These operations convert the synthesis theorem into the following version:

```
[| GSTHEORY ?I ?Out ?I' ?r0 ?Split ?Extract ?subspace ?Satisfies ;
   ?I ?x ⟹ (∃ Fgs. REC Fgs ?I ?I' ?Extract ?Out ?Split ∧
                        ?F ?x ?z = Fgs ?x (?r0 ?x) ?z) |]
⟹ (?I ?x ⟹ ?F ?x ?z = ?Out ?x ?z)
```

Furthermore, GLOBAL_SEARCH successively substitutes the parameters (after parsing and typechecking the string list) into the meta-variables. Finally, the result is applied via rtac to a particular subgoal — this will set the remaining variables (?F, ?x and ?z) and complete the mapping to the problem theory.

There are different versions of tactical sugar conceivable — one could leave more parameters uninstantiated or massage the conclusion into a different syntactical form using more forward resolution steps. More complex tactics based on the synthesis theorem could employ the substitution rule for equality of higher-order logic. We would not break up the equality in (GS) but apply it to a subterm of a possibly complex goal. The choice how to come to the parameters of the global search theory would remain the same as before.

Note that the process of transforming the synthesis theorem into a logical rule which is applied by some ML function is "safe". The transformations used there are all based on proven correct rules and the primitive theorem manipulating functions of Isabelle, so nothing incorrect can happen — assuming the core of Isabelle is sound.

Proving the correctness of tactical sugar is not necessary in the sense that it simply controls the application of basic axioms and lemmas. This control may lead to a dead end and *fail*, or it might prove something that we did not want to prove, but it can never produce a proof for something that would not be provable in the theorem prover without this tactical program. Of course, the *implementation* for "apply an axiom" which represents the atoms of our tactical control programs might be incorrect or the basic rules of the logic might be unsound. But proving the correctness of "apply an axiom" in the absolute sense would require a formalisation and verification of the core theorem prover. As a consequence, this "meta-encoding" can not fully solve the problem since it raises the same problems of correctness on the meta-level.

4 YATS — the System

YATS can be regarded as a step towards an IFDSE (Integrated Formal Development Support Environment) following the philosophy of [BH 95] or [Kri+95]. Such systems support many stages of the formal development, from initial functional specifications, through design specifications and refinement. More elaborated systems will also provide a support for specification animation, version-management etc.

We believe that a high-quality graphical user interface (GUI) plays a key role for both the acceptance and productivity of an IFDSE. To date, there is no common agreement on what could be a good design of a GUI for a theorem prover or an IFDSE (we admit that we have not found the definite answer either), although there are recent remarkable efforts in this direction.

Our GUI should be completely independent from Isabelle and as independent as possible from our system environment and our hardware-platform. In the past, the development of many systems (PROSPECTRA, for example) has been trapped by their complex and monolithic design. An answer to this dilemma of monolithic designs can be a *heterogeneous* one with few complementary components that are systems in their own right. This way it is possible to integrate work of independent research groups. The main task of an heterogeneous design is to provide suitably abstract and flexible interfaces in order to enable an easy and stable integration of new versions of its components.

For our GUI, we chose the toolkit *Tk* [Ous 94] to achieve this goal. Although we wish to profit from Tk as a highly portable interface to X-Windows, we do not believe that the command-language Tcl on top of Tk should be used for larger software developments. The major reason is that Tcl supports only one data-structure — text — in a way similar to LISP and its lists. Tcl is an untyped language without data structures and lacks higher modularisation concepts.

For this reason, we implemented an SML interface for Tk, called *sml_tk*. It is a component in its own right and provides a toolkit for standard windows, e.g. a substitution window, that can be reused by research groups working on, e.g., another theorem prover interface. Based on sml_tk, the GUI itself is implemented as an SML functor, called *isawin*, which is parametrised by a list of tactical-sugar functions. The system YATS is an instantiation of this functor with a list containing the function

GLOBAL_SEARCH (see section 3.5). According to the instantiation, isawin automatically produces new interface components and new dialogues with the user — the implementor of a transformation has only to provide its proof and its tactical sugar.

The following diagram gives a short overview over the stack of main components (the size of the blocks roughly corresponds to their implementation size):

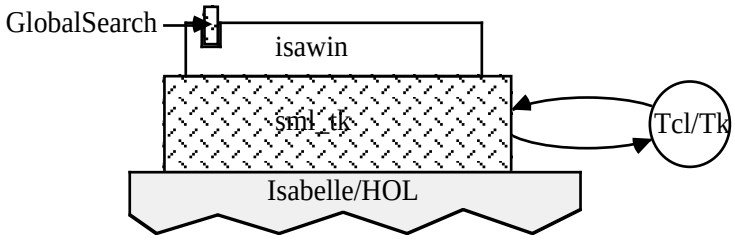

Figure 4.0.1: The System architecture

Note that the formal proof of Global Search and its integration into the system contributes only to a very small part to the whole system. This justifies our claim that the design allows the implementors to concentrate on the real intellectual problem of designing and verifying new transformations. Moreover, the instantiation of isawin with a new transformation is a question of a few seconds. Even if a very different state of technology (hardware and software) has to be taken into consideration, in comparison to PROSPECTRA, for example, where a complete recompilation was necessary that took 2 hours [GL 93], this is quite remarkable.

4.1 The interface sml_tk

Our interface evolved from an imperative version of a purely functional, monad-style encapsulation of Tk in Gofer [VTS 95]. It has a more functional flavour than the interface available for caml/light [PR 95]. It is characterised by the following features:

- abstract data-types for options, configurations, packing information
- abstract data-type for graphical objects, called *widgets*
- events on the interface (mouse clicks, key strokes, etc) are mapped to SML functions associated to widgets via *bindings*.
- a toolkit for defining a problem specific set of window types.

To give a flavour of programming in sml_tk, let us have a brief look at a fragment of the essential tree-like data type for widgets used to describe the content of windows:

```
datatype   Widget =
      Frame of WidId * Widget list * Pack list * Configure list * Binding list
    | Label of WidId * Pack list * Configure list * Binding list
    | Entry of WidId * Pack list * Configure list * Binding list
    ...

type Window   = (WinId * Title * (Widget)list * Action);
```

The following fragment is taken from the description of a small standard-window of the toolkit:

```
fun input enteraction =
    let fun mrs () = let val nm = selectTextAll "e1"
                     in   enteraction nm ();closeWindow "enter" end
    in  Entry("e1",[],[Width  20],  [Bind("<Return>",mrs)])  end;
```

The function input yields an Entry-Widget [Ous 94], that represents a graphical field allowing to enter a string. It has the name e1 and a width of 20 characters. Associated to e1, there is a function mrs that is evaluated whenever the event <Return> happens. mrs selects the inserted text, passes it to the parameter function enteraction and closes the surrounding window called enter.

4.2 The GUI of YATS

The main window of YATS (as well as any other instance of isawin) consists of two major components: An edit-window (where the editing facilities like Cut-Copy-Paste are already provided by Tk without writing any additional line of code in the interface) and a prover-window to which the transformation facilities (a result of the instantiation of isawin) and the operations controlling Isabelle are associated. It is possible to browse theories and ML files in suitable subwindows with their associated axioms and theorems (see Figure 4.2.1 below). The user-interface for the prover part is not in the focus of this paper.

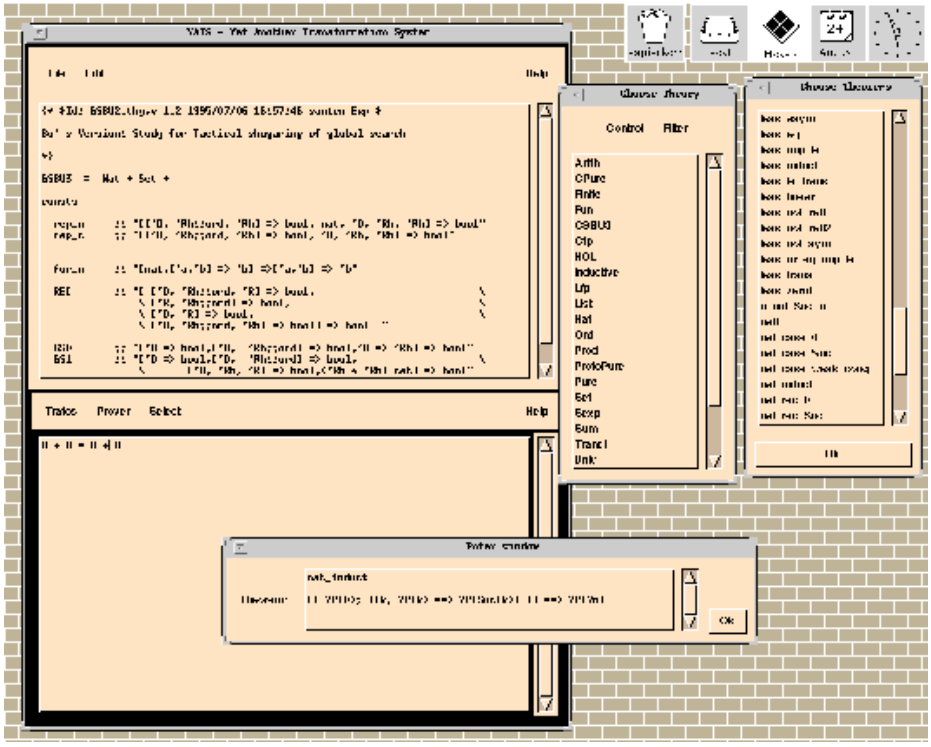

Figure 4.2.1: Screenshot of YATS (Overview)

Usually, by double-click on an arbitrary widget of the interface, the user can get more information, and by triple-clicks suitable operations on the activated unit are executed. For example, when pointing to a subgoal in the prover-state-widget, a double-click will inform the user on what transformation or Isabelle-command will be executed (due to settings and other information inferred by the system), while a triple-click performs this command.

4.3 An Application Example

In this section we use our system to develop a global search algorithm that enumerates all maps from a finite set *U* to a finite set *V*. We take the global search theory for this algorithm, *gs_finite_mappings*, from [Smi 87]. In KIDS, abstract and simple theories like *gs_finite_mappings* are used to describe search patterns on particular data structures. To develop a search algorithm for a particular problem with KIDS means to *specialise* such a "pattern theory" to the given problem specification (see [Smi 90]). Since the specialisation procedure as well as the pattern theories are hard-wired into KIDS, their correctness can not be guaranteed within the system. The development of "pattern theories" can not rely on specialisation but must use different tactical sugar like we have implemented in YATS.

The problem specification for *gs_finite_mappings* is based on a library theory of finite maps:

$F \mapsto$ fin_maps

$\delta \mapsto \alpha$ set $\times \beta$ set $\qquad \rho \mapsto (\alpha, \beta)$ Fmap

$I \mapsto \lambda (U,V).$ Fin $U \wedge$ Fin $V \qquad$ Out $\mapsto \lambda (U,V)$ N. $N \in$ Map(U,V)

We wish to synthesise a function called fin_maps that transforms pairs of sets over types α and β to finite maps whose domains are sets over α and whose ranges are sets over β, i.e. members of (α, β) Fmap. For finite input sets U and V the function must return a map N with domain U and a range in V. The predicate Map is defined by

$$Map(U,V) \equiv \{ M \mid dom\ M = U \wedge \forall b \in dom\ M.\ M\ \hat{}\ b \in V \}$$

Note that we need an explicit operation ^ to apply maps because their type (α, β) Fmap is different form the HOL function type.

To develop an algorithm for the problem, we chose the theory of finite maps as the logical context (just by activating it via a mouse-click) and enter the (slightly massaged) specification as a goal into YATS as shown in Figure 4.3.1.

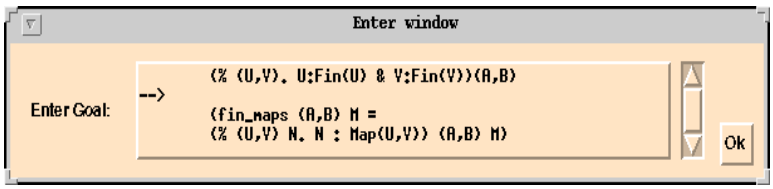

Figure 4.3.1: Entering the transformation goal

We now apply the transformation Globalsearch of section 3.5. First of all, this makes Isabelle match the goal with the transformation rule and set up the substitution of meta-variables and type variables for the problem specification. The next step is the creative part of the transformation: we have to provide the parameters that establish a global search theory. To date, as Figure 4.3.2 shows, this is done by explicitly entering a substitution for the parameters.

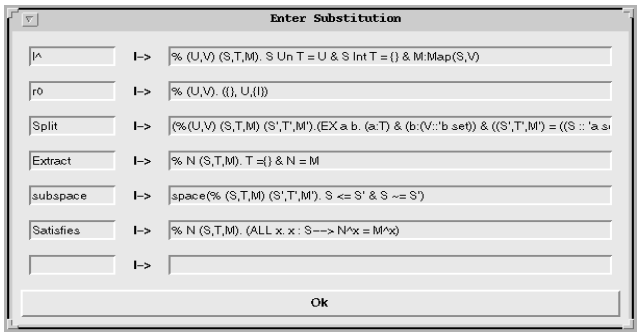

Figure 4.3.2: Entering parameter substitution

In mathematical notation, the substitution looks as follows:

I'	$\mapsto \lambda$ (U,V) (S,T,M). $S \cup T = U \wedge S \cap T = \emptyset \wedge M \in$ Map(S,V)
r'_0	$\mapsto \lambda$ (U,V). ($\emptyset$, U, $[\![\,]\!]$)
Split	$\mapsto \lambda$ (U,V) (S,T,M) (S',T',M'). ($\exists$ a b. a $\in$ T $\wedge$ b $\in$ V $\wedge$ (S',T',M') = (S $\cup$ {a},T − {a}, M $\oplus$ [a $\mapsto$ b]))
Extract	$\mapsto \lambda$ N (S,T,M). T =$\emptyset \wedge$ N = M
subspace	$\mapsto$ space(λ (S,T,M) (S',T',M'). S $\subset$ S')
Satisfies	$\mapsto \lambda$ N (S,T,M). ($\forall$ x $\in$ S. N ^ x = M ^ x)

The crucial part here is to find a representation of search spaces and a suitable Split based on that representation. Our search idea is to successively extend a partial map until its domain encompasses all of U. Therefore, we model search spaces by tri–ples (S,T,M) where S and T partition U and M is a map to V with domain S. Split–ting a search space means to extend M by a new pair mapping a not yet used member a of U to some arbitrary value b of V. The operation $\oplus$ overwrites the first map on the domain of the second, and $[a \mapsto b]$ maps a exactly to b. The subspace relation on search spaces is induced by the strict subset relation on S, which we for–malise using the library function space that converts an ordering function into its graph. Given the transformation above, Isabelle computes the type of search spaces auto–matically.

$\rho' \mapsto \alpha$ set $\times \beta$ set $\times (\alpha, \beta)$ Fmap

After type checking the parameter substitution, YATS responds with the following proof state, containing the proof obligations and the synthesised "program":

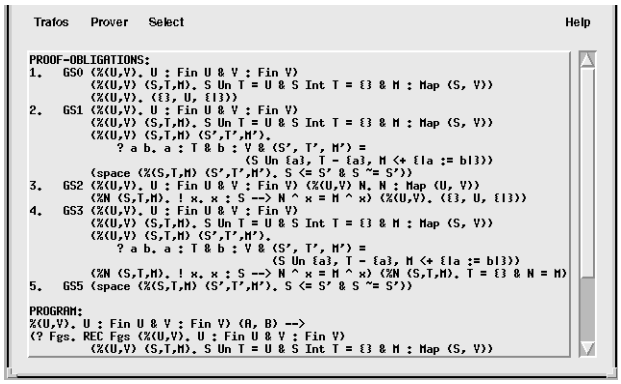

Figure 4.3.3: The resulting proof state

Now, we may use Isabelle to verify the proof obligations. At an arbitrary point of the development, we may decide to "freeze" the complete proof state i.e. convert it into an Isabelle theory containing the proof state as the implication: "if the (remaining) proof obligations hold, then the program is equivalent to the specification". Later, we can reload the frozen proof state and resume the development.

Although a small example, fin_maps demonstrates the virtues of top-down development via stepwise refinement in a transformational setting. Simply by indicating to the system which transformation shall be applied to a specification, YATS can check if the transformation is applicable, and systematically leads the user to the necessary design decisions and proof obligations.

5 Discussion

We have shown how to implement transformation systems in a systematic way that clearly separates the *soundness* issues of transformations, the *pragmatics* of their application, and their *presentation* to developers at the user interface.

For several reasons, representing the logical content of transformations by synthesis theorems highly increases the users' confidence in single transformation steps and thereby in the correctness of software they develop with the implemented system.

- Several attempts to construct a new transformation are usually needed until it is indeed expressed in a correct and useful form. Since *practically useful* transformations must capture large and complex design steps, they are difficult to conceive and implement by human developers. Here, a mechanical proof of the synthesis theorem may be useful to increase confidence in the soundness of the transformation. Moreover, a proof can be useful to *find* the final shape of a transformation.
- Separating transformations into logical core and tactical sugar clearly identifies the parts of the implementation that guarantee correctness of the resulting software. Program errors in the tactical sugar parts of the system or the user interface can in no way lead to logically incorrect results of transformations.
- It is often possible to formalise transformation concepts like "Divide-and-Conquer" in different synthesis theorems. With our approach, it is easy to relate, to specialise and to combine synthesis theorems to form new transformations.
- It is conceivable to extend our system dynamically by proving the synthesis-theorems which are sugared in a standardised way automatically.
- Different specification formalisms like CSP and Z have been represented in HOL. The representation of transformation rules as synthesis theorems provides a means to study rules in combined multi-formalism environments.

The tactical theorem prover is the core of our system. It is not only used to prove synthesis theorems but also, based on its meta-variables and deduction facilities, to implement transformation applications. We have shown that this can be done with little effort. Moreover, in [HSZ 95], we have demonstrated a systematic approach to build tactical sugar.

The user-interface description and command language Tcl/Tk has seen a tremendous success in the recent years. From our experience, the X-Window toolkit Tk seems to offer "the right abstraction" for building user interfaces. However, the design of our parameterized interface and our productivity to code it profited substantially from the embedding of Tk into SML with its typing and modularisation concepts.

5.1 Related Work

The transformational approach to program development has a long tradition. Starting from the Munich CIP project [Bau+85], many studies have stressed the importance of the approach. During the PROSPECTRA project [HK 93], a system has been developed that enabled the formalisation of transformation rules and their use during the software development process.

In KIDS [Smi 91], programs are developed by transforming *problem specifications* to programs. First, high-level transformations such as global search are used to come from the problem specification to a (inefficient) program. This program is then optimised by low-level program transformations like finite differencing or case distinction.

While the research in the context of KIDS has contributed much in the area of mathematically describing complex transformations and tactical sugar for their successful application, a shortcoming of the implemented system is that there is no easy way to convince oneself of the soundness of the implemented transformations. Our work focuses on this aspect and may thus be regarded complementary to the KIDS work.

Kreitz [Kre 93] gives mechanical proofs of global search theories in a constructive type theory, namely Nuprl [Con 86]. He aims at capturing even the pragmatics of transformation applications in a logical framework and attempts to extract synthesis tactics from the computational content of constructive proofs. In our approach, formal logic is used only to treat the soundness of transformations. This admits varying degrees of sophistication of tactical sugar: we can easily have different tactics for the same synthesis theorem. To achieve the same effect, Kreitz would have to provide different *proofs* of the theorem, each encoding a different approach to its application.

Basin's work [Bas 94] represents an approach to logic program synthesis also implemented in Isabelle. It is based on the Whelk logic that has been proposed as foundation. The rules of Whelk are derived in Isabelle. This work focuses on foundational issues rather than on a practical system implementation.

The recent formation of new workshops show a growing interest in the design and implementation of graphical user interfaces for both theorem provers and IFDSEs. A notable implementation is the TkHOLWorkbench currently developed in Cambridge [Sym 95], another one is [CO 95] for Isabelle. Although these interfaces currently are clearly superior to ours it is predominantly implemented in Tcl and not in ML. For this reason, we believe that our approach offers a higher potential of growth and reusability for similar systems.

5.2 Future Work

Our implementation to date is a prototype to illustrate the approach. To make it practically usable, two improvements have to be made: we need to extend the library of transformations and we need to incorporate a standard specification language which is used in practice.

Incorporating more standard transformations from the literature is easy. The standardised form of synthesis theorems even allows us to develop a set of "meta-transformations" — as ML functions — yielding transformations from synthesis theorems automatically. Meta-transformations might implement different ways to deal with application conditions and parameters. As we have mentioned in section 3.5, verificational approaches supplying the parameters directly and leaving application conditions as subgoals to verify are as well as possible as constructive approaches that — auto-

matically or interactively — synthesise parameters while proving application conditions. Different techniques to match an input pattern against a goal are possible.

As for the use of a standard specification language, research is going on to implement support for, e.g., Z in Isabelle. Proving suitable synthesis theorems in the logical representation of such a specification language makes our approach immediately applicable to that language. This work is partly an objective of the project UniForM [Kri+95].

Acknowledgement. We would like to thank Maritta Heisel for many discussions on synthesis theorems, and two anonymous referees for very extensive and constructive comments.

References

[And 86] Andrews, P.B., *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, 1986.

[Bas 94] Basin, D., Isawhelk: Whelk interpreted in Isabelle. Abstract accepted at the *11th International Conference on Logic Programming* (ICLP 94). Full version available via http://www.mpi-sb.mpg.de/guide/staff/basin/-pubs/iclp11.ps.Z.

[Bau+85] Bauer et al. (The CIP Language Group): *The Munich Project CIP. Volume I: The wide spectrum language CIP-L*. LNCS 183. 1985.

[BH 95] Bowen, J. P., Hinchey, M. J., Seven more Myths of Formal Methods: Dispelling Industrial Prejudices, in *FME'94: Industrial Benefit of Formal Methods*, proc. 2nd Int. Symposium of Formal Methods Europe, LNCS 873, Springer Verlag 1994, pp. 105-117.

[BM 93] Bird, R., de Moor, O.: Solving Optimisation Problems with Catamorphisms, in Proc. *Second Conference on the Mathematics of Program Construction*, LNCS 669, Springer Verlag 1993, pp. 49-56.

[Chu 40] Church, A., A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940, pp. 56-68.

[CO 95] Cant, A., Ozohls, M.A. XIsabelle: A graphical User Interface to the Isabelle Theorem Prover. ftp://ftp.cl.cam.ac.uk/ml/XIsabelle-2.0.tar.gz

[Con 86] Constable, R. S. et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[GL 93] Gersdorf, B., Liu, J. personnal communication.

[GM 93] Gordon, M.J.C., Melham, T.M.: *Introduction to HOL: a theorem proving environment for higher order logics*, Cambridge University Press, 1993.

[Gor 95] Gordon. M: Notes on PVS from a HOL perspective. Available via Internet http://www.cl.cam.ac.uk/users/mjcg/PVS.ps.gz, 1995.

[HK 93] Hoffmann, B., Krieg-Brückner, B. (eds.): *PROgram Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System.* LNCS 680, Springer-Verlag 1993.

[HMM 86] Harper, R., MacQueen, R., Milner, R: Standard ML. Technical Report ECS-LFCS-86-2. 1986.

[HSZ 95] Heisel, M., Santen, T., Zimmermann, D.: Tool Support for Formal Software Development: A Generic Architecture, in *Software Engineering — ESEC '95*, LNCS 989, Springer Verlag, 1995, pp. 272-293.

[Kre 93] Kreitz, C.: Meta-Synthesis. Deriving Programs that Develop Programs. Technische Hochschule Darmstadt, 1993.

[Kri+95] Krieg-Brückner, B., Peleska, J., Olderog, E.-R., Balzer, D., Baer, A.: Uniform Workbench — Universelle Entwicklungsumgebung für formale Methoden. Technischer Bericht 8/95, Universität Bremen, 1995.

[KW 95] Kolyang, Wolff, B.: Development by Refinement Revisited: Lessons learnt from a case study. Proc. *Softwaretechnik '95*. Software-Technik Trends, Gesellschaft für Informatik, 1995, pp. 55-66 .

[OSR 93] Owre, S., Shankar, N., Rushby, J.M.: The PVS Specification Language (Beta Release). Comp. Sci. Lab., SRI International, Menlo Park, 1993.

[Ous 94] Ousterhout, J.K.: *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[Pau 94a] Paulson, L. C.: *Isabelle - A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994.

[Pau 94b] Paulson, L. C.: A fixedpoint approach to implementing (co)inductive definitions, in Alan Bundy (ed), *12th International Conference on Automated Deduction*, LNAI 814, 1994, Springer Verlag, pp. 148-161.

[PR 95] Pessaux, F., Rouaix, F.: The Caml/Tk interface, Projet Cristal, INRIA Rocquencourt, July 1995 ftp://ftp.inria.fr/lang/../INRIA/Projects/cristal/caml-light/camltk.dvi.tar.gz

[Smi 87] Smith, D. R.: Structure and Design of Global Search Algorithms, Technical Report Kes.U.87.12, Kestrel Institute, Palo Alto, 1987.

[Smi 90] Smith, D. R.: KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering* Special Issue on Formal Methods in Software Engineering, 16(9), 1990, pp. 1024–1043.

[SPW 95] Smith, D. R., Parra, E. A., Westfold, S. J.: Synthesis of High-Performance Transportation Schedulers, Technical Report, Kestrel Institute, Palo Alto, 1995.

[Sym 95] Syme, D.: A New Interface for HOL – Ideas, Issues and Implementation in *Higher Order Logic: Theorem Proving and its Applications*, LNCS 971, Springer Verlag 1995. pp 325-339.

[VTS 95] Vullinghs,T., Tuijnman, D., Schulte, W., Lightweight GUI's for functional programming. PLILP 95, Utrecht, The Netherlands, 1995.