TAS — A Generic Window Inference System

Christoph Lüth[1] and Burkhart Wolff[2]

[1] FB 3 — Mathematik und Informatik, Universität Bremen
cxl@informatik.uni-bremen.de
[2] Institut für Informatik, Albert-Ludwigs-Universität Freiburg
wolff@informatik.uni-freiburg.de

Abstract. This paper presents work on technology for transformational proof and program development, as used by window inference calculi and transformation systems. The calculi are characterised by a certain class of theorems in the underlying logic. Our transformation system TAS compiles these rules to concrete deduction support, complete with a graphical user interface with command-language-free user interaction by gestures like drag&drop and proof-by-pointing, and a development management for transformational proofs. It is *generic* in the sense that it is completely independent of the particular window inference or transformational calculus, and can be instantiated to many different ones; three such instantiations are presented in the paper.

1 Introduction

Tools supporting formal program development should present proofs and program developments in the form in which they are most easily understood by the user, and should not require the user to adapt to the particular form of presentation as implemented by the system. Here, a serious clash of cultures prevails which hampers the wider usage of formal methods: theorem provers employ presentations stemming from their roots in symbolic logic (e.g. Isabelle uses natural deduction), whereas engineers are more likely to be used to proofs by transformation as in calculus. As a way out of this dilemma, a number of systems have been developed to support *transformational development*. However, many of these systems such as CIP [3], KIDS [21] or PROSPECTRA [12] suffered from a lack of proof support and proven correctness. On the other hand, a variety of calculi have been developed which allow formal proof in a transformational way and are proven correct [8–10, 28, 11, 2], some even with a graphical user interface [14, 6]. However, what has been lacking is a systematic, generic and reusable way to obtain a user-friendly tool implementing transformational reasoning, with an open system architecture capable of coping with the fast changes in technology in user interfaces, theorem provers and formal methods. Reusability of components is crucial, since we hope that the considerable task of developing appropriate GUIs for formal method tools can be shared with other research groups.

In [15], we have proposed an open architecture to build graphical user interfaces for theorem provers in a functional language; here, we instantiate this architecture with a generic transformation system which implements transformational calculi (geared towards refinement proofs) on top of an LCF-like prover. By *generic*, we mean that the system takes a high-level characterisation of a refinement calculus and returns a user-friendly, formally correct transformation or window inference system. The system can be used for various object logics and formal methods (a property for which Isabelle is particularly well suited as a basis). The instantiation of the system is very straightforward once the formal method (including the refinement relation) has been encoded. Various aspects of this overall task have been addressed before, such as logical engines, window-inference packages and prototypical GUIs. In contrast, TAS is an *integrated* solution, bringing existing approaches into one technical framework, and filling missing links like a generic pretty-printer producing markups in mathematical text.

This paper is structured as follows: in Sect. 2 we give an introduction to window inference, surveying previous work and presenting the basic concepts. We explain how the formulation of the basic concepts in terms of ML theorems leads to the implementation of TAS. We demonstrate the versatility of our approach in Sects. 3, 4 and 5 by showing examples of classical transformational program development, for process-oriented refinement proofs and for data-oriented refinement proofs. Sect. 6 finishes with conclusions and an outlook.

2 A Generic Scheme of Window Inference

Window inference [18], structured calculational proof [8, 1, 2] and transformational hierarchical reasoning [11] are closely related formalisations of proof by transformation. In this paper, we will use the format of [1], although we will refer to it as window inference.

2.1 An Introduction to Window Inference

As motivating example, consider the proof for $\vdash (A \wedge B \Rightarrow C) \Rightarrow (B \wedge A \Rightarrow C)$. In natural deduction, a proof would look like (in the notation of [27]; we assume that the reader is roughly familiar with derivations like this):

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[B\wedge A]^1}{A}\wedge E \quad \cfrac{[B\wedge A]^1}{B}\wedge E}{A\wedge B}\wedge I \qquad [A\wedge B\Rightarrow C]^2}{C}\Rightarrow E}{B\wedge A\Rightarrow C}\Rightarrow I_1}{(A\wedge B\Rightarrow C)\Rightarrow(B\wedge A\Rightarrow C)}\Rightarrow I_2 \tag{1}$$

The following equivalent calculational proof is far more compact. We start with $B \wedge A \Rightarrow C$. In the first step, we open a *subwindow* on the sub-expression $B \wedge A$, denoted by the markers. We then transform the sub-window and obtain the desired result for the whole expression:

$$
\begin{array}{ll}
 & \llcorner B \wedge A \lrcorner \Rightarrow C \\
\Leftarrow & \{\text{focus on } B \wedge A\} \\
 & \quad \bullet\ B \wedge A \\
 & \quad = \{\wedge \text{ is commutative}\} \\
 & \qquad A \wedge B \\
\cdot & \ulcorner A \wedge B \urcorner \Rightarrow C
\end{array} \tag{2}
$$

The proof profits from the fact that we can replace equivalent subexpressions. This is formalised by *window rules* [11]. In this case the rule has the form

$$\frac{\Gamma \vdash A = B}{\Gamma \vdash E[A] \Rightarrow E[B]} \tag{3}$$

where the second-order variable E stands for the unchanged *context*, while the subterm A (the *focus* of the transformation) is replaced by the transformation.

Comparing this proof with the natural deduction proof, we see that in the latter we have to decompose the context by applying one rule per operator, whereas the calculational proof employs second-order matching to achieve the same effect directly. Although in this format, which goes back to Dijkstra and Scholten [8], proofs tend to be shorter and more abstract, there are known counterexamples such as proof by contradiction.

In Grundy's work [11], window inference proofs are presented in terms of natural deduction proofs. By showing every natural deduction proof can be constructed using window inference rules, completeness of window inference for first-order logic is shown. This allows the implementation of window inference in a theorem prover. A similar technique underlies our implementation: the system constructs Isabelle proofs from window inference proofs.

As was shown in [11, 1], window inference proofs are not restricted to first-order logic or standard proof refinement, i.e. calculational proofs based on the implication and equality. It is natural to admit a family $\{R_i\}_{i \in I}$ of reflexive and transitive binary relations that enjoy a generalised form of monotonicity (in the form of (3) above).

Extending the framework of window inference in these directions allows to profit from its intuitive conciseness not only in high-school mathematics and traditional calculus, which deals with manipulating equations, but also in formal systems development, where the refinement of specifications is often the central notion. However, adequate user interface support is needed if we want to exploit this intuitive conciseness; the user interaction to set a focus on a subterm should be little more than marking the subterm with the mouse (point&click), otherwise the whole beneficial effect would be lost again.

2.2 The Concepts

Just as equality is at the heart of algebra, at the heart of window inference there is a family of binary preorders (reflexive and transitive relations) $\{\sqsubseteq_i\}_{i \in I}$. These preorders are called the *refinement relations*. Practically relevant examples of refinement relations in formal system development are impliedness $S \Leftarrow P$ (used for algebraic model inclusion, see Sect. 3), process refinement $S \sqsubseteq_{FD} P$ (the process P is more defined and more deterministic than the process S, see Sect. 4), set inclusion (see Sect. 5), or arithmetic orderings for numerical approximations [29]. An example for an infinite family of refinement relations in HOL is the Scott-definedness ordering for higher-order function spaces (where the indexing set I is given by the types):

$$f \sqsubseteq_{(\alpha\to\beta)\times(\alpha\to\beta)\to Bool} g \;\equiv\; \forall x.\, f\, x \sqsubseteq_{\beta\times\beta\to Bool} g\, x \tag{4}$$

The refinement relations have to satisfy a number of properties, given as a number of theorems. Firstly, we require reflexivity and transitivity for all $i \in I$:

$$a \sqsubseteq_i a \qquad [\mathrm{Refl}_i]$$
$$a \sqsubseteq_i b \wedge b \sqsubseteq_i c \Rightarrow a \sqsubseteq_i c \qquad [\mathrm{Trans}_i]$$

The refinement relations can be ordered. We say $\sqsubseteq_i$ is *weaker* than $\sqsubseteq_j$ if $\sqsubseteq_i$ is a subset of $\sqsubseteq_j$, i.e. if $a \sqsubseteq_i b$ implies $a \sqsubseteq_j b$:

$$a \sqsubseteq_i b \Rightarrow a \sqsubseteq_j b \qquad [\mathrm{Weak}_{i,j}]$$

The ordering is optional; in a given instantiation, the refinement relations may not be related at all. However, because of reflexivity, equality is weaker than any other relation, i.e. for all $i \in I$, the following is a derived theorem:[1]

$$a = b \Rightarrow a \sqsubseteq_i b \tag{5}$$

The main device of window inferencing are the window rules shown in the previous section:

$$(A \Rightarrow a \sqsubseteq_i b) \Rightarrow F\, a \sqsubseteq_j F\, b \qquad [\mathrm{Mono}^F_{i,j}]$$

Here, F can either be a meta-variable[2], or a constant-head expression, i.e. a term of the form $\lambda y_1 \ldots y_m.c x_1 \ldots x_n$ with c a constant. Note how there are different refinement relations in the premise and conclusion of the rule. Using a family of rules instead of one monotonicity rule has two advantages: firstly, it allows us to handle, on a case by case basis, instantiations where the refinement relations are not congruences, and secondly, by allowing an additional assumption A in the monotonicity rules, we get more assumptions when refining inside a context. These *contextual assumptions* are crucial, many proofs depend on them.[3]

[1] In order to keep our transformation system independent of the object logic being used, we do not include any equality per default, as different object logics may have different equalities.

[2] In Isabelle, meta-variables are variables in the meta-logic, which are subject to unification. Users of other theorem provers can think of them just as variables.

[3] They already featured in the pioneering CIP-S system [3] in 1984.

Dependencies between refinement relations can be more complicated than the restricted form of weakening rules [$\mathrm{Weak}_{i,j}$] above may be able to express; for example, (4) cannot be expressed by a weakening rule in either direction because of the outermost quantor on the right side. For this reason, there is a further need for *refinement conversions*, i.e. tactical procedures that attempt to rewrite one refinement proof goal into another.

To finish off the picture, we consider transformation rules. A transformation rule is given by a *logical core theorem* of the form

$$A \Rightarrow (I \sqsubseteq_j O) \tag{6}$$

where A is the *application condition*, I the *input pattern* and O the *output pattern*. In other words, transformation rules are theorems the conclusion of which is a refinement relation.

2.3 Parameters

The *parameters* for a transformation rule given by core theorem schema (6) are meta-variables occuring in the output pattern O but not in the input pattern I. After applying the transformation, a parameter occurs as a free meta-variable in the proof state. This is not always useful, hence parameters enjoy special support. In particular, in transformational program development (see Sect. 3) we have rather complex transformations with a lot of parameters and their instantiation is an important design decision. As a simple example, consider the theorem

$$t \Leftrightarrow \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t$$

which as a transformation rule from the left to the right introduces a case distinction on b. This is not very helpful unless we supply a concrete value for b which helps us to further develop t in the two different branches of the conditional expression under the respective assumption that b holds, or does not.

TAS supports parameters by when applying a transformation checking whether it contains parameters, and if so querying for their instantiation. It further allows parameter instantiations to be stored, edited and reused. This avoids having to retype instantiations, which can get quite lengthy, and makes TAS suitable for transformational program development as well as calculational proof.

2.4 The `Trafos` package

The `Trafos` package implements the basic window inferencing operations as Isabelle tactics, such as:

- opening and closing subwindows,
- applying transformations,
- searching for applicable transformations,
- and starting and concluding developments.

In general, our implementation follows Staples' approach [23], for example in the use of the transitivity rules to translate the forward chaining of transformation steps into backwards proofs on top of Isabelle's goal package, or the reflexivity rules to close subwindows or conclude developments.The distinctive features of our implementation are the subterm and search functionalities, so we concentrate on these in the following.

In order to open a subwindow or apply a transformation at a particular subterm, `Trafos` implements an abstract datatype `path` and operations `apply_trafo`, `open_sub` taking such a path (and a transformation) as arguments. To allow direct manipulation by point&click, we extend Isabelle's powerful syntax and pretty-printing machinery by *annotations* [15]. Annotations are markup sequences containing a textual representation of the path, which are attached to the terms. They do not print in the user interface, but instead generate a binding which invokes the respective operations with the corresponding path as argument. In general, users do not need to modify their theories to use the subterm selection facilities, they can be used as they are, including user-defined pretty-printing.[4]

The operations `apply_trafo` and `open_sub` analyse the context, and for each operation making up the context, the most specific [Mono_i^F] rule is selected, and a proof step is generated. In order to speed up this selection, the monotonicity rules are indexed by their head symbol, so we can discard rules which cannot possibly unify; still, the application of the selected rules may fail, so a tactic is constructed which tries to apply any combination of possibly fitting rules, starting with the most specific.

Further, for each refinement relation $\sqsubseteq_i$, we try to find a rule [$\mathrm{Mono}_{i,i}^F$] where F is just a meta-variable and the condition A is void — this rule would state that $\sqsubseteq_i$ is a congruence. If we can find such a rule, we can use it to handle, in one step, large parts of the context consisting of operations for which no more specific rule can be found. If no such congruence rule can be found, we do not construct a step-by-step proof but instead use Isabelle's efficient rewriter, the simplifier, with the appropriate rules to break down larger contexts in one step.

As an example why the more specific rules are applied first, consider the expression $E = x + (\texttt{if}\ x = 0\ \texttt{then}\ u + x\ \texttt{else}\ v + x)$. If we want to simplify $u+x$, then we can do so under the assumption that $x = 0$, and we have $x + 0 \Rightarrow u + x = u$ because of the theorem

$$(B \Rightarrow x = y) \Rightarrow (\texttt{if}\ B\ \texttt{then}\ x\ \texttt{else}\ z = \texttt{if}\ B\ \texttt{then}\ y\ \texttt{else}\ z) \qquad [\mathrm{Mono}_{=}^{\texttt{If}}]$$

But if we had just used the congruence rule for equality $x = y \Rightarrow f\,x = f\,y$ we would have lost the contextual assumption $x = 0$ in the refinement of the if-branch of the conditional.

When looking for applicable transformations, performance becomes an issue, and there is an inherent trade-off between the speed and accuracy of the search. In principle, we have to go through all theorems in Isabelle's database and check

[4] Except if Isabelle's freely programmable so-called *print translations* are used (which is rarely the case). In this case, there are facilities to aid in programming markup-generation analogously to these print-translations.

whether they can be considered as transformation rule, and if so if the input pattern of the rule matches. Many theorems can be excluded straight away since their conclusion is not a refinement. For the rest, we can either superficially check whether they might fit, which is much faster but bears the risk of returning rules which actually do not fit, or we can construct and apply the relevant tactic. We let users decide (by setting a search option) whether they want fast or accurate search. Another speed-up heuristic is to be able to specify that rules are only collected from certain theories (called *active theories*). Finally, users can exclude expanding rules (where the left-hand side is only a variable), because most (but not all) of the time these are not really helpful. In this way, users can guide the search for applicable transformations by selecting appropriate heuristics.

When instantiating the functor `Trafos`, the preprocessing of the monotonicity rules as described above takes place (calculation of the simplifier sets, head constants etc.) Further, some consistency checks are carried out (e.g. that there are transitivity and reflexivity rules for all refinement relations).

2.5 Genericity by Functors

In Standard ML (SML), modules are called *structures*. *Signatures* are module types, describing the interface, and *functors* are parameterised modules, mapping structures to structures. Since in LCF provers theorems are elements of an abstract SML datatype, we can describe the properties of a window inference calculus as described in Sect. 2.2 above using SML's module language, and implement TAS a functor, taking a structure containing the necessary theorems, and returning a transformation or window inferencing system complete with graphical user interface built on top of this:

```
  functor TAS(TrfThy: TRAFOTHY) = ...
```

The signature `TRAFOTHY` specifies a structure which contains all the theorems of Sect. 2.2. Abstracted a little (by omitting some parameters for special tactical support), it reads as follows:

```
signature TRAFOTHY =
    sig val topthy   : string
        val refl     : thm list
        val trans    : thm list
        val weak     : thm list
        val mono     : thm list
        val ref_conv : (string* (int-> tactic)) list
        ...
    end
```

To instantiate TAS, we need to provide a theory (named `topthy`) which encodes the formal method of our choice and where our refinement lives, theorems describing the transitivity, reflexivity and monotonicity of the refinement relation(s), and a list of refinement conversions, which consist of a name, and a tactic when when applied to a particular subgoal converts the subgoal into another refinement relation.

When applying this functor by supplying appropriate arguments, we obtain a structure which implements a window inferencing system, complete with a graphical user interface. The graphical user interface abstracts from the command line interface of most LCF provers (where functions and values are referred to by names) by implementing a *notepad*, on which objects (theorems, theories, etc.) can be manipulated by drag&drop. It provides a *construction area* where the current on-going proof is displayed, and which has a *focus* to open subwindows, apply transformations to subterms or search the theorem database for applicable transformations. We can navigate the *history* (going backwards and forwards), and display the history concisely, or in detail through an active display, which allows us to show and hide subdevelopments. Further, the user interface provides an active *object management* (keeping track of changes to external objects like theories), and a *session management* which allows to save the system state and return to it later. All of these features are available for *any* instance of TAS, and require no additional implementation; and this is what we mean by calling TAS *generic*.

The implementation of `TAS` consists of two components: a kernel transformation system, which is the package `Trafos` as described in Sect. 2.4, and a graphical user interface on top of this. We can write this simplified as

```
functor TAS(TrfThy : TRAFOTHY) = GenGUI(Trafos(TrfThy : TRAFOTHY))
```

The graphical user interface is implemented by the functor `GenGUI`, and is independent of `Trafos` and Isabelle. For a detailed description, we refer to [15], but in a nutshell, the graphical user interface is implemented entirely in SML, using a typed functional encapsulation of Tcl/Tk called `sml_tk`. Most of the GUI features mentioned above (such as the notepad, and the history, object and session management) are implemented at this more general level.

The division of the implementation into a kernel system and a generic graphical user interface has two major advantages: firstly, the GUI is reusable for similar applications (for example, we have used it to implement a GUI `IsaWin` to Isabelle itself); and secondly, it allows us to run the transformation system without the graphical user interface, e.g. as a scripting engine to check proofs.

3 Design Transformations in Classical Program Transformation

In the design of algorithms, certain schemata can be identified [7]. When such a schema is formalised as a theorem in the form of (6), we call the resulting transformation rule a *design transformation*. Examples include *divide and conquer* [20], *global search* [22] or *branch and bound*. Recall from Sect. 2.2 that transformation rules are represented by a logical core theorem with an input pattern and an output pattern. Characteristically, design transformations have as input pattern a *specification*, and as output pattern a *program*.

Here, a specification is given by a pre- and a postcondition, i.e. a function $f : X \to Y$ is specified by an implication $Pre(x) \longrightarrow Post(x, f(x))$, where $Pre : X \to Bool, Post : X \times Y \to Bool$. A program is given by a recursive scheme, such as well-founded recursion; the proof of the logical core theorem must accordingly be based on the corresponding induction principles, i.e. here well-founded induction. Thus, a function $f : X \to Y$ can be given as

$$\texttt{let fun } f(x) = E \texttt{ in } f \texttt{ end measure} < \tag{7}$$

where E is an expression of type Y, possibly containing f, and $< \subseteq X \times X$ is a well-founded relation, the *measure*, which must decrease with every recursive call of f. The notational proximity of (7) to SML is intended: (7) can be considered as a functional program.

As refinement relation, we will use model-inclusion — when refining a specification of some function f, the set of possible interpretations for f is reduced. The logical equivalent of this kind of refinement is the implication, which leads to the following definition:

$$\sqsubseteq : Bool \times Bool \to Bool \qquad P \sqsubseteq Q \stackrel{def}{=} Q \longrightarrow P$$

Based on this definition, we easily prove the theorems `ref_trans` and `ref_refl` (transitivity and reflexivity of $\sqsubseteq$). We can also prove that $\sqsubseteq$ is monotone for all boolean operators, e.g.

$$s \sqsubseteq t \Rightarrow s \wedge u \sqsubseteq t \wedge u \qquad\qquad \texttt{ref\_conj1}$$

Most importantly, we can show that

$$(B \Rightarrow s \sqsubseteq t) \Rightarrow \texttt{if } B \texttt{ then } s \texttt{ else } u \sqsubseteq \texttt{if } B \texttt{ then } t \texttt{ else } u \qquad \texttt{ref\_if}$$
$$(\neg B \Rightarrow u \sqsubseteq v) \Rightarrow \texttt{if } B \texttt{ then } s \texttt{ else } u \sqsubseteq \texttt{if } B \texttt{ then } s \texttt{ else } v \qquad \texttt{ref\_then}$$

which provides the contextual assumptions mentioned above. When instantiating the functor, we also have to specify equality as a refinement relation. Since we can reuse the relevant definitions for all theories based on HOL, they have been put in a separate functor `functor HolEqTrfThy(TrfThy : TRAFOTHY) : TRAFOTHY` In particular, this functor proves the weakening theorems (5) for all refinement relations, and appends them to the list `weak`. Thus, the full functor instantiation reads

```
structure HolRefThy =
  struct val name  = "HolRef"
         val trans = [ref_trans]
         val refl  = [ref_refl]
         val weak  = []
         val mono  = [ref_if, ref_else, ref_conj1, ref_conj2,
                      ref_disj1, ref_disj2, ...]
         val ref_conv = []
         ...
  end
structure TAS = TAS(HolEqTrfThy(HolRefThy))
```

The divide and conquer design transformation [20] implements a program $f : X \to Y$ by splitting X into two parts: the termination part of f, which can be directly embedded into the codomain Y of f, and the rest, where the values are divided into smaller parts, processed recursively, and reassembled. The core theorem for divide and conquer based on model-inclusion refinement and well-founded recursion reads:[5]

$$
\begin{array}{ll}
A \longrightarrow & (Pre(x) \longrightarrow Post(x, f(x)) \\
& \sqsubseteq \\
& Pre(x) \longrightarrow f = \texttt{let fun}\ F(x) = \texttt{if}\ isPrim(x)\ \texttt{then}\ Dir(x) \\
& \qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ Com(\langle G, F\rangle(Decom(x))) \\
& \qquad\qquad\quad \texttt{in}\ F\ \texttt{end measure} <)
\end{array}
\tag{8}
$$

As explained above, the parameters of the transformation are the meta-variables appearing in the output pattern but not in the input pattern of the logical core theorem (8). Here, these are

- the termination criterion $isPrim : X \to Bool$;
- the embedding of terminal values $Dir : X \to Y$;
- the decomposition function of input values $Decom : X \to Z \times X$;
- a function $G : Z \to U$ for those values which are not calculated by recursive calls of F;
- the composition function $Com : U \times Y \to Y$ that joins the subsolutions given by G and recursive calls of F;
- and the measure $<$ assuring termination.

We will now apply this transformation to synthesise a sorting algorithm in the theory of lists. We start with the usual specification of *sort*, as shown on the left of Fig. 1. We can see the notepad, on which the transformation object `Divide & Conquer` is represented by an icon. The workspace shows the current state of the already started development. The highlighting indicates the focus set by the user. Now we drag the transformation onto the focus; TAS interprets this gesture as application of the transformation at the focus. In this case, TAS infers that there are parameters to be provided by the user, who is thus guided to the necessary design decisions. The parameter instantiations are fairly simple: the termination condition is the empty list, which is sorted (hence *Dir* is the identity). The decomposition function splits off the head and the tail; the tail is sorted recursively, and the head is inserted into the sorted list (hence, G is the identity). Finally, the measure relates non-empty lists to their tails (since the recursive call always passes the tail of the argument; a relation easily proven to be well-founded).

This transformation step readily produces the desired program (right of Fig. 1). However, this step is only valid if the application conditions of the transformation hold. When applying a transformation, these conditions turn into proof obligations underlying a special bookkeeping. The proof obligations

[5] $\langle f, g\rangle$ is the pairing of functions defined as $\langle f, g\rangle(x, y) \stackrel{def}{=} (f(x), f(y))$.

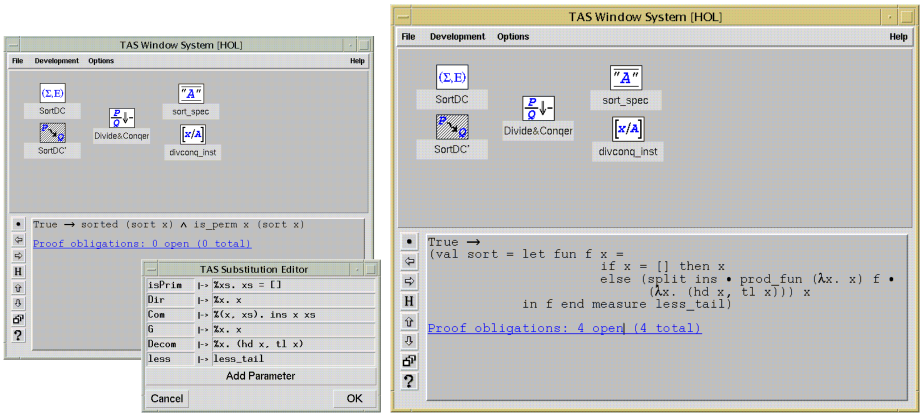

Fig. 1. TAS and its graphical user interface. To the left, the initial stage of the development, and the parameters supplied for the transformation; to the right, the development after applying the divide and conquer transformation. On the top of the window, we can see the notepad with the theory `SortDC`, the transformation `Divide&Conquer`, the specification `sort_spec`, the ongoing development (shaded) and the parameter instantiation `divconq_inst`.

can be proven with a number of proof procedures. Typically, these include automatic proof via Isabelle's simplifier or classical reasoner and interactive proof via IsaWin. Depending on the particular logic, further proof procedures may be at our disposal, such as specialised tactics or model-checkers integrated into Isabelle.

Another well-known scheme in algorithm design is *global search* which has been investigated formally in [22]. It represents another powerful design transformation which has already been formalised in an earlier version of TAS [13].

4 Process Modelling with CSP

This section shows how to instantiate TAS for refinement with CSP [19], and will briefly present an example how the resulting system can be used. CSP is a language designed to describe systems of interacting components. It is supported by an underlying theory for reasoning about their equivalences, and in particular their refinements. In this section, we use the embedding HOL-CSP [26] of CSP into Isabelle/HOL. Even though shortage of space precludes us the set out the basics of CSP here, a detailed understanding of CSP is not required in the following; suffice it to say that CSP is a language to model distributed programs as communicating processes.

CSP is interesting in this context because it has three refinement relations, namely trace refinement, failures refinement and failures-divergence refinement. Here, we only use the third, since it is the one most commonly used when devel-

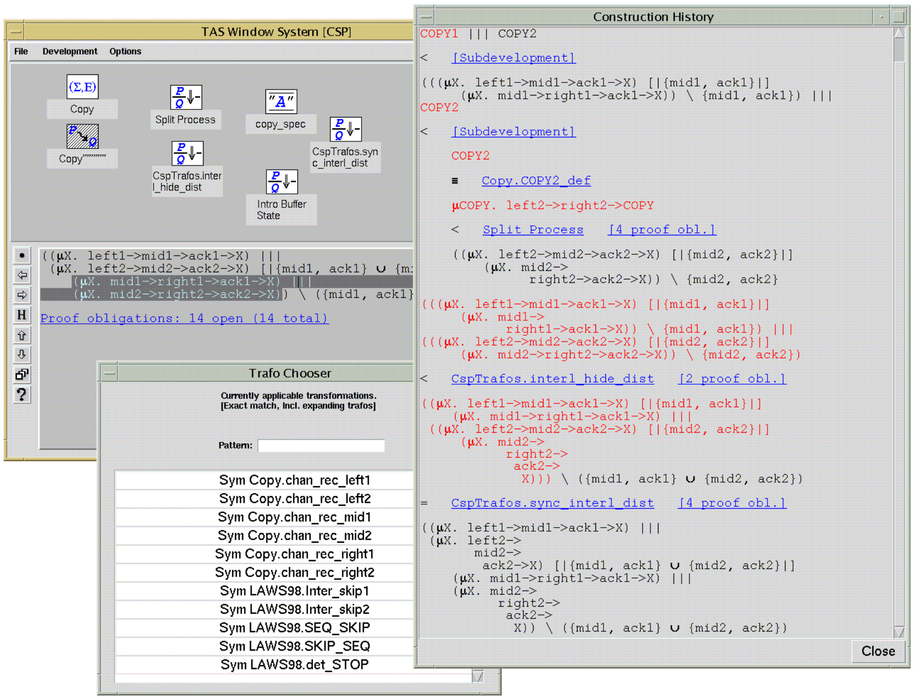

Fig. 2. TAS in the CSP instance. On the right, the construction history is shown. The development proceeded by subdevelopments on `COPY1` and `COPY2`, which can be shown and hidden by clicking on `[Subdevelopment]`. Similarly, proof obligations can be shown and hidden. In the lower part of the main window, the focus is set on a subterm, and all applicable transformations are shown. By clicking on the name of the transformations, their structure can be displayed (not shown).

oping systems from specifications, but e.g. trace refinement can be relevant to show security properties.

Recall from Sect. 2.5 that to instantiate TAS we need a theory encoding our formal method, and theorems describing the refinement relation. The relevant theory is called `CspTrafos`, which contains the core theorems of some (simple) transformations built on top of `Csp`, the encoding of CSP into Isabelle/HOL.

For brevity, we only describe instantiation with failure-divergence refinement; the other two refinements would be similar. The theorems stating transitivity and reflexivity of failure-divergence refinement are called `ref_ord_trans` and `ref_ord_refl`, respectively. For monotonicity, we have a family of theorems describing monotonicity of the operators of CSP over this relation, but since the relation is monotone only with respect to the CSP relations it is not a proper congruence. This gives us the following functor instantiation:

```
structure CspRefThy = struct
```

```
  val name  = "CspTrafos"
  val trans = [ref_ord_trans]
  val refl  = [ref_ord_refl]
  val mono  = [mono_mprefix_ref,mono_prefix_ref,mono_ndet_ref,
               mono_det_ref,mono_Ren_ref,mono_hide_set_ref,
               mono_PaI_ref,mono_Inter_ref]
  val weak  = []
  val ref_conv = []
  ...
  end
structure TAS = TAS(HolEqTrfThy(CspRefThy))
```

Fig. 2 shows the resulting, instantiated system in use. We can see an ongoing development on the left, and the opened construction history showing the development up to this point on the left. As we see, the development started with two processes in parallel; we focussed on both of these in turn to develop them separately, and afterwards rearranged the resulting process, using algebraic laws of CSP such as **sync_interl_dist** which states the distributivity of synchronisation over interleaving under some conditions. The development does not use powerful design transformations as in Sect. 3, but just employs a couple of the algebraic laws of CSP, showing how we can effectively use previously proven theorems for transformational development. Finding design transformations like divide and conquer for CSP is still an open research problem.

If we restrict ourselves to finite state processes (by requiring that the channels only carry finite messages), then we can even check the development above with the CSP model checker FDR [19], connected to Isabelle as a so-called *oracle* (a trusted external prover). This speeds up development at the cost of generality and can e.g. be used for rapid prototyping.

5 Data Refinement in the Refinement Calculus

In this section, we will emphasise a particular aspect of the genericity of TAS and demonstrate its potential for reuse of given logical embeddings. As we mentioned, TAS is generic with respect to the underlying refinement calculus, which in particular means that it is generic with respect to the underlying object logic. In the previous examples, we used higher-order logic (as encoded in Isabelle/HOL); in this example, we will use Zermelo-Fränkel set theory (as encoded in Isabelle/ZF). On top of Isabelle/ZF, Mark Staples has built a substantial theory for imperative program refinement and data refinement [24, 25] following the lines of Back's Refinement Calculus RC [2].

RC is based on a weakest precondition semantics, where predicates and predicate transformers are represented as sets of states and functions taking sets of states to sets of states respectively. The distinctive feature of Staples' work over previous implementations of refinement calculi is the use of sets in the sense of ZF based on an open type universe. This allows derivations where the types of program variables are unknown at the beginning, and become more and more concrete after a sequence of development steps.

In order to give an idea of Staples' formalisation, we very briefly review some of the definitions of Back's core language in his presentation:[6]

$$
\begin{aligned}
\mathtt{Skip}_{\mathtt{A}} &\stackrel{def}{=} \lambda q : \mathbb{P}(\mathtt{A}).q \\
a \ ; \ b &\stackrel{def}{=} \lambda q : dom(b).a \text{ ` } b \text{ ` } q \\
\mathtt{if}\ g\ \mathtt{then}\ a\ \mathtt{else}\ b\ \mathtt{fi} &\stackrel{def}{=} \lambda q : dom(a) \cup dom(b). \\
&\qquad (g \cap a \text{ ` } q) \cup ((\bigcup(dom(a) \cup dom(b)) - g) \cap b \text{ ` } q) \\
\mathtt{while}\ g\ \mathtt{do}\ c\ \mathtt{od} &\stackrel{def}{=} \lambda q : \mathbb{P}(A).\, lfp_A\, N.(g \cap c \text{ ` } N) \cup ((A - g) \cap q) \\
&\ldots
\end{aligned}
$$

This theory could be used for an instantiation of TAS, called TAS/RC. The instantiation follows essentially the lines discussed in the previous sections; with respect to the syntactic presentation, the configuration for the pretty-printing engine had to provide special support for 5 print-translations comprising 100 lines of code, and a particular set-up for the tactics providing reasoning over well-typedness, regularity and monotonicity. (We omit the details here for space reasons). As a result, a larger case study in [24] for the development of an BDD-related algorithm as a data-refinement from truth tables to decision trees can be represented *inside* TAS.

6 Conclusions and Outlook

This paper has presented the transformation system TAS. TAS is generic in the sense that it takes a set of theorems, describing a refinement relation, and turns them into a window inference or transformation system, complete with an easy-to-use, graphical user interface. This genericity means that the system can be instantiated both to a transformation system for transformational program development in the vein of traditional transformation systems such as CIP, KIDS or PROSPECTRA, or as system for window inference. We have demonstrated this versatility by showing instantiations from the provenance of each the two areas just mentioned, complemented with an instantiation from a different area, namely reasoning about processes using CSP.

The effort required for the actual instantiation of TAS is very small indeed, since merely the values for the parameters of the functor need to be provided. (Only rarely will tactical programming be needed, such as mentioned in Sect. 5, and even then it only amounts to a few lines of code.) It takes far more effort to set up the logical encoding of the formal method, in particular if one does so conservatively.

TAS' graphical user interface complements the intuitiveness of transformational calculi with a command-language-free user interface based on gestures

[6] Note that the backquote operator ` is infix function application in Isabelle/ZF.

such as drag&drop and proof-by-pointing. It further provides technical infrastructure such as development management (replay, reuse, history navigation), object management and session management.

TAS is implemented on top of the prover Isabelle, such that the consistency of the underlying logics and its rules can be ensured by the LCF-style architecture of Isabelle and well-known embedding techniques. It benefits further from the LCF architecture, because we can use SML's structuring mechanisms (such as functors) to implement reusable, generic proof components across a wide variety of logics.

Internally, we spent much effort to organise TAS componentwise, easing the reuse of as much code as possible for completely different logical environments. The GUI and large parts of TAS (except the package `Trafos`) are designed to work with a different SML-based prover, and are readily available for other research groups to provide GUI support for similar applications. On the other hand, the logical embeddings (such as HOL-CSP) which form the basis of the transformation calculi do not depend on TAS either. This allowed the easy integration of Staples' encoding of the refinement calculus into our system, as presented in Sect. 5.

6.1 Discussion and Related Work

This work attempts to synthesise previous work on transformational program development [3, 21, 12] which developed a huge body of formalised developments and design schemes, but suffered from ad-hoc, inflexible calculi, correctness problems and lack of proof support, with the work on window inferencing [18, 11] and structured calculational proof [2, 1], which provides proven correctness by LCF design and proof support from HOL or Isabelle.

PRT [6] is a program refinement tool (using window inference) which is built on top of the Ergo theorem prover. It offers an interface based on Emacs, which allows development management and search functionalities. However, the Tk-WinHOL system [14] comes closest to our own system conception: it is based on Tcl/Tk (making it platform independent), and offers focusing with a mouse, drag&drop in transformational goals, and a formally proven sound calculus implemented by derived rules in HOL. On the technical side it uses Tcl directly instead of an encapsulation (which in our estimate will make it much harder to maintain). On the logical side, it is also generic in the sense that it can be used with different refinement relations, but requires more work to be adapted to a new refinement relation; for example, users need to provide a pretty-printer which generates the correct mark-up code to be able to click on subterms. In contrast, TAS extends Isabelle's infrastructure (like the pretty-printer) into the graphical user interface, leaving the user with less work when instantiating the system.

The essential difference between window inferencing and structured calculational proof [1] is that the latter can live with more than one transformational goal. This difference is not that crucial for TAS since it can represent more than one transformational development on the notepad and is customisable for appropriate interaction between them via drag&drop operations.

Another possible generalisation would be to drop the requirement that all refinement relations be reflexive. However, this would complicate the tactical programming considerably without offering us perceivable benefit at the moment, so we have decided against it.

6.2 Future Work

Future work can be found in several directions. Firstly, the user interaction can still be improved in a variety of ways. Although in the present system, the user can ask for transformations which are applicable, this can considerably be improved by a best-fit strategy and, for example, stronger matching algorithms like AC-matching. The problem here is to help the user to find the few interesting transformations in the multitude of uninteresting (trivial, misleading) ones. Supporting design decisions at the highest possible user-oriented level must still count as an open problem, in particular in a generic setting.

Secondly, the interface to the outside world can be improved. Ideally, the system should interface to a variety of externally available proof formats, and export web-browsable proof scripts.

A rather more ambitious research goal is the reuse and abstraction of transformational developments. A first step in this direction would be to allow to cut&paste manipulation of the history of a proof.

Thirdly, going beyond classical hierarchical transformational proofs the concept of *indexed window inferencing* [29] appears highly interesting. The overall idea is to add an additional parameter to the refinement relation that allows to calculate the concrete refinement relation on the fly during transformational deduction. Besides the obvious advantage of relaxing the requirements to refinement relations to irreflexive ones (already pointed out in [23]), indexed window inferencing can also be used for a very natural representation of operational semantics rules. Thus, the system could immediately be used as an animator for, say, CSP, given the operational semantics rules for this language.

Finally, we would like to see more instances for TAS. Transformational development and proof in the specification languages Z and CASL should not be too hard, since for both embeddings into Isabelle are available [13, 16]. The main step here is to formalise appropriate notions of refinement. A rather simple different instantiation is obtained by turning the refinement relation around. This amounts to *abstracting* a concrete program to a specification describing aspects of its behaviour, which can then be validated by a model-checker. For example, deadlock checks using CSP and FDR have been carried out in this manner, where the abstraction has been done manually[4, 5, 17]. Thus we believe that TAS represents an important step towards our ultimate goal of a transformation system which is similarly flexible with respect to underlying specification languages and refinement calculi as Isabelle is for conventional logical calculi.

Acknowledgements We would like to thank Mark Staples and Jim Grundy for providing us with the sources for their implementations of window inference and the refinement calculus respectively, and the anonymous referees for threir constructive criticism. Ralph Back pointed out several weaknesses of a previous version of TAS and made suggestions for improvements.

References

1. R. Back, J. Grundy, and J. von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9:467–483, 1997.
2. R.-J. Back and J. von Wright. *Refinement Calculus*. Springer Verlag, 1998.
3. F. L. Bauer. *The Munich Project CIP. The Wide Spectrum Language CIP-L.* Number 183 in LNCS. Springer Verlag, 1985.
4. B. Buth, J. Peleska, and H. Shi. Combining methods for the deadlock analysis of a fault-tolerant system. In *Algebraic Methodology and Software Technology AMAST'97*, number 1349 in LNCS, pages 60–75. Springer Verlag, 1997.
5. B. Buth, J. Peleska, and H. Shi. Combining methods for the livelock analysis of a fault-tolerant system. In *Algebraic Methodology and Software Technology AMAST'98*, number 1548 in LNCS, pages 124–139. Springer Verlag, 1999.
6. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A Program Refinement Tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and New York: McGraw-Hill, 1989.
8. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1990.
9. D. Gries. *A Science of Programming*. Springer Verlag, 1981.
10. D. Gries. Teaching calculation and discrimination: A more effecticulum. *Communications of the ACM*, 34:45–54, 1991.
11. J. Grundy. Transformational hierarchical reasoning. *Computer Journal*, 39:291–302, 1996.
12. B. Hoffmann and B. Krieg-Brückner. *PROSPECTRA: Program Development by Specification and Transformation*. Number 690 in LNCS. Springer Verlag, 1993.
13. Kolyang, T. Santen, and B. Wolff. Correct and user-friendly implementations of transformation systems. In M. C. Gaudel and J. Woodcock, editors, *Formal Methods Europe FME'96*, number 1051 in LNCS, pages 629– 648. Springer Verlag, 1996.
14. T. Långbacka, R. Rukšėna, and J. von Wright. TkWinHOL: A tool for doing window inferencing in HOL. In *Proc.* 8^{th} *International Workshop on Higher Order Logic Theorem Proving and Its Applications*, number 971 in LNCS, pages 245–260. Springer Verlag, 1995.
15. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167– 189, March 1999.
16. T. Mossakowski, Kolyang, and B. Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In *Recent trends in algebraic development techniques. Proc* 13^{th} *International Workshop*, number 1376 in LNCS, pages 333– 348. Springer Verlag, 1998.
17. R. S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University, 1999.

18. P. J. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal for Logic and Computation*, 14(1):43–52, 1993.
19. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
20. D. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.
21. D. R. Smith. KIDS — a semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024– 1043, 1991.
22. D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305– 321, 1990.
23. M. Staples. Window inference in Isabelle. In *Proc. Isabelle Users Workshop*. University of Cambridge Computer Laboratory, 1995.
24. M. Staples. *A Mechanised Theory of Refinement*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.
25. M. Staples. Representing wp semantics in isabelle/zf. In G. Dowek, C. Paulin, and Y. Bertot, editors, *TPHOLs: The 12th International Conference on Theorem Proving in Higher-Order Logics*, number 1690 in lncs. springer, 1999.
26. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe FME '97*, number 1313 in LNCS, pages 318–337. Springer Verlag, 1997.
27. D. van Dalen. *Logic and Structure*. Springer Verlag, 1994.
28. A. J. M. van Gasteren. On the shape of mathematical arguments. In *Advances in Software Engineering and Knowledge Engineering*, number 445 in LNCS, pages 1–39. Springer Verlag, 1990.
29. J. von Wright. Extending window inference. In *Proc. TPHOLs '98*, number 1497 in LNCS, pages 17–32. Springer Verlag, 1998.