A Tool for Data Refinement

Rimvydas Rukšėnas

Turku Centre for Computer Science (TUCS)
Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14, FIN-20520 Turku, Finland

Joakim von Wright

Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14, FIN-20520 Turku, Finland

Turku Centre for Computer Science
TUCS Technical Report No 119
August 1997

ISBN 952-12-0031-6
ISSN 1239-1891

Abstract

We describe a tool for data refinement based on the Refinement Calculator. The tool supports the calculational approach to data refinement. As a consequence of the program calculation, a refinement theorem is automatically derived. The operation of the tool is illustrated with a case study.

Keywords: data refinement, calculation, proof, automation, HOL

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Refinement is a process by which specifications are transformed into executable programs. Frequently, two forms of refinement are recognised — algorithmic refinement and data refinement [11]. In the former, the operation of a program is made more explicit, whereas the latter replaces abstract data structures with more efficiently implementable ones.

The *refinement calculus* [1, 16, 18] is a formalisation of the stepwise refinement method of program construction, based on the weakest precondition calculus of Dijkstra [6]. The refinement calculus supports transformational development of programs. In each step, a new program is derived from the old one by application of a refinement rule.

Typical refinement transformations involve a considerable amount of formula manipulation which makes them error-prone. This is especially true for data refinement. Thus, tool support is needed for application of the refinement calculus in practice. The Refinement Calculator [13] is an environment for program development using the refinement calculus of [1, 2] and uses the HOL system [8] as an underlying engine.

In the paper, we present an extension to the Refinement Calculator — a tool for data refinement. In the tool, data refinement is modelled as a special case of algorithmic refinement. The implementation is based on the calculational approach to data refinement [17, 19, 21] which preserves the structure of an abstract program.

Overview of the paper In Section 2 we give a brief introduction to the Refinement Calculator and a theory behind it. Section 3 describes the approach to data refinement supported by our tool. >From the general rules of data refinement we also derive in Section 4 more specific rules for a number of special cases. These rules allow our tool to produce simpler programs with less proof effort. An implementation of the tool as an extension of the Refinement Calculator is described in Section 5. Finally, we develop a small example in Section 6 illustrating the usage of the Refinement Calculator and data refinement extension to it.

2 The Refinement Calculator

The Refinement Calculator is a user friendly environment supporting development of provably correct programs within the refinement calculus. It is built on top of the HOL theorem proving system [8] and uses the Window Library [10] to implement program transformations. The Window Library supports a natural way of transforming a term by restricting attention to a subterm, called a *focus*, and transforming it. The remainder of the term is left unchanged. The Tcl/Tk [20] programming system provides a graphical user interface to the environment. The user selects (using the mouse) objects to manipulate and chooses the desired transformation (refinement) from a menu. A detailed list of refinement rules supported by the Refinement Calculator may be found in [4].

2.1 The refinement calculus

The refinement calculus is a formalisation of the stepwise refinement method of program construction. An initial specification is given as an abstract, often nonexecutable, program. By a series of correctness preserving steps, the specification is then refined into an executable program.

In the refinement calculus, programs are given a weakest precondition semantics [6]. The refinement relation between program statements S and S' is defined using their weakest preconditions: we say that a statement S is *refined* by S', written $S \sqsubseteq S'$, iff

$$\forall Q.\ wp(S, Q) \Rightarrow wp(S', Q)$$

Intuitively, refinement means that the new statement S' preserves the total correctness of S. The refinement relation is a preorder (i.e., it is reflexive and transitive). Transitivity of the relation justifies a program development by a series of refinement steps:

$$S_1 \sqsubseteq S_2 \sqsubseteq \ldots \sqsubseteq S_n$$

Data refinement is a special case of refinement where abstract data structures are replaced with more concrete ones. Typically, 'more concrete' means more easily or efficiently implementable. Data refinement is formally defined in terms of ordinary refinement as follows.

A relation between abstract and concrete data structures, called an *abstraction* relation, is given. This relation is then encoded in an abstraction statement E, which replaces the abstract component of the program state by a component of another type according to the relation and leaves the rest of the state unchanged. For a relation abs (i.e., a boolean expression over global variables v, abstract local variables a and concrete local variables c) and a postcondition Q over the abstract variables, the weakest precondition of the abstraction statement E is defined as

$$wp(E, Q) = (\exists a.\ abs \wedge Q)$$

Finally, we say that an abstract statement S is data-refined by a concrete statement S' iff the following ordinary refinement holds:

$$E; S \sqsubseteq S'; E$$

2.2 Formalisation in HOL

The HOL theory of refinement [21] is a *shallow embedding*, meaning that the theory manipulates semantic objects, rather than the actual syntax of programs. The type of the state space is not fixed, so we model it with a type variable 's. For any given program, this type variable is instantiated to a product with one component for each program variable. If, for example, a program works on a natural number variable and a boolean variable, then the program state has type num#bool.

Predicates, pred, are then functions of type 's→bool. The order relation on predicates is defined as follows:

$$\vdash \mathsf{p\ implies\ q} = \forall \mathsf{s.\ p\ s} \Rightarrow \mathsf{q\ s}$$

A predicate transformer is a function of type pred→pred from predicates to predicates. The refinement relation ref between two predicate transformers S and S' is simply an order relation lifted from the type pred:

$$\vdash \mathsf{S\ ref\ S'} = \forall \mathsf{q.\ S\ q\ implies\ S'\ q}$$

2.3 Programming notation

The following syntax of programs (specifications) is supported in the Refinement Calculator:

$$\begin{array}{rcl}
\mathit{Prog} & ::= & \textbf{program}\ \mathit{Name}\ \textbf{var}\ v : \mathit{Type} \cdot \mathit{Com} \\
\mathit{Com} & ::= & \mathit{Com};\ \mathit{Com} \\
 & | & \{\mathit{Pred}\} \\
 & | & [\mathit{Pred}] \\
 & | & v := \mathit{Exp} \\
 & | & v := v' \bullet \mathit{Pred} \\
 & | & \textbf{if}\ \mathit{Pred}\ \textbf{then}\ \mathit{Com}\ \textbf{else}\ \mathit{Com}\ \ \textbf{fi} \\
 & | & \textbf{if}\ \mathit{Actionlist}\ \textbf{fi} \\
 & | & \textbf{do}\ \mathit{Actionlist}\ \textbf{od} \\
 & | & (\textbf{rec}\ X \cdot \mathit{Com}\ ) \\
 & | & |[\ \textbf{var}\ v : \mathit{Type} \mid \mathit{Pred} \cdot \mathit{Com}\ ]| \\
\mathit{Actionlist} & ::= & \mathit{Action}\ \#\ \mathit{Actionlist} \\
\mathit{Action} & ::= & \mathit{Pred} \rightarrow \mathit{Com}
\end{array}$$

In the syntax definition, *Type* denotes data types used in programs, *Pred* is any boolean expression on program variables, and *Exp* stands for an expression of a suitable type.

The notation is an extension of Dijkstra's guarded command language. It includes *assertions* and *assumptions*, representing context information, as well as *nondeterministic assignment*, *recursive* construct and *block*. An occurrence of an assertion, $\{p\}$, in a program asserts that condition p holds at that point in the program. Dually, an assumption, $[p]$, represents our anticipation that p holds at that point (however, the validity of the anticipation needs to be demonstrated at some point). A nondeterministic assignment, $v := v' \bullet post$, is used for specification purposes. It is interpreted as an assignment to v of some value such that predicate *post* is satisfied (in *post*, v refers to initial values and v' to final values of the variable v).

Every program statement denotes a predicate transformer. As an example, the meaning of the sequential composition $S_1; S_2$ is given as the predicate transformer $\mathsf{S1\ seq\ S2}$. Here, $\mathsf{S1}$ and $\mathsf{S2}$ are predicate transformers representing program statements S_1 and S_2, respectively, and the sequential composition seq is defined as

$$\vdash \mathsf{(S1\ seq\ S2)\ q = S1(S2\ q)}$$

The nondeterministic assignment $v := v' \bullet post$ denotes the predicate transformer $\mathsf{nondass}\ (\lambda \mathsf{v\ v'}.\ \mathsf{post})$, where the constant $\mathsf{nondass}$ is defined as follows:

$$\vdash \mathsf{nondass\ P\ q} = \lambda \mathsf{v}.\ \forall \mathsf{v'}.\ \mathsf{P\ v\ v'} \Rightarrow \mathsf{q\ v'}$$

A block (which is the target of data refinement in our approach)

$$|[\ \mathbf{var}\ x \mid ini \cdot S\ ]|$$

adds local variables x to the program state and initialises them according to the initialisation predicate *ini*. The block is represented by the predicate transformer $\mathsf{block}\ (\lambda \mathsf{x,v}.\ \mathsf{ini})\ \mathsf{S}$, where S is a predicate transformer corresponding to the block body S. All local variables (x) of the block are modelled as one tuple ($\mathsf{x{:}'x}$). The constant block is defined in HOL as

$$\vdash \mathsf{block\ p\ c\ q} = \lambda \mathsf{v}.\ (\forall \mathsf{x}.\ \mathsf{p(x,v)} \Rightarrow \mathsf{c}\ (\lambda \mathsf{u}.\ \mathsf{q(SND\ u)})\ \mathsf{(x,v)})$$

where the predicate transformer S works on the extended program state of type $\mathsf{'x\#'s}$.

3 Calculation of data refinement

Recall that data refinement was defined using the abstraction statement. In the Refinement Calculator, the abstraction statement is not a part of the program syntax. This means that programs cannot operate on both the abstract and the concrete state at the same time. Thus, a data refinement transformation must replace all occurrences of the abstract variables in a program in one step. Since the scope of the local variable is a block statement, data refinement is modelled as a transformation of the whole block. In this way, data refinement becomes a special case of algorithmic refinement.

Lemma 1 *Let* $|[\ \mathbf{var}\ a \mid ini \cdot S\ ]|$ *be an abstract program, and* E *be an abstraction statement. Assume that a concrete statement,* S'*, and concrete predicate,* ini'*, are such that:*

(i) $ini' \;\Rightarrow\; wp(E, ini)$,

(ii) $E; S \;\sqsubseteq\; S'; E$.

Then the concrete program $|[\ \mathbf{var}\ c \mid ini' \cdot S'\ ]|$ *is a (data) refinement of the abstract program:*

$$|[\ \mathbf{var}\ a \mid ini \cdot S\ ]| \;\sqsubseteq\; |[\ \mathbf{var}\ c \mid ini' \cdot S'\ ]|$$

Calculator of data refinement It has been shown [17, 19] that the body S' of concrete block can be calculated from the abstract body S and abstraction relation. We take an approach described there with some adaptations (extensions) owing to our programming notation.

Without loss of generality, let us assume that abstract variables, a, of a block, $|[\ \mathbf{var}\ a \mid ini \cdot S\ ]|$, are data-refined by concrete variables, c, while the remaining (global) variables, v, are left unchanged. If only some of local variables are replaced, then the remaining variables are simply treated as (unchanged) global variables. Furthermore, we assume that the list c does not mention variables from a and/or v. The body S of the abstract block works on the program state (a, v). Hence, a body of the concrete block will operate on the program state (c, v). Assume that abstract and concrete variables are related by the abstraction relation R. We then define *calculators* of data refinement, D_R for predicates and $\mathcal{D}_R$ for statements, such that

$$|[\ \mathbf{var}\ a \mid ini \cdot S\ ]| \;\sqsubseteq\; |[\ \mathbf{var}\ c \mid D_R(ini) \cdot \mathcal{D}_R(S)\ ]|$$

The definitions are shown to be valid in the Section 5.

For an abstract predicate p, D_R and its *dual calculator* D°_R are defined as follows:

$$\begin{array}{rcl} D_R(p) & = & (\exists a \cdot R \wedge p) \\ D^{\circ}_R(p) & = & (\forall a \cdot R \Rightarrow p) \end{array}$$

The calculator for statements $\mathcal{D}_R$ is defined recursively over the structure of program notation and reflects data refinement theorems for the corresponding program statements.

Basic statements For a nondeterministic assignment, $w, b := w', b' \bullet post$, where b and w are some of the local variables a and the global variables v, respectively, we calculate its data refinement as follows:

$$\begin{array}{l} \mathcal{D}_R(b, w := b', w' \bullet post) = \\ \qquad c, w := c', w' \bullet (\forall b \cdot R \;\; \Rightarrow \;\; (\exists b' \cdot R[b, c, w := b', c', w'] \;\; \wedge \;\; post)) \end{array}$$

Since predicate *post* is binary (may refer to the values of variables before and after execution of the assignment), the rule is a generalisation of similar ones in [17, 19].

An ordinary assignment statement, $v := e$, is first replaced by the equivalent nondeterministic assignment $v := v' \bullet v' = e$. Then the above rule can be applied. In general, the result is a nondeterministic assignment.

For the context statements (assertions and assumptions) $\mathcal{D}_R$ is defined as follows:

$$\begin{array}{rcl} \mathcal{D}_R(\{p\}) & = & \{D_R(p)\} \\ \mathcal{D}_R(\,[p]\,) & = & [D^{\circ}_R(p)] \end{array}$$

Sequential composition The calculator of data refinement distributes over sequential composition:

$$\mathcal{D}_R(S_1; S_2) \;\; = \;\; \mathcal{D}_R(S_1); \mathcal{D}_R(S_2)$$

Conditional statement The data refinement calculator also distributes into a conditional statement, but in this case an assumption statement must be added into one branch of the concrete conditional:

$$\begin{array}{l} \mathcal{D}_R(\textbf{if } g \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \\ \qquad \textbf{if } D_R(g) \textbf{ then } [D^{\circ}_R(g)]; \mathcal{D}_R(S_1) \textbf{ else } \mathcal{D}_R(S_2) \textbf{ fi} \end{array}$$

Block Let us assume that data refinement is to be distributed over an inner block, $|[\ \textbf{var}\ b \mid q \cdot T\ ]|$. Then the calculator is pushed into the block as follows:

$$\mathcal{D}_R(\,|[\ \textbf{var}\ b \mid q \cdot T\ ]|\,) \;\; = \;\; |[\ \textbf{var}\ b \mid D^{\circ}_R(q) \cdot \mathcal{D}_R(T)\ ]|$$

A similar rule is given in [19], using an (explicit) assignment instead of an initialisation predicate.

Alternation and loop An abstract action, $g \to S$, is data-refined as follows:

$$\mathcal{D}_R(g \to S) \quad = \quad D_R(g) \to [D_R^\circ(g)]; \mathcal{D}_R(S)$$

The assumption of the form $[D_R^\circ(g)]$, in a concrete action, is needed for the distribution of data refinement over alternation and loop statements:

$$\begin{array}{lcl} \mathcal{D}_R(\textbf{if } A_1 \# \ldots \# A_n \textbf{ fi}) & = & \textbf{if } \mathcal{D}_R(A_1) \# \ldots \# \mathcal{D}_R(A_n) \textbf{ fi} \\ \mathcal{D}_R(\textbf{do } A_1 \# \ldots \# A_n \textbf{ od}) & = & \textbf{do } \mathcal{D}_R(A_1) \# \ldots \# \mathcal{D}_R(A_n) \textbf{ od} \end{array}$$

The resulting alternation can give a less general refinement than the the rules in [17, 19]. However, our rule is somewhat more flexible in the case of the loop statement. It does not require that we include the loop invariant into the abstraction relation, which is necessary when a proof obligation is generated instead of an explicit assumption statement. This is particularly desirable when a loop invariant mentions local variables of an inner block statements. The more general rule for the alternation [19] could easily be implemented in our tool as well.

Recursion Assume that the body S of recursive parameterless statement, $(\textbf{rec } X \cdot S)$, is a *program context*, i.e., it contains X as a free statement variable. The calculator applied to a variable gives just the same variable:

$$\mathcal{D}_R(X) \quad = \quad X$$

Then a rule for the distribution of data refinement over the recursive construct is as follows:

$$\mathcal{D}_R(\,(\textbf{rec } X \cdot S)\,) \quad = \quad (\textbf{rec } X \cdot \mathcal{D}_R(S))$$

This rule is more in the calculational style than the one given in [7]. It allows a concrete program context to be calculated directly from the abstract one.

Assumptions vs. proof obligations We have seen that data refinement calculations of alternation and loop statements introduce assumptions of the form $[D_R^\circ(g_i)]$, where g_i is the guard (predicate) of an abstract action A_i. These assumptions can be discharged later using rules for context handling [12]. An alternative approach would be to generate proof obligations of the form $D_R(g_i) \Rightarrow \bigvee_i D_R^\circ(g_i)$, without adding assumptions into the program. However, we feel that the former approach is advantageous since it gives more freedom. For example, context information can later be propagated to demonstrate the validity of an assumption.

4 Special cases of data refinement

In general, the result of calculating a data refinement can be very complex. By considering special cases, we get simpler rules and calculations that are easier to handle. By incorporating rules for special cases, our tool for data refinement is able to produce simpler programs.

Handling concrete invariant Assume that an abstraction relation R of the form $abs \wedge ci$ is given, where ci (the *concrete invariant*) refers only to the concrete variables c and the global variables v. The initialisation of a concrete block, $D_R(ini)$, guarantees that ci holds before the execution of its body. Furthermore, the calculator of data refinement ensures that local variables may be only assigned values that satisfy the predicate ci. Therefore, ci is an invariant within the concrete block, i.e. it holds before and after the execution of any statement in the block. This means that ci can be dropped from the guards of concrete alternations and loops. We get the following rule for calculation of a concrete action:

$$\mathcal{D}_R(g \rightarrow S) \quad = \quad D'_R(g) \rightarrow [D^{\circ}_R(g)]; \mathcal{D}_R(S)$$

where $D'_R(g)$ is defined to be $(\exists a \cdot abs \wedge g)$.

Functional data refinement Assume that the abstraction relation R can be written as

$$a = f(c, v) \quad \wedge \quad ci$$

Such relation is *functional*. It makes the calculation of the concrete program simpler. Specialised techniques for functional relations are considered in [17, 19]. In our case, the calculators of concrete predicates, D_R and D°_R, are reduced to:

$$\begin{array}{lcl} D_R(p) & = & ci \;\; \wedge \;\; p[a := f(c, v)] \\ D^{\circ}_R(p) & = & ci \;\; \Rightarrow \;\; p[a := f(c, v)] \end{array}$$

This simplifies calculations of the assertion and assumption statements. For the nondeterministic assignment, the calculator $\mathcal{D}_R$ becomes the following:

$$\begin{array}{l} \mathcal{D}_R(a, v := a', v' \bullet post) = \\ \qquad c, v := c', v' \bullet (ci \;\; \Rightarrow \;\; post[a, a' := f(c, v), f(c', v')] \;\; \wedge \;\; ci[c, v := c', v']) \end{array}$$

Since $D_R(p) \Rightarrow D^{\circ}_R(p)$ is trivially true in the case of a functional relation R, no assumptions are needed in the alternation and loop statements. The calculational rule for the abstract action $g \rightarrow S$ becomes:

$$\mathcal{D}_R(g \rightarrow S) \quad = \quad D'_R(g) \rightarrow \mathcal{D}_R(S)$$

where $D'_R(g)$ is simply $g[a := f(c,v)]$, i.e. a concrete invariant is dropped from the guards of alternations and loops.

Superposition *Superposition* is a refinement where only new variables are added under the abstraction relation. Since no variables are removed, the abstraction relation R is just a (concrete) invariant, ci, on some of program variables. Again, the calculator $\mathcal{D}_R$ is simplified for the nondeterministic assignment:

$$\mathcal{D}_R(v := v' \bullet post) \quad = \quad c, v := c', v' \bullet (ci \;\; \Rightarrow \;\; post \;\; \wedge \;\; ci[c, v := c', v'])$$

Moreover, predicate calculators D_R and D°_R become as below:

$$\begin{array}{rcl} D_R(p) & = & ci \;\; \wedge \;\; p \\ D^{\circ}_R(p) & = & ci \;\; \Rightarrow \;\; p \end{array}$$

The guards of alternations and loops remain unchanged in a superposition. As a result, the whole transformation introduces a block:

$$S \;\; \sqsubseteq \;\; |[\; \mathbf{var} \; c \mid ci \cdot \mathcal{D}_R(S) \;]|$$

In combination with a rule for block elimination, superposition can also be used as part of a two-step method for data refinement with *auxiliary* (ghost) variables [14] for data refinement [15].

5 Implementation in HOL

In this section we first give HOL definitions which our formalisation of data refinement is based upon. Then an implementation of the calculator of data refinement is described.

Abstraction statements Let the predicate r:'a#'c#'s→bool be the HOL representation of an abstraction relation R. We then define the corresponding abstraction statement as a predicate transformer abst r, as follows:

⊢ abst r q = λv. ∃a. r(a,FST v,SND v) ∧ q(a,SND v)

Dually to the abstraction statement, a *representation* statement is defined as a predicate transformer repr r:

⊢ repr r q = λw. ∀k. r(FST w,k,SND w) ⇒ q(k,SND w)

This statement replaces the abstract component of a program state by the concrete component according to the relation r. Intuitively, it operates as an inverted abstraction statement.

Data refinement The data refinement relation with respect to abstraction relation r is defined in HOL as follows (for readability, we use the symbol $\sqsubseteq$ for the refinement relation ref):

$$\vdash \mathsf{dataref\ r\ c\ c'} = \mathsf{(abst\ r)\ seq\ c} \sqsubseteq \mathsf{c'\ seq\ (abst\ r)}$$

For monotonic predicate transformers c and c', the condition

$$\mathsf{(abst\ r)\ seq\ c} \sqsubseteq \mathsf{c'\ seq\ (abst\ r)}$$

is equivalent to the condition

$$\mathsf{(abst\ r)\ seq\ c\ seq\ (repr\ r)} \sqsubseteq \mathsf{c'}$$

Thus we have the following refinement theorem for the block construct

$$\vdash \mathsf{block\ p\ c} \sqsubseteq \mathsf{block\ (abst\ r\ p)\ ((abst\ r)\ seq\ c\ seq\ (repr\ r))} \qquad (1)$$

which corresponds the Lemma 1. This theorem is a starting point for the calculation of data refinement.

Calculation of a concrete block The predicate transformer abst r now represents the predicate calculator D_R. Rewriting abst r p gives us a new initialisation of the block, p'.

Similarly, we have written a HOL conversion[1] that represents the data refinement calculator for statements. This conversion, called DR_CALC, calculates a new predicate transformer c' from (abst r) seq c (repr r) and proves the refinement theorem

$$\vdash \mathsf{(abst\ r)\ seq\ c\ seq\ (repr\ r)} \sqsubseteq \mathsf{c'}$$

Here, if c represents program statement S, then c' represents program statement $\mathcal{D}_R(S)$.

Since the block construct is monotonic in its body argument, the predicate transformer (abst r) seq c seq (repr r) can be replaced by its refinement c' in the theorem (1), giving the following data refinement:

$$\vdash \mathsf{block\ p\ c} \sqsubseteq \mathsf{block\ p'\ c'}$$

The conversion DR_CALC is implemented so that it recursively pushes the pair of abstraction statements, abst r and repr r, through the structure of the term c. The effect of this is that the resulting predicate transformer is a refinement of the original one. For each program statement written as a HOL term, a theorem corresponding to the definition of the calculator $\mathcal{D}_r$ is used to distribute data refinement. As an example, we have the following theorem for the sequential composition of c1 and c2:

[1] A conversion is a program in the metalanguage ML that generates a theorem from the input arguments.

$$\begin{array}{l} \vdash \ \mathsf{monotonic\ c1} \quad \Rightarrow \\ \quad (\mathsf{abst\ r})\ \mathsf{seq}\ (\mathsf{c1\ seq\ c2})\ \mathsf{seq}\ (\mathsf{repr\ r}) \\ \quad \sqsubseteq \ ((\mathsf{abst\ r})\ \mathsf{seq\ c1\ seq}\ (\mathsf{repr\ r}))\ \mathsf{seq}\ ((\mathsf{abst\ r})\ \mathsf{seq\ c2\ seq}\ (\mathsf{repr\ r})) \end{array}$$

In the above theorem, the side condition monotonic c1 says that pushing the abstraction pair into sequential composition is valid only when its left argument is a monotonic predicate transformer. Monotonicity of predicate transformers is defined as follows:

$$\vdash \mathsf{monotonic\ c} = \forall \mathsf{p\ q.\ p\ implies\ q} \quad \Rightarrow \quad (\mathsf{c\ p})\ \mathsf{implies}\ (\mathsf{c\ q})$$

Monotonicity conditions also arise in the data refinement of loops. In the Refinement Calculator, they are automatically established for any term representing a program statement in our language.

Inner block Recall that the state space of a program is changed by the block construct. Let the predicate transformer block p c operate on the program state (a,v), where a is the abstract component of the program state. Then the predicate transformer c operates on the program state (b,a,v), where b is the local variable(s) of the (inner) block. The HOL implementation of data refinement assumes that the abstract component is the first one in the program state. Therefore, the predicate transformer c has to be 'adapted' before the transformation so that it operates on the program state (a,b,v). The adaptation operator is defined as follows:

$$\vdash \mathsf{adapt\ f\ g\ c} = (\mathsf{assign\ f})\ \mathsf{seq\ c\ seq}\ (\mathsf{assign\ g})$$

where f and g are state functions of the types state1→state2 and state2→state1, respectively. The assignment assign f (assign g) stands for a transformation of the program state specified by the function f (g).

For our purposes, we define adaptation functions, f1 and g1, so that f1 moves the abstract component of the program state one position to the left, whereas g1 moves it one position to the right (returns it to the initial place). Before data refinement, an adapted predicate transformer is automatically calculated from the predicate transformer

$$(\mathsf{assign\ f1})\ \mathsf{seq\ c\ seq}\ (\mathsf{assign\ g1})$$

After the transformation, a concrete predicate transformer must be 're-adapted' so that b is again the first component in the program state. To achieve this we define new adaptation functions, f2 and g2. The function f2 moves the concrete component one position to the right, whereas g2 moves it one position to the left. As a result, the theorem for the distribution of data refinement over the block construct is as follows:

$$
\begin{array}{ll}
\vdash & \mathsf{(abst\ r)\ seq\ (block\ p\ c)\ seq\ (repr\ r)} \\
& \sqsubseteq\ \mathsf{block}\quad \mathsf{(dual(abst\ \bar{r})\ p)} \\
& \qquad\qquad \mathsf{(adapt\ f2\ g2\ ((abst\ \bar{r})\ seq\ (adapt\ f1\ g1\ c)\ seq\ (repr\ \bar{r})))}
\end{array}
$$

Here $\mathsf{\bar{r}}$ is the adapted abstraction relation r:

$$\mathsf{\bar{r}(a,c,b,v)} \quad = \quad \mathsf{r(a,c,v)}$$

In the initialisation of the concrete block, the predicate transformer $\mathsf{dual(abst\ \bar{r})}$ corresponds to the dual predicate calculator $D^{\circ}_{\overline{R}}$.

Recursive construct The recursive construct is represented by the least fixpoint of a corresponding function. Let f be such a function on abstract predicate transformers. Then data refinement is distributed over the recursive construct as follows:

$$
\begin{array}{ll}
\vdash & \mathsf{regular\ f} \quad \Rightarrow \\
& \mathsf{(abst\ r)\ seq\ (mu\ f)\ seq\ (repr\ r)} \\
& \sqsubseteq\ \mathsf{mu\ (\lambda y.\ (abst\ r)\ seq\ f((repr\ r)\ seq\ y\ seq\ (abst\ r))\ seq\ (repr\ r))}
\end{array}
$$

where mu is the least fix-point operator. The side condition $\mathsf{regular\ f}$ essentially requires the function f to be monotonic, and to map monotonic predicate transformers to monotonic predicate transformers. Regularity is proved automatically for any function representing a recursive statement.

A concrete function is constructed from the abstract one by distributing data refinement over its structure. Therefore before the distribution, the parameter (y) of the concrete function is enclosed into an abstract wrapping:

$$\mathsf{(repr\ r)\ seq\ y\ seq\ (abst\ r)}$$

The wrapping is removed using the theorem

$$
\begin{array}{ll}
\vdash & \mathsf{monotonic\ c} \quad \Rightarrow \\
& \mathsf{(abst\ r)\ seq\ ((repr\ r)\ seq\ c\ seq\ (abst\ r))\ seq\ (repr\ r)} \sqsubseteq \mathsf{c}
\end{array}
$$

whenever the distribution of the abstraction pair reaches such a wrapped parameter (this agrees with the fact that the data refinement calculator maps a statement variable X to X itself).

Data refinement in the Refinement Calculator The Refinement Calculator provides a menu of transformations for refining programs. Our tool for data refinement is an extension of it which adds new refinement rules. When the rule *Data Refine* is chosen from a menu, the user is asked to supply three arguments: a list of variables to be replaced (a), a list of new variables (c), and an abstraction relation (r). The arguments are translated into corresponding HOL terms, and the data refinement conversion is invoked. The focus must be a block for the transformation to be valid. The calculated predicate transformer is pretty-printed in our programming notation. Similarly, the superposition rule *Add Block Inv* must be supplied with variables to be added (c), and a concrete invariant (ci). It can be applied to any focus and introduces a block with the local variables c under the invariant ci.

6 An example of data refinement

As an example, we consider a program that produces a number of distinct keys (natural numbers). The specification of the program is as follows:

$$
\begin{array}{l}
\textbf{program}\ keys\ \textbf{var}\ l : (num)list\ \cdot \\
\quad l := l' \bullet LENGTH\ l' = N\ \ \wedge \\
\qquad\qquad (\forall j, k \cdot j < k\ \ \wedge\ \ k < N\ \ \Rightarrow\ \ (EL\ j\ l') \neq (EL\ k\ l')
\end{array}
$$

(here $EL\ n$ returns the nth element of a list, with numbering starting from zero). The following (abstract) program is constructed in several refinement steps which include *Block Introduction* and *Loop introduction* transformations:

$$
\begin{array}{l}
\textbf{program}\ keys\ \textbf{var}\ l : (num)list \cdot l := [\,]; \\
\quad |[\ \ \textbf{var}\ a : (num)set\ \mid a = \emptyset\ \cdot \\
\qquad \textbf{do}\ \ LENGTH\ l < N \rightarrow \\
\qquad\qquad a, l := a', l' \bullet \exists m \cdot l' = l\,@[m]\ \ \wedge\ \ m \notin a\ \ \wedge\ \ a' = (m\ \ INS\ \ a) \\
\qquad \textbf{od} \\
\quad]|
\end{array}
$$

Here the set-valued variable a accumulates the keys that have already been generated. The symbol @ stands for the list append operation, whereas *INS* simply adds an element into a set.

For the data refinement transformation, we focus on the block statement in the above program. The local variable (a) is replaced with a new variable k which ranges over the natural numbers. For this the *Block Introduction* rule is provided with the following arguments:

$$
\begin{array}{ll}
Replace: & a : (num)set \\
By: & k : num \\
Relation: & \forall n \cdot n \in a\ \ \Rightarrow\ \ n < k
\end{array}
$$

The abstraction relation says that the value of the concrete variable k must be greater than any value in the (abstract) set a. Given these arguments, the rule calculates a new block:

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ k : num \mid \forall n \cdot n \in \emptyset \ \Rightarrow\ n < k \cdot \\
\quad \mathbf{do}\ \ (\exists a \cdot (\forall n \cdot n \in a \ \Rightarrow\ n < k) \ \wedge\ LENGTH\ l < N) \rightarrow \\
\qquad\quad [\forall a \cdot (\forall n \cdot n \in a \ \Rightarrow\ n < k) \ \Rightarrow\ LENGTH\ l < N]; \\
\qquad\quad k, l := k', l' \bullet \\
\qquad\qquad (\forall a \cdot (\forall n \cdot n \in a \ \Rightarrow\ n < k) \ \Rightarrow \\
\qquad\qquad\quad (\exists a' \cdot (\forall n \cdot n \in a' \ \Rightarrow\ n < k') \ \wedge \\
\qquad\qquad\qquad (\exists m \cdot l' = l\,@[m] \ \wedge\ m \notin a \ \wedge\ a' = (m\ \ INS\ \ a)))) \\
\quad \mathbf{od} \\
]|
\end{array}
$$

The calculated program can be simplified further as follows. The initialisation condition

$$\forall n \cdot n \in \emptyset \ \Rightarrow\ n < k$$

is replaced with T by rewriting, i.e., by selecting a menu alternative for rewriting and pointing to the theorem $\forall a \cdot a \notin \emptyset$. The guard of the loop action

$$\exists a \cdot (\forall n \cdot n \in a \ \Rightarrow\ n < k) \ \wedge\ LENGTH\ l < N$$

is simplified using the *General Logic* extension [4] of the Refinement Calculator and providing the empty set an existential witness. The result is a new guard of the loop:

$$LENGTH\ l < N$$

Both these simplifications are simple, but they require domain-specific knowledge (set theory). This means that the user must supply the names of the theorems used for rewriting.

The assumption

$$[\forall a \cdot (\forall n \cdot n \in a \ \Rightarrow\ n < k) \ \Rightarrow\ LENGTH\ l < N]$$

is easily shown to be true using context handling transformations [12]. First the guard of the loop is added to its body as an assertion. Then the *Pull Assumption* transformation produces a new assumption

$$[LENGTH\ l < N \ \Rightarrow\ (\forall a \cdot (\forall n \cdot n \in a \ \Rightarrow\ n < k) \ \Rightarrow\ LENGTH\ l < N)]$$

which is discharged by focusing on the rightmost occurrence of $LENGTH\ l < N$ and rewriting it with a new assumption (antecedent of the outermost implication) of the window.

Domain specific knowledge is also required to refine the nondeterministic assignment within the loop. The *Replace Similar* rule is supplied with a new body of the loop:

$$k, l := k', l' \bullet k < k' \quad \wedge \quad l' = l \mathbin{@} [k]$$

As a result, a proof obligation is generated which says that the relation in the new assignment implies the relation in the old assignment. The proof obligation is discharged using the *General Logic* extension. All in all the data-refined program (block) is:

$$\begin{array}{l} |[\ \ \mathbf{var}\ k : num \mid T \cdot \\ \quad \mathbf{do}\ LENGTH\ l < N \rightarrow k, l := k', l' \bullet k < k' \quad \wedge \quad l' = l \mathbin{@} [k]\ \mathbf{od} \\ ]| \end{array}$$

Next, the *Nondass To Assign* rule replaces the nondeterministic assignment with an ordinary assignment:

$$k, l := k + 1, l \mathbin{@} [k]$$

Finally, the block initialisation (T) is made deterministic by strengthening it to $k = 0$. This yields the following program:

$$\begin{array}{l} \mathbf{program}\ keys\ \mathbf{var}\ l : (num)list \cdot l := [\,]; \\ \quad |[\ \ \mathbf{var}\ k : num \mid k = 0 \cdot \\ \quad\quad \mathbf{do}\ LENGTH\ l < N \rightarrow k, l := k + 1, l \mathbin{@} [k]\ \mathbf{od} \\ \quad]| \end{array}$$

7 Conclusions

We have described a tool for data refinement which is an extension of the Refinement Calculator. The tool supports the calculational style of data refinement: a concrete program is automatically constructed from an abstract one. Furthermore, the tool proves data refinement as a HOL theorem which give us a high degree of confidence. The calculational style of refinement preserves the structure of an abstract program. The complexity of the calculated program (predicates in the assignment, alternation and loop statements) depends essentially on the complexity of the abstraction relation used in the calculation. In the case of a functional relation, the calculated program is rather simple. For more sophisticated abstraction relations, the concrete program needs subsequent transformations (refinements) as was shown by our example in Section 6. These transformations are mostly based on domain-specific knowledge, therefore, their automation is quite a challenging task.

Nevertheless, it must be considered (to some extent at least) for the tool to be applicable in practice.

A one-to-one correspondence between the structures of abstract and concrete programs is too restrictive in some cases. This is especially true for the loop statement where, for example, the introduction of new actions or many-to-many mapping between abstract and concrete actions may be desirable. A loop within a block can be thought of as an *action system* [3] which is a model for parallel and reactive programs. Since a refinement relation for such systems (trace refinement) is essentially reduced to data refinement, the extensions to our tool would make it possible to refine parallel and reactive programs as well. These extensions will be considered in the future.

A refinement tool called PRT [5] is similar to the Refinement Calculator, but it has no support for data refinement. A refinement editor [9] provides a similar data refinement transformation based on [17]. However, this editor, does not include proofs of the transformation. Moreover, a concrete program is calculated in rather general form (without use of specialised techniques) and needs a number of subsequent refinements.

References

[1] R.-J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[2] R.-J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[3] R.-J. R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[4] M. J. Butler, T. Långbacka, R. Rukšėnas, and J. von Wright. Refinement Calculator tutorial and manual. Draft — available upon request.

[5] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A tool for developing correct programs by refinement. In H. Jifeng, editor, *BCS FACS 7th Refinement Workshop*. Springer-Verlag, July 1996.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, 1976.

[7] P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Acta Informatica*, 87(1):143–162, Sept. 1991.

[8] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[9] L. J. Groves, R. G. Nickson, and M. Utting. A tactic driven refinement tool. In C. B. Jones, B. T. Denvir, and R. C. F. Shaw, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272–297, London, Jan. 1992. BCS-FACS, Springer-Verlag.

[10] J. Grundy. The HOL window library. In *The HOL System*, volume Libraries. SRI International, Cambridge Research Center, Millers Yard, Mill Lane, Cambridge CB2 1RQ, England, 2.01 edition, July 1991.

[11] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[12] L. Laibinis and J. von Wright. Context handling in the refinement calculus framework. TUCS Technical Report 118, Turku Centre for Computer Science, August 1997.

[13] T. Långbacka, R. Rukšėnas, and J. von Wright. TkWinHOL: A tool for doing window inference in HOL. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 245–260, Aspen Grove, Utah, Sept. 1995. Springer-Verlag.

[14] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, 1968.

[15] C. C. Morgan. Auxiliary variables in data refinement. *Information Processing Letters*, 29:293–296, 1988.

[16] C. C. Morgan. *Programming from Specifications.* Prentice Hall, 2 edition, 1994.

[17] C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27(6):481–503, 1990.

[18] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, Dec. 1987.

[19] J. M. Morris. Laws of data refinement. *Acta Informatica*, 26(4):287–308, 1989.

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

[21] J. von Wright. Program refinement by theorem prover. In D. Till and R. C. F. Shaw, editors, *6th Refinement Workshop*, Workshops in Computing, pages 121–150, London, Jan. 1994. BCS-FACS, Springer-Verlag.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**