A Mechanised Calculus of Refinement in HOL

J. von Wright *

January 27, 1994

Abstract

This is the documentation of the RefCalc contribution to the HOL system. The contribution contains a mechanised theory of the refinement calculus and some infrastructure for proving refinement rules and doing program refinements using the HOL system.

1 Introduction

This is the documentation of the RefCalc contribution to the HOL system. This contribution is available with Version 2.02 of the HOL system. It resides in the RefCalc subdirectory of the contrib directory.

The aim of the contribution is to show how a theory of program refinement can be machanised and how HOL can be used as a tool in program refinement. In particular, we show how window inference (as implemented in the Window Library of the HOL system) can be used in program refinement.

For a detailed description of the refinement calculus itself we refer to, e.g., [1, 2, 4]. Data refinement, which is an important part of the RefCalc contribution, is described in more detail in [3, 12]. The principles behind the mechanisation of the refinement calculus are described in [5, 6, 11, 13].

We assume that the reader is familiar with the higher-order logic of the HOL system and with the system itself. Excellent introduction and tutorials can be found in the documentation [8] and in Gordon's and Melham's book [9]. For documentation on window inference and the Window Library we refer to the HOL documentation and to Grundy's paper [10].

The RefCalc contribution is a collection of theories organised according to the hierarchy in Figure 1.

For each theory there is a subdirectory in the RefCalc directory. The subdirectories contain the sources and examples for the theory in question. For details on how to build the theories, see the READ-ME file. A theory is loaded by starting HOL in the RefCalc directory and then loading the appropriate ld_???.ml file. The question marks stand for the theory name. For example, the RCcorrect theory is loaded with the command

*Åbo Akademi University, Dept. of Computer Science, Lemminkäinengatan 14 A, FIN-20520 Turku, Finland. E-mail: jwright@aton.abo.fi

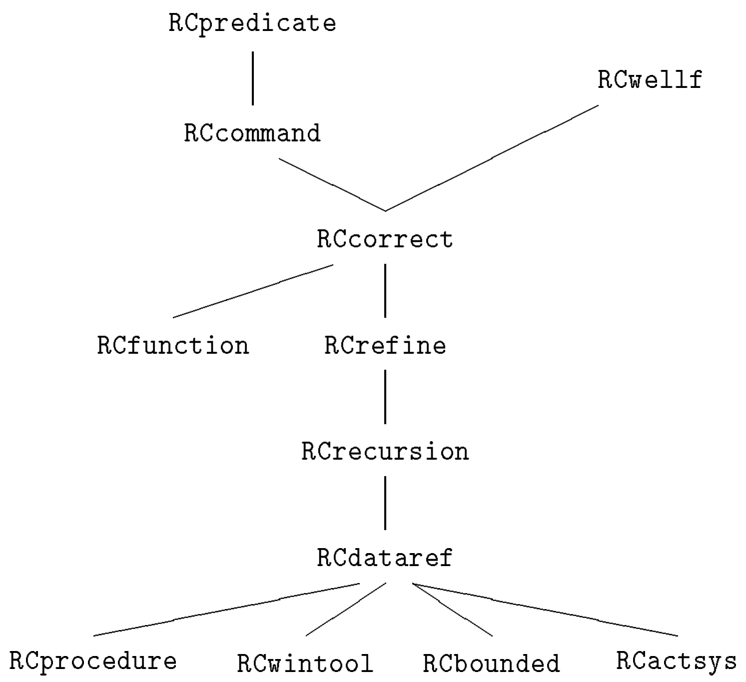

Figure 1: RefCalc theory hierarchy

```
#loadf `Correct/ld_RCcorrect`;;
```

The examples are in files that have the letter combination ex in their names. Examples of total correctness reasoning, e.g., can be found in the file Correct/RCcorrect_ex1.ml.

We use sans serif font for names of constants that are defined in the HOL system or the RefCalc theory. The same font is also used for names of definitions and theorems. Names of theorems are written immediately to the left of the turnstile.

To make theorems more readable, we usually omit outermost universal quantifiers. Thus, any free variables in a theorem are implicitly universally quantified.

Sample dialogs with the HOL system are shown in `teletype` font.

2 Modelling states and predicates: the RCpredicate theory

A state is modelled as a tuple of values. When we consider states in general, we assume that they have polymorphic type $*s$. In particular cases, this type will be instantiated to a product type with one component for each state component (program variable). Note that program variables are anonymous; they are identified by their position in the state tuple rather than by name. A predicate is a boolean function on states. For example, the predicate

$$\lambda(b, x, y).\ \neg b \wedge (x < y) \tag{1}$$

holds in the state $(F, 2, 3)$.

From now on we use the abbreviation pred for the predicate type $*s \rightarrow \mathsf{bool}$. When we need to consider more than one state space, we use the abbreviation pred1 for $*s1 \rightarrow \mathsf{bool}$, pred2 for $*s2 \rightarrow \mathsf{bool}$ etc.

Substitution in predicates is represented by a combination of function application and lambda abstraction. As an example, substitution of 2 for x in the predicate $\lambda(b, x, y).\ x < y$ is represented as

$$\lambda(b, x, y).\ (\lambda(b, x, y).\ x < y)(b, 2, y)$$

which by the HOL rule of beta conversion is equal to $\lambda(b, x, y).\ 2 < y$. This corresponds exactly to the syntactic substitution of 2 for x in $x < y$.

Note that the state components are *anonymous*; they have no global names. In the above example, we used (b, x, y) as a local (bound) name for the state. If we consistently use (b, x, y) as the bound tuple in predicates in this way, then we can think of the state components as being called b, x and y.

2.1 Definitions

The partial order on predicates is defined by lifting:

$$\mathsf{implies\_DEF} \quad \vdash \quad p \ \mathsf{implies}\ q \ = \ \forall u.\ p\ u \Rightarrow q\ u$$

(in definitions such as this one, we implicitly assume that u has generic type $*s$). Note that implies is defined to be an infix operator.

Operators on predicates are also defined by lifting:

$$
\begin{array}{lll}
\mathsf{false\_DEF} & \vdash & \mathsf{false} \;=\; \lambda u.\ F \\
\mathsf{true\_DEF} & \vdash & \mathsf{true} \;=\; \lambda u.\ T \\
\mathsf{not\_DEF} & \vdash & \mathsf{not}\ q \;=\; \lambda u.\ \neg\ q\ u \\
\mathsf{and\_DEF} & \vdash & p\ \mathsf{andd}\ q \;=\; \lambda u.\ p\ u \wedge q\ u \\
\mathsf{or\_DEF} & \vdash & p\ \mathsf{or}\ q \;=\; \lambda u.\ p\ u\ \vee\ q\ u \\
\mathsf{imp\_DEF} & \vdash & p\ \mathsf{imp}\ q \;=\; \lambda u.\ p\ u\ \Rightarrow\ q\ u
\end{array}
$$

Since the predicates over a given state space are a complete lattice, we define the greatest lower bound and least upper bound operators. We represent sets by characteristic functions, so in the expression glb P, the set P has type $\mathit{pred} \to \mathit{bool}$.

$$
\begin{array}{lll}
\mathsf{glb\_DEF} & \vdash & \mathsf{glb}\ P \;=\; \lambda u.\ \forall p.\ P\ p \Rightarrow p\ u \\
\mathsf{lub\_DEF} & \vdash & \mathsf{lub}\ P \;=\; \lambda u.\ \exists p.\ P\ p \wedge p\ u
\end{array}
$$

Monotonicity of functions on predicates is defined as follows:

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

and the least and greatest fixpoints are defined in closed form:

$$
\begin{array}{lll}
\mathsf{fix\_DEF} & \vdash & \mathsf{fix}\ f \;=\; \mathsf{glb}\ (\lambda p.\ (f\ p)\ \mathsf{implies}\ p) \\
\mathsf{gfix\_DEF} & \vdash & \mathsf{gfix}\ f \;=\; \mathsf{lub}\ (\lambda p.\ p\ \mathsf{implies}\ (f\ p))
\end{array}
$$

2.2 Theorems

The fact that implies is a partial order is encoded in the following theorem:

$$
\begin{array}{lll}
\mathsf{implies\_prop} & \vdash & (\forall p.\ p\ \mathsf{implies}\ p) \wedge \\
 & & (\forall p\ q.\ p\ \mathsf{implies}\ q \wedge q\ \mathsf{implies}\ p \Rightarrow (p = q)) \wedge \\
 & & (\forall p\ q\ r.\ p\ \mathsf{implies}\ q \wedge q\ \mathsf{implies}\ r \Rightarrow p\ \mathsf{implies}\ r))
\end{array}
$$

The fixpoint properties of fix are as follows:

$$
\begin{array}{lll}
\mathsf{fix\_fp} & \vdash & \mathsf{monotonic}\ f \Rightarrow (f(\mathsf{fix}\ f) = \mathsf{fix}\ f) \\
\mathsf{fix\_least} & \vdash & (f\ p)\ \mathsf{implies}\ p \Rightarrow (\mathsf{fix}\ f)\ \mathsf{implies}\ p
\end{array}
$$

The theorems for gfix are dual (gfix_fp and gfix_greatest). These theorems rely on a number of properties of glb that are also stored in the theory file.

Another useful theorem in the RCpredicate theory are the following:

$$
\mathsf{shunt} \;\vdash\; (b\ \mathsf{andd}\ p)\ \mathsf{implies}\ q = p\ \mathsf{implies}\ ((\mathsf{not}\ b)\ \mathsf{or}\ q))
$$

2.3 ML functions and tactics

We do not have very much infrastructure for reasoning about predicates. There is a simple tactic PRED_TAUT_TAC which proves tautologous predicate formulas. The following example shows how this tactic can be used:

```
#let thm = prove("!p:^pred. false or p = p",PRED_TAUT_TAC);;
thm = |- !p. false or p = p
```

We have not proved and stored any large number of basic algebraic facts about predicates (such as this one). Instead, we assume that the user makes use of PRED_TAUT_TAC to prove them whenever they are needed.

2.4 Example

As an example, we show how a simple theorem about predicates is proved. This example can be found in the file RCpredicate_ex1.ml.

```
g "!p:^pred. (!x. ~p x) ==> (p = false)";;

e(RT[false_DEF] THEN REPEAT STRIP_TAC);;
"p = (\v. F)"
    [ "!x. ~p x" ]

e(FUN_TAC);;
"p x = F"
    [ "!x. ~p x" ]

e(ART[]);;
goal proved
. |- p x = F
. |- p = (\v. F)
. |- p = (\v. F)
|- !p. (!x. ~p x) ==> (p = false)
```

First, the goal $\forall p.\ (\forall x. \neg p\ x) \Rightarrow (p = \mathsf{false})$ is set up using the predefined function g. This function and many more are defined when the RefCalc theory is loaded.

The goal is next expanded (e) first by rewriting using false_DEF (RT is an abbreviation for REWRITE_TAC) and then by repeated stripping (removing outermost universal quantifiers and undischarging antecedents. At this point, it remains to prove $p = (\lambda v.\mathsf{F})$ under the assumption $\forall x. \neg p\ x$.

Now we apply FUN_TAC which uses extensionality of functions to rewrite the goal (it also does beta reductions and removes outermost universal quantifiers).

Finally, we rewrite using the assumptions (ART abbreviates the ASM_REWRITE_TAC). The systems shows all the intermediate theorems and the proved goal. This theorem can now be accessed by evaluating top_thm().

3 Commands: the RCcommand theory

Commands are modelled as predicate transformers, i.e., as functions from predicates to predicates. Since predicates are polymorphic, commands can have arbitrary initial and final state spaces. This means that a command generally has type $\mathsf{pred1} \rightarrow \mathsf{pred2}$, which we abbreviate $\mathsf{ptrans12}$. Similarly, we use the abbreviation $\mathsf{ptrans23}$ for $\mathsf{pred2} \rightarrow \mathsf{pred3}$, etc. Note that in the intuitive (relational) interpretation, a command of type $\mathsf{ptrans12}$ has initial state of type $*s2$ and final state of type $*s1$.

3.1 Definitions

The refinement order on commands is the partial order on predicates, lifted one level:

$$\mathsf{ref\_DEF} \quad \vdash \quad c \ \mathsf{ref}\ c' \ = \ \forall q.\ (c\ q)\ \mathsf{implies}\ (c'\ q)$$

Basic commands The basic commandsof our command notation are defined as follows:

$$\begin{array}{lll}
\mathsf{abort_DEF} & \vdash & \mathsf{abort}\ q \ = \ \mathit{false} \\
\mathsf{skip_DEF} & \vdash & \mathsf{skip}\ q \ = \ q \\
\mathsf{assert_DEF} & \vdash & \mathsf{assert}\ p\ q \ = \ p\ \mathsf{andd}\ q \\
\mathsf{guard_DEF} & \vdash & \mathsf{guard}\ p\ q \ = \ p\ \mathsf{imp}\ q \\
\mathsf{assign_DEF} & \vdash & \mathsf{assign}\ e\ q \ = \ (\lambda u.\ q\ (e\ u)) \\
\mathsf{nondass_DEF} & \vdash & \mathsf{nondass}\ P\ q \ = \ (\lambda u.\ \forall u'.\ P\ u\ u' \Rightarrow q\ u')
\end{array}$$

Here, the assignment command $\mathsf{assign}\ e$ has a state function e as its argument. This state function indicates how the state is to be changed. For example, the command

$$\mathsf{assign}\ \lambda(b, x, y).\ (b, x + y, y)$$

models the assignment $x := x + y$ in a state containing three variables; b, x and y. Note that the command explicitly says that the values of b and y are left unchanged.

The command $\mathsf{nondass}\ P$ models a nondeterministic assignment according to the state relation P. For example, the command

$$\mathsf{nondass}\ \lambda(b, x, y)(b', x', y').\ (b' = b) \wedge (x' > x) \wedge (y' = y)$$

models the nondeterministic assignment which increases x arbitrarily (by demonic choice) and leaves the other variables (b and y) unchanged.

A general *specification command* is of the form

$$(\mathsf{assert}\ p)\ \mathsf{seq}\ (\mathsf{nondass}\ P)$$

where p is the *precondition* and P is the input-output relation.

Command constructors The lattice-oriented command constructors are defined as

$$
\begin{array}{lll}
\mathsf{dch\_DEF} & \vdash & (c1 \;\mathsf{dch}\; c2)\; q \;=\; (c1\; q) \;\mathsf{andd}\; (c2\; q) \\
\mathsf{Dch\_DEF} & \vdash & \mathsf{Dch}\; C\; q \;=\; \mathsf{glb}\; (\lambda p.\; \exists c.\; C\; c \wedge (p = c\; q)) \\
\mathsf{ach\_DEF} & \vdash & (c1 \;\mathsf{ach}\; c2)\; q \;=\; (c1\; q) \;\mathsf{or}\; (c2\; q) \\
\mathsf{Ach\_DEF} & \vdash & \mathsf{Ach}\; C\; q \;=\; \mathsf{lub}\; (\lambda p.\; \exists c.\; C\; c \wedge (p = c\; q))
\end{array}
$$

Here, dch is binary demonic choice while Dch is demonic choice over a set C of commands (recall that sets are represented by characteristic functions). Dually, ach is binary angelic choice while Ach is angelic choice over a set of commands.

Further command constructors are defined in the following theorems:

$$
\begin{array}{lll}
\mathsf{seq\_DEF} & \vdash & (c1 \;\mathsf{seq}\; c2)\; q \;=\; c1\; (c2\; q) \\
\mathsf{cond\_DEF} & \vdash & \mathsf{cond}\; g\; c1\; c2\; q \;=\; (g \;\mathsf{andd}\; (c1\; q)) \;\mathsf{or}\; ((\mathsf{not}\; g) \;\mathsf{andd}\; (c2\; q)) \\
\mathsf{mu\_DEF} & \vdash & \mathsf{mu}\; g\; c\; q \;=\; \mathsf{Dch}\; (\lambda c.\; \mathsf{monotonic}\; c \wedge (f\; c) \;\mathsf{ref}\; c) \\
\mathsf{do\_DEF} & \vdash & \mathsf{do}\; g\; c\; q \;=\; \mathsf{mu}\; (\lambda x.\; \mathsf{cond}\; g\; (c \;\mathsf{seq}\; x)\; \mathsf{skip}) \\
\mathsf{block\_DEF} & \vdash & \mathsf{block}\; p\; c\; q \;=\; (\lambda v.\; \forall x.\; p(x,v) \Rightarrow c(\lambda(x',v').\; q\; v')(x,v))
\end{array}
$$

Note that iteration is defined as a special case of recursion. Recursion (mu) is in turn defined (in closed form) as a least fixpoint. We shall see below that mu is in fact a least fixpoint operator.

Healthiness properties Operators for the healthiness properties are defined as follows (monotonicity was defined already in the RCpredicate theory).

$$
\begin{array}{lll}
\mathsf{strict\_DEF} & \vdash & \mathsf{strict}\; c \;=\; (c\; \mathsf{false} \;=\; \mathsf{false}) \\
\mathsf{terminating\_DEF} & \vdash & \mathsf{terminating}\; c \;=\; (c\; \mathsf{true} \;=\; \mathsf{true}) \\
\mathsf{conjunctive\_DEF} & \vdash & \mathsf{conjunctive}\; c \;= \\
 & & \quad (\forall P.\; (\exists p.\; P\; p) \Rightarrow (c\; (\mathsf{glb}\; P) = \mathsf{glb}\; (\lambda q.\; \exists p.\; P\; p \wedge (q = c\; p)))) \\
\mathsf{uniconjunctive\_DEF} & \vdash & \mathsf{uniconjunctive}\; c \;= \\
 & & \quad (\forall P.\; c\; (\mathsf{glb}\; P) = \mathsf{glb}\; (\lambda q.\; \exists p.\; P\; p \wedge (q = c\; p)))
\end{array}
$$

Thus, conjunctive stands for (positive) conjunctivity while uniconjunctive stands for universal conjunctivity.

Functions on commands Furthermore, we need a few definitions for reasoning about recursion.

$$
\begin{array}{lll}
\mathsf{pmonotonic\_DEF} & \vdash & \mathsf{pmonotonic}\; f \;= \\
 & & \quad \forall c\; c'.\; \mathsf{monotonic}\; c \;\wedge\; \mathsf{monotonic}\; c' \;\wedge\; c \;\mathsf{ref}\; c' \;\Rightarrow\; (f\; c) \;\mathsf{ref}\; (f\; c') \\
\mathsf{mono\_mono\_DEF} & \vdash & \mathsf{mono\_mono}\; f \;=\; \forall c.\; \mathsf{monotonic}\; c \Rightarrow \mathsf{monotonic}\; (f\; c) \\
\mathsf{regular\_DEF} & \vdash & \mathsf{regular}\; f \;=\; \mathsf{pmonotonic}\; f \;\wedge\; \mathsf{mono\_mono}\; f
\end{array}
$$

Thus, a function on commands is regular if it maps monotonic commands to monotonic commands and its restriction to monotonic commands is monotonic.

3.2 Theorems

The basic fact that ref is a partial order is encoded in the following theorem:

$$\begin{array}{lll} \mathsf{ref\_prop} & \vdash & (\forall c.\ c\ \mathsf{ref}\ c) \wedge \\ & & (\forall c\ c'.\ c\ \mathsf{ref}\ c' \wedge c'\ \mathsf{ref}\ c \Rightarrow (c = c')) \wedge \\ & & (\forall c\ c'\ c''.\ c\ \mathsf{ref}\ c' \wedge c'\ \mathsf{ref}\ c'' \Rightarrow c\ \mathsf{ref}\ c'')) \end{array}$$

Algebraic properties of commands Commands have a large number of algebraic properties. Many of these results can be directly derived from basic lattice and function theory (commutativity, associativity etc). As example we have proved two basic properties of sequential composition:

$$\begin{array}{lll} \mathsf{seq\_assoc} & \vdash & c1\ \mathsf{seq}\ (c2\ \mathsf{seq}\ c3) = (c1\ \mathsf{seq}\ c2)\ \mathsf{seq}\ c3 \\ \mathsf{skip\_unit} & \vdash & (\mathsf{skip}\ \mathsf{seq}\ c = c) \wedge (c\ \mathsf{seq}\ \mathsf{skip} = c) \end{array}$$

We also have some special degenerate cases for a number of commands:

$$\begin{array}{lll} \mathsf{cond\_true} & \vdash & \mathsf{cond}\ \mathsf{true}\ c1\ c2 = c1 \\ \mathsf{cond\_false} & \vdash & \mathsf{cond}\ \mathsf{false}\ c1\ c2 = c2 \\ \mathsf{cond\_same} & \vdash & \mathsf{cond}\ g\ c\ c = c \\ \mathsf{do\_false} & \vdash & \mathsf{monotonic}\ c\ \Rightarrow\ (\mathsf{do}\ \mathsf{false}\ c = \mathsf{skip}) \\ \mathsf{do\_true} & \vdash & \mathsf{monotonic}\ c\ \wedge\ \mathsf{strict}\ c\ \Rightarrow\ (\mathsf{do}\ \mathsf{true}\ c = \mathsf{abort}) \end{array}$$

Rewriting commands Our command notation is redundant. For example, skip and abort can both be written as assert commands. The following theorems show how commands can be reduced to other commands. Again, this is just a small part of a potentially very long list.

$$\begin{array}{lll} \mathsf{skip\_eq\_assert} & \vdash & \mathsf{skip} = \mathsf{assert}\ \mathsf{true} \\ \mathsf{assign\_eq\_nondass} & \vdash & \mathsf{assign}\ e = \mathsf{nondass}\ \lambda u\ u'.\ u' = e\ u \end{array}$$

We can also rewrite a sequence of two assignments as a single assignment:

$$\mathsf{assign\_seq} \quad \vdash \quad (\mathsf{assign}\ e)\ \mathsf{seq}\ (\mathsf{assign}\ f) = \mathsf{assign}\ \lambda u.\ f(e\ u)$$

Recursion and iteration The fixpoint properties of mu are as follows:

$$\begin{array}{lll} \mathsf{mu\_fp} & \vdash & \mathsf{regular}\ f \Rightarrow (f(\mathsf{mu}\ f) = \mathsf{mu}\ f) \\ \mathsf{mu\_least} & \vdash & \mathsf{monotonic}\ c \wedge ((f\ c)\ \mathsf{ref}\ c) \Rightarrow (\mathsf{mu}\ f)\ \mathsf{ref}\ c \end{array}$$

These theorems rely on a number of properties of glb that are also stored in the theory file. Now we can prove that do is a least fixpoint on the predicate level:

$$\mathsf{do\_thm} \quad \vdash \quad \mathsf{monotonic}\ c \Rightarrow (\mathsf{do}\ g\ c\ q = \mathsf{fix}\ (\lambda p.\ (g\ \mathsf{andd}\ (c\ p))\ \mathsf{or}\ ((\mathsf{not}\ g)\ \mathsf{andd}\ q)))$$

From this characterisation of do it is possible to derive the following two theorems which are useful for proving properties about iterations.

$$\begin{array}{lll} \mathsf{do\_unfold} & \vdash & \mathsf{monotonic}\ c \\ & & \quad \Rightarrow (\mathsf{do}\ g\ c\ q = (g\ \mathsf{andd}\ (c(\mathsf{do}\ g\ c\ q)))\ \mathsf{or}\ ((\mathsf{not}\ g)\ \mathsf{andd}\ q))) \\ \mathsf{do\_implies} & \vdash & \mathsf{monotonic}\ c \wedge ((g\ \mathsf{andd}\ (c\ q))\ \mathsf{or}\ ((\mathsf{not}\ g)\ \mathsf{andd}\ p))\ \mathsf{implies}\ q) \\ & & \quad \Rightarrow (\mathsf{do}\ g\ c\ p)\ \mathsf{implies}\ q \end{array}$$

Healthiness properties In some reasoning we need to know that a certain command satisfies a healthiness propertie, e.g., monotonicity or conjunctivity. Fir this, we have proved a number of theorems about healthiness.

First, we have theorems relating the different kinds of healthiness:

$$\begin{array}{lll} \mathsf{conj\_mono} & \vdash & \mathsf{conjunctive}\ c \Rightarrow \mathsf{monotonic}\ c \\ \mathsf{uniconj\_mono} & \vdash & \mathsf{uniconjunctive}\ c \Rightarrow \mathsf{monotonic}\ c \\ \mathsf{uniconj\_conj} & \vdash & \mathsf{uniconjunctive}\ c \Rightarrow \mathsf{conjunctive}\ c \end{array}$$

Then we have theorems about the commands. Essentially, we have theorems saying that all commands built up using the notation introduced so far are monotonic, strict and conjunctive. As examples, we show two monotonicity theorems:

$$\begin{array}{lll} \mathsf{mono\_abort} & \vdash & \mathsf{monotonic\ abort} \\ \mathsf{mono\_seq} & \vdash & \mathsf{monotonic}\ c \wedge \mathsf{monotonic}\ c' \Rightarrow \mathsf{monotonic}\ (c\ \mathsf{seq}\ c') \end{array}$$

The following is a list of monotonicity, strictness and conjunctivity theorems that we have proved. The reader should be aware that the list is not complete. For example, the demonic and angelic choice operators also preserve monotonicity.

mono_abort	strict_abort	conj_abort
mono_skip	strict_skip	conj_skip
mono_assert	strict_assert	conj_assert
mono_assign	strict_assign	conj_assign
mono_nondass	strict_nondass	conj_nondass
mono_seq	strict_seq	conj_seq
mono_cond	strict_cond	conj_cond
mono_do	strict_do	conj_do
mono_block	strict_block	conj_block

A normal form theorem Finally, we show a theorem that is needed in some proofs about universally conjunctive commands:

$$\mathsf{nondass\_complete} \quad \vdash \quad \mathsf{uniconjunctive}\ c \Rightarrow (c = \mathsf{nondass}\ \lambda u\ u'.\ \mathsf{glb}\ (\lambda q.\ c\ q\ u)\ u'))$$

This theorem states that every universally conjunctive command can be written as a nondeterministic assignment. Thus nondass is a kind of normal form for universally comjunctive commands.

3.3 ML functions and tactics

We have written ML conversions that prove monotonicity, strictness and conjunctivity of arbitrary commands that are built up using the core of our command notation: abort, skip, assert, assign, nondass, seq, cond, do and block. These conversion have the following names:

mono_CONV
strict_CONV
conj_CONV

The following example shows how this tactic can be used:

```
#let thm = prove_mono "do (\(x,y). x>0) (assign \(x,y). (x-1,y+a))";;
thm = |- monotonic (do (\(x,y). x>0) (assign \(x,y). (x-1,y+a)))
```

Corresponding to these three conversions, there are tactics for proving monotonicity, strictness and conjunctivity goals:

mono_TAC
strict_TAC
conj_TAC

3.4 Example

As an example, we show how commands can be evaluated in predicates, i.e., how weakest preconditions can be computed. This example can be found in the file RCcommand_ex1.ml.

We consider the problem of computing the weakest precondition of the command $x := x + y; y := x + y$ with respect to the postcondition $y \leq x$. We do this as a forward proof, starting from reflexivity.

```
#let th1 = REFL
#  "((assign \(x,y).(x+y,y)) seq
#   (assign\(x,y).(x,x+y)))(\(x,y).y<=x)";;
th1 =
|- ((assign(\(x,y). (x + y,y))) seq (assign(\(x,y). (x,x + y))))
   (\(x,y). y <= x) =
   ((assign(\(x,y). (x + y,y))) seq (assign(\(x,y). (x,x + y))))
   (\(x,y). y <= x)
```

Next we rewrite on the right-hand side using definitions of seq and assign.

```
#let th2 = RHS_RRL[seq_DEF;assign_DEF] th1;;
th2 =
|- ((assign(\(x,y). (x + y,y))) seq (assign(\(x,y). (x,x + y))))
   (\(x,y). y <= x) =
   (\v.
     (\v. (\(x,y). y <= x)((\(x,y). (x,x + y))v))((\(x,y). (x + y,y))v))
```

Now beta reduce the result using PBETA_RULE (tuple handling beta rule from the "pair" library). We also rewrite using standard rewrite theorems (this gets rid of some extra FST and SND).

```
#let th3 = RR[](PBETA_RULE th2);;
th3 =
|- ((assign(\(x,y). (x + y,y))) seq (assign(\(x,y). (x,x + y))))
   (\(x,y). y <= x) =
   (\v. (((FST v) + (SND v)) + (SND v)) <= ((FST v) + (SND v)))
```

Now we need some extra theorems to rewrite the right-hand side. ADD_MONO_LESS_EQ, ADD_SYM, NOT_LESS_0, LESS_OR_EQ and ADD_CLAUSES are all preproved theorems.

```
#let th4 = PORR[ADD_SYM]ADD_MONO_LESS_EQ;;
th4 = |- !m n p. (n + m) <= (p + m) = n <= p

#let th5 = RR[NOT_LESS_0](RHS_RRL[LESS_OR_EQ](RR[ADD_CLAUSES]
   (SPECL["m:num";"n:num";"0"] ADD_MONO_LESS_EQ)));;
th5 = |- (m + n) <= m = (n = 0)
```

Now we use these theorems for rewriting.

```
#let thm1 = RR[th4;th5] th3;;
thm1 =
|- ((assign(\(x,y). (x + y,y))) seq (assign(\(x,y). (x,x + y))))
   (\(x,y). y <= x) =
   (\v. SND v = 0)
```

Finally, we use paired alpha conversion to introduce the same state tuple as before.

```
#let thm2 =
  RR[] (CONV_RULE(RHS_CONV(PALPHA_CONV "x:num,y:num"))thm1);;
thm2 =
|- ((assign(\(x,y). (x + y,y))) seq (assign(\(x,y). (x,x + y))))
   (\(x,y). y <= x) =
   (\(x,y). y = 0)
```

In traditional programming notation, our computation shows that

$$\mathrm{wp}(x := x + y;\, y := x + y, y \leq x) \;=\; (y = 0)$$

4 Total correctness: the RCcorrect theory

The aim of the RCcorrect theory is to provide infrastructure for reasoning about the total correctness of commands.

4.1 Definitions

The defining theorem for total correctness is

$$\mathsf{correct\_DEF} \quad \vdash \quad \mathsf{correct}\ p\ c\ q \;=\; p\ \mathsf{implies}\ (c\ q)$$

Thus $\mathsf{correct}\ p\ c\ q$ stands for the (total correctness) Hoare triple $\{p\}c\{q\}$.

4.2 Theorems

The Rcorrect theory contains one total correctness theorem for each of the program constructs that we intend to use in programs:

$$
\begin{array}{lll}
\textsf{correct\_assign} & \vdash & (\forall v.\ p\ v \Rightarrow q(e\ v)) \Rightarrow \textsf{correct}\ p\ (\textsf{assign}\ e)\ q \\
\textsf{correct\_nondass} & \vdash & (\forall v.\ p\ v\ \Rightarrow\ (\forall v'.\ P\ v\ v'\ \Rightarrow\ q\ v'))\ \Rightarrow\ \textsf{correct}\ p\ (\textsf{nondass} P)\ q \\
\textsf{correct\_seq} & \vdash & \textsf{monotonic}\ c1 \wedge \textsf{correct}\ p\ c1\ q \wedge \textsf{correct}\ q\ c2\ r \\
 & & \quad \Rightarrow \textsf{correct}\ p\ (c1\ \textsf{seq}\ c2)\ r \\
\textsf{correct\_cond} & \vdash & \textsf{correct}\ (g\ \textsf{andd}\ p)\ c1\ q \wedge \textsf{correct}\ ((\textsf{not}\ g)\ \textsf{andd}\ p)\ c2\ q \\
 & & \quad \Rightarrow \textsf{correct}\ p\ (\textsf{cond}\ g\ c1\ c2)\ q \\
\textsf{correct\_do} & \vdash & \textsf{monotonic}\ c\ \wedge \\
 & & (\exists inv\ t.\ p\ \textsf{implies}\ inv \wedge ((\textsf{not}\ g)\ \textsf{andd}\ inv)\ \textsf{implies}\ q\ \wedge \\
 & & (\forall x.\ \textsf{correct}\ (inv\ \textsf{andd}\ (g\ \textsf{andd}\ (\lambda u.\ t\ u = x))) \\
 & & \qquad\qquad c \\
 & & \qquad\qquad (inv\ \textsf{andd}\ (\lambda u.\ (t\ u) < x)))) \\
 & & \quad \Rightarrow \textsf{correct}\ p\ (\textsf{do}\ g\ c)\ q \\
\textsf{correct\_block} & \vdash & \textsf{correct}\ (\lambda(x,u).\ p\ u \wedge p0(x,u))\ c\ (\lambda(x,u).\ q\ u) \\
 & & \quad \Rightarrow \textsf{correct}\ p\ (\textsf{block}\ p0\ c)\ q
\end{array}
$$

Each theorem can be seen as a rule in a Hoare logic of total correctness. For example, correct_seq corresponds to the sequencing rule

$$\frac{\{p\}c_1\{q\} \quad \{q\}c_2\{r\}}{\{p\}c_1;\ c_2\{r\}}$$

Note, however, that we have an explicit monotonicity requirement in the HOL theorem corresponding to this rule.

The correctness rule for do uses a termination function t that ranges over the natural numbers with the ordinary "less-than" ordering $<$. In the Appendix we show how well-founded orderings can be used. The use of well-founded orderings is necessary in some cases involving unbounded nondeterminism.

4.3 ML functions and tactics

Using the theorems above (the rules for proving total correctness), we can decompose a total correctness proof of a complex command into total correctness proofs of the basic assignments involved. Of course, this requires that we can supply the right intermediate predicate in the rule for seq and the right invariant and termination function in the rule for do.

We have made tactics for reducing total correctness goals involving the five first constructors that we have rules for. The calls to these tactics are as follows:

assign_correct_TAC *thl*
nondass_correct_TAC *thl*
seq_correct_TAC *med*
cond_correct_TAC
do_correct_TAC *inv t thl*

Basically, these tactics match the goal against the appropriate rule and reduces the goal to sobgoals. In seq_correct_TAC the user supplies the intermediate predicate *med* and in do_correct_TAC the user supplies the invariant *inv* and the termination function t. The tactics for seq and do proves monotonicity subgoal automatically. The argument *thl* in some of the calls is a list of theorems supplied by the user. This list is used for rewriting purposes.

4.4 Example

As an example, we show the outline of a total correctness proof. The whole example can be found in the file RCcorrect_ex1.ml.

We want to show that the algorithm for multiplication by repeated addition,

$$\textbf{do}\ x > 0 \rightarrow x, r := x - 1, r + Y\ \textbf{od}$$

is correct with respect to precondition $x = X$ and postcondition $r = X * Y$.

We begin by expressing the algorithm and the pre- and postconditions in our command notation. Note that Y is represented by a global constant; the state has only two components (x and r).

```
let LOOP = "do (\(x:num,r:num). x>0) (assign (\(x,r).(x-1,r+Y)))";;
let P = "\(x:num,r:num). (x=X) /\ (r=0)";;
let Q = "\(x:num,r:num). r=X*Y";;
```

We obviously need a loop invariant and a termination (bound) function.

```
let INV = "\(x:num,r:num). (x<=X) /\ (r=(X-x)*Y)";;
let BOUND = "\(x:num,r:num).x";;
```

Now we set up the goal and match it against the rule of total correctness for iteration.

```
#g "correct ^P ^LOOP ^Q";;
...

#e(do_correct_TAC INV BOUND [LESS_OR_EQ;SUB_EQUAL_0;MULT]);;
OK..
2 subgoals
"!x.
  correct
  ((\(x',r). x' <= X /\ (r = (X - x') * Y)) andd
   ((\(x',r). x' > 0) andd (\s. (\(x,r). x)s = x)))
  (assign(\(x',r). (x' - 1,r + Y)))
  ((\(x',r). x' <= X /\ (r = (X - x') * Y)) andd
   (\s. ((\(x,r). x)s) < x))"

"(X - (FST v)) * Y = X * Y"
    [ "~(FST v) > 0" ]
```

```
[ "(FST v) <= X" ]
[ "SND v = (X - (FST v)) * Y" ]
```

The first (bottom) subgoal is easily solved using arithmetic reasoning. The second subgoal is then matched against the total correctness rule for assignment.

```
#e(GEN_TAC THEN assign_correct_TAC [] THEN RT[and_DEF;assign_DEF]
#  THEN PBETA_TAC THEN RT[]);;
OK..
"!v.
  ((FST v) <= X /\ (SND v = (X - (FST v)) * Y)) /\
  (FST v) > 0 /\
  (FST v = x) ==>
  (((FST v) - 1) <= X /\ ((SND v) + Y = (X - ((FST v) - 1)) * Y)) /\
  ((FST v) - 1) < x"
```

The remaining goal is not very comples, but it requires some tedious arithmetic reasoning.

5 Laws of refinement: the RCrefine theory

The theory RCrefine collects theorems that can be considered to be refinement laws, i.e., theorems that express some general fact about when a command is refined by another command. This theory has no new definitions.

5.1 Theorems

The most basic theorem is the one which links total correctness and refinement.

$$\mathsf{ref\_correct} \quad \vdash \quad (\forall p\ q.\ \mathsf{correct}\ p\ c\ q \Rightarrow \mathsf{correct}\ p\ c'\ q) \ = \ c\ \mathsf{ref}\ c'$$

This theorem can be seen as a rule for reducing refinement to total correctness.

Rules for implementing specifications A specification can generally be expressed as an assignment. The following theorems are rules for implementing such specifications:

$$\begin{array}{lll} \mathsf{impl\_assign} & \vdash & \mathsf{monotonic}\ c\ \wedge\ (\forall u0.\ \mathsf{correct}\ (\lambda u.\ u = u0)\ c\ (\lambda u.\ u = e\ u0)) \\ & & \quad \Rightarrow\ (\mathsf{assign}\ e)\ \mathsf{ref}\ c' \\ \mathsf{impl\_nondass} & \vdash & \mathsf{monotonic}\ c\ \wedge\ (\forall u0.\ \mathsf{correct}\ (\lambda u.\ u = u0)\ c\ (\lambda v.\ P\ u0\ v)) \\ & & \quad \Rightarrow\ (\mathsf{nondass}\ P)\ \mathsf{ref}\ c' \end{array}$$

Rules for assertions Assertion (assert commands) provide a way of making information about the state visible. Below is a collection of rules that show how assertions can be introduced and propagated in commands. The most basic rule is one which relates assertions and total correctness:

$$\begin{array}{lll} \mathsf{assert\_fwd} & \vdash & \mathsf{conjunctive}\ c\ \wedge\ \mathsf{correct}\ p\ c\ q \\ & & \quad \Rightarrow\ ((\mathsf{assert}\ p)\ \mathsf{seq}\ c)\ \mathsf{ref}\ (c\ \mathsf{seq}\ (\mathsf{assert}\ q)) \end{array}$$

This can be seen as a rule for propagating assertions forward through commands. A dual rule for propagating assertions backward is

$$\mathsf{assert\_bwd} \quad \vdash \quad \mathsf{conjunctive}\ c \ \Rightarrow\ (c\ \mathsf{seq}\ (\mathsf{assert}\ p)\ =\ (\mathsf{assert}\ (c\ p))\ \mathsf{seq}\ c)$$

We have a rule for moving assertions into conditionals and iterations.

$$\begin{array}{lll}
\mathsf{assert_cond} & \vdash & ((\mathsf{assert}\ p)\ \mathsf{seq}\ (\mathsf{cond}\ g\ c1\ c2))\ \mathsf{ref} \\
 & & (\mathsf{cond}\ g\ ((\mathsf{assert}\ (p\ \mathsf{andd}\ g))\ \mathsf{seq}\ c1)\ ((\mathsf{assert}\ (p\ \mathsf{andd}\ (\mathsf{not}\ g)))\ \mathsf{seq}\ c2)) \\
\mathsf{assert_do} & \vdash & \mathsf{conjunctive}\ c\ \wedge\ (\mathsf{correct}\ (p\ \mathsf{andd}\ g)\ c\ p) \\
 & & \Rightarrow\ ((\mathsf{assert}\ p)\ \mathsf{seq}\ (\mathsf{do}\ g\ c))\ \mathsf{ref} \\
 & & \quad (\mathsf{do}\ g((\mathsf{assert}\ (p\ \mathsf{andd}\ g))\ \mathsf{seq}\ c))
\end{array}$$

Note that in the rule for iteration, p plays the role of an invariant.

Finally, we have some rules for manipulating and eliminating assertions:

$$\begin{array}{lll}
\mathsf{assert_weaken} & \vdash & (p\ \mathsf{implies}\ q)\ \Rightarrow\ (\mathsf{assert}\ p)\ \mathsf{ref}\ (\mathsf{assert}\ q) \\
\mathsf{assert_split} & \vdash & \mathsf{assert}\ (p\ \mathsf{andd}\ q)\ =\ (\mathsf{assert}\ p)\ \mathsf{seq}\ (\mathsf{assert}\ q) \\
\mathsf{assert_remove} & \vdash & (\mathsf{assert}\ p)\ \mathsf{ref}\ \mathsf{skip}
\end{array}$$

6 Functional abstraction: the RCfunction theory

The RCfunction theory provides what is needed to encapsulate commands that implement given functions.

6.1 Definitions

Assume that a program (command) c is supposed to compute a function f. We take the view that functions are uncurried and they always have at least one argument. This means that we can without loss of generality assume that f has type $*s1 \rightarrow *s2$. In an instantiation, the initial state : $*s1$ contains one component for each argument of the function, while the final state : $*s2$ has only one component (the result returned by the function). Since this component may contain subcomponents, a multi-valued function can be treated as a single-valued function whose value is a tuple.

Now assume that we have written a program fragment (command) c of which we can prove the following:

$$\vdash \quad c = \mathsf{assign}\ f$$

Then it is fair to say that c is an implementation of the function f, and that any occurrence of f in a program can be replaced by a call to c.

This idea has been formalized in the following way. We define an operator fcall, which extracts the function f from a command c in the case when $c = \mathsf{assign}\ f$ holds.

$$\mathsf{fcall\_DEF} \quad \vdash \quad \mathsf{fcall}\ c = (\lambda u.\ \epsilon v.\ \mathsf{glb}\ (\lambda q.\ c\ q\ u)\ v)$$

6.2 Theorems

The definition of the fcall operator is not very illuminating. However, the important thing is that we can prove the crucial theorem:

$$\textsf{assign\_fcall} \quad \vdash \quad \textsf{fcall}\ (\textsf{assign}\ f) \ = \ f$$

We have also proved the following theorems which are important as proof rules:

$$\begin{array}{rll} \textsf{fcall\_thm} & \vdash & \textsf{conjunctive}\ c \wedge \textsf{strict}\ c \wedge (\forall u0.\ \textsf{correct}\ (\lambda u.\ u = u0)\ c\ (\lambda v.\ v = f\ u0)) \\ & & \ \Rightarrow\ (\textsf{fcall}\ c = f) \\ \textsf{fcall\_rule} & \vdash & \textsf{conjunctive}\ c\ \wedge\ \textsf{strict}\ c\ \wedge\ ((\textsf{assign}\ f)\ \textsf{ref}\ c) \\ & & \ \Rightarrow\ (\textsf{fcall}\ c = f) \end{array}$$

Since conjunctivity and strictness can be proved automatically, we only have a total correctness proof obligation (or a simple refinement) left to prove, in order to verify the correctness of a functional procedure.

In the case when a functional procedure is correct only under some precondition, we can use the following theorems as proof rules:

$$\begin{array}{rll} \textsf{fcall\_thm\_pre} & \vdash & \textsf{conjunctive}\ c \wedge \textsf{strict}\ c \wedge (\forall u0.\ \textsf{correct}\ (\lambda u.\ u = u0)\ c\ (\lambda v.\ v = f\ u0)) \\ & & \ \Rightarrow\ (\forall u.\ g\ u\ \Rightarrow\ \textsf{fcall}\ c\ u = f\ u) \\ \textsf{fcall\_rule\_pre} & \vdash & \textsf{conjunctive}\ c\ \wedge\ \textsf{strict}\ c\ \wedge\ ((\textsf{assert}\ g)\ \textsf{seq}\ (\textsf{assign}\ f))\ \textsf{ref}\ c) \\ & & \ \Rightarrow\ (\forall u.\ g\ u\ \Rightarrow\ \textsf{fcall}\ c\ u = f\ u) \end{array}$$

6.3 Examples

As an example, we show the correctness proof for a simple functional procedure, which computes the length of a given list. The whole example can be found in the file RCfunction_ex1.ml.

The functional procedure is introduced as a new constant lenf, defined by the theorem

```
|- lenf =
   (assign \l.(0,l)) seq
   (do (\(r,l).~(l=[])) (assign \(r,l).(SUC r,TL l))) seq
   (assign \(r,l).r)
```

This procedure takes a parameter $l : (*)$list as argument and then computes its length in the local variable r. At the end, it returns the value or r.

The correctness theorem is

```
|- fcall lenf = LENGTH
```

where LENGTH is the built in function for the length of a list. The theorem is proved by matching against the theorem fcall_thm and then using tactics described in earlier sections; the full tactic is

```
MATCH_MP_TAC fcall_thm THEN PORT[lenf_DEF] THEN REPEAT CONJ_TAC THENL
  [conj_TAC
  ;strict_TAC
  ;GEN_TAC THEN seq_correct_TAC "\(r,l:(*)list).(r=0)/\(l=u0)" THENL
   [assign_correct_TAC []
   ;seq_correct_TAC "\(r,l:(*)list).(r=LENGTH(u0:(*)list))" THENL
    [do_correct_TAC "\(r,l:(*)list).LENGTH(u0:(*)list) = r + LENGTH l"
                    "\(r:num,l:(*)list).LENGTH l" [ADD_CLAUSES;LENGTH] THEN
     GEN_TAC THEN assign_correct_TAC [] THEN
     PRT[and_DEF] THEN GEN_TAC THEN PBETA_TAC THEN RT[] THEN
     CASE_TAC "SND(v:num#(*)list)" list_CASES [NOT_CONS_NIL;LENGTH;TL] THEN
     BOUND_TAC THEN RT[LESS_SUC_REFL;ADD_CLAUSES]
    ;assign_correct_TAC []
  ]]]
```

Here, CASE_TAC makes a case split on its first argument according to the theorem in the second argument. In the case above this theorem is list_CASES:

$$\vdash \forall l.\ (l = [\,]) \vee (\exists t\ h.\ l = \mathsf{CONS}\ h\ t)$$

The tactic BOUND_TAC makes some simplifications. Both these tactics are defined under the RCcorrect theory in the file RCcorrect_convs.ml.

7 Procedural abstraction: the RCprocedure theory

We will take the view that a procedure is a program fragment which implements (refines) a specification. A specification, in turn, is generally a nondeterministic assignment.

In this section, we will show how procedures can be handled in our HOL notation. We define a procedure call operator pcall and we show how it handles (reference) parameters. We also define a procedure call operator xpcall which can handle both value and reference parameters.

7.1 Definitions

The definition of the procedure call operators is as follows:

$$\begin{array}{lll} \mathsf{pcall\_DEF} & \vdash & \mathsf{pcall}\ c\ G\ R\ q\ u\ = \\ & & \quad \exists p.\ (\forall u'.\ p(G\ u') \wedge (R\ u' = R\ u) \Rightarrow q\ u') \wedge c\ p\ (G\ u) \\ \mathsf{xpcall\_DEF} & \vdash & \mathsf{xpcall}\ c\ G\ R\ V\ q\ u\ = \\ & & \quad \exists p.\ (\forall u'.\ p(G\ u') \wedge (R\ u' = R\ u) \Rightarrow q\ u') \wedge c\ (V\ u)\ p\ (G\ u) \end{array}$$

Here c is the procedure, G is a parameter expression and R indicates the unchanged part of the state (the *R*est). In the second definition, V is an expression which evaluates to the value parameter. The role of the expressions G, R and V will be explained by examples.

7.2 Theorems

We do not have any theorems that can be used as proof rules directly. However, the conversions described in the next section make use of theorems showing that the procedure call operators are monotonic and regular.

$$\begin{array}{rll} \mathsf{pcall\_mono} & \vdash & c \ \mathsf{ref}\ c' \ \Rightarrow \ (\mathsf{pcall}\ c\ G\ R)\ \mathsf{ref}\ (\mathsf{pcall}\ c'\ G\ R) \\ \mathsf{xpcall\_mono} & \vdash & c \ \mathsf{ref}\ c' \ \Rightarrow \ (\mathsf{xpcall}\ c\ G\ R)\ \mathsf{ref}\ (\mathsf{xpcall}\ c'\ G\ R) \\ \mathsf{regular\_pcall} & \vdash & \mathsf{regular}\ (\lambda x.\ \mathsf{pcall}\ x\ G\ R) \end{array}$$

7.3 ML functions and tactics

The ML function prove_pcall_ref automatically proves the correctness of a procedure call pcall $c\ G\ R$ in a given state space, given a procedure correctness theorem

$$\vdash ((\mathsf{assert}\ g)\ \mathsf{seq}\ (\mathsf{nondass}\ P))\ \mathsf{ref}\ c$$

and a parameter expression G. The call to this ML function is

$$\mathsf{prove\_pcall\_ref}\ \mathit{thm}\ G$$

where *thm* is the name of the procedure correctness theorem.

Similarly, the call

$$\mathsf{prove\_xpcall\_ref}\ \mathit{thm}\ G\ V$$

proves the correctness of a procedure call pcall $c\ G\ R\ V$ in a given state space, given a procedure correctness theorem *thm*, a parameter expression G and a value parameter expression V.

7.4 Examples

As an example, we show the correctness proof for a simple functional procedure, which reverses a given list. The whole example can be found in the file RCfunction_ex1.ml, together with examples of recursive procedures and procedures with value parameters.

The procedure is introduced as a new constant preverse, defined by the theorem

```
|- preverse =
   (assign \l. ([],l)) seq
   (do (\(rl,l). ~(l = [])) (assign \(rl,l). (CONS(HD l)rl,TL l))) seq
   (assign \(rl,l). rl))
```

We first prove a correctness theorem for this procedure, stating that it refines a nondeterministic assignments which reverses a list directly:

```
thm2 = |- (nondass(\l l'. l' = REVERSE l)) ref preverse
```

This theorem states that preverse implements an abstract specification of list reversal (the HOL REVERSE function from the more_lists library is used here). For the details of the proof of thm2 we refer to the example file.

Now assume that we want to call preverse to reverse the second component in a three-component state space. We then need to use a "gluing" function which indicates that the second state components is to be mapped to the parameter of the procedure:

```
#let G = "\(x:num,y:(num)list,z:bool).y";;
```

Now we get the correctness theorem for the call automatically:

```
#prove_pcall_ref thm2 G;;
|- (nondass
    (\(x,y,z) (x',y',z'). (y' = REVERSE y) /\ (x = x') /\ (z = z'))) ref
   (pcall preverse(\(x,y,z). y)(\(x,y,z). (x,z)))
```

This theorem shows that the call

```
pcall preverse (\(x,y,z). y) (\(x,y,z). (x,z))
```

actually reverses the second state component and leaves the two other components unchanged. The last argument of the call is the dual gluing function which indicates what state components are lect untouched (it is computed by prove_pcall_ref).

8 Data refinement: the RCdataref theory

The RCdataref theory defines a notion of data refinement that is based on the theory in [4], using abstraction commands.

8.1 Definitions

We assume that the abstract state is of type $:*a\times:*s$ while the concrete state is of type $:*k\times:*s$. Abstraction relations are then typed $r:*a\times*k\times*s\rightarrow bool$. We first define abstraction and representation commands (parameterised by the abstraction relation).

$$\begin{array}{lll} \mathsf{abst\_DEF} & \vdash & \mathsf{abst}\ r\ q\ =\ \lambda(k,u).\ \exists a.\ r(a,k,u)\ \wedge\ q(a,u) \\ \mathsf{repr\_DEF} & \vdash & \mathsf{repr}\ r\ q\ =\ \lambda(a,u).\ \forall k.\ r(a,k,u)\ \Rightarrow\ q(k,u) \end{array}$$

The data refinement relation is now defined as follows:

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

8.2 Theorems

Abstraction and representation commands are monotonic. Furthermore, the representation command is conjunctive, while the abstraction command is disjunctive (i.e., it distributes over lub's of predicates. These facts are stored in theorems mono_abst, conj_repr etc.

The fact that abstraction and representation are a Galois connection is the content of the following theorem.

$$\textsf{abst\_repr} \quad \vdash \quad ((\textsf{abst}\ r)\ \textsf{seq}\ (\textsf{repr}\ r))\ \textsf{ref skip}\ \wedge\ \textsf{skip ref}\ ((\textsf{repr}\ r)\ \textsf{seq}\ (\textsf{abst}\ r))$$

This fact can be used to prove an alternative characterisation of data refinement:

$$\begin{aligned} \textsf{dataref\_alt} \quad \vdash \quad & \textsf{monotonic}\ c\ \wedge\ \textsf{monotonic}\ c' \\ & \Rightarrow\ \textsf{dataref}\ r\ c\ c'\ =\ ((\textsf{abst}\ r)\ \textsf{seq}\ c\ \textsf{seq}\ (\textsf{repr}\ r))\ \textsf{ref}\ c' \end{aligned}$$

Rules of data refinement For each program construct, we have a rule of data refinement. This rule tells us under what conditions a data refinement is permitted.

$$\begin{array}{rcl} \textsf{abort\_dataref} & \vdash & \textsf{dataref}\ r\ \textsf{abort abort} \\ \textsf{skip\_dataref} & \vdash & \textsf{dataref}\ r\ \textsf{skip skip} \\ \textsf{nondass\_dataref} & \vdash & (\forall a\ k\ k'\ u\ u'.\ r(a,k,u)\ \wedge\ Q(k,u)(k',u') \\ & & \quad \Rightarrow\ \exists a'.\ r(a',k',u')\ \wedge\ P(a,u)(a',u')) \\ & & \ \Rightarrow\ \textsf{dataref}\ r\ (\textsf{nondass}\ P)\ (\textsf{nondass}\ Q) \\ \textsf{seq\_dataref} & \vdash & \textsf{monotonic}\ c1'\ \wedge\ \textsf{dataref}\ r\ c1\ c1'\ \wedge\ \textsf{dataref}\ r\ c2\ c2' \\ & & \ \Rightarrow\ \textsf{dataref}\ r\ (c1\ \textsf{seq}\ c2)\ (c1'\ \textsf{seq}\ c2') \\ \textsf{cond\_dataref} & \vdash & \textsf{dataref}\ r\ c1\ c1'\ \wedge\ \textsf{dataref}\ r\ c2\ c2'\ \wedge \\ & & (\forall a\ k\ u.\ r(a,k,u)\ \Rightarrow\ (g(a,u) = g'(k,u)) \\ & & \ \Rightarrow\ \textsf{dataref}\ r\ (\textsf{cond}\ g\ c1\ c2)\ (\textsf{cond}\ g'\ c1'\ c2') \\ \textsf{mu\_dataref} & \vdash & \textsf{regular}\ f' \wedge (\forall c\ c'.\ \textsf{dataref}\ r\ c\ c'\ \Rightarrow\ \textsf{dataref}\ r\ (f\ c)\ (f'\ c') \\ & & \ \Rightarrow\ \textsf{dataref}\ r\ (\textsf{mu}\ f)\ (\textsf{mu}\ f') \\ \textsf{do\_dataref} & \vdash & \textsf{monotonic}\ c\ \wedge\ \textsf{dataref}\ r\ c\ c'\ \wedge \\ & & (\forall a\ k\ u.\ r(a,k,u)\ \Rightarrow\ (g(a,u) = g'(k,u)) \\ & & \ \Rightarrow\ \textsf{dataref}\ r\ (\textsf{do}\ g\ c)\ (\textsf{do}\ g'\ c') \end{array}$$

These theorems allow us to reduce data refinement of a compound command to data refinement of the basic commands (nondaterministic assignments). Since assignments are a special case of nondeterministic assignments, we can always data-refine an assignment by rewriting it as a nondeterministic assignment and then do the data refinement. The result is then a nondeterministic assignment which can be refined into an assignment by making it more deterministic.

For specifications, we have derived the following rule of data refinement:

$$\begin{aligned} \textsf{spec\_dataref} \quad \vdash \quad & (\forall a\ k\ u.\ r(a,k,u)\ \wedge\ p(a,u)\ \Rightarrow\ q(k,u))\ \wedge \\ & \ (\forall a\ k\ k'\ u\ u'.\ r(a,k,u)\ \wedge\ p(a,u)\ \wedge\ q(k,u)\ \wedge\ Q(k,u)(k',u') \\ & \quad \Rightarrow\ \exists a'.\ r(a',k',u')\ \wedge\ P(a,u)(a',u')) \\ & \Rightarrow\ \textsf{dataref}\ r\ ((\textsf{assert}\ p)\ \textsf{seq}\ (\textsf{nondass}\ P))\ ((\textsf{assert}\ q)\ \textsf{seq}\ (\textsf{nondass}\ Q)) \end{aligned}$$

Data refinement of local variables Finally, we have the result which connects data refinement to ordinary refinement; we can refine a block with local variables by data refining the contents of the block.

$$
\begin{array}{lll}
\textsf{block\_dataref} & \vdash & \textsf{monotonic}\ c\ \wedge\ \textsf{dataref}\ r\ c\ c' \\
& & \wedge\ (\forall k\ u.\ p'(k,u)\ \Rightarrow\ \exists a.\ r(a,k,u)\ \wedge\ p(a,u)) \\
& & \Rightarrow\ (\textsf{block}\ p\ c)\ \textsf{ref}\ (\textsf{block}\ p'\ c')
\end{array}
$$

8.3 ML functions and tactics

The only ML functions that belong to the data refinement theory is an enhanced monotonicity prover mono_CONV which can also handle abstraction and representation commands. In Section 11 we will use window inference to perform data refinements.

8.4 Examples

For examples on data refinement, we refer to section 11.4.

9 Bounded nondeterminism and backward data refinement: the RCbounded theory

Dijkstra's original theory of guarded commands [7] assumed that commands were continuous, i.e., that nondeterminism was bounded. In this section we consider a few special features of bounded nondeterminism, e.g., a weaker semantics for iteration and backward data refinement.

9.1 Definitions

Continuity (upward continuity) is defined through the concepts of upward (ascending) chains and upward limits.

$$
\begin{array}{rll}
\textsf{uchain\_DEF} & \vdash & \textsf{uchain}\ Q\ =\ \forall n.\ (Q\ n)\ \textsf{implies}\ Q(\textsf{SUC}\ n) \\
\textsf{ulimit\_DEF} & \vdash & \textsf{ulimit}\ Q\ =\ \lambda s.\ \exists n.\ Q\ n\ s \\
\textsf{ucontinuous\_DEF} & \vdash & \textsf{ucontinuous}\ f\ = \\
& & \quad \forall Q.\ \textsf{uchain}\ Q\ \Rightarrow\ (f\ (\textsf{ulimit}\ Q) = \textsf{ulimit}\ (\lambda n.\ f\ (Q\ n)))
\end{array}
$$

where $Q : \mathit{num} \rightarrow \mathit{pred}$ is an upward chain of predicates.

Backward data refinement is defined using duals of abstraction commands.

$$
\begin{array}{rll}
\textsf{dual\_DEF} & \vdash & \textsf{dual}\ c\ q =\ \textsf{not}\ (c\ (\textsf{not}\ q)) \\
\textsf{bwdref\_DEF} & \vdash & \textsf{bwdref}\ r\ c\ c'\ = \\
& & ((\textsf{dual}\ (\textsf{abst}\ r))\ \textsf{seq}\ c)\ \textsf{ref}\ (c'\ \textsf{seq}\ (\textsf{dual}(\textsf{abst}\ r)))
\end{array}
$$

We also have a number of auxiliary definitions, that will be needed when reasoning about the semantics of iteration and in backward data refinement.

$$
\begin{array}{rll}
\mathsf{H\_DEF} & \vdash & (\mathsf{H}\ g\ c\ 0\ q = (\mathsf{not}\ g)\ \mathsf{andd}\ q) \wedge \\
 & & (\mathsf{H}\ g\ c\ (\mathsf{SUC}\ n)\ q\ =\ ((\mathsf{not}\ g)\ \mathsf{andd}\ q)\ \mathsf{or}\ (g\ \mathsf{andd}\ (c\ (\mathsf{H}\ g\ c\ n\ q)))) \\
\mathsf{max\_DEF} & \vdash & (\mathsf{max}\ N\ 0\ =\ N\ 0) \wedge \\
 & & (\mathsf{max}\ N\ (\mathsf{SUC}\ n)\ =\ ((\mathsf{max}\ N\ n < N(\mathsf{SUC}\ n)) \to N(\mathsf{SUC}\ n) \mid \mathsf{max}\ N\ n)) \\
\mathsf{finite\_DEF} & \vdash & \mathsf{finite}\ A\ = \\
 & & \ \exists f\ n.\ (\forall a.\ A\ a\ \Rightarrow\ \exists i.\ i < n\ \wedge\ (f\ i = a)))
\end{array}
$$

Here H and max are defined by primitive recursion over the natural numbers. In the definition of max, we use the notation $(b \to x|y)$ for conditional expressions. H represents the function originally used by Dijkstra for the semantics of iteration [7]. $\mathsf{max}\ N\ n$ stands for the maximum of the natural numbers $\{N\ 0, \ldots, N\ n\}$, where the typing is $N : num \to num$. Finally, $\mathsf{finite}\ A$ means that the set $A : * \to bool$ is finite (a set A is finite if there exists a surjection from the natural numbers smaller than some n to A).

9.2 Theorems

We first have some basic theorems about the concepts introduced in this theory.

$$
\begin{array}{rll}
\mathsf{ucont\_mono} & \vdash & \mathsf{ucontinuous}\ c\ \Rightarrow\ \mathsf{monotonic}\ c \\
\mathsf{dual\_abst} & \vdash & \mathsf{dual}\ (\mathsf{abst}\ r)\ q\ =\ (\lambda(k,u).\ \forall a.\ r(a,k,u)\ \Rightarrow\ q(a,u)) \\
\mathsf{max\_prop} & \vdash & i \le n\ \Rightarrow\ (N\ i\ \le\ \mathsf{max}\ N\ n)
\end{array}
$$

Semantics of iteration It is well known that in the case when the body c of an iteration $\mathsf{do}\ g\ c$ is continuous (i.e., boundedly nondeterministic), the semantics of iteration can be written as the limit of a countable chain of approximations. We use the H function to state this fact:

$$
\mathsf{do\_bounded}\ \ \vdash\ \ \mathsf{ucontinuous}\ c\ \Rightarrow\ (\mathsf{do}\ g\ c\ q\ =\ \mathsf{ulimit}\ (\lambda n.\ H\ g\ c\ n\ q))
$$

Continuity Almost all the constructs we have encoundered so far are continuous.

$$
\begin{array}{rll}
\mathsf{ucont\_abort} & \vdash & \mathsf{ucontinuous\ abort} \\
\mathsf{ucont\_skip} & \vdash & \mathsf{ucontinuous\ skip} \\
\mathsf{ucont\_assign} & \vdash & \mathsf{ucontinuous}\ (\mathsf{assign}\ e) \\
\mathsf{ucont\_assert} & \vdash & \mathsf{ucontinuous}\ (\mathsf{assert}\ p) \\
\mathsf{ucont\_seq} & \vdash & \mathsf{ucontinuous}\ c\ \wedge\ \mathsf{ucontinuous}\ c'\ \Rightarrow\ \mathsf{ucontinuous}\ (c\ \mathsf{seq}\ c') \\
\mathsf{ucont\_cond} & \vdash & \mathsf{ucontinuous}\ c\ \wedge\ \mathsf{ucontinuous}\ c'\ \Rightarrow\ \mathsf{ucontinuous}\ (\mathsf{cond}\ g\ c\ c') \\
\mathsf{ucont\_do} & \vdash & \mathsf{ucontinuous}\ c\ \Rightarrow\ \mathsf{ucontinuous}\ (\mathsf{do}\ g\ c) \\
\mathsf{ucont\_abst} & \vdash & \mathsf{ucontinuous}\ (\mathsf{abst}\ r)
\end{array}
$$

For nondeterministic assignment and dual abstraction we have restricted continuity theorems.

$$
\begin{array}{rll}
\textsf{ucont\_nondass} & \vdash & (\forall u.\ \textsf{finite}\ (\lambda v.\ P\ u\ v))\ \Rightarrow\ \textsf{ucontinuous}\ (\textsf{nondass}\ P) \\
\textsf{ucont\_block} & \vdash & \textsf{ucontinuous}\ c \wedge\ (\forall u.\ \textsf{finite}\ (\lambda x.\ p(x,u))) \\
& & \ \Rightarrow\ \textsf{ucontinuous}\ (\textsf{block}\ p\ c) \\
\textsf{ucont\_dualabst} & \vdash & (\forall k\ u.\ \textsf{finite}\ (\lambda a.\ r(a,k,u)))\ \Rightarrow\ \textsf{ucontinuous}\ (\textsf{dual}\ (\textsf{abst}\ r))
\end{array}
$$

Basically, these theorems state that nondeterministic assignment and dual abstraction are continuous if the nondeterminism involved is bounded.

Backward data refinement Backward data refinement of compound commands can be decomposed in the same way as ordinary data refinement. However, in the case of iteration we have a number of extra assumptions.

$$
\begin{array}{rll}
\textsf{abort\_bwdref} & \vdash & (\forall k\ u.\ \exists a.\ r(a,k,u))\ \Rightarrow\ \textsf{bwdref}\ r\ \textsf{abort}\ \textsf{abort} \\
\textsf{skip\_bwdref} & \vdash & \textsf{bwdref}\ r\ \textsf{skip}\ \textsf{skip} \\
\textsf{nondass\_bwdref} & \vdash & (\forall a'\ k\ k'\ u\ u'.\ r(a',k',u')\ \wedge\ Q(k,u)(k',u') \\
& & \quad \Rightarrow\ \exists a.\ r(a,k,u)\ \wedge\ P(a,u)(a',u')) \\
& & \ \Rightarrow\ \textsf{bwdref}\ r\ (\textsf{nondass}\ P)\ (\textsf{nondass}\ Q) \\
\textsf{seq\_bwdref} & \vdash & \textsf{monotonic}\ c1'\ \wedge\ \textsf{bwdref}\ r\ c1\ c1'\ \wedge\ \textsf{bwdref}\ r\ c2\ c2' \\
& & \ \Rightarrow\ \textsf{bwdref}\ r\ (c1\ \textsf{seq}\ c2)\ (c1'\ \textsf{seq}\ c2') \\
\textsf{cond\_bwdref} & \vdash & \textsf{bwdref}\ r\ c1\ c1'\ \wedge\ \textsf{bwdref}\ r\ c2\ c2' \wedge \\
& & (\forall a\ k\ u.\ r(a,k,u)\ \Rightarrow\ (g(a,u) = g'(k,u)) \\
& & \ \Rightarrow\ \textsf{bwdref}\ r\ (\textsf{cond}\ g\ c1\ c2)\ (\textsf{cond}\ g'\ c1'\ c2') \\
\textsf{do\_bwdref} & \vdash & \textsf{ucontinuous}\ c\ \wedge\ \textsf{ucontinuous}\ c'\ \wedge\ \textsf{bwdref}\ r\ c\ c' \wedge \\
& & (\forall a\ k\ u.\ r(a,k,u)\ \Rightarrow\ (g(a,u) = g'(k,u)) \wedge \\
& & \textsf{ucontinuous}\ (\textsf{dual}\ (\textsf{abst}\ r))\ \wedge\ (\forall k\ u.\ \exists a.\ r(a,k,u)) \\
& & \ \Rightarrow\ \textsf{bwdref}\ r\ (\textsf{do}\ g\ c)\ (\textsf{do}\ g'\ c')
\end{array}
$$

Finally, we have the theorem which shows that backward data refinement can actually be used to refine local variables.

$$
\begin{array}{rll}
\textsf{block\_bwdref} & \vdash & \textsf{monotonic}\ c\ \wedge\ (\forall a\ k\ u.\ r(a,k,u)\ \wedge\ p'(k,u)\ \Rightarrow\ p(a,u)) \\
& & \wedge\ \textsf{bwdref}\ r\ c\ c'\ \wedge\ (\forall k\ u.\ \exists a.\ r(a,k,u)) \\
& & \Rightarrow\ (\textsf{block}\ p\ c)\ \textsf{ref}\ (\textsf{block}\ p'\ c')
\end{array}
$$

9.3 ML functions and tactics

We have written an ML conversion that proves continuity of arbitrary commands that are built up using the continuous core of our command notation: abort, skip, assert, assign, seq, cond and do. The conversion is called mono_CONV and the corresponding tactic is ucont_TAC.

9.4 Example

The example shown in this section can be found in the file RCbounded_ex1.ml.

Our example is a small backward data refinement:

$$
\begin{array}{l}
|[\ \mathbf{var}\ a.\ a \leq x;\ \mathbf{do}\ a < x \rightarrow x := x - 1\ ]| \\
\leq\ |[\ \mathbf{var}\ c.\ 0{<}x;\ \mathbf{do}\ c\ \wedge\ 0{<}x \rightarrow c, x :=?, x - 1\ ]|
\end{array}
$$

where c is a boolean variable and c :=? asigns c an arbitrary value (i.e., T or F). This is an example of where a choices is postponed in the refining loop (the final value of x is chosen in the initalisation of a, but in the second loop it is postponed). Such a situation calls for backward data refinement.

Our goal is expressed as

```
"(block(\(a,x). a <= x)(do(\(a,x). a < x)(assign \(a,x). (a,x - 1)))) ref
 (block (\(c,x). 0 < x)
        (do (\(c,x). c /\ 0 < x)
            ((assign \(c,x). (c,x - 1)) dch
            (assign \(c,x). (~c,x - 1)))))"
```

We will prove this refinement using backward data refinement and we define or abstraction relation as follows

```
#let R = "\(a,c,x). (c /\ (0<x) = (a<x)) /\ (a<=x)";;
```

This data refinement replaces the natural-valued local variable (a) by a boolean-valued one (c).

We begin by matching against the rule of backward data refinement for blocks:

```
#e(MATCH_MP_TAC actsys_bwdref THEN EXISTS_TAC R THEN REPEAT CONJ_TAC);;
```

As a result, we get seven subgoals. Two continuity goals can be solved immediately using ucont_TAC and three of the other goals require only some basic arithmetic reasoning. The two remaining goals are the following:

```
"bwdref
 (\(a,c,x). (c /\ 0 < x = a < x) /\ a <= x)
 (assign \(a,x). (a,x - 1))
 ((guard \(c,x). c /\ 0 < x) seq
  ((assign \(c,x). (c,x - 1)) dch (assign \(c,x). (~c,x - 1))))"

"ucontinuous(dual(abst(\(a,c,x). (c /\ 0 < x = a < x) /\ a <= x)))"
```

The first (bottom) goal is solved using the theorem ucont_dualabst and a theorem showing that R is image-finite, which we have proved separately:

```
finiteR =
|- finite(\a. (\(a,c,x). (c /\ 0 < x = a < x) /\ a <= x)(a,k,u))
```

The second goal is proved by rewriting the refining both commands as nondeterministic assignments and and matching against the rule for backward data refinement of nondass. After that only straightforward arithmetic reasoning remains.

10 Action systems: the RCactsys theory

The theory RCactsys extends the programming notation with Dijkstra's nondeterministic conditional composition and with iterations with more than one guarded command (action).

10.1 Definitions

In some data refinement rules we will need the function EVERY2, characterised by the following theorem.

$$
\begin{array}{lll}
\mathsf{EVERY2} & \vdash & (\mathsf{EVERY2}\ P\ []\ []\ =\ T) \wedge \\
 & & (\mathsf{EVERY2}\ P\ (\mathsf{CONS}\ x\ xl)\ (\mathsf{CONS}\ y\ yl)\ =\ P\ x\ y\ \wedge\ \mathsf{EVERY2}\ P\ xl\ yl)
\end{array}
$$

(in fact, this is not the definition of EVERY2; it is a theorem).

Next we introduce a miracle command and Dijkstra's binary nondeterministic conditional.

$$
\begin{array}{rll}
\mathsf{magic\_DEF} & \vdash & \mathsf{magic}\ q\ =\ \mathsf{true} \\
\mathsf{if2\_DEF} & \vdash & \mathsf{if2}\ (g1, c1)\ (g2, c2)\ q\ = \\
 & & \quad (g1\ \mathsf{or}\ g2)\ \mathsf{andd}\ (g1\ \mathsf{imp}\ (c1\ q))\ \mathsf{andd}\ (g2\ \mathsf{imp}\ (c2\ q))
\end{array}
$$

Note that guarded commands (actions) are now represented as pairs $a = (g, c)$ where g is the guard and c is the body.

Now we can introduce list versions of demonic choice, conditional composition and iteration. In each case, a set of commands is represented as a HOL list and the definitions use primitive recursion over list.

$$
\begin{array}{rll}
\mathsf{ldch\_DEF} & \vdash & (\mathsf{ldch}\ []\ =\ \mathsf{magic}) \wedge \\
 & & (\mathsf{ldch}\ (\mathsf{CONS}\ c\ cl)\ =\ c\ \mathsf{dch}\ (\mathsf{ldch}\ cl)) \\
\mathsf{lguard\_DEF} & \vdash & (\mathsf{lguard}\ []\ =\ \mathsf{false}) \wedge \\
 & & (\mathsf{lguard}\ (\mathsf{CONS}\ a\ al)\ =\ (\mathsf{FST}\ a)\ \mathsf{or}\ (\mathsf{lguard}\ al)) \\
\mathsf{lif\_DEF} & \vdash & (\mathsf{lif}\ []\ =\ \mathsf{abort}) \wedge \\
 & & (\mathsf{lif}\ (\mathsf{CONS}\ a\ al)\ =\ \mathsf{if2}\ a\ (\mathsf{lguard}\ al, \mathsf{lif}\ al) \\
\mathsf{ldo\_DEF} & \vdash & (\mathsf{ldo}\ al\ =\ \mathsf{do}\ (\mathsf{lguard}\ al)\ (\mathsf{lif}\ al)
\end{array}
$$

An *action system* is now any command of the form

$$\mathsf{block}\ p\ (\mathsf{ldo}\ al)$$

for some initialisation predicate p and some list of actions al.

10.2 Theorems

We first note the basic properties of the new constructs.

$$
\begin{array}{lll}
\mathsf{mono\_magic} & \vdash & \mathsf{monotonic\ magic} \\
\mathsf{mono\_if2} & \vdash & \mathsf{monotonic}\ c1\ \wedge\ \mathsf{monotonic}\ c2 \\
 & & \quad \Rightarrow\ \mathsf{monotonic}\ (\mathsf{if2}\ (g1, c1)\ (g2, c2)) \\
\mathsf{mono\_lif} & \vdash & \mathsf{EVERY\ monotonic}\ (\mathsf{MAP\ SND}\ al) \\
 & & \quad \Rightarrow\ \mathsf{monotonic}\ (\mathsf{lif}\ al) \\
\mathsf{mono\_ldo} & \vdash & \mathsf{EVERY\ monotonic}\ (\mathsf{MAP\ SND}\ al) \\
 & & \quad \Rightarrow\ \mathsf{monotonic}\ (\mathsf{ldo}\ al)
\end{array}
$$

Here, EVERY is a generalised conjunction and MAP maps an operator over a list. Both are HOL built-ins.

We can also prove conjunctivity of magic and strictness and conjunctivity properties for the if2, lif and ldo commands.

Total correctness for action systems Exactly as for the previously defined constructs (Section 4), we can decompose total correctness proofs for the new constructs.

$$
\begin{array}{lll}
\mathsf{correct\_if2} & \vdash & \mathsf{correct}\ (g1\ \mathsf{andd}\ p)\ c1\ q \wedge \mathsf{correct}\ (g2\ \mathsf{andd}\ p)\ c2\ q \\
 & & \quad \Rightarrow \mathsf{correct}\ ((g1\ \mathsf{or}\ g2)\ \mathsf{andd}\ p)\ (\mathsf{if2}\ (g1, c1)\ (g2, c2))\ q \\
\mathsf{correct\_ldo} & \vdash & \mathsf{EVERY\ monotonic}\ (\mathsf{MAP\ SND}\ al) \wedge \\
 & & p\ \mathsf{implies}\ inv \wedge ((\mathsf{not}\ (\mathsf{lguard}\ al))\ \mathsf{andd}\ inv)\ \mathsf{implies}\ q \wedge \\
 & & \mathsf{EVERY}\ (\lambda(g, c). \\
 & & \quad (\forall x.\ \mathsf{correct}\ (inv\ \mathsf{andd}\ (g\ \mathsf{andd}\ (\lambda u.\ t\ u = x))) \\
 & & \qquad\qquad c \\
 & & \qquad\qquad (inv\ \mathsf{andd}\ (\lambda u.\ (t\ u) < x)))\ al \\
 & & \Rightarrow \mathsf{correct}\ p\ (\mathsf{ldo}\ al)\ q
\end{array}
$$

Data refinement of action systems One of the main aims of our theory is to be able to do data refinements of action systems. The following theorems give us a basis for this.

$$
\begin{array}{lll}
\mathsf{magic\_dataref} & \vdash & \mathsf{dataref}\ r\ \mathsf{magic\ magic} \\
\mathsf{assert\_dataref} & \vdash & (\forall a\ k\ u.\ r(a, k, u)\ \wedge\ g(a, u)\ \Rightarrow\ g'(k, u)) \\
 & & \quad \Rightarrow\ \mathsf{dataref}\ r\ (\mathsf{assert}\ g)\ (\mathsf{assert}\ g') \\
\mathsf{lif\_dataref} & \vdash & \mathsf{EVERY\ monotonic}\ (\mathsf{MAP\ SND}\ al)\ \wedge\ (\mathsf{LENGTH}\ al = \mathsf{LENGTH}\ al') \wedge \\
 & & \mathsf{EVERY2}\ (\lambda(g, c)(g', c').\ \mathsf{dataref}\ r\ c\ ((\mathsf{guard}\ g')\ \mathsf{seq}\ c'))\ al\ al' \wedge \\
 & & \mathsf{EVERY2}\ (\lambda(g, c)(g', c').\ (\forall a\ k\ u.\ r(a, k, u)\ \wedge\ g'(k, u)\ \Rightarrow\ g(a, u)))\ al\ al' \wedge \\
 & & (\forall a\ k\ u.\ r(a, k, u)\ \wedge\ (\mathsf{lguard}\ al\ (a, u))\ \Rightarrow\ (\mathsf{lguard}\ al'\ (k, u)) \\
 & & \quad \Rightarrow\ \mathsf{dataref}\ r\ (\mathsf{lif}\ al)\ (\mathsf{lif}\ al') \\
\mathsf{ldo\_dataref} & \vdash & \mathsf{EVERY\ monotonic}\ (\mathsf{MAP\ SND}\ al)\ \wedge\ (\mathsf{LENGTH}\ al = \mathsf{LENGTH}\ al') \wedge \\
 & & \mathsf{EVERY2}\ (\lambda(g, c)(g', c').\ \mathsf{dataref}\ r\ c\ ((\mathsf{guard}\ g')\ \mathsf{seq}\ c'))\ al\ al' \wedge \\
 & & \mathsf{EVERY2}\ (\lambda(g, c)(g', c').\ (\forall a\ k\ u.\ r(a, k, u)\ \wedge\ g'(k, u)\ \Rightarrow\ g(a, u)))\ al\ al' \wedge \\
 & & (\forall a\ k\ u.\ r(a, k, u)\ \wedge\ (\mathsf{lguard}\ al\ (a, u))\ \Rightarrow\ (\mathsf{lguard}\ al'\ (k, u)) \\
 & & \quad \Rightarrow\ \mathsf{dataref}\ r\ (\mathsf{ldo}\ al)\ (\mathsf{ldo}\ al')
\end{array}
$$

As a special case of lif_dataref we immediately get the rule for binary nondeterministic conditional.

$$\begin{array}{lll}
\textsf{if2_dataref} & \vdash & \textsf{monotonic}\ c1' \ \wedge\ \textsf{monotonic}\ c2' \ \wedge \\
& & \textsf{dataref}\ r\ c1\ ((\textsf{guard}\ g1')\ \textsf{seq}\ c1') \ \wedge \\
& & \textsf{dataref}\ r\ c2\ ((\textsf{guard}\ g2')\ \textsf{seq}\ c2') \ \wedge \\
& & (\forall a\ k\ u.\ r(a,k,u)\ \wedge\ g1'(k,u)\ \Rightarrow\ g1(a,u)) \ \wedge \\
& & (\forall a\ k\ u.\ r(a,k,u)\ \wedge\ g2'(k,u)\ \Rightarrow\ g2(a,u)) \ \wedge \\
& & (\forall a\ k\ u.\ r(a,k,u)\ \wedge\ (g1(a,u)\ \vee\ g2(a,u))\ \Rightarrow\ (g1'(k,u)\ \vee\ g2'(k,u)) \\
& & \ \Rightarrow\ \textsf{dataref}\ r\ (\textsf{if2}\ (g1,c1)\ (g2,c2))\ (\textsf{if2}\ (g1',c1')\ (g2',c2'))
\end{array}$$

Finally, we have the data refinement rule for action systems.

$$\begin{array}{lll}
\textsf{actsys_dataref} & \vdash & \textsf{EVERY monotonic}\ (\textsf{MAP SND}\ al)\ \wedge\ (\textsf{LENGTH}\ al = \textsf{LENGTH}\ al') \ \wedge \\
& & (\forall k\ u.\ p'(k,u)\ \Rightarrow\ \exists a.\ r(a,k,u)\ \wedge\ p(a,u)) \ \wedge \\
& & \textsf{EVERY2}\ (\lambda(g,c)(g',c').\ \textsf{dataref}\ r\ c\ ((\textsf{guard}\ g')\ \textsf{seq}\ c'))\ al\ al' \ \wedge \\
& & \textsf{EVERY2}\ (\lambda(g,c)(g',c').\ (\forall a\ k\ u.\ r(a,k,u)\ \wedge\ g'(k,u)\ \Rightarrow\ g(a,u)))\ al\ al' \ \wedge \\
& & (\forall a\ k\ u.\ r(a,k,u)\ \wedge\ (\textsf{lguard}\ al\ (a,u))\ \Rightarrow\ (\textsf{lguard}\ al'\ (k,u)) \\
& & \ \Rightarrow\ (\textsf{block}\ p\ (\textsf{ldo}\ al))\ \textsf{ref}\ (\textsf{block}\ p'\ (\textsf{ldo}\ al'))
\end{array}$$

10.3 Example

There are examples of data refinement of action systems in the files RCactsys_ex1.ml, RCactsys_ex2.ml and RCactsys_ex3.ml. One example using a well-founded ordering is described in the Appendix.

11 Refinement using window inference: the RCwintool theory

This section describes the RCwintool theory which makes it possible to use the HOL Window Inference Library [10] to perform refinements. A full description of the Window Inference Library can be found in the documentation of the HOL system.

A more detailed description of the Window refinement tool that is part of the RefCalc contribution can be found in [13]. In this document we concentrate on the contents of the theory and the examples.

11.1 Definitions

Because of an idiosyncrasy of the Window Inference Library, we have to use the refinement relation reversed:

$$\textsf{refines\_DEF}\ \ \vdash\ \ c'\ \textsf{refines}\ c\ =\ c\ \textsf{ref}\ c'$$

We also introduce the reverse of implies and a version of implies for relations.

$$\begin{array}{lll}
\textsf{implied_by_DEF} & \vdash & p'\ \textsf{implied_by}\ p\ =\ p\ \textsf{implies}\ p' \\
\textsf{implies2_DEF} & \vdash & P\ \textsf{implies2}\ P'\ =\ \forall u\ v.\ P\ u\ v\ \Rightarrow\ Q\ u\ v
\end{array}$$

11.2 Theorems

The window inference tool needs reflexivity and transitivity theorems for each relation it handles.

refines_refl	$\vdash$	$\forall c.\ c$ refines c
refines_trans	$\vdash$	$\forall c\ c'\ c''.\ c'$ refines $c\ \wedge\ c''$ refines $c'\ \Rightarrow\ c''$ refines c
implies_refl	$\vdash$	$\forall p.\ p$ implies p
implies_trans	$\vdash$	$\forall p\ q\ r.\ q$ implies $p\ \wedge\ r$ implies $q\ \Rightarrow\ r$ implies p
implied_by_refl	$\vdash$	$\forall p.\ p$ implied_by p
implied_by_trans	$\vdash$	$\forall p\ q\ r.\ q$ implied_by $p\ \wedge\ r$ implied_by $q\ \Rightarrow\ r$ implied_by p
implies2_refl	$\vdash$	$\forall p.\ p$ implies2 p
implies2_trans	$\vdash$	$\forall P\ Q\ R.\ Q$ implies2 $P\ \wedge\ R$ implies2 $Q\ \Rightarrow\ R$ implies2 P

Refinement rules We need to recast refinement rules into the new form, with refines rather than ref. We have not proved a great number of laws. Below are a few rules that are used in the examples. A user will probably need to prove many more laws.

First, we have a law for transforming a specification (using nondass) into an initialised loop.

$$
\begin{array}{lll}
\mathsf{spec\_to\_loop} & \vdash & \mathsf{monotonic}\ c \\
& & \Rightarrow\ (\lambda(x,u).\ p0\ u\ \wedge\ q(x,u))\ \mathsf{implies}\ inv \\
& & \Rightarrow\ ((\mathsf{not}\ g)\ \mathsf{andd}\ inv)\ \mathsf{implies}(\lambda(x,u).\ p\ u) \\
& & \Rightarrow\ (\forall x.\ \mathsf{correct}\ (inv\ \mathsf{andd}\ g\ \mathsf{andd}\ (\lambda u.\ t\ u = x))) \\
& & \qquad\qquad c \\
& & \qquad\qquad (inv\ \mathsf{andd}\ (\lambda u.\ t\ u < x))) \\
& & \Rightarrow\ ((\mathsf{nondass}\ \lambda u\ v.\ p0\ v)\ \mathsf{seq}\ (\mathsf{block}\ q\ (\mathsf{do}\ g\ c))) \\
& & \quad\ \mathsf{refines}\ (\mathsf{nondass}\ \lambda u\ v.\ p\ v)
\end{array}
$$

Next, we have a rule for making a nondeterministic assignment deterministic.

$$\mathsf{nondass\_to\_assign}\ \vdash\ (\forall u.\ P\ u\ (e\ u))\ \Rightarrow\ (\mathsf{assign}\ e)\ \mathsf{refines}\ (\mathsf{nondass}\ P)$$

Rules of data refinement For the window refinement tool, we use a calculational form of data refinement. This means that a block with local variables is refined using the following theorem:

$$
\begin{array}{lll}
\mathsf{dr\_block} & \vdash & \mathsf{monotonic}\ c \\
& & \Rightarrow\ (\mathsf{block}\ (\mathsf{abst}\ r\ p)\ ((\mathsf{abst}\ r)\ \mathsf{seq}\ c\ \mathsf{seq}\ (\mathsf{repr}\ r)))\ \mathsf{refines}\ (\mathsf{block}\ p\ c)
\end{array}
$$

Here, abst r p is the *calculated data refinement* of predicate p while

$$(\mathsf{repr}\ r)\ \mathsf{seq}\ c\ \mathsf{seq}\ (\mathsf{repr}\ r)$$

is the calculated data refinement of command c.

Data refinements are then pushed into program structures using the rules for command constructors:

$$
\begin{array}{lll}
\mathsf{dr\_seq} & \vdash & \mathsf{monotonic}\ c1 \\
& & \Rightarrow\ (((\mathsf{abst}\ r)\ \mathsf{seq}\ c1\ \mathsf{seq}(\mathsf{repr}\ r))\ \mathsf{seq}\ ((\mathsf{abst}\ r)\ \mathsf{seq}\ c2\ \mathsf{seq}(\mathsf{repr}\ r))) \\
& & \quad \mathsf{refines}\ ((\mathsf{abst}\ r)\ \mathsf{seq}\ (c1\ \mathsf{seq}\ c2)\ \mathsf{seq}(\mathsf{repr}\ r)) \\
\mathsf{dr\_cond} & \vdash & (\forall a\ a'\ k\ u.\ r(a,k,u)\ \wedge\ r(a',k',u')\ \wedge\ g(a',u)\ \Rightarrow\ g(a,u)) \\
& & \Rightarrow\ (\mathsf{cond}\ (\mathsf{abst}\ r\ g)\ ((\mathsf{abst}\ r)\ \mathsf{seq}\ c1\ \mathsf{seq}(\mathsf{repr}\ r)) \\
& & \qquad\qquad ((\mathsf{abst}\ r)\ \mathsf{seq}\ c2\ \mathsf{seq}(\mathsf{repr}\ r))) \\
& & \quad \mathsf{refines}\ ((\mathsf{abst}\ r)\ \mathsf{seq}\ (\mathsf{cond}\ g\ c1\ c2)\ \mathsf{seq}(\mathsf{repr}\ r)) \\
\mathsf{dr\_do} & \vdash & \mathsf{monotonic}\ c\ \wedge \\
& & (\forall a\ a'\ k\ u.\ r(a,k,u)\ \wedge\ r(a',k',u')\ \wedge\ g(a',u)\ \Rightarrow\ g(a,u)) \\
& & \Rightarrow\ (\mathsf{do}\ (\mathsf{abst}\ r\ g)\ ((\mathsf{abst}\ r)\ \mathsf{seq}\ c\ \mathsf{seq}(\mathsf{repr}\ r)) \\
& & \quad \mathsf{refines}\ ((\mathsf{abst}\ r)\ \mathsf{seq}\ (\mathsf{do}\ g\ c)\ \mathsf{seq}(\mathsf{repr}\ r))
\end{array}
$$

Finally, basic commands are data refined explicitly. For specification commands, we have the following calculational rule:

$$
\begin{array}{lll}
\mathsf{dr\_spec} & \vdash & ((\mathsf{assert}\ (\mathsf{abst}\ r\ p))\ \mathsf{seq} \\
& & \quad (\mathsf{nondass}\ (\lambda(k,u)(k',u'). \\
& & \qquad \forall a.\ \exists a'.\ r(a,k,u)\ \wedge\ p(a,u)\ \wedge\ r(a',k',u')\ \wedge\ P(a,u)(a',u')))) \\
& & \quad \mathsf{refines}\ ((\mathsf{assert}\ p)\ \mathsf{seq}\ (\mathsf{nondass}\ P))
\end{array}
$$

For pure nondeterministic assignments, this rule reduces to

$$
\begin{array}{lll}
\mathsf{dr\_nondass} & \vdash & (\mathsf{nondass}\ (\lambda(k,u)(k',u'). \\
& & \qquad \forall a.\ \exists a'.\ r(a,k,u)\ \wedge\ p(a,u)\ \wedge\ r(a',k',u')\ \wedge\ P(a,u)(a',u'))) \\
& & \quad \mathsf{refines}\ (\mathsf{nondass}\ P)
\end{array}
$$

Since the expressions in the refining commands in both the above theorems are quite complicated, the corresponding proof rule might be more useful:

$$
\begin{array}{lll}
\mathsf{dr\_spec\_rule} & \vdash & (\forall a\ k\ u.\ r(a,k,u)\ \wedge\ p(a,u)\ \Rightarrow\ q(k,u))\ \wedge \\
& & (\forall a\ k\ k'\ u\ u'.\ r(a,k,u)\ \wedge\ p(a,u)\ \wedge\ q(k,u)\ \wedge\ Q(k,u)(k',u') \\
& & \qquad \Rightarrow\ \exists a'.\ r(a',k',u')\ \wedge\ P(a,u)(a',u')) \\
& & \Rightarrow\ ((\mathsf{assert}\ q)\ \mathsf{seq}\ (\mathsf{nondass}\ Q)) \\
& & \quad \mathsf{refines}\ ((\mathsf{assert}\ p)\ \mathsf{seq}\ (\mathsf{nondass}\ P)) \\
\mathsf{dr\_nondass\_rule} & \vdash & (\forall a\ k\ k'\ u\ u'.\ r(a,k,u)\ \wedge\ Q(k,u)(k',u') \\
& & \qquad \Rightarrow\ \exists a'.\ r(a',k',u')\ \wedge\ P(a,u)(a',u')) \\
& & \Rightarrow\ (\mathsf{nondass}\ Q)\ \mathsf{refines}\ (\mathsf{nondass}\ P)
\end{array}
$$

11.3 ML functions and tactics

Window transformation commands The Window Inference System has a number of built-in transformation commands. For details, we refer to the documentation. When the RCwintool theory is loaded, a number of additional commands are defined. They are described below.

C_MATCH_TRANSFORM_WIN th Transforms the current focus by matching it against the consequent of the theorem th. Additional antecedents in th are added as unproved ("bad") conjectures.

SPEC_TO_LOOP_WIN $p\ q\ g\ c\ inv\ t$ Transforms a specification of the form nondass $\lambda u\ v.\ p\ v$ into the action system command block q (do $g\ c$), using invariant inv and termination function t. Conditions that cannot be proved automatically are added as unproved conjectures.

BLOCK_DR_WIN r Transforms a block by data refining it using abstraction relation r. This transformation transforms the initialisations and adds abstraction and representation commands in the body (see the theorem dr_block).

BLOCK_DR_WIN r Pushes abstraction and representation commands into a do command.

COND_DR_WIN() Pushes abstraction and representation commands into a cond command.

NONDASS_DR_WIN Q Transforms (abst r) seq (nondass P) seq (repr r) into nondass Q, adding proof obligations (see dr_nondass_rule) as unproved conjectures.

Window closing rules The Window Inference System permits us to open a subwindow (focus) on a subcomponent of the command we are interest. We can then transform (refine) this subcomponent, and finally close the subwindow. At this point the system uses a *window closing rule* to justify that the subcomponent replacement operation is valid. Each rule is a piece of ML code, which given a refinement theorem for the subcomponent produces a refinement theorem for the original command. Below, we show these rules, written as inference rules. First come the two rules for sequential composition.

$$\text{SEQ\_LEFT\_CLOSE} \qquad \frac{\vdash\ c1 \text{ refines } c2}{\vdash\ (c1 \text{ seq } c3) \text{ refines } (c2 \text{ seq } c3)}$$

$$\text{SEQ\_RIGHT\_CLOSE} \qquad \frac{\vdash\ \text{monotonic } c1 \qquad c2 \text{ refines } c3}{\vdash\ (c1 \text{ seq } c2) \text{ refines } (c1 \text{ seq } c3)}$$

Next some the rules for conditional composition:

$$\text{COND\_LEFT\_CLOSE} \qquad \frac{\vdash\ c1 \text{ refines } c2}{\vdash\ (\text{cond } g\ c1\ c3) \text{ refines } (\text{cond } g\ c2\ c3)}$$

$$\text{COND\_RIGHT\_CLOSE} \qquad \frac{\vdash\ c2 \text{ refines } c3}{\vdash\ (\text{cond } g\ c1\ c2) \text{ refines } (\text{cond } g\ c1\ c3)}$$

Finally, we have the rules for do and block.

$$\text{DO\_CLOSE} \qquad \frac{\vdash\ c' \text{ refines } c}{\vdash\ (\text{do } g\ c') \text{ refines } (\text{do } g\ c)}$$

$$\text{BLOCK\_CLOSE} \qquad \frac{\vdash\ c' \text{ refines } c}{\vdash\ (\text{block } p\ c') \text{ refines } (\text{block } p\ c)}$$

We also have theorems that are used by rules for opening subwindows on predicates rather than commands:

$$\textsf{ASSERTP\_CLOSE} \qquad \frac{\vdash\ p' \ \textsf{implied\_by}\ p}{\vdash\ (\textsf{assert}\ p')\ \textsf{refines}\ (\textsf{assert}\ p)}$$

$$\textsf{NONDASSP\_CLOSE} \qquad \frac{\vdash\ P' \ \textsf{implies2}\ P}{\vdash\ (\textsf{nondass}\ P')\ \textsf{refines}\ (\textsf{nondass}\ P)}$$

$$\textsf{BLOCKP\_CLOSE} \qquad \frac{\vdash\ p' \ \textsf{implies}\ p}{\vdash\ (\textsf{block}\ p'\ c)\ \textsf{refines}\ (\textsf{block}\ p\ c)}$$

The user never has to call these rules explicitly; the Window Inference System uses the appropriate rule when a window is closed, and it chains rules together automatically. The user should also be aware that it is always permitted to preserve equality in a subwindow and that there are built-in rules for boolean implication ($\Rightarrow$). For details, we refer to the manual for the Window Inference Library.

Controlling the window stack Since many of the transformation and closing rules have side conditions (extra assumptions), their application generate proof obligation that are treated as "bad conjectures" (unproved assumptions) by the Window Inference System. Since the number of such obligations may grow quickly, we have introduced a flag conj_print that determines whether they are printed or not. Evaluating

```
#conj_print false;;
```

turns off the printing of unproved proof obligations.

11.4 Example

We do a small data refinement using window inference. This example can be found in the file RCwintool_ex3.ml in the Wintool directory of the RefCalc contribution. For a more detailed description of this example, we refer to [13].

The example starts off with a specification, given as a nondeterministic assignment, guaranteeing that the elements in the list l are distinct natural numbers. the idea behind this example is to produce N distinct keys, to be consumed later on.

```
#let SPECIF = "nondass \l:(num)list. \l':(num)list.
     (LENGTH l' = N) /\ (!j k. j<k /\ k<N ==> ~(EL j l' = EL k l'))";;
```

We now set up the initial goal:

```
#BEGIN_STACK `keys` "(refines) (^SPECIF)" [] [];;
refines * nondass
          (\l l'.
            (LENGTH l' = N) /\
            (!j k. j < k /\ k < N ==> ~(EL j l' = EL k l')))
```

We then apply the rule SPEC_TO_LOOP_WIN with appropriate arguments. The arguments are global initialisation, local initialisation,guard, body, invariant and termination function (for readability, we have omitted the type information). The new program keeps all the keys in a set a and each time nondeterministically choses a new key not in a.

```
#SPEC_TO_LOOP_WIN "\l'. l' = []""\(a,l). a = (\n. F)""\(a,l). (LENGTH l) < N"
#  "nondass (\(a,l) (a',l'). ?m.
#       (l' = CONS m l) /\ ~a m /\ (!n. a' n = a n \/ (n = m)))"
# "\(a,l). (LENGTH l) <= N /\
#           (!j k. j < k /\ k < (LENGTH l) ==> ~(EL j l = EL k l)) /\
#           (!n. a n = (?k. k < (LENGTH l) /\ (n = EL k l)))"
# "\(a,l). N - (LENGTH l)";;
refines * (nondass(\u u'. u' = [])) seq
          (block (\(a,l). a = (\n. F))
                 (do (\(a,l). (LENGTH l) < N)
                     (nondass
                       (\(a,l) (a',l'). ?m.
                          (l' = CONS m l) /\ ~a m /\
                          (!n. a' n = a n \/ (n = m))))))
```

We have set the conj_print flag to false, so the Window Inference System does not print the unproved proof obligations. By looking at the current window theorem (WIN_THM) we can see that there are three of them, waiting to be proved. We start proving the first one by making it the focus of a new window and starting the goal stack for backward proof.

```
#ESTABLISH (hd(hyp(WIN_THM())));;
...
#g (focus(TOP_WIN()));;
```

Having finished the proof (it is simple), we used the theorem to discharge the proof obligation and then close.

```
#REWRITE\_WIN[top_thm()];;
...
#CLOSE\_WIN();;
```

After this, we prove the two other proof obligations in the same way.

Now we move to the data refinement, replacing the local set-valued variable a by a variable k which stands for some number greater than all numbers in a. The abstraction relation is defined as follows:

```
#let absrel = "\(a,k,l:(num)list). (!n. a n ==> n < k)";;
```

The data refinement is performed by focusing on the block and applying the rule for data refinement of blocks.

```
#OPEN_WIN[RAND];;
...
#BLOCK_DR_WIN absrel;;
refines * block
          (abst(\(a,k,l). !n. a n ==> n < k)(\(a,l). a = (\n. F)))
          ((abst(\(a,k,l). !n. a n ==> n < k)) seq
           ((do (\(a,l). (LENGTH l) < N)
                (nondass
                  (\(a,l) (a',l'). ?m.
                    (l' = CONS m l) /\ ~a m /\
                    (!n. a' n = a n \/ (n = m))))) seq
            (repr(\(a,k,l). !n. a n ==> n < k))))
```

After this, we can focus on the initialisation and transform it using REWRITE_WIN and CONVERT_WIN. For details, see the file containing the full text of the example. We move to the data refinement of the iteration: we focus on it and apply the appropriate rule.

```
#OPEN_WIN[RAND];;
...
#DO_DR_WIN();;
refines * do
          (abst(\(a,k,l). !n. a n ==> n < k)(\(a,l). (LENGTH l) < N))
          ((abst(\(a,k,l). !n. a n ==> n < k)) seq
           ((nondass
             (\(a,l) (a',l'). ?m.
                (l' = CONS m l) /\ ~a m /\
                (!n. a' n = a n \/ (n = m)))) seq
            (repr(\(a,k,l). !n. a n ==> n < k))))
```

The abstraction and representation commands have now been pushed into the guard and into the body. Again, we first focus on the guard and simplify it. Then we focus on the body and apply the rule for data refinement of a nondeterministic assignment, supplying the state relation for the refining nondass command.

```
#OPEN_WIN[RAND];;
...
#NONDASS_DR_WIN "\(k,l)(k',l'). (k < k') /\ (l' = CONS k l)";;
refines * nondass(\(k,l) (k',l'). k < k' /\ (l' = CONS k l))
```

Now we can close all open windows by repeatedly evaluating CLOSE_WIN(). There are some more proof obligations left to prove, but once this is done, we have the final result:

```
#WIN_THM();;
|- ((nondass(\u u'. u' = [])) seq
    (block true
          (do (\(c,u). (LENGTH u) < N)
```

```
            (nondass(\(k,l) (k',l'). k < k' /\ (l' = CONS k l))))))
refines
 (nondass (\l l'. (LENGTH l' = N) /\
     (!j k. j < k /\ k < N ==> ~(EL j l' = EL k l'))))
```

This concludes the example.

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*, 25:593–624, 1988.

[3] R.J.R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*, January 1989.

[4] R.J.R. Back and J. von Wright. Refinement calculus, part I: Sequential programs. In *REX Workshop for Refinement of Distributed Systems,* volume 430 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, 1989. Springer–Verlag.

[5] R.J.R. Back and J. von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing*, 2:247–272, 1990.

[6] R.J.R. Back and J. von Wright. Predicate transformers and higher order logic. In *REX Workshop for Refinement of Distributed Systems*, The Netherlands, 1992.

[7] E.W. Dijkstra. *A Discipline of Programming.* Prentice–Hall International, 1976.

[8] The HOL System Documentation. Cambridge, 1989.

[9] M. Gordon and T. Melham. *Introduction to HOL.* Cambridge University Press, New York, 1993.

[10] J. Grundy. A window inference tool for refinement. In Jones et al, editor, *Proc. 5th Refinement Workshop*, London, Jan. 1992. Springer–Verlag.

[11] J. von Wright, J. Hekanaho, P. Luostarinen and T. Långbacka. Mechanising some advanced refinement concepts. *Formal Methods in System Design*, 3:49–81, 1993.

[12] J. von Wright. The lattice of data refinement. *Acta Informatica.* to appear.

[13] J. von Wright. Program refinement by theorem prover. In *Proc. 6th Refinement Workshop*, London, January 1994. Springer–Verlag.

Appendix: the RCwellf theory

The RCwellf theory is a theory of well-founded orders. It defines the notion of a well-founded order on a type and proves the correctness of the principle of well-founded induction.

Definitions

We define the notions of order, minimal element and well-foundedness.

$$
\begin{array}{rll}
\textsf{order\_DEF} & \vdash & \textsf{order}\ po\ =\ (\forall x\ y.\ po\ x\ y\ =\ \neg(x = y)) \wedge \\
& & \qquad (\forall x\ y\ z.\ po\ x\ y\ \wedge\ po\ y\ z\ \Rightarrow\ po\ x\ z) \\
\textsf{minimal\_DEF} & \vdash & \textsf{minimal}\ x\ M\ po\ =\ M\ x\ \wedge\ (\forall y.\ M\ y\ \Rightarrow\ \neg(po\ y\ x)) \\
\textsf{wellf\_DEF} & \vdash & \textsf{wellf}\ po\ = \\
& & \quad \textsf{order}\ po\ \wedge\ \forall M.\ (\exists x.\ M\ x)\ \Rightarrow\ (\exists x.\ \textsf{minimal}\ x\ M\ po)
\end{array}
$$

Here, minimal $x\ M\ po$ means that the element x is minimal in the set M with respect to the order po.

Theorems

The standard example of a well-founded ordering is the less-than ordering on the natural numbers. We have proved this fact:

$$\textsf{num\_wellf}\ \ \vdash\ \ \textsf{wellf}\ \$<$$

The principle of well-founded induction shows how we can prove that all elements of a well-founded type have some property P.

$$
\begin{array}{rll}
\textsf{wellf\_INDUCT} & \vdash & \textsf{wellf}\ po\ \Rightarrow \\
& & \quad (\forall x.\ (\forall y.\ po\ y\ x\ \Rightarrow\ P\ y)\ \Rightarrow\ P\ x)\ \Rightarrow\ (\forall x.\ P\ x)
\end{array}
$$

The RCwellf theory is in fact an ancestor of the RCcorrect theory. In the RCcorrect theory we have proved the following generalisation of the total correctness rule for iteration:

$$
\begin{array}{rll}
\textsf{correct\_do\_wellf} & \vdash & \textsf{wellf}\ po\ \wedge\ \textsf{monotonic}\ c\ \wedge \\
& & p\ \textsf{implies}\ inv \wedge\ ((\textsf{not}\ g)\ \textsf{andd}\ inv)\ \textsf{implies}\ q\ \wedge \\
& & (\forall x.\ \textsf{correct}\ (inv\ \textsf{andd}\ (g\ \textsf{andd}\ (\lambda u.\ t\ u = x))) \\
& & \qquad\qquad c \\
& & \qquad\qquad (inv\ \textsf{andd}\ (\lambda u.\ po\ (t\ u)\ x))) \\
& & \quad \Rightarrow \textsf{correct}\ p\ (\textsf{do}\ g\ c)\ q
\end{array}
$$

which replaces the less-that ordering on the natural numbers by an arbitrary well-founded ordering po.

Example

The following example shows how termination of a loop with unbounded nondeterminism can be proved. This is the example found in the file RCactsys_ex3.ml in the Actsys directory. The proof uses a variant function that ranges over the natural numbers, ordered

$$1, 2, 3, \ldots, 0$$

i.e., with 0 considered to be greater than all other numbers.

Proving that this is a well-founded order is quite time-consuming, though not difficult:

$$\mathsf{wellf\_0\_grt} \quad \vdash \quad \mathsf{wellf}\ \lambda x\ y.\ 0 < x\ \wedge\ (x < y\ \vee\ (y = 0))$$

Our aim is to prove that the following program terminates:

$$\mathbf{var}\ x = 0;\ \mathbf{do}\ x = 0 \rightarrow x :=?\ \|\ x > 1 \rightarrow x := x - 1\ \mathbf{od}$$

The first iteration sets x to an arbitrary natural number (x :=? is a random assignment on the natural numbers). After this, x is decremented on each iteration. This example has been used to justify the notion of *weak termination* (termination not within any time bound that can be computed in advance). The termination proof uses the order defined above.

We assume that there is a global component $u : *$ (it does not matter what type this component has. Now the goal to be proved is

$$\begin{array}{lll} \mathsf{correct} & \mathsf{true} & \\ & (\mathsf{block} & (\lambda(x,u).\ x = 0) \\ & & (\mathsf{ldo}\ [((\lambda(x,u).\ x = 0), \mathsf{nondass}\ \lambda(x,u)(x',u').\ u' = u); \\ & & \quad ((\lambda(x,u).\ x > 1), \mathsf{assign}\ \lambda(x,u).\ (x-1,u))]) \\ & \mathsf{true} & \end{array}$$

We match against the generalised loop correctness rule correct_do_wellf, instantiating the well-founded order above and get a number of trivial subgoals, which are solved using simple arithmetic reasoning.