A Verification Environment for Sequential Imperative Programs in Isabelle/HOL [⋆]

Norbert Schirmer

Technische Universität München, Institut für Informatik
`http://www4.in.tum.de/~schirmer`

Abstract. We develop a general language model for sequential imperative programs together with a Hoare logic. We instantiate the framework with common programming language constructs and integrate it into Isabelle/HOL, to gain a usable and sound verification environment.

1 Introduction

The main goal of this work is to develop a suitable programming language model and proof calculus, to support program verification in the interactive theorem prover Isabelle/HOL. The model should be lightweight so that program verification can be carried out on the abstraction level of the programming language. The design of a framework for program verification in an expressive logic like HOL is driven by two main goals. On the one hand we want to derive the proof calculus in HOL, so that we can guarantee soundness of the calculus with respect to the programming language semantics. On the other hand we want to apply the proof calculus to verify programs. During program verification we focus on one single program for which we want to derive some properties. But for a general soundness proof of the calculus we have to regard the whole programming language, not just a single program. The tradeoff can be illustrated by a simple program that only concerns two local variables: `i` and `b` of type `int` and `bool`, respectively. It is desirable to obtain verification conditions in terms of logical variables i and b of type *int* and *bool* in HOL. But of course we do not want to fix the state space of the general proof calculus only to programs on these two variables. One solution is to model local variables as a function, mapping variable names to values: $name \Rightarrow value$. But in this model all variables are represented by the same logical type, namely *value*. If the programming language supports more than one type for variables we can define the type *value* as datatype, e.g.: $value = Int\ int \mid Bool\ bool \mid \ldots$. We tag values with their programming language type. The drawback of this approach is that we have to explicitely deal with programming language typing in assertions and proofs. If for example a program adds the constant `2` to the variable `i` we have to know that the current environment holds a value of the kind $Int\ i$ and not $Bool\ b$ to properly reason about the addition. This knowledge has to stem from a typing

⋆ This research is funded by the project Verisoft (`http://www.verisoft.de`)

constraint. We somehow always have to deal with type safety of the programming language during program verification.

The main contribution of this work is to present a programming language model that operates on a polymorphic state space, but still can handle local and global variables throughout procedure calls. By this we can achieve both desired goals. We can once and for all develop a sound proof calculus as well as later on tailor the state space to fit to the current program verification task. Moreover the model is expressive enough to handle abrupt termination, runtime faults and dynamic procedure calls. Finally we instantiate the framework with a state space representation that allows us to match programming language typing with logical typing. So type inference will take care of basic type safety issues, which simplifies the assertions and proof obligations. Parts of the frame condition for procedure specifications can be naturally expressed in this state space representation and can already be handled during verification condition generation.

We start with a brief introduction to Isabelle/HOL in Section 2; in Section 3 we introduce the syntax and semantics of the basic programming language model; Section 4 describes a Hoare logic for partial correctness; Section 5 a Hoare logic for total correctness; Section 6 will instantiate the framework with common language features and sketch the integration into Isabelle; Section 7 concludes.

Related Work The tradition of embedding a programming language in HOL goes back to the work of Gordon [11], where a while language with variables ranging over natural numbers is introduced. A polymorphic state space was already used by Harrison in his formalisation of Dijkstra [4] and by Prensa to verify parallel programs [18]. Still procedures are not present. Homeier [5] introduces procedures, but the variables are again limited to numbers. Later on detailed semantics for Java [16,6] and C [15] where embedded in a theorem prover. But verification of even simple programs suffers from the complex models.

The Why tool [3] implements a program logics for annotated functional programs (with references) and produces verification conditions for an external theorem prover. It can handle uninterpreted parts of annotations that are only meaningful to the external theorem prover. With this approach it is possible to map imperative languages like Java to the tool by representing the heap in a reference variable. Splitting up verification condition generation and their proofs to different tools is also followed in [10,17].

2 Preliminary Notes on Isabelle/HOL

Isabelle is a generic logical framework which allows one to encode different object logics. In this article we are only concerned with Isabelle/HOL [14], an encoding of higher order logic augmented with facilities for defining data types, records, inductive sets as well as primitive and total general recursive functions.

The syntax of Isabelle is reminiscent of ML, so we will not go into detail here. There are the usual type constructors $T_1 \times T_2$ for product and $T_1 \Rightarrow T_2$ for function space. The syntax $[\![P;\ Q]\!] \Longrightarrow\ R$ should be read as an inference rule with the two premises P and Q and the conclusion R. Logically it is just a shorthand for $P \Longrightarrow Q \Longrightarrow\ R$. There are actually two implications $\longrightarrow$ and $\Longrightarrow$. The two mean the same thing except that $\longrightarrow$ is HOL's "real" implication, whereas $\Longrightarrow$ comes from Isabelle's meta-logic and expresses inference rules. Thus $\Longrightarrow$ cannot appear inside a HOL formula. For the purpose of this paper the two may be identified. Similarly, we use $\bigwedge$ for the universal quantifier in the meta logic.

To emulate partial functions the polymorphic option type is frequently used:

datatype *'a option* = *None* | *Some 'a*

Here *'a* is a type variable, *None* stands for the undefined value and *Some x* for a defined value x. A partial function from type T_1 to type T_2 can be modelled as $T_1 \Rightarrow (T_2\ \mathit{option})$.

There is also a destructor for the constructor *Some*, the function *the*:: *'a option* $\Rightarrow$ *'a*. It is defined by the sole equation *the* (*Some x*) = x and is total in the sense that *the None* is a legal, but indefinite value.

Appending two lists is written as *xs* @ *ys* and "consing" as x # *xs*.

3 Programming Language Model

3.1 Abstract Syntax

The basic model of the programming language is quite general. We want to be able to represent a sequential imperative programming language with mutually recursive procedures, local and global variables and heap. Abrupt termination like `break`, `continue`, `return` or exceptions should also be expressible in the model. Moreover we support a dynamic procedure call, which allows us to represent procedure pointers or dynamic method invocation.

We only fix the statements of the programming language. Expressions are ordinary HOL-expressions, therefore they do not have any side effects. Nevertheless we want to be able to express faults during expression evaluation, like division by *0* or dereferencing a *Null* pointer. We introduce *guards* in the language, which check for those runtime faults.

The state space of the programming language and also the representation of procedure names is polymorphic. The canonical type variable for the state space is *'s* and for procedure names *'p*. The programming language is defined by a **datatype** (*'s*, *'p*) *com* with the following constructors:

Skip: Do nothing.
Basic f: Basic commands like assignment.
Seq c_1 c_2: Sequential composition, also written as $c_1;c_2$.
Cond b c_1 c_2: Conditional statement.

*Guard g c***:** Guarded command, also written as $g \mapsto c$.
*While g b c***:** Loop.
*Call init p return result***:** Static procedure call.
*DynCall init p return result***:** Dynamic procedure call.
*Throw***:** Initiate abrupt termination.
Catch c_1 c_2**:** Handle abrupt termination.

3.2 Semantic

State Space Representation Although the semantics is defined for polymorphic state spaces we introduce the state space representation which we will use later on to give some illustrative examples. We represent the state space as a **record** [14,12] in Isabelle/HOL. This idea goes back to Wenzel [19]. A simple state space with three local variables B, N and M can be modelled with the following record definition:

record *vars* = B::*bool* N::*int* M::*int*

Records of type *vars* have three fields, named B, N and M of type *bool* resp. *int*. An example instance of such a record is $(\!| B = \mathit{True},\ N = 42,\ M = 3 |\!)$. For each field there is a *selector* function of the same name, e.g. $N\ (\!| B = \mathit{True},\ N = 42,\ M = 3 |\!) = 42$. The *update* operation is functional. For example, $v(\!| N := 0 |\!)$ is a record where component N is 0 and whose B and M component are copied from v. Selections of updated components can be simplified automatically e.g. $N\ (r(\!| N := 43 |\!)) = 43$. The representation of the state space as record has the advantage that the typing of variables can be expressed by means of typing in the logic. Therefore basic type safety requirements are already ensured by type inference.

Operational Semantics We give an operational (big step) semantics for the programming language, written as $\Gamma \vdash s\ {-c\rightarrow}\ t$. Starting in state s, execution of command c leads to the final state t. Γ is the procedure environment, which maps procedure names to procedure bodies. The states s and t are not just plain state spaces of type $'s$, but extended states of type $'s$ *xstate* which allow us to identify runtime faults, stuck calculations and abrupt termination. During normal execution such an extended state has the form *Normal s*, during abrupt termination the form *Abrupt s*, runtime faults are captured by the (extended) state *Fault* and stuck calculation by the (extended) state *Stuck*. The execution relation is defined inductively.

Basic commands The command *Basic f* just applies the function f to the current state. An example of a basic operation may be an assignment `N = 2`. This can be represented as *Basic* $(\lambda s.\ s(\!| N{:=}2 |\!))$ in our language model. We can also model field assignment or memory allocation as basic commands.

$\Gamma \vdash \mathit{Normal}\ s\ {-\mathit{Skip}\rightarrow}\ \mathit{Normal}\ s$ $\qquad$ $\Gamma \vdash \mathit{Normal}\ s\ {-\mathit{Basic}\ f\rightarrow}\ \mathit{Normal}\ (f\ s)$

Composition Sequential composition combines the execution of the two commands.

$$\frac{\Gamma \vdash Normal\ s\ -c_1\rightarrow\ s' \qquad \Gamma \vdash s'\ -c_2\rightarrow\ t}{\Gamma \vdash Normal\ s\ -Seq\ c_1\ c_2\rightarrow\ t}$$

Conditional The conditional statement executes the first or the second command depending on the branching condition b. We represent boolean expressions as state sets.

$$\frac{s \in b \qquad \Gamma \vdash Normal\ s\ -c_1\rightarrow\ t}{\Gamma \vdash Normal\ s\ -Cond\ b\ c_1\ c_2\rightarrow\ t} \qquad \frac{s \notin b \qquad \Gamma \vdash Normal\ s\ -c_2\rightarrow\ t}{\Gamma \vdash Normal\ s\ -Cond\ b\ c_1\ c_2\rightarrow\ t}$$

Guards The guarded command is used to model runtime faults which may occur during expression evaluation. The guard g is a boolean expression that checks for possible faults in the expressions of command c and only executes c if the test is passed. Otherwise the fault will be signalled. Once a fault has occurred we cannot leave the error state *Fault* anymore.

$$\frac{\begin{array}{c} s \in g \\ \Gamma \vdash Normal\ s\ -c\rightarrow\ t \end{array}}{\Gamma \vdash Normal\ s\ -Guard\ g\ c\rightarrow\ t} \qquad \frac{s \notin g}{\Gamma \vdash Normal\ s\ -Guard\ g\ c\rightarrow\ Fault}$$

$$\Gamma \vdash Fault\ -c\rightarrow\ Fault$$

Loop If the guard g for the condition b fails the while loop will end up in the state *Fault*. If the guard and the loop condition hold, first the loop body c is executed, followed by the recursive execution of the while loop. If the guard holds, but the loop condition does not, we exit the loop.

$$\frac{s \notin g}{\Gamma \vdash Normal\ s\ -While\ g\ b\ c\rightarrow\ Fault}$$

$$\frac{\begin{array}{c} s \in g \qquad s \in b \\ \Gamma \vdash Normal\ s\ -c\rightarrow\ s' \\ \Gamma \vdash s'\ -While\ g\ b\ c\rightarrow\ t \end{array}}{\Gamma \vdash Normal\ s\ -While\ g\ b\ c\rightarrow\ t} \qquad \frac{s \in g \qquad s \notin b}{\Gamma \vdash Normal\ s\ -While\ g\ b\ c\rightarrow\ Normal\ s}$$

Abrupt termination The *Throw* statement transforms a *Normal* state to an *Abrupt* state. For *Abrupt* states execution is skipped. A *Catch* c_1 c_2 statement will handle an *Abrupt* final state of c_1 by continuing execution of c_2 in a *Normal* state. Otherwise execution of c_2 is skipped.

$$\Gamma \vdash Normal\ s\ -Throw\rightarrow\ Abrupt\ s \qquad \Gamma \vdash Abrupt\ s\ -c\rightarrow\ Abrupt\ s$$

$$\frac{\begin{array}{c} \Gamma \vdash Normal\ s\ -c_1\rightarrow\ Abrupt\ s' \\ \Gamma \vdash Normal\ s'\ -c_2\rightarrow\ t \end{array}}{\Gamma \vdash Normal\ s\ -Catch\ c_1\ c_2\rightarrow\ t} \qquad \frac{\Gamma \vdash Normal\ s\ -c_1\rightarrow\ t \qquad \neg\ isAbr\ t}{\Gamma \vdash Normal\ s\ -Catch\ c_1\ c_2\rightarrow\ t}$$

$$\begin{aligned}&\mathit{isAbr}\ (\mathit{Normal}\ s) = \mathit{False}\\&\mathit{isAbr}\ (\mathit{Abrupt}\ s) = \mathit{True}\\&\mathit{isAbr}\ \mathit{Fault} = \mathit{False}\\&\mathit{isAbr}\ \mathit{Stuck} = \mathit{False}\end{aligned}$$

Procedure call To execute a procedure call *Call init p return result* we first pass the parameters by applying *init* to the starting state *s*. Then we execute the procedure body that is given by a lookup in the procedure environment $\Gamma\ p$. If this lookup fails $\Gamma\ p = \mathit{None}$ execution gets *Stuck*. For *Stuck* states execution is skipped. If we find a procedure body in the environment the further execution depends on the kind of state, resulting from the body:

$$\frac{\Gamma\ p = \mathit{None}}{\Gamma\vdash \mathit{Normal}\ s\ -\mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result}\rightarrow\ \mathit{Stuck}}$$

$$\Gamma\vdash \mathit{Stuck}\ -c\rightarrow\ \mathit{Stuck}$$

$$\frac{\Gamma\ p = \mathit{Some}\ \mathit{bdy}\qquad \Gamma\vdash \mathit{Normal}\ (\mathit{init}\ s)\ -\mathit{bdy}\rightarrow\ \mathit{Fault}}{\Gamma\vdash \mathit{Normal}\ s\ -\mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result}\rightarrow\ \mathit{Fault}}$$

$$\frac{\Gamma\ p = \mathit{Some}\ \mathit{bdy}\qquad \Gamma\vdash \mathit{Normal}\ (\mathit{init}\ s)\ -\mathit{bdy}\rightarrow\ \mathit{Stuck}}{\Gamma\vdash \mathit{Normal}\ s\ -\mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result}\rightarrow\ \mathit{Stuck}}$$

$$\frac{\Gamma\ p = \mathit{Some}\ \mathit{bdy}\qquad \Gamma\vdash \mathit{Normal}\ (\mathit{init}\ s)\ -\mathit{bdy}\rightarrow\ \mathit{Normal}\ t}{\Gamma\vdash \mathit{Normal}\ s\ -\mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result}\rightarrow\ \mathit{Normal}\ (\mathit{result}\ s\ t)}$$

$$\frac{\Gamma\ p = \mathit{Some}\ \mathit{bdy}\qquad \Gamma\vdash \mathit{Normal}\ (\mathit{init}\ s)\ -\mathit{bdy}\rightarrow\ \mathit{Abrupt}\ t}{\Gamma\vdash \mathit{Normal}\ s\ -\mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result}\rightarrow\ \mathit{Abrupt}\ (\mathit{return}\ s\ t)}$$

If execution of the body fails or gets stuck, the whole call fails or gets stuck. If execution of the body ends up in a *Normal* state *t*, the outcome of the call is given by *result s t*. If execution of the body ends up in an *Abrupt* state *t*, the outcome of the call is given by *return s t*. The function *return* passes back the global variables (and heap components), and *result* additionally assigns results to local variables of the caller.

The *return*/*result* functions get both the initial state *s* before the procedure call and the final state *t* after execution of the body. It is the purpose of *return* to restore the local variables of the caller and update the global variables. The *result* function will additionally assign the result to the caller. If the body terminates abruptly we apply the *return* function, thus the global state will be propagated to the caller but no result will be assigned. This is the expected semantics of an exception. Note that we can also store a description of the raised exception in a global variable so that a *Catch* can peek at it, to decide whether to handle the exception or to re-raise it.

As an example for a procedure call, consider a function definition `int fac(int n)` and a call to this function `m = fac(m)`. When we do not regard global variables, we can model this call by: *Call* $(\lambda s.\ s(\!|N := M\ s|\!))$ *fac* $(\lambda s\ t.\ s)$ $(\lambda s\ t.\ s(\!|M := N\ t|\!))$. The state space of the programming language is flat. We do not explicitly model a stack. Locality of variables and parameters is maintained by the *return* and *result* functions. The body of *fac* expects the input to be stored in the formal parameter N. But we call the function with the actual parameter M. So the *init* function $\lambda s.\ s(\!|N := M\ s|\!)$ copies the content of M to component N. The trivial *return* function $\lambda s\ t.\ s$ gives back the initial state. State s is the initial state of the caller, and t is the state after executing the body. By this all local variables of the caller are restored. Consider that the body of *fac* internally holds the result of the factorial calculation in its local variable N. Then the *result* function has to copy the content of N to component M because of the assignment `m = fac(m)`. This is implemented by the *result* function $\lambda s\ t.\ s(\!|M := N\ t|\!)$. The *result* function just takes the initial state and performs the necessary update by peeking on the actual state. These ideas can be extended to global variables. Consider B to be a global variable. We just have to adapt the *return*/*result* function, so that they will copy B back to the caller: *return* $= (\lambda s\ t.\ s(\!|B := B\ t|\!))$, *result* $= (\lambda s\ t.\ s(\!|B := B\ t,\ M := N\ t|\!))$.

In contrast to the static procedure call, a dynamic procedure call first calculates the procedure from the actual state. The rest is handled by the ordinary procedure call rules.

$$\frac{\Gamma \vdash \mathit{Normal}\ s\ -\mathit{Call}\ \mathit{init}\ (p\ s)\ \mathit{return}\ \mathit{result}\rightarrow\ t}{\Gamma \vdash \mathit{Normal}\ s\ -\mathit{DynCall}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result}\rightarrow\ t}$$

Termination To characterise terminating programs we introduce the inductively defined judgement $\Gamma \vdash c \downarrow s$ expressing that in procedure environment Γ program c will terminate when it is started in state s. The rules should be self-explanatory:

Basic commands

$$\Gamma \vdash \mathit{Skip} \downarrow \mathit{Normal}\ s \qquad\qquad \Gamma \vdash \mathit{Basic}\ f \downarrow \mathit{Normal}\ s$$

Composition

$$\frac{\Gamma \vdash c_1 \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma \vdash \mathit{Normal}\ s\ -c_1\rightarrow\ s' \longrightarrow \Gamma \vdash c_2 \downarrow s'}{\Gamma \vdash \mathit{Seq}\ c_1\ c_2 \downarrow \mathit{Normal}\ s}$$

Conditional

$$\frac{s \in b \qquad \Gamma \vdash c_1 \downarrow \mathit{Normal}\ s}{\Gamma \vdash \mathit{Cond}\ b\ c_1\ c_2 \downarrow \mathit{Normal}\ s} \qquad\qquad \frac{s \notin b \qquad \Gamma \vdash c_2 \downarrow \mathit{Normal}\ s}{\Gamma \vdash \mathit{Cond}\ b\ c_1\ c_2 \downarrow \mathit{Normal}\ s}$$

Guards

$$\frac{s \in g \qquad \Gamma \vdash c \downarrow \mathit{Normal}\ s}{\Gamma \vdash \mathit{Guard}\ g\ c \downarrow \mathit{Normal}\ s} \qquad \frac{s \notin g}{\Gamma \vdash \mathit{Guard}\ g\ c \downarrow \mathit{Normal}\ s} \qquad \Gamma \vdash c \downarrow \mathit{Fault}$$

Loop

$$\frac{s \notin g}{\Gamma \vdash \mathit{While}\ g\ b\ c \downarrow \mathit{Normal}\ s} \qquad\qquad \frac{s \in g \qquad s \notin b}{\Gamma \vdash \mathit{While}\ g\ b\ c \downarrow \mathit{Normal}\ s}$$

$$\frac{\begin{array}{c} s \in g \qquad s \in b \\ \Gamma \vdash c \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma \vdash \mathit{Normal}\ s\ -c\rightarrow\ s' \longrightarrow \Gamma \vdash \mathit{While}\ g\ b\ c \downarrow s' \end{array}}{\Gamma \vdash \mathit{While}\ g\ b\ c \downarrow \mathit{Normal}\ s}$$

Abrupt termination

$\Gamma\vdash \mathit{Throw} \downarrow \mathit{Normal}\ s$ $\qquad\qquad$ $\Gamma\vdash c \downarrow \mathit{Abrupt}\ s$

$$\frac{\Gamma\vdash c_1 \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma\vdash \mathit{Normal}\ s\ {-c_1\rightarrow}\ \mathit{Abrupt}\ s' \longrightarrow \Gamma\vdash c_2 \downarrow \mathit{Normal}\ s'}{\Gamma\vdash \mathit{Catch}\ c_1\ c_2 \downarrow \mathit{Normal}\ s}$$

Procedure call

$$\frac{\Gamma\ p = \mathit{Some}\ \mathit{bdy} \qquad \Gamma\vdash \mathit{bdy} \downarrow \mathit{Normal}\ (\mathit{init}\ s)}{\Gamma\vdash \mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result} \downarrow \mathit{Normal}\ s}$$

$$\frac{\Gamma\ p = \mathit{None}}{\Gamma\vdash \mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result} \downarrow \mathit{Normal}\ s} \qquad \Gamma\vdash c \downarrow \mathit{Stuck}$$

$$\frac{\Gamma\vdash \mathit{Call}\ \mathit{init}\ (p\ s)\ \mathit{return}\ \mathit{result} \downarrow \mathit{Normal}\ s}{\Gamma\vdash \mathit{DynCall}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result} \downarrow \mathit{Normal}\ s}$$

4 Hoare Logic for Partial Correctness

The first question concerning a Hoare logic is how to represent the assertions. The model of the imperative programming language is quite general. The state space is polymorphic. So the variables and their types are not fixed until we regard a program to verify. Therefore the assertion language is not fixed either. An assertion on states of type $'s$ is a set of states: $'s\ \mathit{set}$.

We first define a Hoare logic for partial correctness. The judgement is of the general form $\Gamma,\Theta\vdash P\ c\ Q,A$ where P is the precondition, c the program, Q the postcondition for normal termination, A the postcondition for abrupt termination, Γ the procedure environment and Θ is a set of Hoare quadruples that we may assume. Θ is used to handle recursive procedures as we will see later on. The approach to split up the postcondition for normal and abrupt termination is also followed by [3,7].

The semantics of these judgements is given by the notion of validity:

$$\Gamma\models P\ c\ Q,A \equiv$$
$$\forall s\ t.\ \Gamma\vdash s\ {-c\rightarrow}\ t \longrightarrow s \in \mathit{Normal}\ `\ P \longrightarrow t \in \mathit{Normal}\ `\ Q \cup \mathit{Abrupt}\ `\ A$$

Given an execution of command c which takes us from the starting state s to the final state t, if s is a *Normal* state for which the precondition P holds, then the final state t will either be a *Normal* state for which the postcondition Q holds, or an *Abrupt* state for which postcondition A holds. The *Fault* and *Stuck* states are no valid outcomes. This extends the traditional partial correctness interpretation to abrupt termination and the additional constraint that no runtime fault may occur. The assertions P, Q and A are of type $'s\ \mathit{set}$, whereas s and t are of type $'s\ \mathit{xstate}$. We do not have to deal with the extended state in assertions, which makes them easier. The operator $`$ is the set image (like *map* for lists). So $s \in \mathit{Normal}\ `\ P$ can be rephrased by the set comprehension $\{\mathit{Normal}\ s.\ s \in P\}$.

When designing the Hoare logic we should always keep soundness and completeness in mind, which we have both proven:

– **theorem** *soundness*: $\Gamma,\{\}\vdash P\ c\ Q,A \longrightarrow \Gamma\models P\ c\ Q,A$
  We can only derive valid Hoare quadruples out of the empty context.
– **theorem** *completeness*: $\Gamma\models P\ c\ Q,A \longrightarrow \Gamma,\{\}\vdash P\ c\ Q,A$
  We can derive every valid Hoare quadruple out of the empty context.

The Hoare logic is defined inductively. The rules are syntax directed, and most of them are defined in a weakest precondition style. This makes it easy to automate rule application in a verification condition generator. Handling abrupt termination is surprisingly simple. The postcondition for abrupt termination is left unmodified by most of the rules. Only if we actually encounter a *Throw* it has to be a consequence of the precondition. This means that the proof rules do not complicate the verification of programs where abrupt termination is not present.

Basic Commands The rule for *Basic f* commands is a variation of the classical assignment rule. If the postcondition is Q, then the precondition is the set of all states that will lead into Q after applying *f*.

$$\Gamma,\Theta\vdash Q\ \mathit{Skip}\ Q,A \qquad\qquad \Gamma,\Theta\vdash\{s.\ f\ s \in Q\}\ \mathit{Basic}\ f\ Q,A$$

Composition and Conditional The rule for sequential composition and the conditional are almost standard. In case of sequential composition the postcondition for abrupt termination has to hold in either statement independently, in contrast to the intermediate assertion R for normal termination. This is simply because in case of abrupt termination of the first statement the second one will be skipped.

$$\frac{\Gamma,\Theta\vdash P\ c_1\ R,A \qquad \Gamma,\Theta\vdash R\ c_2\ Q,A}{\Gamma,\Theta\vdash P\ \mathit{Seq}\ c_1\ c_2\ Q,A} \qquad\qquad \frac{\begin{array}{c}\Gamma,\Theta\vdash(P \cap b)\ c_1\ Q,A\\ \Gamma,\Theta\vdash(P \cap -\ b)\ c_2\ Q,A\end{array}}{\Gamma,\Theta\vdash P\ \mathit{Cond}\ b\ c_1\ c_2\ Q,A}$$

Guards To prove a guarded command correct, we have to show that both the precondition P of the statement c and the guard g hold. This ensures that no runtime fault occurs.

$$\frac{\Gamma,\Theta\vdash P\ c\ Q,A}{\Gamma,\Theta\vdash(g \cap P)\ \mathit{Guard}\ g\ c\ Q,A}$$

Loop The rule for the while loop is also almost the traditional invariant rule. But we also have to ensure that the guard g for the conditional b always holds. Otherwise a runtime fault could occur. The verification condition generator will use a derived rule which takes an invariant annotation into account.

$$\frac{\Gamma,\Theta\vdash(g \cap P \cap b)\ c\ (g \cap P),A}{\Gamma,\Theta\vdash(g \cap P)\ \mathit{While}\ g\ b\ c\ (g \cap P \cap -\ b),A}$$

Abrupt Termination In case of a *Throw* the abrupt postcondition has to stem from the precondition. The rule for *Catch* is dual to sequential composition. Here the postcondition for normal termination can be derived independently. The intermediate assertion R is the precondition for the second statement and the postcondition for abrupt termination of the first statement.

$$\Gamma,\Theta\vdash A\ \mathit{Throw}\ Q,A \qquad\qquad \frac{\Gamma,\Theta\vdash P\ c_1\ Q,R \qquad \Gamma,\Theta\vdash R\ c_2\ Q,A}{\Gamma,\Theta\vdash P\ \mathit{Catch}\ c_1\ c_2\ Q,A}$$

Consequence We have a quite general form of the consequence rule. The traditional rules like precondition strengthening or postcondition weakening can easily be derived from it.

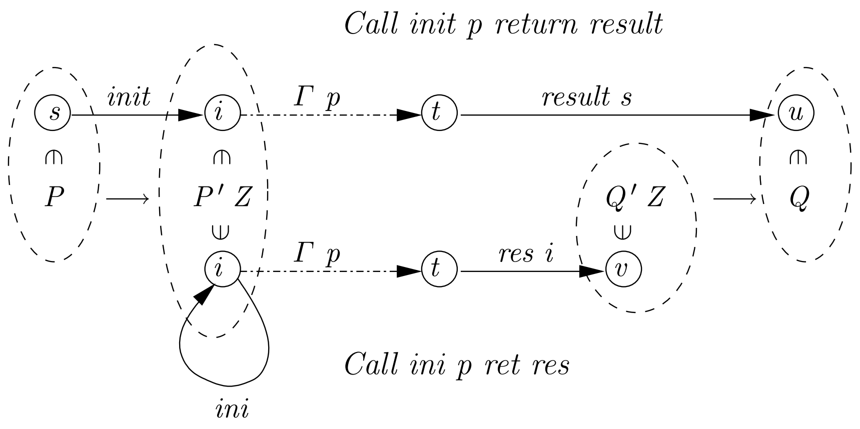
$$\frac{\begin{array}{c}\forall Z.\ \Gamma,\Theta\vdash(P'\ Z)\ c\ (Q'\ Z),(A'\ Z)\\ \forall s.\ s\in P\longrightarrow(\exists Z.\ s\in P'\ Z\wedge Q'\ Z\subseteq Q\wedge A'\ Z\subseteq A)\end{array}}{\Gamma,\Theta\vdash P\ c\ Q,A}$$

The consequence rule can be used to adapt a given specification $\Gamma,\Theta\vdash(P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$ about command c to $\Gamma,\Theta\vdash P\ c\ Q,A$. The auxiliary variable Z can be used to transport state information from the pre state to the post state. This is a crucial tool to deal with procedure specifications, where the postcondition is defined by means of the pre state. For completeness issues it is sufficient that Z has the type $'s$ of the state space. Z can be used to fix the pre-state logically. That is why the given specification must be valid for all Z and the side-condition allows us to select a specific Z dependent on the current state s. A more detailed discussion of consequence rules and auxiliary variables can be found in [8,13,16].

Procedure Call If we encounter a procedure call during verification condition generation, like $\Gamma,\Theta\vdash P\ \mathit{Call\ init\ p\ return\ result}\ Q,A$, we do not look inside the procedure body, but instead use a specification $\Gamma,\Theta\vdash(P'\ Z)\ \mathit{Call\ ini\ p\ ret\ res}\ (Q'\ Z),(A'\ Z)$ of the procedure. We then adapt the specification to the actual calling context mainly by a variant of the consequence rule, where we also take parameter and result passing into account.

$$\frac{\begin{array}{l}\qquad\qquad\forall Z.\ \Gamma,\Theta\vdash(P'\ Z)\ \mathit{Call\ ini\ p\ ret\ res}\ (Q'\ Z),(A'\ Z)\\ \qquad\qquad\qquad\qquad\forall s.\ \mathit{ini}\ (\mathit{init}\ s)=\mathit{init}\ s\\ P\subseteq\{s.\ (\exists Z.\ \mathit{init}\ s\in P'\ Z\ \wedge\\ \qquad\qquad(\forall t.\ \mathit{res}\ (\mathit{init}\ s)\ t\in Q'\ Z\longrightarrow \mathit{result}\ s\ t\in Q)\ \wedge\\ \qquad\qquad(\forall t.\ \mathit{ret}\ (\mathit{init}\ s)\ t\in A'\ Z\longrightarrow \mathit{return}\ s\ t\in A))\}\end{array}}{\Gamma,\Theta\vdash P\ \mathit{Call\ init\ p\ return\ result}\ Q,A}$$

The central idea of this rule is to simulate the actual call *Call init p return result* in state s, with a call of the specification *Call ini p ret res* in state *init s*. The following figure shows the sequence of intermediate states for normal termination of both executions. On the top the actual call, on the bottom the call of the specification:



We start in state s for which the precondition P holds. To be able to make use of the procedure specification we have to find a suitable instance of the auxiliary variable Z so that the precondition of the specification holds: $\mathit{init}\ s \in P'\ Z$. Let t be the state immediately after execution of the procedure body, before returning to the caller and passing results. We know from the procedure specification that when exiting the procedure according to *res* the postcondition will hold: $\mathit{res}\ (\mathit{init}\ s)\ t \in Q'\ Z$. From this we have to conclude that exiting the procedure according to the actual function *result* will lead us to a state in Q. For abrupt termination the analogous simulation idea applies.

The side-condition $\forall s.\ \mathit{ini}\ (\mathit{init}\ s) = \mathit{init}\ s$ is no real burden. A procedure specification will be given with the canonical procedure parameters, according to the procedure declaration. So *ini* will be the identity. Also in actual program verification the formal procedure exit protocol defined by *ret* and *res* and the actual protocol *return* and *result* will be closely related to each other. Just because these protocols are modelled so generically here it might seem tedious at first sight. But *ret* and *return* will both copy back global variables to the caller (so they will actually be the same), and *res* will just store the result of the procedure at the formal result parameter in the callers local variables, whereas *result* will store it to the actual one. In the record implementation, the verification condition generator can use simplification of the record updates and selections encoded in the functions and the assertions, to achieve the expected adaption of the specification to the actual calling context.

Procedure Implementation To verify a procedure implementation against its specification we also need a rule that descends into the procedure body. The Hoare logic can deal with (mutually) recursive procedures. The basic idea of a Hoare rule for recursive procedures is simple. We prove that the procedure body respects the specification, under the assumption that recursive calls to the procedure will meet the specification. To model this assumption the context Θ comes in. If a procedure specification is in this context, we can immediately derive this specification within the Hoare logic.

$$\frac{(P,\ c,\ Q,\ A) \in \Theta}{\Gamma,\Theta \vdash P\ c\ Q,A}$$

To handle a set *Procs* of mutually recursive procedures we enrich the context by all the procedure specifications, while we prove their bodies.

$$\frac{\begin{array}{c}\Theta' {=} \Theta \cup (\bigcup p {\in} Procs.\\ \quad \bigcup Z.\{(P\ p\ Z, Call\ (Init\ p)\ p\ (Ret\ p)\ (Res\ p), Q\ p\ Z, A\ p\ Z)\})\\ \forall p {\in} Procs.\ \forall Z.\ \Gamma,\Theta' {\vdash} (P\ p\ Z)\ the\ (\Gamma\ p)\ (Q'\ p\ Z),(A'\ p\ Z)\\ \forall p {\in} Procs.\ \forall Z\ s\ t.\ s \in P\ p\ Z \longrightarrow\\ \quad (t \in Q'\ p\ Z \longrightarrow Res\ p\ s\ t \in Q\ p\ Z) \wedge (t \in A'\ p\ Z \longrightarrow Ret\ p\ s\ t \in A\ p\ Z)\\ \forall p {\in} Procs.\ Init\ p = (\lambda s.\ s) \qquad Procs \subseteq dom\ \Gamma\end{array}}{\forall Z.\ \forall p {\in} Procs.\ \Gamma,\Theta {\vdash} (P\ p\ Z)\ Call\ (Init\ p)\ p\ (Ret\ p)\ (Res\ p)\ (Q\ p\ Z),(A\ p\ Z)}$$

Since we deal with the set *Procs* of procedures we also have to give the pre- and postconditions and the parameter and return/result passing protocols for all these procedures. We use the functions P, Q, A, *Init*, *Ret* and *Res* which map procedure names to the desired entities. Z plays the role of an auxiliary (or logical) variable. It usually fixes (parts of) the pre state, so that we can refer to it in the post state. In the Hoare rule for procedure specifications, which we have described before, we had the freedom to pick a particular Z so that $s \in P \longrightarrow init\ s \in P'\ Z$ holds. Since we have the freedom there, we now have to prove the procedure bodies for all possible Z. Think of Z as the pre state. We prove that the specification holds for all pre states (that satisfy the precondition). When we later on use the specification to prove a procedure call we instantiate Z to the actual state to adapt the specification to the current calling context. Whereas the postconditions for the procedures are given by $Q\ p\ Z$ and $A\ p\ Z$ we prove different postconditions for the procedure bodies, namely $Q'\ p\ Z$ and $A'\ p\ Z$. This stems from the fact that the final state of a procedure body is not the final state of the corresponding procedure call, since exiting the procedure lies in-between them. We take the intermediate assertions $Q'\ p\ Z$ and $A'\ p\ Z$ to describe the final state of the procedure body. The big side-condition then links $Q'\ p\ Z$ and $Q\ p\ Z$ or $A'\ p\ Z$ and $A\ p\ Z$ together. It is worth noticing that in most cases Q' and Q or A' and A will actually be the same, since a proper postcondition will only talk about global variables or result variables of the final state and their relation to the initial state and not about local variables. Keep in mind that we verify a procedure specification here. So the *init* functions will not pass any parameters ($Init\ p = (\lambda s.\ s)$), and the return/result functions will only assign to global variables or formal result parameters. So global variables and result variables at the end of the procedure body will be the same as after exiting the procedure. Finally, with $Procs \subseteq dom\ \Gamma$, we make sure that the calculation will not get stuck.

Dynamic Procedure Call The rule for dynamic procedure call is a slight generalisation of the rule for static procedure call. Since the selected procedure depends on the state, we have the liberty to select a suitable specification dependent on the state.

$$\frac{\begin{array}{l}\forall s{\in}P.\ \forall Z.\ \Gamma,\Theta\vdash(P'\ s\ Z)\ \mathit{Call\ ini}\ (p\ s)\ \mathit{ret\ res}\ (Q'\ s\ Z),(A'\ s\ Z)\\ \qquad\qquad \forall s.\ \mathit{ini}\ (\mathit{init}\ s) = \mathit{init}\ s\\ P \subseteq \{s.\ (\exists Z.\ \mathit{init}\ s \in P'\ s\ Z\ \wedge\\ \qquad (\forall t.\ \mathit{res}\ (\mathit{init}\ s)\ t \in Q'\ s\ Z \longrightarrow \mathit{result}\ s\ t \in Q)\ \wedge\\ \qquad (\forall t.\ \mathit{ret}\ (\mathit{init}\ s)\ t \in A'\ s\ Z \longrightarrow \mathit{return}\ s\ t \in A))\}\end{array}}{\Gamma,\Theta\vdash P\ \mathit{DynCall\ init}\ p\ \mathit{return\ result}\ Q,A}$$

5 Hoare Logic for Total Correctness

The Hoare logic for total correctness ensures both partial correctness and termination. The judgement is written as $\Gamma,\Theta\vdash_t P\ c\ Q,A$. The intended semantics of this judgement is described by the notion of validity.

$\Gamma\models_t\ P\ c\ Q,A \equiv \Gamma\models\ P\ c\ Q,A \wedge (\forall s{\in}\mathit{Normal}\ `\ P.\ \Gamma\vdash c \downarrow s)$

Validity for total correctness directly captures the informal description given above. The quadruple must be valid in the sense of partial correctness, and the program c has to terminate for all *Normal* states that satisfy the precondition P.

Again we have proven both soundness and completeness of the Hoare logic.

– **theorem** *soundness*: $\Gamma,\{\}\vdash_t P\ c\ Q,A \longrightarrow \Gamma\models_t\ P\ c\ Q,A$
We can only derive valid Hoare quadruples out of the empty context.
– **theorem** *completeness*: $\Gamma\models_t\ P\ c\ Q,A \longrightarrow \Gamma,\{\}\vdash_t P\ c\ Q,A$
We can derive every valid Hoare quadruple out of the empty context.

Most of the Hoare logic rules for total correctness are structurally equivalent to their partial correctness counterparts. We will only focus on those interesting rules with an impact on termination, namely loops and recursion. The basic idea is to justify termination by a well-founded relation on the state space.

Loop We have to supply a well-founded relation r on the state space, which decreases by evaluation of the loop body. Formally this is expressed by first fixing the pre-state with the singleton set $\{\tau\}$. In the postcondition for *Normal* termination of the loop body we end up in a state s and have to show that this state is "smaller" as τ according to the relation: $(s,\ \tau) \in r$. For *Abrupt* termination we do not have to take any care, since it will exit the loop anyway.

$$\frac{\mathit{wf}\ r \qquad \forall\tau.\ \Gamma,\Theta\vdash_t(\{\tau\} \cap g \cap P \cap b)\ c\ (\{s.\ (s,\ \tau) \in r\} \cap g \cap P),A}{\Gamma,\Theta\vdash_t(g \cap P)\ \mathit{While}\ g\ b\ c\ (g \cap P \cap -\ b),A}$$

Procedure Implementation In contrast to partial correctness we now only assume "smaller" recursive procedure calls correct while verifying the procedure bodies. Here "smaller" again is in the sense of a well-founded relation r. To be able to handle mutually recursive procedures the relation r not only relates state spaces but also takes the procedure names into account. We fix the pre-state of the procedure p with the singleton set $\{\tau\}$. For every call to a procedure q in a state s which is "smaller" than the initial call of p in state τ according to the relation $(((s,q),(\tau,p)) \in r)$, we can safely assume the specification of q while verifying the body of p.

$$\frac{\begin{array}{c} \mathit{wf}\ r \\ \begin{array}{l} \Theta'{=}\lambda\tau.\ \Theta \cup (\bigcup q{\in}\mathit{Procs}.\ \bigcup Z. \\ \quad \{(P\ q\ Z \cap \{s.\ ((s,q),(\tau,p)) \in r\}, \mathit{Call}\ (\mathit{Init}\ q)\ q\ (\mathit{Ret}\ q)\ (\mathit{Res}\ q), Q\ q\ Z, A\ q\ Z)\}) \end{array} \\ \forall p{\in}\mathit{Procs}.\ \forall \tau\ Z.\ \Gamma,\Theta'\ \tau{\vdash_t}(\{\tau\} \cap P\ p\ Z)\ \mathit{the}\ (\Gamma\ p)\ (Q'\ p\ Z),(A'\ p\ Z) \\ \begin{array}{l} \forall p{\in}\mathit{Procs}.\ \forall Z\ s\ t.\ s \in P\ p\ Z \longrightarrow \\ \qquad (t \in Q'\ p\ Z \longrightarrow \mathit{Res}\ p\ s\ t \in Q\ p\ Z) \wedge (t \in A'\ p\ Z \longrightarrow \mathit{Ret}\ p\ s\ t \in A\ p\ Z) \end{array} \\ \forall p{\in}\mathit{Procs}.\ \mathit{Init}\ p = (\lambda s.\ s) \qquad \mathit{Procs} \subseteq \mathit{dom}\ \Gamma \end{array}}{\forall Z.\ \forall p{\in}\mathit{Procs}.\ \Gamma,\Theta{\vdash_t}(P\ p\ Z)\ \mathit{Call}\ (\mathit{Init}\ p)\ p\ (\mathit{Res}\ p)\ (\mathit{Ret}\ p)\ (Q\ p\ Z),(A\ p\ Z)}$$

6 Utilising the Framework

In this section we will sketch the integration of the Hoare logics in Isabelle/HOL and how we can express and deal with typical programming language constructs in our framework. Our purpose is to give an impression of how program verification "feels" like in our verification environment. The main tool is a verification condition generator that is implemented as tactic called *vcg*. The Hoare logic rules are defined in a weakest precondition style, so that we can almost take them as they are. We derive variants of the Hoare rules where all assertions in the conclusions are plain variables so that they are applicable to every context. We get the following format: $\dfrac{P \subseteq \mathit{WP}\ \ldots}{\Gamma,\Theta{\vdash}P\ c\ Q,A}$. The ... may be recursive Hoare quadruples or side-conditions which somehow lead to the weakest precondition *WP*. If we recursively apply rules of this format until the program c is completely processed, then we have calculated the weakest precondition *WP* and are left with the verification condition $P \subseteq \mathit{WP}$. The set inclusion is then transformed to an implication. Then we can split the state records so that the record representation will not show up in the resulting verification condition. This leads to quite comprehensible proof obligations that closely resemble the specifications. Moreover we supply some concrete syntax for programs. The mapping to the abstract syntax should be obvious. As a shorthand an empty set Θ can be omitted and writing a Hoare triple instead of the quadruples is an abbreviation for an empty postcondition for abrupt termination.

Basics If we refer to components (variables) of the state-space of the program we always mark these with ´ (in assertions and also in the program itself). Assertions are ordinary Isabelle/HOL sets. As we usually want to refer to the state space in the assertions, we provide special brackets $\{\!|\ldots|\!\}$ for them. Internally, an assertion of the from $\{\!|\acute{I} \le 3|\!\}$ gets expanded to $\{s.\ I\ s \le 3\}$ in ordinary set comprehension notation of Isabelle.

Although our assertions work semantically on the state space, stepping through verification condition generation "feels" like the expected syntactic substitutions of traditional Hoare logic. This is achieved by light simplification on the assertions calculated by the Hoare rules.

lemma $\Gamma\vdash \{\!|\acute{M} = a \wedge \acute{N} = b|\!\}$
$\acute{I} := \acute{M};\ \acute{M} := \acute{N};\ \acute{N} := \acute{I}$
$\{\!|\acute{M} = b \wedge \acute{N} = a|\!\}$
apply *vcg-step*

$1.\ \Gamma\vdash\{\!|\acute{M} = a \wedge \acute{N} = b|\!\}\ \acute{I} := \acute{M};\ \acute{M} := \acute{N}\ \{\!|\acute{M} = b \wedge \acute{I} = a|\!\}$

apply *vcg-step*

$1.\ \Gamma\vdash\{\!|\acute{M} = a \wedge \acute{N} = b|\!\}\ \acute{I} := \acute{M}\ \{\!|\acute{N} = b \wedge \acute{I} = a|\!\}$

apply *vcg-step*

$1.\ \{\!|\acute{M} = a \wedge \acute{N} = b|\!\} \subseteq \{\!|\acute{N} = b \wedge \acute{M} = a|\!\}$

apply *vcg-step*

$1.\ \bigwedge M\ N.\ N = N \wedge M = M$

by *simp*

Loops The following example calculates multiplication by an iterated addition. The user annotates the loop with an invariant.

lemma $\Gamma\vdash \{\!|\acute{M} = 0 \wedge \acute{S} = 0|\!\}$
WHILE $\acute{M} \neq a$ ***INV*** $\{\!|\acute{S} = \acute{M} * b|\!\}$
DO $\acute{S} := \acute{S} + b;\ \acute{M} := \acute{M} + 1$ ***OD***
$\{\!|\acute{S} = a * b|\!\}$
apply *vcg*

$1.\ \bigwedge M\ S.\ [\![M = 0;\ S = 0]\!] \Longrightarrow S = M * b$
$2.\ \bigwedge M\ S.\ [\![S = M * b;\ M \neq a]\!] \Longrightarrow S + b = (M + 1) * b$
$3.\ \bigwedge M\ S.\ [\![S = M * b;\ \neg\ M \neq a]\!] \Longrightarrow S = a * b$

The verification condition generator gives us three proof obligations, stemming from the path from the precondition to the invariant, from the invariant together with the loop condition through the loop body to the invariant, and finally from the invariant together with the negated loop condition to the postcondition.

For total correctness the user also has to supply the variant, which in our case is a well-founded relation. We make use of the infrastructure for well-founded recursion that is already present in Isabelle/HOL [14]. In the example the distance of the loop variable M to a decreases in every iteration. This is expressed by the measure function $a - {}^{\acute{}}M$ on the state-space.

lemma $\Gamma\vdash_t \{\!|\, {}^{\acute{}}M = 0 \wedge {}^{\acute{}}S = 0 |\!\}$
$\boldsymbol{WHILE}\ {}^{\acute{}}M \neq a\ \boldsymbol{INV}\{\!|\, {}^{\acute{}}S = {}^{\acute{}}M * b \wedge {}^{\acute{}}M \leq a |\!\}\ \textbf{VAR}\ \mathit{MEASURE}\ a - {}^{\acute{}}M$
$\boldsymbol{DO}\ {}^{\acute{}}S := {}^{\acute{}}S + b;\ {}^{\acute{}}M := {}^{\acute{}}M + 1\ \boldsymbol{OD}$
$\{\!|\, {}^{\acute{}}S = a * b |\!\}$
apply *vcg*

1. $\bigwedge M\ S.\ [\![M = 0;\ S = 0]\!] \Longrightarrow S = M * b \wedge M \leq a$
2. $\bigwedge M\ S.\ [\![S = M * b;\ M \leq a;\ M \neq a]\!]$
   $\Longrightarrow a - (M + 1) < a - M \wedge S + b = (M + 1) * b \wedge M + 1 \leq a$
3. $\bigwedge M\ S.\ [\![S = M * b;\ M \leq a;\ \neg\ M \neq a]\!] \Longrightarrow S = a * b$

The variant annotation results in the proof obligation $a - (M + 1) < a - M$ after verification condition generation.

Abrupt Termination We can implement breaking out of a loop by a ***THROW*** inside the loop body and enclosing the loop into a ***TRY−CATCH*** block.

lemma $\Gamma\vdash \{\!|\, {}^{\acute{}}I \leq 3 |\!\}$
$\boldsymbol{TRY\ WHILE}\ \mathit{True}\ \boldsymbol{INV}\ \{\!|\, {}^{\acute{}}I \leq 10 |\!\}$
$\quad \boldsymbol{DO\ IF}\ {}^{\acute{}}I < 10\ \boldsymbol{THEN}\ {}^{\acute{}}I := {}^{\acute{}}I + 1\ \boldsymbol{ELSE\ THROW\ FI\ OD}$
$\boldsymbol{CATCH\ SKIP\ YRT}$
$\{\!|\, {}^{\acute{}}I = 10 |\!\},\{\}$
apply *vcg*

1. $\bigwedge I.\ I \leq 3 \Longrightarrow I \leq 10$
2. $\bigwedge I.\ [\![I \leq 10;\ \mathit{True}]\!]$
   $\Longrightarrow (I < 10 \longrightarrow I + 1 \leq 10) \wedge (\neg\ I < 10 \longrightarrow I = 10)$
3. $\bigwedge I.\ [\![I \leq 10;\ \neg\ \mathit{True}]\!] \Longrightarrow I = 10$

The first subgoal stems from the path from the precondition to the invariant. The second one from the loop body. We can assume the invariant and the loop condition and have to show that the invariant is preserved when we execute the ***THEN*** branch, and that the ***ELSE*** branch will imply the assertion for abrupt termination, which will be $\{\!|\, {}^{\acute{}}I = 10 |\!\}$ according to the rule for *Catch*. The third subgoal expresses that normal termination of the while loop has to imply the postcondition. But the loop will never terminate normally and so the third subgoal will trivially hold. All subgoals are quite simple and can be proven automatically.

To model a `continue` we can use the same idea and put a ***TRY***−***CATCH*** around the loop body. Or for `return` we can put the procedure body into a ***TRY***−***CATCH***. To distinguish the kind of abrupt termination we can add a ghost variable *Abr* to the state space and store this information before the ***THROW***. For example `break` can be translated to *´Abr* := *''Break''*; ***THROW***, and the matching ***CATCH*** will peek for this variable to decide whether it is responsible or not: ***IF*** *´Abr* = *''Break''* ***THEN SKIP ELSE THROW FI***. This idea can immediately be extended to exceptions. We just have to make sure to use a global variable to store the kind of exception, so that it will properly pass procedure boundaries.

Procedures We provide the command **procedures**, to declare, define and specify a procedure.

procedures *Fac* (*N*|*R*) =
 IF *´N* = *0* ***THEN*** *´R* := *1*
 ELSE *´R* := ***CALL*** *Fac*(*´N* − *1*); *´R* := *´N* ∗ *´R*
 FI

 Fac-spec: ∀ *n*. *Γ*⊢⦃*´N* = *n*⦄ *´R* := ***CALL*** *Fac*(*´N*) ⦃*´R* = *fac n*⦄

A procedure is given by the signature of the procedure followed by the procedure body and named specifications. The signature consists of the name of the procedure and a list of parameters. The parameters in front of the pipe | are value parameters and behind the pipe are the result parameters. Value parameters model call by value semantics. The value of a result parameter at the end of the procedure is passed back to the caller.

The procedure specifications are ordinary Hoare quadruples. The precondition here fixes the current value *´N* to the logical variable *n*. Universal quantification of *n* enables us to adapt the specification to an actual parameter. The specification will be used in the rule for procedure call when we come upon a call to *Fac*. Thus *n* plays the role of the auxiliary variable *Z*.

The procedures command provides convenient syntax for procedure calls (that creates the proper *init*, *return* and *result* functions on the fly), defines a constant for the procedure body (named *Fac-body*) and creates two *locales*. The purpose of locales is to set up logical contexts to support modular reasoning [1].

One locale is named like the specification, in our case *Fac-spec*. This locale contains the procedure specification. The second locale is named *Fac-impl* and contains the assumption *Γ ''Fac''* = *Some Fac-body*, which expresses that the procedure is defined in the current context. The purpose of these locales is to give us easy means to setup the context in which we will prove programs correct.

By including the locale *Fac-spec*, the following lemma assumes that the specification of the factorial holds. The *vcg* will make use the specification to handle the procedure call. The lemma also illustrates locality of *I*.

lemma includes *Fac-spec* **shows**

$\Gamma \vdash \{\!|\, {}^\prime M = 3 \wedge {}^\prime I = 2 |\!\}\ {}^\prime R := \textbf{\textit{CALL}}\ Fac\ ({}^\prime M)\ \{\!|\, {}^\prime R = 6 \wedge {}^\prime I = 2 |\!\}$
apply *vcg*

1. $\bigwedge I\ M.\ [\![M = 3;\ I = 2]\!] \Longrightarrow fac\ M = 6 \wedge I = 2$

To verify the procedure body we use the rule for recursive procedures. We extend the context with the procedure specification. In this extended context the specification will hold by the assumption rule. We then verify the procedure body by using *vcg*, which will use the assumption to handle the recursive call.

lemma includes *Fac-impl* **shows**
$\forall n.\ \Gamma \vdash \{\!|\, {}^\prime N = n |\!\}\ \textbf{\textit{CALL}}\ Fac({}^\prime N, {}^\prime R)\ \{\!|\, {}^\prime R = fac\ n |\!\}$
apply (*hoare-rule CallRec1-SamePost*)

1. $\forall n.\ \Gamma,(\bigcup n\ \{(\{\!|\, {}^\prime N = n |\!\},\ {}^\prime R := \textbf{\textit{CALL}}\ Fac({}^\prime N),\ \{\!|\, {}^\prime R = fac\ n |\!\},\ \{\})\})$
   $\vdash \{\!|\, {}^\prime N = n |\!\}$
   $\textbf{\textit{IF}}\ {}^\prime N = 0\ \textbf{\textit{THEN}}\ {}^\prime R := 1$
   $\textbf{\textit{ELSE}}\ {}^\prime R := \textbf{\textit{CALL}}\ Fac({}^\prime N - 1);\ {}^\prime R := {}^\prime N * {}^\prime R\ \textbf{\textit{FI}}$
   $\{\!|\, {}^\prime R = fac\ n |\!\}$

apply *vcg*

1. $\bigwedge N.\ (N = 0 \longrightarrow 1 = fac\ N) \wedge (N \neq 0 \longrightarrow N * fac\ (N - 1) = fac\ N)$

The rule *CallRec1-SamePost* is a specialised version of the general rule for recursion, tailored for one (mutual recursive) procedure, and where the intermediate assertions for the procedure body and the actual postcondition are the same. The method *hoare-rule* applies a single rule and solves the canonical side-conditions concerning the parameter passing and returning protocols. Moreover it expands the procedure body.

For total correctness the user supplies a well-founded relation. For the factorial the input parameter N decreases in the recursive call. This is expressed by the measure function $\lambda(s,p).\ {}^sN$. The relation can depend on both the state-space s and the procedure name p. The latter is useful to handle mutual recursion. The prefix superscript in ${}^sN$ is a shorthand for record selection $N\ s$ and is used to refer to state components of a named state.

lemma includes *Fac-impl* **shows**
$\forall n.\ \Gamma \vdash_t \{\!|\, {}^\prime N = n |\!\}\ {}^\prime R := \textbf{\textit{CALL}}\ Fac({}^\prime N)\ \{\!|\, {}^\prime R = fac\ n |\!\}$
apply (*hoare-rule CallRec1-SamePost*$_t$ [**where** *r=measure* $(\lambda(s,p).\ {}^sN)$])

$$
\begin{array}{l}
1.\ \forall \tau\ n.\ \Gamma,(\bigcup_n \{(\{\!|\,{}'N = n|\!\} \cap \{\!|\,{}'N < {}^{\tau}N|\!\},\ {}'R := \textbf{\textit{CALL}}\ \mathit{Fac}({}'N),\\
\qquad\qquad\qquad \{\!|\,{}'R = \mathit{fac}\ n|\!\},\ \{\})\})\\
\qquad\qquad \vdash_t(\{\tau\} \cap \{\!|\,{}'N = n|\!\})\\
\qquad\qquad\quad \textbf{\textit{IF}}\ {}'N = 0\ \textbf{\textit{THEN}}\ {}'R := 1\\
\qquad\qquad\quad \textbf{\textit{ELSE}}\ {}'R := \textbf{\textit{CALL}}\ \mathit{Fac}({}'N - 1);\ {}'R := {}'N * {}'R\ \textbf{\textit{FI}}\\
\qquad\qquad\quad \{\!|\,{}'R = \mathit{fac}\ n|\!\}
\end{array}
$$

We may only assume the specification for "smaller" states $\{\!|\,{}'N < {}^{\tau}N|\!\}$, where state τ gets fixed in the precondition.

apply *vcg*

$$
\begin{array}{l}
1.\ \bigwedge N.\ (N = 0 \longrightarrow 1 = \mathit{fac}\ N) \wedge\\
\qquad\quad (N \neq 0 \longrightarrow N - 1 < N \wedge N * \mathit{fac}\ (N - 1) = \mathit{fac}\ N)
\end{array}
$$

The measure function results in the proof obligation $N - 1 < N$ in the verification condition.

Heap The heap can contain structured values like `structs` in C or records in Pascal. Our model of the heap follows Bornat [2]. We have one heap variable f of type *ref* ⇒ *value* for each component f of type *value* of the `struct`.

A typical structure to represent a linked list in the heap is `struct {int cont; list *next} list`. The structure contains two components, `cont` and `next`. So we will also get two heap variables, *cont* of type *ref* ⇒ *int* and *next* of type *ref* ⇒ *ref* in our state space record:

record *list-vars* =
  next::*ref* ⇒ *ref*
  cont::*ref* ⇒ *ref*
  p::*ref*
  q::*ref*
  r::*ref*

In this state space *next* and *cont* are global variables and *p* and *q* are local ones. This is given to Isabelle by the **globals** command.

globals *list-vars* = *next* **and** *cont*

The only effect of this command is that the *return*/*result* functions that are created by the syntax translations will actually pass all global variables back to the caller. References *ref* are isomorphic to the natural numbers and contain *Null*.

The approach to specify procedures on lists basically follows [9]. From the pointer structure in the heap we (relationally) abstract to HOL lists of references. Then we can specify further properties on the level of HOL lists, rather then on the heap:

$\mathit{List}\ x\ h\ [] = (x = \mathit{Null})$
$\mathit{List}\ x\ h\ (p\ \#\ ps) = (x = p \wedge x \neq \mathit{Null} \wedge \mathit{List}\ (h\ x)\ h\ ps)$

The list of references is obtained from the heap h by starting with the reference x, following the references in h up to *Null*.

We define in place list reversal. The list pointed to by p in the beginning is *Ps*. In the end q points to the reversed list *rev Ps*. The notation $r{\rightarrow}f$ mimics the field selection syntax of C and is translated to ordinary function application for field lookup and function update for field assignment.

```
lemma $\Gamma\vdash\{\!|\mathit{List}\ \acute{}p\ \acute{}\mathit{next}\ \mathit{Ps}|\!\}$
$\acute{}q := \mathit{Null}$;
WHILE $\acute{}p \neq \mathit{Null}$
INV $\{\!|\exists \mathit{Ps}'\ \mathit{Qs}'.\ \mathit{List}\ \acute{}p\ \acute{}\mathit{next}\ \mathit{Ps}' \wedge \mathit{List}\ \acute{}q\ \acute{}\mathit{next}\ \mathit{Qs}' \wedge$
        $\mathit{set}\ \mathit{Ps}' \cap \mathit{set}\ \mathit{Qs}' = \{\} \wedge \mathit{rev}\ \mathit{Ps}'\ @\ \mathit{Qs}' = \mathit{rev}\ \mathit{Ps}|\!\}$
DO $\acute{}r := \acute{}p;\ \acute{}p := \acute{}p{\rightarrow}\acute{}\mathit{next};\ \acute{}r{\rightarrow}\acute{}\mathit{next} := \acute{}q;\ \acute{}q := \acute{}r$ OD
$\{\!|\mathit{List}\ \acute{}q\ \acute{}\mathit{next}\ (\mathit{rev}\ \mathit{Ps})|\!\}$
by (vcg,fastsimp+)
```

In the loop, pointer p sequentially steps through the list *Ps* and q accumulates the reversed list. Therefore the desired outcome *rev Ps* can be obtained by appending the the reversed list pointed to by p and the list pointed to by q. This is expressed by $\mathit{rev}\ \mathit{Ps}'\ @\ \mathit{Qs}' = \mathit{rev}\ \mathit{Ps}$ in the invariant. Separation of the two lists Ps' and Qs' is captured by the empty intersection of references: $\mathit{set}\ \mathit{Ps}' \cap \mathit{set}\ \mathit{Qs}' = \{\}$.

The specification of list reversal above, does not capture the information about the parts of the heap that do not change. But this information is crucial to properly use the specification in different contexts. We encapsulate this code fragment in a procedure and give the following, more detailed specification.

```
procedures $\mathit{Rev}(p|q)$ =
 $\acute{}q := \mathit{Null}$;
 WHILE $\acute{}p \neq \mathit{Null}$
 DO $\acute{}r := \acute{}p;\ \acute{}p := \acute{}p{\rightarrow}\acute{}\mathit{next};\ \acute{}r{\rightarrow}\acute{}\mathit{next} := \acute{}q;\ \acute{}q := \acute{}r$ OD

 Rev-spec:
  $\forall \sigma\ \mathit{Ps}.\ \Gamma\vdash\ \{\!|\sigma.\ \mathit{List}\ \acute{}p\ \acute{}\mathit{next}\ \mathit{Ps}|\!\}\ \acute{}q :=$ CALL $\mathit{Rev}(\acute{}p)$
  $\{\!|\mathit{List}\ \acute{}q\ \acute{}\mathit{next}\ (\mathit{rev}\ \mathit{Ps}) \wedge (\forall p.\ p \notin \mathit{set}\ \mathit{Ps} \longrightarrow (\acute{}\mathit{next}\ p = {}^{\sigma}\mathit{next}\ p))|\!\}$

 Rev-modifies:
  $\forall \sigma.\ \Gamma\vdash\{\sigma\}\ \acute{}q :=$ CALL $\mathit{Rev}(\acute{}p)\ \{t.\ t\ \mathit{may\text{-}only\text{-}modify}\ \sigma\ \mathit{in}\ [\mathit{next},q]\}$
```

We have given two specifications this time. The first one captures the functional behaviour and additionally expresses that all parts of the *next* heap not contained in *Ps*, will stay the same (σ denotes the pre-state). The second one is a modifies clause that lists all the state components that may be changed by the procedure. Therefore we know that the *cont* parts will not be changed. The assertion t $\mathit{may\text{-}only\text{-}modify}\ \sigma\ \mathit{in}\ [\mathit{next},\ p]$ abbreviates the following relation between the final state t and the initial state σ: $\exists\, next\ p.\ t{=}\sigma(\!|next{:=}next{,}p\ {:=}p|\!)$. This modifies clause can be exploited during verification condition generation. We derive that we can reduce the *result* function in the call to *Rev*, which copies the global components *next* and *cont* back, to one that only copies *next* back. So *cont* will actually behave like a local variable in the resulting proof obligation. This is an effective way to express separation of different pointer structures in the heap and can be handled completely automatic during verification condition generation. For example, reversing a list will only modify the *next* heap but not some *left* and *right* heaps of a tree structure. Moreover the modifies clause itself can be verified automatically. The following example illustrates the effect of the modifies clause.

lemma includes *Rev-spec* + *Rev-modifies* **shows**
$\Gamma\vdash\{\!|\,\acute{}cont{=}c \wedge List\ \acute{}p\ \acute{}next\ Ps|\!\}\ \acute{}p := \textbf{\textit{CALL}}\ Rev(\acute{}p)$
$\quad\{\!|\,\acute{}cont{=}c \wedge List\ \acute{}p\ \acute{}next\ (rev\ Ps)|\!\}$
apply *vcg*

$$\begin{aligned}
&1.\ \bigwedge next\ cont\ p.\\
&\qquad List\ p\ next\ Ps \Longrightarrow\\
&\qquad \forall\, nexta\ q.\\
&\qquad\quad List\ q\ nexta\ (rev\ Ps) \wedge (\forall\, p.\ p \notin set\ Ps \longrightarrow nexta\ p = next\ p) \longrightarrow\\
&\qquad\quad cont = cont \wedge List\ q\ nexta\ (rev\ Ps)
\end{aligned}$$

The impact of the modifies clause shows up in the verification condition. The content heap results in the same variable before and after the procedure call ($cont = cont$), whereas the next heap is described by *next* in the beginning and by *nexta* in the end. The specification of *Rev* relates both next heap states.

Memory Management To model allocation and deallocation we need some bookkeeping of allocated references. This can be achieved by an auxiliary ghost variable *alloc* in the state space. A good candidate is a list of allocated references. A list is per se finite, so that we can always get a new reference. By the length of the list we can also handle space limitations. Allocation of memory means to append a new reference to the allocation list. Deallocation of memory means to remove a reference from the allocation list. To guard against dangling pointers we can regard the allocation list: $\{\!|\,\acute{}p{\neq}Null \wedge \acute{}p \in set\ \acute{}alloc|\!\}{\mapsto}\ \acute{}p{\rightarrow}\acute{}cont := 2$.

The use of guards is a flexible mechanism to adapt the model to the kind of language we are looking at. If it is type safe like Java and there is no explicit deallocation by the user, we can remove some guards. If the `new` instruction of the programming language does not initialise the allocated memory we can add another ghost variable to watch for initialised memory through guards.

7 Conclusion

We have presented a flexible, sound and complete Hoare calculus for sequential imperative programs with mutually recursive procedures and dynamic procedure call. We have elaborated how to model various kinds of abrupt termination like `break`, `continue`, `return` and exceptions, how to deal with global variables, heap and memory management issues. The polymorphic state space of the programming language allows us to choose the adequate representation for the current verification task. Depending on the context we can for example decide, whether it is preferable to model certain variables as unbounded integers in HOL or as bit-vectors, without changing the program representation or logics. Guards make it possible to customise the runtime faults we are interested in. The usage of records as state space representation gives us a natural way to express typing of program variables and yields comprehensible verification conditions. Moreover in combination with the modifies clause we can lift separation of heap components, which are directly expressible in the split heap model, to the level of procedures. Crucial parts of the frame problem can then already be handled during verification condition generation. The calculus is developed, verified and integrated in the theorem prover Isabelle and the resulting verification environment is seamless fitting into the infrastructure of Isabelle/HOL.

References

1. C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs: International Workshop, TYPES 2003, Torino, Italy, April 30–May 4, 2003, Selected Papers*, number 3085 in Lect. Notes in Comp. Sci., pages 34–50. Springer-Verlag, 2004.
2. R. Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lect. Notes in Comp. Sci.*, pages 102–126. Springer-Verlag, 2000.
3. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
4. J. Harrison. Formalizing Dijkstra. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lect. Notes in Comp. Sci.*, pages 171–188, Canberra, Australia, 1998. Springer-Verlag.
5. P. V. Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures.* PhD thesis, Department of Computer Science, University of California, Los Angeles, 1995.
6. M. Huisman. *Java program verification in higher order logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2000.
7. B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.
8. T. Kleymann. Hoare Logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

9. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *Lect. Notes in Comp. Sci.*, pages 121–135. Springer-Verlag, 2003.
10. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lect. Notes in Comp. Sci.*, pages 63–77. Springer-Verlag, 2000.
11. M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer, Berlin.
12. W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1479 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998.
13. T. Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.
14. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.
15. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
16. D. v. Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
17. C. Pierik and F. S. de Boer. Computer-aided specification and verification of annotated object-oriented programs. In *Formal Methods for Open Object-Based Distributed Systems 2003*, volume 2884 of *Lect. Notes in Comp. Sci.*, pages 163–177. Springer-Verlag, 2002.
18. L. Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.
19. M. Wenzel. Miscellaneous Isabelle/Isar examples for higher order logic. Isabelle/Isar proof document, 2001.