A Verification Environment for Sequential Imperative Programs in Isabelle/HOL *

Norbert Schirmer

Technische Universität München, Institut für Informatik
url://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.

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, side-effecting expressions, 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.

* This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project (http://www.verisoft.de) under grant 01 IS C38. The responsibility for this article lies with the author.

Related Work The tradition of embedding a programming language in HOL goes back to the work of Gordon [12], where a while language with variables ranging over natural numbers is introduced. A polymorphic state space was already used by Wright et. al. [21] in their machnisation of refinement concepts, by Harrison in his formalisation of Dijkstra [5] and by Prensa to verify parallel programs [18]. Still procedures were not present. Homeier [6] introduces procedures, but the variables are again limited to numbers. Later on detailed semantics for Java [16,7] and C [15] were embedded in a theorem prover. But verification of even simple programs suffers from the complex models.

The Why tool [4] 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 C to the tool by representing the heap in reference variables. Although the Why tool and the work we present in this paper both provide comparable verification environments for imperative programs the theoretical foundations to achieve this are quite different: Filliâtre builds up a sophisticated type theory incorporating an effect analysis on the input language, whereas the framework of Hoare logics and the simple type system of HOL is sufficient for our needs. Moreover our entire development, the calculus together with its soundness and completeness proof, is carried out in Isabelle/HOL, in contrast to the pen and paper proofs of Filliâtre [3].

The rest of the paper is structured as follows. We start with a brief introduction to Isabelle/HOL in Section 2; in Section 3 we introduce the programming language model and the Hoare logics; Section 4 describes the integration into Isabelle and shows how we deal with various language constructs; Section 5 concludes.

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 $T_1 \Rightarrow (T_2\ option)$. The domain of such a partial function *f* is *dom f*.

There is also a destructor for the constructor *Some*, the function *the*:: *'a option* ⇒ *'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; *f* is a state-update: *'s* ⇒ *'s*
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 b c: Loop
Call p: Static procedure call, *p*::*'p*
Throw: Initiate abrupt termination
Catch c_1 c_2: Handle abrupt termination of c_1 with c_2
DynCom c: Dynamic (state dependant) command: *c*::*'s* ⇒ (*'s*,*'p*) *com*

The procedure call above is parameterless. In 4.4 we implement a call with parameters: *call init p return result*.

The dynamic command *DynCom* allows to abstract a statement over the state. It is fairy general, and we implement side-effecting expressions (4.3) and

real "dynamic" statements, like pointers to procedures or dynamic method invocation with it. We model the latter with:

dynCall init p return result ≡ *DynCom* (λ*s*. *call init* (*p s*) *return result*)

3.2 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** in Isabelle/HOL. This idea goes back to Wenzel [23]. A simple state space with three 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 = *True*, N = *42*, M = *3*|). For each field there is a *selector* function of the same name, e.g. N (|B = *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.

3.3 Hoare Logics

We have defined two Hoare logic judgements, for partial correctness of the general form Γ,$\Theta\vdash$ P c Q,A and Γ,$\Theta\vdash_t$ P c Q,A for total correctness. P is the precondtion and Q and A are the postconditions for normal and abrupt termination. If we start in a state satisfying P, execution of command c will end up in a state satisfying Q in case of normal termination and in a state satisfying A in case of abrupt termination. Total correctness additionally guarantees termination of the program. Γ is the procedure environment, which maps procedure names to their bodies, 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. We have proven soundness and completeness of the Hoare logics with respect to an operational semantics [20]. But this paper will focus on the application of the logic.

The assertions P, Q and A are represented as set of states: *'s set*. This means we do not introduce a special assertion language, but can use ordinary HOL sets to describe the states.

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. The approach to split up the postcondition for normal and abrupt termination is also followed by [4,8].

The rules for the basic language constructs are standard:

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

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

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

The command *Basic f* 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 represent field assignment or memory allocation as basic commands.

To model runtime faults that may occur during expression evaluation (like division by zero), we use the guarded command. In order to prove a guarded command we have to ensure that the guard holds.

The remaining rules will be described in the following section. Most of 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.

4 Verification Environment

Our 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: $\dfrac{P\subseteq\ WP\ \ldots}{\Gamma,\Theta\vdash\ P\ c\ Q,A}$. The $\ldots$ 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 $P\subseteq WP$. The set inclusion is then transformed to an implication. Finally we 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.

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 $\{\!|\,\acute{I} \leq \acute{J}|\!\}$ gets expanded to $\{s.\ I\,s \leq \acute{J}\}$ 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 simplification of the record updates in 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. $\bigwedge M\ N.\ N = N \wedge M = M$

4.1 Loops

To verify a loop, the user annotates an invariant. For total correctness the user also supplies the variant, which in our case is a well-founded relation on the state-space, which decreases by evaluation of the loop body. Formally this is expressed by first fixing the pre-state with the rsingleton 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 than" τ according to the relation: $(s,\ \tau) \in r$. Since abrupt termination will exit the loop immediately we do not have to take any care in this case.

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

We make use of the infrastructure for well-founded recursion that is already present in Isabelle/HOL [14]. The following example calculates multiplication by iterated addition. 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 $\Gamma\vdash_t \{\!|\,\acute{M} = 0 \wedge \acute{S} = 0|\!\}$
WHILE $\acute{M} \neq a$ ***INV***$\{\!|\,\acute{S} = \acute{M} * b \wedge \acute{M} \leq a|\!\}$ ***VAR*** *MEASURE* $a - \acute{M}$
DO $\acute{S} := \acute{S} + b;\ \acute{M} := \acute{M} + 1$ ***OD***
$\{\!|\,\acute{S} = a * b|\!\}$
apply *vcg*

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

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

4.2 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. Only if the first statement terminates abruptly the second statement is executed. Thinking of exceptions the first statement forms the protected `try` part, whereas the second statement is the exception handler. Thus the precondition R for the second statement is the postcondition for abrupt termination of the first statement.

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

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

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

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 $\{\!|\ I = 10\ |\!\}$ according to the rule for *Catch* and *Skip*. 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.

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***. 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.

4.3 Expressions with Side Effects

Expressions in our language model are ordinary HOL expressions (functions over the state-space) and though do not have side effects. The trivial approach is to reduce side-effecting expressions to statements and expressions without side effects. A program transformation step introduces temporary variables to store the result of subexpressions. For example we can get rid of the increment expression in `r = m++ + n` by first saving the initial value of `m` in a temporary variable: `tmp = m; m = m + 1; r = tmp + n`. But in our state-space model this approach is somehow annoying since the temporary variables directly affect the shape of the state record. The essence of the temporary variables is to fix the value of an expression at a certain program state, so that we can later on refer to this value. Since our dynamic command *DynCom* allows to abstract over the state-space we already have the means to refer to certain program states. In contrast to Oheimb [16] we do not have to invent a special kind of postcondition that explicitely depends on the result value of an expression. Similar to the state monad in functional programming [22] we introduce the command *bind e c*, which binds the value of expression *e* (of type $'s \Rightarrow 'v$) at the current program state and feeds it into the following command *c* (of type $'v \Rightarrow ('s,'p)\ com$): *bind e c* ≡ *DynCom* ($\lambda s.\ c\ (e\ s)$). The Hoare rule for *bind* is the following:

$$\frac{P \subseteq \{s.\ s \in P'\ s\} \qquad \forall s.\ \Gamma,\Theta\vdash\ (P'\ s)\ c\ (e\ s)\ Q,A}{\Gamma,\Theta\vdash\ P\ bind\ e\ c\ Q,A}$$

The initial state is *s*. The intuitive reading of the rule is backwards in the style of the weakest precondition calculation. The postcondition we want to reach is *Q* or *A*. Since statement *c* depends on the initial state *s* via expression *e*, the intermediate assertion P' depends on *s*, too. The actual precondition *P* describes a subset of the states of $P'\ s$.

In the following example the notation $e \gg m.\ c$ is syntax for *bind e* ($\lambda m.$ *c*), whereas the second ≫ is just a syntactic variant of sequential composition to indicate the scope of the bound variable *m*.

lemma
$\Gamma \vdash \{\!|\mathit{True}|\!\}\ \acute{M} \gg m.\ \acute{M} := \acute{M} + 1 \gg \acute{R} := m + \acute{N}\ \{\!|\acute{R} = \acute{M} + \acute{N} - 1|\!\}$
apply *vcg*
1. $\bigwedge M\ N.\ \mathit{True} \Longrightarrow M + N = M + 1 + N - 1$

M and N are the initial values of the variables. So in the postcondition $\acute{R}$ gets substituted by $M + N$ and $\acute{M}$ by $M + 1$.

4.4 Procedures

To introduce a new procedure we provide the command **procedures**.

procedures *Fac* $(N|R) =$
IF $\acute{N} = 0$ ***THEN*** $\acute{R} := 1$
ELSE $\acute{R} :=$ ***CALL*** *Fac*$(\acute{N} - 1);\ \acute{R} := \acute{N} * \acute{R}$ ***FI***

Fac-spec: $\forall n.\ \Gamma \vdash \{\!|\acute{N} = n|\!\}\ \acute{R} :=$ ***PROC*** *Fac*$(\acute{N})\ \{\!|\acute{R} = \mathit{fac}\ n|\!\}$

A procedure is given by its signature followed by its body and some named specifications. The parameters in front of the pipe | are value parameters and behind the pipe are 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. Most common programming languages do not have the concept of a result parameter. But our language is a model for sequential programs rather than a "real" programming language. We represent `return e` as an assignment of `e` to the result variable. In order to capture the abrupt termination stemming from a `return` we can use the techniques described in 4.2.

To call a procedure we write $\acute{M} :=$ ***CALL*** *Fac*$(\grave{I})$. This translates to the internal form *call init ''Fac'' return result* with the proper *init*, *return* and *result* functions. Starting in an initial state s first the *init* function is applied, in order to pass the parameters. Then we execute the procedure body according to the environment Γ. Upon normal termination of the body in a state t, we first exit the procedure according to the function *return* s t and then continue execution with *result* s t. In case of an abrupt termination the final state is given by *return* s t. The function *return* passes back the global variables (and heap components) and restores the local variables of the caller, and *result* additionally assigns results to the scope 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. If the body terminates abruptly we only 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. We use the dynamic command to capture the states s and t in the definition of the procedure call with parameters:

call init p return result $\equiv$
DynCom
$(\lambda s.$ ***TRY*** *Basic init*; *Call p* ***CATCH*** *Basic* $(\mathit{return}\ s)$; ***THROW YRT***;
DynCom $(\lambda t.$ *Basic* $(\mathit{return}\ s)$; *result* s $t))$

Back to our example $´M := \textbf{\textit{CALL}}\ Fac(´I)$. The *init* function copies the actual parameter I to the formal parameter N: $init\ s = s(\!|N := I\ s|\!)$. The *return* function updates the global variables of the initial state with their values in the final state. The global variables are all grouped together in a single record field: $return\ s\ t = s(\!|globals := globals\ t|\!)$. The *result* function is not just a state update function like *return*, but yields a complete command, like the second argument in the *bind* command. This allows us to use the same technique as described for side-effecting expressions to model nested procedure calls. In our example the *result* statement is an assignment that copies the formal result parameter R to M: $result\ s\ t = Basic\ (\lambda u.\ u(\!|M := R\ t|\!))$. Here s is the initial state (before parameter passing), t the final state of the procedure body, and u the state after the *return* from the procedure. In the example the initial state s is not used. But if we assign the result of the procedure to a complex left expression and implement a left to right evaluation strategy like in C we can consider s. For example consider a pointer manipulating function call: `p->next = rev(q)`. The left value of `p->next` is the address where the result is assigned to. It is evaluated *before* the procedure call, according to the left to right evaluation strategy. We can refer to the initial state s to properly implement this semantics.

Procedure specifications are ordinary Hoare quadruples. We use the parameterless call for the specification; $´R := \textbf{\textit{PROC}}\ Fac(´N)$ is syntactic sugar for $Call\ ''Fac''$. This emphasises that the specification describes the internal behaviour of the procedure, whereas parameter passing corresponds to the procedure call. The precondition of the factorial specification fixes the current value $´N$ to the logical variable n. Universal quantification of n enables us to adapt the specification to an actual parameter. Besides providing convenient syntax, the command procedures also 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 $\Gamma\ ''Fac'' = Some\ Fac\text{-}body$, which expresses that the procedure is defined in the current environment. The purpose of these locales is to give us easy means to setup the context in which we will prove programs correct.

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

lemma includes *Fac-spec* **shows**
$\Gamma \vdash \{\!|´M = 3 \wedge ´I = 2|\!\}\ ´R := \textbf{\textit{CALL}}\ Fac\ (´M)\ \{\!|´R = 6 \wedge ´I = 2|\!\}$
apply *vcg*

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

If the verification condition generator encounters a procedure call, like $\Gamma,\Theta \vdash P\ call\ ini\ p\ ret\ res\ Q,A$, it does not look inside the procedure body, but instead

uses a specification $\forall Z.\ \Gamma,\Theta\vdash\ (P'\ Z)\ \mathit{Call}\ p\ (Q'\ Z),(A'\ Z)$ of the procedure. It adapts the specification to the actual calling context by a variant of the consequence rule, which also takes parameter and result passing into account. In the factorial example n plays the role of the auxiliary variable Z. It transports state information from the pre- to the postcondition. A detailed discussion of consequence rules and auxiliary variables can be found in [9,13].

$$\frac{\begin{array}{c}P \subseteq \{s.\ \exists Z.\ \mathit{ini}\ s \in P'\ Z \wedge (\forall t{\in}Q'\ Z.\ \mathit{ret}\ s\ t \in R\ s\ t) \wedge (\forall t{\in}A'\ Z.\ \mathit{ret}\ s\ t \in A)\} \\ \forall s\ t.\ \Gamma,\Theta\vdash\ (R\ s\ t)\ \mathit{res}\ s\ t\ Q,A \qquad \forall Z.\ \Gamma,\Theta\vdash\ (P'\ Z)\ \mathit{Call}\ p\ (Q'\ Z),(A'\ Z)\end{array}}{\Gamma,\Theta\vdash\ P\ \mathit{call}\ \mathit{ini}\ p\ \mathit{ret}\ \mathit{res}\ Q,A}$$

The idea of this rule is to adapt the specification of *Call p* to *call ini p ret res*. Figure 1 shows the sequence of intermediate states for normal termination. We

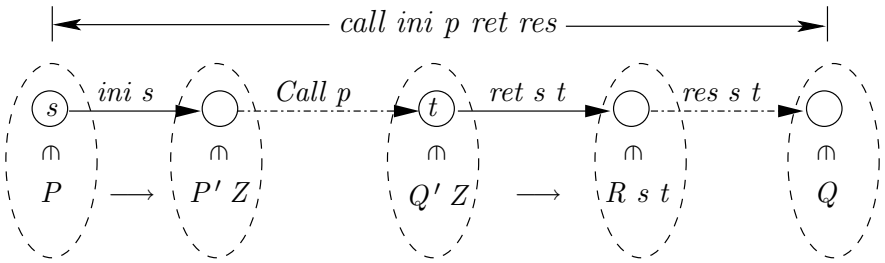

Fig. 1. Procedure call and specification

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: $\mathit{ini}\ s \in P'\ Z$. Let t be the state immediately after execution of the procedure body, before returning to the caller and passing results. We know from the specification that the postcondition will hold: $t \in Q'\ Z$. From this we have to conclude that leaving the procedure according to the function *ret* will lead to a state in $R\ s\ t$. From this state execution of *res s t* ends up in a state in Q. For abrupt termination the analogous idea applies, but without the intermediate assertion $R\ s\ t$, since execution of *res s t* is skipped. Simplifying the record updates and selections of the side-condition yields the natural proof obligation we have seen before.

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

$$\frac{\begin{array}{c}P \subseteq \{s.\ \exists Z.\ \mathit{ini}\ s \in P'\ s\ Z \wedge \\ \qquad (\forall t \in Q'\ s\ Z.\ \mathit{ret}\ s\ t \in R\ s\ t) \wedge (\forall t \in A'\ s\ Z.\ \mathit{ret}\ s\ t \in A)\} \\ \forall s\ t.\ \Gamma,\Theta\vdash\ (R\ s\ t)\ \mathit{res}\ s\ t\ Q,A \\ \forall s{\in}P.\ \forall Z.\ \Gamma,\Theta\vdash\ (P'\ s\ Z)\ \mathit{Call}\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)\end{array}}{\Gamma,\Theta\vdash\ P\ \mathit{dynCall}\ \mathit{ini}\ p\ \mathit{ret}\ \mathit{res}\ Q,A}$$

Procedure Implementation — Partial Correctness 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**
$\forall\, n.\ \Gamma \vdash \{\!|\, \acute{N} = n \,|\!\}\ \acute{R} := \textbf{\textit{PROC}}\ \mathit{Fac}(\acute{N})\ \{\!|\, \acute{R} = \mathit{fac}\ n \,|\!\}$
apply (*hoare-rule ProcRec1*)

$$\begin{array}{l} 1.\ \forall\, n.\ \Gamma, (\bigcup n\ \{(\{\!|\, \acute{N} = n \,|\!\},\ \acute{R} := \textbf{\textit{PROC}}\ \mathit{Fac}(\acute{N}),\ \{\!|\, \acute{R} = \mathit{fac}\ n \,|\!\},\ \{\})\}) \\ \qquad \vdash \{\!|\, \acute{N} = n \,|\!\} \\ \qquad\quad \textbf{\textit{IF}}\ \acute{N} = 0\ \textbf{\textit{THEN}}\ \acute{R} := 1 \\ \qquad\quad \textbf{\textit{ELSE CALL}}\ \mathit{Fac}(\acute{N} - 1);\ \acute{R} := \acute{N} * \acute{R}\ \textbf{\textit{FI}} \\ \qquad\quad \{\!|\, \acute{R} = \mathit{fac}\ n \,|\!\} \end{array}$$

apply *vcg*

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

The rule *ProcRec1* is a specialised version of the general rule for recursion, tailored for one recursive procedure. The method *hoare-rule* applies a single rule and solves canonical side-conditions. Moreover it expands the procedure body.

Let us now have a look at the general recursion rule. The Hoare logic can deal with (mutually) recursive procedures. We prove that the procedure bodies respect their specification, under the assumption that recursive calls to the procedures will meet their specifications. 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 $\mathcal{P}$ of mutually recursive procedures we enrich the context by all the procedure specifications, while we prove their bodies.

$$\frac{\begin{array}{c} \Theta' = \Theta \cup (\bigcup_{p \in \mathcal{P}} \bigcup_{Z} \{(P\ p\ Z,\ \mathit{Call}\ p,\ Q\ p\ Z,\ A\ p\ Z)\}) \\ \forall\, p \in \mathcal{P}.\ \forall\, Z.\ \Gamma, \Theta' \vdash (P\ p\ Z)\ \mathit{the}\ (\Gamma\ p)\ (Q\ p\ Z), (A\ p\ Z) \qquad \mathcal{P} \subseteq \mathit{dom}\ \Gamma \end{array}}{\forall\, p \in \mathcal{P}.\ \forall\, Z.\ \Gamma, \Theta \vdash (P\ p\ Z)\ \mathit{Call}\ p\ (Q\ p\ Z), (A\ p\ Z)}$$

Since we deal with the set $\mathcal{P}$ of procedures we also have to give the pre- and postconditions for all these procedures. We use the functions P, Q and A, 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 $s \in P \longrightarrow \mathit{init}\ s \in P'\ Z$ holds. Since we have the freedom there, we now have to prove the procedure bodies for all possible Z. Finally, with $\mathcal{P} \subseteq \mathit{dom}\ \Gamma$, we make sure that the calculation will not get stuck.

Procedure Implementation — Total Correctness 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 $\lambda(s,p).\ {}^{s}N$. 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 ${}^{s}N$ 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**
$\forall\, n.\ \Gamma\vdash_t \{\!|\acute{N} = n|\!\}\ \acute{R} := \textbf{\textit{PROC}}\ Fac(\acute{N})\ \{\!|\acute{R} = fac\ n|\!\}$
apply (*hoare-rule ProcRec1*$_t$ [**where** r=*measure* $(\lambda(s,p).\ {}^{s}N)$])

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

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

apply *vcg*

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

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

In contrast to partial correctness we 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. 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.

$$
\frac{\begin{array}{c}
\Theta' = \lambda\tau\ p.\ \Theta \cup (\bigcup_{q\in\mathcal{P}} \bigcup_{Z} \{(P\ q\ Z \cap \{s.\ ((s,q),\tau,p) \in r\}, Call\ q, Q\ q\ Z, A\ q\ Z)\})\\
\forall\, p{\in}\mathcal{P}.\ \forall\,\tau\ Z.\ \Gamma,\Theta'\ \tau\ p \vdash_t (\{\tau\} \cap P\ p\ Z)\ the\ (\Gamma\ p)\ (Q\ p\ Z),(A\ p\ Z)\\
wf\ r \qquad \mathcal{P} \subseteq dom\ \Gamma
\end{array}}{\forall\, p{\in}\mathcal{P}.\ \forall\, Z.\ \Gamma,\Theta \vdash_t (P\ p\ Z)\ Call\ p\ (Q\ p\ Z),(A\ p\ Z)}
$$

4.5 Heap

The heap can contain structured values like `structs` in C or records in Pascal. Our model of the heap follows Burstall [2]. We have one heap variable f of type *ref* $\Rightarrow$ *value* for each component f of type *value* of the `struct`. References *ref* are isomorphic to the natural numbers and contain *Null*.

A typical structure to represent a linked list in the heap is `struct list {int cont; list *next}`. 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 *heap* =
  next::*ref* ⇒ *ref*
  cont::*ref* ⇒ *int*

record *state* =
  globals::*heap*
  p::*ref*
  q::*ref*
  r::*ref*

We follow the approach of [10], and abstract the pointer structure in the heap to HOL lists of references. Then we can specify further properties on the level of HOL lists, rather than on the heap:

List x h [] = (*x* = *Null*)
List x h (*p* # *ps*) = (*x* = *p* ∧ *x* ≠ *Null* ∧ *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*. With a generalised predicate that describes a path in the heap, Mehta and Nipkow [11] show how this idea can canonically be extended to cyclic lists.

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*→*f* mimics the field selection syntax of C and is translated to ordinary function application for field lookup and function update for field assignment.

procedures *Rev*(*p*|*q*) =
 ´*q* := *Null*;
 WHILE ´*p* ≠ *Null*
 DO ´*r* := ´*p*; ´*p* := ´*p*→´*next*; ´*r*→´*next* := ´*q*; ´*q* := ´*r* ***OD***

Rev-spec:
 ∀ σ *Ps*. Γ⊢ ⦃σ. *List* ´*p* ´*next Ps*⦄ ´*q* := ***PROC*** *Rev*(´*p*)
  ⦃*List* ´*q* ´*next* (*rev Ps*) ∧ (∀ *p*. *p* ∉ *set Ps* ⟶ (´*next p* = ^σ*next p*))⦄
Rev-modifies:
 ∀ σ. Γ⊢{σ} ´*q* := ***PROC*** *Rev*(´*p*) {*t*. *t may-only-modify-globals* σ *in* [*next*]}

We give 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). Fixing a state is part of the assertion syntax: ⦃σ. ...⦄ translates to {*s*. *s*=σ ...} and ^σ*next* to *next* σ. The second specification 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 remain unchanged. Thus the main specification can focus on the relevant parts of the state-space. The assertion *t may-only-modify-globals* σ *in* [*next*] abbreviates the following relation between the final state *t* and the initial state σ: ∃ *next*. *globals t* = (*globals* σ)(|*next*:=*next*|). 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 automatically 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*

1. $\bigwedge next\ cont\ p.\ List\ p\ next\ Ps \Longrightarrow$
$\quad \forall\, nexta\ q.$
$\quad\quad List\ q\ nexta\ (rev\ Ps) \wedge (\forall\, p.\ p \notin set\ Ps \longrightarrow nexta\ p = next\ p) \longrightarrow$
$\quad\quad cont = cont \wedge List\ q\ nexta\ (rev\ Ps)$

The impact of the modifies-clause shows up in the verification condition. The *cont*-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.

5 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 side-effecting expressions, 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. Using 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, without having to introduce a new logic like separation logic [19]. 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 seamless fits into the infrastructure of Isabelle/HOL.

This work is part of the Verisoft project, a long-term research project aiming at the pervasive verification of computer systems (hard- and software). We translate a subset of C to the verification environment and have started to verify parts of an operating system, a compiler and an email client. We also verify the translation into the verification environment. Moreover we validated the feasibility of our approach by verifying algorithms for binary decision diagrams, involving a high degree of side effects due to sharing in the pointer structure [17]. Applying the verification condition generator to the annotated programs results in quite sizable proof obligations. But since they closely resemble the control flow the connection to the input program is not lost. To prove them, we used the structured proof language of Isar [24] that allows us to focus and keep track of the various different aspects, so that we can conduct the proof in a sensible order. Moreover it turned out that the Isar proofs are quite robust with regard to the iterative adaptation of the invariants resulting from failed proof attempts. The already established lines of reasoning remained stable, while adding new aspects to, or strengthening parts of the invariant. Altogether we gained confidence that our approach is practically useful.

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 LNCS, pages 34–50. Springer, 2004.
2. R. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.
3. J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.
4. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

5. 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 *LNCS*, pages 171–188, Canberra, Australia, 1998. Springer.
6. 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.
7. M. Huisman. *Java program verification in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2000.
8. B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.
9. T. Kleymann. Hoare Logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.
10. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.
11. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 2005. To appear.
12. 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.
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 *LNCS*. Springer, 2002.
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. V. Ortner. Verification of BDD Algorithms. Master's thesis, Technische Universität München, 2004. http://www.veronika.langlotz.info/.
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. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.
20. N. Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In G. Klein, editor, *Proc. NICTA Workshop on OS Verification 2004*, 2004. ID: 0401005T-1, http://www4.in.tum.de/~schirmer.
21. J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanizing some advanced refinement concepts. *Formal Methods in System Design*, 3:49–81, 1993.
22. P. Wadler. The essence of functional programming. In *Proc. 19th ACM Symp. Principles of Programming Languages*, 1992.
23. M. Wenzel. Miscellaneous Isabelle/Isar examples for higher order logic. Isabelle/Isar proof document, 2001.
24. M. Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.