Hoare Logic with Explicit Contexts

Michael Franssen
Eindhoven University of Technology
Dept. Computing Science
Den Dolech 2, 5612 AZ Eindhoven
The Netherlands
e-mail: m.franssen@tue.nl

Abstract

In this paper, we combine a Hoare logic with a typed λ-calculus to create a reliable tool for deriving correct programs. In this combined system proofs become part of the programs. The main advantages of our approach are (1) that the Hoare logic conforms to the de Bruijn criterion and hence, can be implemented in a reliable way; (2) that the Hoare logic and the typed λ-calculus co-exist at the same level and therefore programs do not have to be encoded within a theorem prover and (3) scopes of variables are dealt with explicitly and hence, the specification language is strictly separated from the programming language.

1 Introduction

Most papers about Hoare logics are about soundness and completeness of the Hoare rules for certain programming constructs. Transformation of specifications of these programs is then done in meta-theory, using the validity of certain logical formulas. The formal systems needed to prove these validities are usually left implicit.

When reliably implementing a Hoare logic, the logic to prove specification transformations has to be made explicit. Often this is done by embedding the Hoare logic in a theorem prover and using the theorem prover's deductive system to prove the logical transformations. For instance, Homeier et all used a deep embedding in [HM95] to implement a verification condition generator. Von Wright used a shallow embedding in [vW94] to create a tool to derive correct programs. In both approaches, programs need to be encoded in the theorem prover used.

In this paper, we present a Hoare logic combined with a typed λ-calculus that conforms to the de Bruijn criterion (see [OG02, BB]). That is, derived programs can be checked for correctness once they are completed and, if desired, the entire correctness proof of the program can be communicated to other systems that implement the same logic. Hence, the Hoare logic can be implemented in a reliable way. Since the Hoare logic is not embedded within a theorem prover, it is not necessary to encode programs in a theorem prover's language. Therefore, it is easier to present programmers a user-interface aimed at programming instead of theorem proving. Also, since most typed λ-calculi can be used for the proposed combination, we can restrict the specification language to first-order logic, which is not possible when the Hoare logic is embedded in a theorem prover. By splitting the λ-calculus' context in three parts, we can also strictly separate the programming language from the specification language.

In section 2 we define the logic and the language we want to support. In section 3 we discuss the construction of the Hoare logic and its most important properties and in section 4 we will briefly discuss the results.

2 Preliminaries

Throughout this paper, we will use a number of well-known concepts and definitions related to logic and Hoare logic. In order to avoid confusion, we will briefly describe them in this section.

2.1 Logic

To specify programs, we will use predicates of first-order logic. We assume that this logic is multi-sorted; i.e. that variables have an associated type symbol. These types can be chosen from the set *Set* of type symbols, which is a parameter of the logic. Other parameters of the logic are $\mathcal{F}$, the set of function symbols and *Prop*, the set of predicate symbols. The function and predicate symbols have associated type-tuples that state the required types of their arguments and, in case of function-symbols, the result type.

The semantics of this first-order logic are, as usual, defined by an interpretation of variables, set-, function- and predicate-symbols. A model is then defined by consistently lifting the domain of the interpretation to the set of formulas of the logic. If a formula P is valid in all models (independent of the interpretation), we denote this as $\models P$.

2.2 The Language

The focus of this paper is not the construction of a Hoare logic for real-life programming constructs. Hence, we do not need to consider an extensive language.

Instead, we describe only a simple language called *While* (see [NN92]), that has the following abstract syntax:

$$\begin{aligned} W ::= &\ \mathbf{skip} \mid V := T \mid \mathbf{if}\ T\ \mathbf{then}\ W\ \mathbf{else}\ W\ \mathbf{fi} \mid \mathbf{while}\ T\ \mathbf{do}\ W\ \mathbf{od} \\ &\mid W;W \mid |[\mathbf{var}\ V : Set\ \bullet\ W]| \end{aligned}$$

In this syntax T's denote terms of the logic. Hence, every function in the logic can occur in programs of *While*. If one wants to restrict the functions available in *While*, one needs to define a T', in which only a subset of $\mathcal{F}$ is allowed. $\mathcal{F}$ must be a superset of the functions in *While* to avoid problems with the expressibility requirement of the Hoare logic (see below).

The guards of **if** and **while** statements must always have associated the set-symbol **bool** which we assume exists. Furthermore, we assume that the constants **true** and **false** exist in $\mathcal{F}$ (constants are functions with arity 0). The interpretations of **bool**, **true** and **false** are $\{True, False\}$, *True* and *False* respectively.

The last syntactic construct allows the use of local variables, that only have a limited scope (namely in the program occurring within their brackets). Usually, the scope rules are hard to define within a Hoare logic, but as we will see, not in the Hoare logic we will introduce in this paper.

2.3 Hoare Logic

A Hoare logic is a derivation system which is used to derive Hoare triples. A Hoare triple is a triple consisting of two predicates over states, say P and Q and a program S. It is denoted as

$$\{P\}S\{Q\}$$

A Hoare triple is valid if S only terminates in states satisfying Q when it is started in a state satisfying P. Note that it is not claimed that S does terminate at all. This interpretation is called *partial* correctness. There is also an interpretation called *total* correctness, which claims that S terminates, but this is not considered in this paper.

In this case, P is called the *precondition* of S and Q is called its *postcondition*.

The idea of a Hoare logic for a language like *While* is, off course, that a Hoare triple $\{P\}S\{Q\}$ is only derivable if it is valid; i.e. the Hoare logic is sound. A Hoare logic is complete if every valid Hoare triple can be derived. Creating a sound an complete Hoare logic is involved and will not be discussed in this paper.

The Hoare logic for the language *While* is fairly straightforward and will not be given explicitly. The only hard rule is the rule for local variables and this will be defined after we introduced explicit contexts, in which case its definition is simple.

The Hoare rule for assignment, $\{P[x := e]\}x := e\{P\}$, however, has some consequences for the logic. Since through assignment, every expression of the programming language can occur in a precondition of a Hoare triple, the set of functions of the specification language must be a superset of the set of functions of the programming language. This is taken care for by our definition of the abstract syntax of *While*, which is based on the parameters of the logic.

The remainder of this paper will focus on a way to implement the Hoare logic in such a way that one obtains a reliable tool.

3 Hoare Logic with Contexts

If a Hoare logic is embedded in a theorem prover, this theorem prover must support higher order logic (see [Fra00a]). Hence, automatic proof search becomes very difficult. However, in an environment for program derivation, one will often encounter proof obligations that are very simple. Hence, in practice, the user will have a need for automatic support to solve these proof obligations.

If we want first-order logic, we need an approach to implement the Hoare logic that does not require embedding. Hence, we do not want the programming logic to be expressed within the proof logic.

In this section, we will link a Hoare logic to a theorem prover without embedding it. Instead, we will create derivation rules for programs, that depend on derivations in a typed lambda-calculus. This allows us to restrict the theorem prover to first-order logic and hence, the tool can support automated theorem proving (see [Fra97]). Note that this approach to combine a Hoare logic with a theorem prover also works for higher-order logic, but then automated theorem proving becomes (much) more difficult.

The only rule in the Hoare logic of *While* that refers to theorems, is the rule of consequence:

$$\frac{\models P' \Rightarrow P \qquad \{P\}S\{Q\} \qquad \models Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

Stated this way, one assumes $\models$ to denote semantic validity of formulas in a logic, which is left implicit. This logic, however, must be powerful enough to deal with all possible expressions allowed in the language. This is usually referred to as the expressibility requirement.

Since, in our definition of *While*, the expressions in the programs are those defined in the logic, we automatically fulfill the expressibility requirement. One can use a typed lambda-calculus to construct the required proofs. The rule of consequence then becomes:

$$\frac{\Gamma_{\mathcal{L}} \vdash p{:}P' \Rightarrow P \qquad \{P\}S\{Q\} \qquad \Gamma_{\mathcal{L}} \vdash q{:}Q \Rightarrow Q'}{\{P'\}S\{Q'\}},$$

where $\Gamma_{\mathcal{L}}$ is the context for the typed λ-calculus defining the sets *Set* and $\mathcal{F}$ used to define *While*.

Any typed λ-calculus with enough expressive power can be used. A λ-calculus is suitable if it supports at least the following:

- types to denote the set of all propositions, say $*_p$, and the set of all sets, say $*_s$.
- a falsum symbol, say $\bot$: $*_p$ and the logical connectives $\wedge$ and $\Rightarrow$ for conjunction and implication respectively.
- a set-type for booleans, say **bool** : $*_s$, with constants **true** and **false** as its only elements. These are used as expression types for guards in **if** and **while** statements.
- Leibniz equality for all terms of any set-type $U : *_s$, denoted as $=$.

Also, we assume that type-inference, and hence type checking, of the λ-calculus is decidable. That is, given a lambda-term representing a proof, it must be possible to compute the type of this term and hence, the corresponding theorem it proves.

Since one of our goals was a high degree of automated theorem proving, we choose for the typed lambda calculus $\lambda P-$, a pure type system described in [LF01], which exactly corresponds to first order logic. The construction of $\Gamma_{\mathcal{L}}$ is also described in more detail there. Proofs can then automatically be constructed by a tableau based theorem prover and then translated into a λ-term of $\lambda P-$ as described in [Fra00b]. When using λ-calculi for higher-order logic, other methods for automated theorem proving should be used.

The advantages of this approach are:

- Programs do not have to be constructed using encodings. Instead, the tool can provide a specialized interface for the programming part of the tool.
- Boolean expressions allowed in programs are defined within the logic, but are separated from the specification language (e.g. one cannot use $(\forall p : nat.\ \exists q : nat.\ q > n \wedge prime(q))$ as a guard in an if- or a while-construction , since it is a propositional formula, not a boolean). Having booleans as ordinary types in the programming language also resembles real-world programming languages more closely.
- The system's logic can be restricted to first-order logic and hence, meaningful automatic proof search is possible.

Even though this "new" rule of consequence is sufficient to implement a sound and relatively complete Hoare logic, it still suffers from several drawbacks:

- Programs cannot be checked once they are completed, since the proofs constructed during programming have not been stored. Hence, there is no term representing a correctness proof of the entire program.
- There is no way to prevent the usage of specification functions in programs. For instance, $x := \mathit{fibonacci}(N)$ is a valid, although undesirable, program.

In the following, we show how these drawbacks can be eliminated.

Since proofs are syntactically represented in λ-calculus, we can easily incorporate those proofs by extending the program-syntax and change the rule of consequence to something like:

$$\frac{\Gamma_{\mathcal{L}} \vdash p{:}P' \Rightarrow P \qquad \{P\}S\{Q\} \qquad \Gamma_{\mathcal{L}} \vdash q{:}Q \Rightarrow Q'}{\{P'\}\mathbf{cons}\ p\ S\ q\{Q'\}}$$

Since program S now has become embedded in *cons* $p\ S\ q$, it is harder to change, since this requires changing both p and q along with it. If one regards the change of P to P' as a re-formulation of a state-property, one can regard application of the rule of consequence to be an application of the theorem $P' \Rightarrow P$ to a state in which P holds. The same can be said about $Q \Rightarrow Q'$. We denote this application of a theorem by the program **fake** r, which has the same semantics as **skip**, since the state does not change. The rule of consequence can now be replaced by the simpler rule:

$$\frac{\Gamma_{\mathcal{L}} \vdash r{:}P \Rightarrow Q}{\{P\}\mathbf{fake}\ r\{Q\}}$$

The original rule of consequence can now be derived as follows: From $\Gamma_{\mathcal{L}} \vdash p{:}P' \Rightarrow P$ we derive $\{P'\}\mathbf{fake}\ p\{P\}$ and from $\Gamma_{\mathcal{L}} \vdash q{:}Q \Rightarrow Q'$ we derive $\{Q\}\mathbf{fake}\ q\{Q'\}$. Since $\{P\}S\{Q\}$, we use the composition rule to conclude

$$\{P'\}\mathbf{fake}\ p; S; \mathbf{fake}\ q\{Q'\}.$$

The fake-statement has the advantage that it allows separate treatment of pre- and postconditions. Also, all proofs are now stored in separate statements, not having other programs as sub-programs.

Using *While*, extended with fake-statements, yields programs that can automatically be checked once they are completed. This is nearly a trivial matter, since every statement can only be derived by a single rule, including the fake-statement.

However, since proofs have now become part of the program, our Hoare logic is different from regular Hoare logics: when the specification of the program is changed, the program itself also has to be changed, even if its semantics remain the same. For instance, given the propositions P and Q with $P \Rightarrow Q$, the program $\{Q\}\mathbf{skip}\{Q\}$ is correct; whereas $\{P\}\mathbf{skip}\{Q\}$ is not (it should read $\{P\}\mathbf{fake}\ p; \mathbf{skip}\{Q\}$ or $\{P\}\mathbf{skip}; \mathbf{fake}\ p\{Q\}$ instead, where $\Gamma_{\mathcal{L}} \vdash p{:}P \Rightarrow Q$).

Since our tool only needs to check if programs meet the specification for which they were derived, this difference is of no concern.

Having proofs explicitly stated in programs seems unnatural. However, this is not necessarily true, since one can consider programs to be proofs of the satisfiability of their specification. From this point of view, programs are the λ-terms of a Hoare logic. Since, in our tool, the Hoare logic is linked to a typed lambda-calculus, this view is also more consistent with the formalism used for proofs. Therefore, we will introduce a different notation for programs and their specifications.

Definition 1 (Program Specification)
Let P and Q be a pre- and postcondition respectively. Then $P \triangleright Q$ is a program specification. The Hoare triple $\{P\}S\{Q\}$ can now be denoted as $S : P \triangleright Q$, stating that program S satisfies specification $P \triangleright Q$.

The reason that programs like $x := \mathit{fibonacci}(N)$ are allowed, is that program expressions are based on exactly the same logic as specifications. Therefore, all function-symbols and expressions available in the specification are also available in programs.

However, in a typed lambda-calculus, like $\lambda P-$, all function symbols are explicitly declared within the context $\Gamma_{\mathcal{L}}$. If we add contexts to the Hoare logic, we can use a larger context for specifications than for programs. The context used to construct expressions for use in programs must always be a subset of the context used to construct specifications (see section 2.3). Conversely, a function needed to specify a programming problem (e.g. the Fibonacci numbers) does not need to be available to the program.

Therefore, we split contexts for Hoare triples into three parts:

- The first part is accessible from programs as well as specifications. Programs can use this context, but not alter its variables. Typically, it contains all function symbols, constants and definitions that are default to the language (e.g. the type **bool** of booleans). This context is referred to as the *language context*.
- The second part contains (local) program variables that can be altered by programs. The second context can depend on the first context, e.g. a program variable could have pre-defined type **bool**. Typically, this context is used to store constants and variables needed to specify a programming problem, for instance the variable x from the postcondition $x = \mathit{fibonacci}(N)$. This context is called the *program context*.
- The third context contains all other logical elements needed to specify programs, like abstract data types or auxiliary functions. This context cannot be used by programs, only by specifications. It may depend on both previous contexts, since a postcondition may specify that some language expression must be equal to an auxiliary function (e.g. $x = \mathit{fibonacci}(N)$, where *fibonacci* is an element of the third context and hence, cannot be used by the program). This context is referred to as the *specification context*.

The order of these contexts is quite natural: there would be little need for functions and variables accessible only from programs and not from specifications.

Now, both drawbacks mentioned before are eliminated. The fake-statements enable us to check the correctness of a program once it is completed. The use of a triple of contexts allows us to have functions available in the specification languages, that are not available in expressions of the programming language. The formal definitions of this Hoare logic with explicit contexts and the proof of these claims is given below.

We will add triples of contexts to the Hoare logic in the following manner:

Definition 2 (Hoare Contexts)
Let Γ_1, Γ_2 and Γ_3 be contexts of a typed lambda-calculus, then $\langle\Gamma_1, \Gamma_2, \Gamma_3\rangle$ is a Hoare context.

Note the following important differences between $\langle\Gamma_1, \Gamma_2, \Gamma_3\rangle$ and $\Gamma_1, \Gamma_2, \Gamma_3$: The comma-separated contexts Γ_1, Γ_2 and $\Gamma_1, \Gamma_2, \Gamma_3$ denote concatenations of λ-contexts and hence, are themselves contexts of a λ-calculus. The bracketed triple $\langle\Gamma_1, \Gamma_2, \Gamma_3\rangle$ denotes a triple of λ-contexts and hence is a Hoare context.

Definition 3 (Hoare logic with explicit contexts)
Let $\vdash_\lambda$ be the type-judgment relation of a typed lambda-calculus. Then we define $\vdash_H$ to be a Hoare derivation system defined by the rules shown in figure 1.

We briefly comment on the use of the context for each rule:

Spec This rule formalizes the definition of a specification. The rule expresses the well-formednes of a specification. It is added because we regard specifications as a sort of type.

Skip In order for **skip** : $P \triangleright P$ to hold, $P \triangleright P$ must be a well-formed specification.

Assign For $x := e : P[x := e] \triangleright P$ to hold, $P[x := e] \triangleright P$ must be a specification and e must be an expression of the same type as variable x. This type must represent a set. Moreover, x must occur in the program context and e may only depend on the language and program context. This is expressed by the three premisses.

$$[Spec] \quad \frac{\Gamma_1, \Gamma_2, \Gamma_3 \vdash P{:}*_p \qquad \Gamma_1, \Gamma_2, \Gamma_3 \vdash Q{:}*_p}{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash P \triangleright Q{:}\mathbf{Spec}}$$

$$[Skip] \quad \frac{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash P \triangleright P{:}\mathbf{Spec}}{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash \mathbf{skip}{:}P \triangleright P}$$

$$[Assign] \quad \frac{\begin{array}{l}\Gamma_1, (\Delta_1, x : U, \Delta_2) \vdash U{:}*_s \\ \Gamma_1, (\Delta_1, x : U, \Delta_2) \vdash e{:}U \\ \langle \Gamma_1, (\Delta_1, x : U, \Delta_2), \Gamma_3 \rangle \vdash P[x := e] \triangleright P{:}\mathbf{Spec}\end{array}}{\langle \Gamma_1, (\Delta_1, x : U, \Delta_2), \Gamma_3 \rangle \vdash x := e{:}P[x := e] \triangleright P}$$

$$[If] \quad \frac{\begin{array}{l}\Gamma_1, \Gamma_2 \vdash e{:}\mathbf{bool} \\ \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash S_1{:}P \wedge e = \mathbf{true} \triangleright Q \\ \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash S_2{:}P \wedge e = \mathbf{false} \triangleright Q\end{array}}{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}{:}P \triangleright Q}$$

$$[While] \quad \frac{\begin{array}{l}\Gamma_1, \Gamma_2 \vdash e{:}\mathbf{bool} \\ \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash S{:}P \wedge e = \mathbf{true} \triangleright P\end{array}}{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash \mathbf{while}\ e\ \mathbf{do}\ S\ \mathbf{od}{:}P \triangleright P \wedge e = \mathbf{false}}$$

$$[Comp] \quad \frac{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash S_1{:}P \triangleright Q \qquad \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash S_2{:}Q \triangleright R}{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash S_1; S_2{:}P \triangleright R}$$

$$[Fake] \quad \frac{\Gamma_1, \Gamma_2, \Gamma_3 \vdash p{:}P \Rightarrow Q}{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash \mathbf{fake}\ p{:}P \triangleright Q}$$

Figure 1: A Hoare logic with explicit contexts.

If The first premise claims that e is a boolean expression that can be derived from the language and program context. The other premises are direct translations from the original Hoare logic.

While Similar to *If*.

Comp This rule is a direct translation of the original rule from the Hoare logic.

Fake The *Fake* rule was explained before. Note that since $P \Rightarrow Q$ is a proposition, so are P and Q. Hence, the program is correctly specified.

By using explicit contexts, we can now also easily introduce a formal rule to allow the introduction of local variables (variables with limited scope) into the program's state space:

$$[Block] \quad \frac{\begin{array}{ll}\Gamma_1, \Gamma_2 \vdash U{:}*_s & \\ \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash P \triangleright Q{:}\mathbf{Spec} & x \text{ is } \Gamma_1, \Gamma_2, \Gamma_3\text{-fresh} \\ \langle \Gamma_1, (\Gamma_2, x : U), \Gamma_3 \rangle \vdash S{:}P \triangleright Q & \end{array}}{\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle \vdash |[\mathbf{var}\ x : U \ \bullet\ S\ ]|{:}P \triangleright Q}$$

The Hoare logic now has a notation and a set of derivation rules similar to those of a pure type system. Through the following theorems, we will prove that the Hoare logic also has some important meta-theoretical properties in common with typed lambda-calculi: programs can be checked once they are completed. This also enables the communication of those programs: most importantly, it conforms to the de Bruijn criterion: programs which include **fake**-statement are self-contained, including all correctness proofs required.

Definition 4
Given a Hoare logic with explicit contexts as defined above and a context $\Gamma \equiv \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$, *we define the following questions:*

Backward inference: Given a program S *and a postcondition* Q, *does a predicate* P *exist, such that* $\Gamma \vdash S{:}P \triangleright Q$*?*

Forward inference: Given a program S *and a precondition* P, *does a predicate* Q, *such that* $\Gamma \vdash S{:}P \triangleright Q$*?*

Specification inference: Given a program S, *do predicates* P *and* Q *exist, such that* $\Gamma \vdash S{:}P \triangleright Q$*?*

Program checking: Given precondition P, *postcondition* Q *and program* S, *does* $\Gamma \vdash S{:}P \triangleright Q$ *hold?*

Backward inference and forward inference are used to recompute a correct Hoare triple if the user wants to change the specification after a part of the program has already been derived. Specification inference can be used to recompute the specification of a derived program if for some reason the original specification is no longer available. Program checking is used to verify the correctness of the derived program once it is complete. This last action is the most important one, since it allows us to detect errors made by the tool during program derivation. It is, in fact, nothing less than type checking for imperative programs.

Theorem 5 (Backward inference is decidable)
Let Γ *be a Hoare context, let* S *be a program and let* Q *be a postcondition. One can compute a unique* P *such that* $\Gamma \vdash S{:}P \triangleright Q$ *if it exists. If such a* P *does not exist, the computation will abort with an error.*

Proof: By induction on the structure of S (using type-inference for proofs contained within **fake**-statements). $\square$

Corollary 6 (Program checking is decidable)
From theorem 5 it immediately follows that given a Hoare context Γ, *a program* S *and a specification* $P \triangleright Q$, *it is decidable whether or not* $\Gamma \vdash S{:}P \triangleright Q$.

In the following theorems we will sometimes need the set of postconditions that correspond to a certain precondition. Since programs may contain assignments, we need to invert the substitutions performed by these assignments. Therefore we define inverse substitutions.

Definition 7 (Inverse substitution)
Let ϕ *be a mapping from variables to expressions (i.e. a substitution). We define* ϕ^{-1} *as the function from propositions to sets of propositions by:*

$$\phi^{-1}(P) = \{Q \mid P \equiv \phi(Q)\}$$

Without further proof we claim that ϕ^{-1} is computable and that $\phi^{-1}(Q)$ is finite for every Q, provided that $\phi(x) \neq x$ only for a finite number of known variables x.

For example, consider the substitution $\phi \equiv \{x \mapsto 30\}$. Then $\phi^{-1}(r = 30+30) = \{r = 30+30, r = x+30, r = 30+x, r = x+x\}$ and $\phi^{-1}(r = x+1) = \emptyset$. For $\psi \equiv \{x \mapsto x+1\}$, we get $\psi^{-1}(x+1 < 8) = \{x < 8\}$ and $\psi^{-1}(x+3 < 6) = \emptyset$. Note that it is important to know the domain of the substitution and that this domain is finite if we want to compute the inverse substitution.

Theorem 8 (Forward inference is decidable)
Given a Hoare context Γ*, a program* S *and a precondition* P*, one can compute a finite set* Q^s *of propositions, such that*

$$Q \in Q^s \text{ if and only if } \Gamma \vdash S{:}P \triangleright Q.$$

From this it immediately follows that forward inference is computable.

Proof: By induction on the structure of S (using type-inference for proofs contained within **fake**-statements). □

To compute answers for specification inference, we consider two cases: either a program contains one or more fake-statements or it does not. In the first case, specification inference boils down to finding the fake statement; computing its specification based on type inference of the lambda calculus; and using the result to compute the complete program's specification as with forward and backward inference. In the second case, determining a specification becomes much harder, since there is no predicate to start out with. Therefore, we introduce propositional patterns, that represent predicates consisting of a left associative series of conjuncts. The leftmost predicate in such a pattern is a meta-variable *alpha* and the other conjuncts represent parts of the specification that correspond to guards of the program's statements. Instead of computing the specification directly, we the compute a relationship between the pre- and postcondition, along with a set of constraints. If this succeeds, we can always compute at least one pre-and postcondition that satisfy the constraints and conform to the relationship. Also, it is then easy to decide whether a geven predicate would suffice.

Lemma 9 (Specification inference with fake)
Let Γ *be a Hoare context and* S *be a program containing at least one* **fake***-statement. It is possible to compute a unique proposition* P *and a set of propositions* Q^s*, such that*

$$Q \in Q^s \text{ if and only if } \Gamma \vdash S{:}P \triangleright Q.$$

Proof: By induction on the structure of S (using type-inference for proofs contained within this **fake**-statements and using these to construct P and the set Q^s as within forward and backward inference). □

What remains is to compute specifications for *While* programs without **fake** statements. In all previous inference theorems, the structure of the program specification was derived from the types of the proofs stored in **fake**-statements. Since this is no longer possible, we define the relation between pre- and postcondition in such a way that we can search for specifications with this relation.

Definition 10 (Propositional patterns)
The set of propositional patterns is defined as:

- *The special symbol* α *is a propositional pattern.*
- *If* $\mathcal{P}$ *is a propositional pattern and* Q *is a proposition, then* $(\mathcal{P} \wedge Q)$ *is a propositional pattern.*

We also define the application of a pattern $\mathcal{P}$ *to a propositional pattern or a proposition* X *as:*

$$\begin{array}{ll} \mathcal{P}(X) \equiv X & \text{if } \mathcal{P} \equiv \alpha \\ \mathcal{P}(X) \equiv \mathcal{P}_1(X) \wedge P_2 & \text{if } \mathcal{P} \text{ has the form } \mathcal{P}_1 \wedge P_2 \end{array}$$

Note that if X is a pattern, $\mathcal{P}(X)$ is also a pattern and if X is a proposition, $\mathcal{P}(X)$ is a proposition. For sets $\mathcal{P}^s$ of patterns we define

$$\begin{array}{rcl} \mathcal{P}^s(X) & = & \{\mathcal{P}(X) \mid \mathcal{P} \in \mathcal{P}^s\} \text{ and} \\ \mathcal{P}^s(X^s) & = & \{\mathcal{P}(X) \mid \mathcal{P} \in \mathcal{P}^s, X \in X^s\} \end{array}$$

Also, define for pattern $\mathcal{P}$ and substitution ϕ that

$$\begin{array}{ll} \phi(\mathcal{P}) \equiv \alpha & \text{if } \mathcal{P} \equiv \alpha \\ \phi(\mathcal{P}) \equiv \phi(\mathcal{P}_1) \wedge \phi(P_2) & \text{if } \mathcal{P} \text{ has the form } \mathcal{P}_1 \wedge P_2 \end{array}$$

For sets $\mathcal{P}^s$ of patterns we define $\phi(\mathcal{P}^s) = \{\phi(\mathcal{P}) \mid \mathcal{P} \in \mathcal{P}^s\}$.

For example, let $((\alpha \wedge P_1) \wedge P_2)$ be a pattern. Then $((\alpha \wedge P_1) \wedge P_2)(P_0)$ yields the proposition $((P_0 \wedge P_1) \wedge P_2)$. When applied to $(\alpha \wedge P_0)$ we get propositional pattern $(((\alpha \wedge P_0) \wedge P_1) \wedge P_2)$

Definition 11 (Constraints)
A constraint C is a pair of substitutions. If P is a proposition, P is said to satisfy constraint $C = \langle \phi_1, \phi_2 \rangle$ if and only if $\phi_1(P) \equiv \phi_2(P)$. P is said to satisfy the set C^s of constraints if and only if it satisfies all constraints in this set. For a constraint set C and a substitution σ, we define $C \circ \sigma = \{\langle \phi_1 \circ \sigma, \phi_2 \circ \sigma \rangle \mid \langle \phi_1, \phi_2 \rangle \in C\}$.

For instance, let $\langle \phi_1, \phi_2 \rangle$ be a constraint with $\phi_1 = id$ and $\phi_2 = \{x \mapsto y\}$. Proposition $P(x)$ does not satisfy this constraint, since $\phi_1(P(x)) = P(x)$, whereas $\phi_2(P(x)) = P(y)$. However, $P(z)$ does meet the constraint, since $\phi_1(P(z)) = \phi_2(P(z)) = P(z)$.

Lemma 12 (Inference of specification constraints without fake)
Let Γ be a Hoare context and S be a program without **fake** *statements. We can compute a triple $T = \langle \phi, \mathcal{Q}^s, C^s \rangle$ where ϕ is a substitution, $\mathcal{Q}^s$ is a set of propositional patterns and C^s is a set of constraints, such that $\Gamma \vdash S{:}P \triangleright Q$ if and only if there exists a P' satisfying C^s such that $Q \in \mathcal{Q}^s(P')$ and $P \equiv \phi(P')$.*

Hence, for each P' satisfying C^s we have for each propositional pattern $\mathcal{Q} \in \mathcal{Q}^s$ that

$$\Gamma \vdash S{:}\phi(P') \triangleright \mathcal{Q}(P').$$

Intuitively, P' satisfying C^s is the piece of the postcondition that corresponds to the precondition. The remaining part of the postcondition, introduced by *If* guards and blocks, is given by the patterns stored in $\mathcal{Q}^s$.

Proof: The proof is tedious and involved. As an example, the cases *If* and *Comp*are given in appendix A. □

Programs without **fake** statements for which valid specifications exist will always be somewhat artificial. For an example see appendix B. However, for the sake of completeness of the theory, these programs must also be considered.

Corollary 13 (Specification inference is decidable)
From lemma 9 and lemma 12 it immediately follows that specification inference is decidable. (For programs without **fake***, the proposition* **true** = **true** *will always meet all constraints).*

4 Results

The combined system of typed lambda calculus and a Hoare logic presented has many desirable properties:

- The system conforms to the de Bruijn criterion, which means that completed programs can be checked by a small, reliable program.
- The logic system is independent of the Hoare logic. This enables us to experiment with different logics. For instance, we can restrict the logic to first-order logic (using $\lambda P-$), which allows for a high degree of automation. When using other λ-calculi, like λC, we get a stronger specification language and more expressive power in the expressions of the programming language. On the other hand, these more powerful calculi are (a lot) harder to automate.
- Programs are directly represented in the system and can easily be edited by the programmer. In Cocktail (see [Fra00a]), programs and proofs are both edited in similar structure editors, allowing for a large amount of software re-use.

The main disadvantage of the approach is that a tool based on this logic cannot be used to prove meta-theorems about the language itself. However, since our goal was a tool for deriving correct programs, this was never the intention of this logic.

References

[BB] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *To appear in Journal of Symbolic Computing.*

[Fra97] Michael Franssen. λP−: A pure type system for first order logic with automated theorem proving. Computing Science Report 97–15, Eindhoven University of Technology, 1997.

[Fra00a] Michael Franssen. *Cocktail: A Tool for Deriving Correct Programs.* PhD thesis, Eindhoven University of Technology, 2000.

[Fra00b] Michael Franssen. Embedding first-order tableaux into a pure type system. In Didier Galmiche, editor, *Electronic Notes in Theoretical Computer Science*, volume 17. Elsevier Science Publishers, 2000.

[HM95] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, 1995.

[LF01] Twan Laan and Michael Franssen. Embedding first-order logic in a pure type system with parameters. *Journal of Logic and Computation*, 11(4):545–557, 2001.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing. Wiley, 1992.

[OG02] Martijn Oostdijk and Herman Geuvers. Proof by computation in the Coq system. *Theoretical Computer Science*, 272(1-2):293–314, 2002. Special Issue on the MSJ regional workshop on Theories of Types and Proofs (TTP'97), Tokyo, Japan.

[vW94] Joakim von Wright. Program refinement by theorem prover. In David Till, editor, *6th Refinement Workshop*, pages 121–150. Springer, 1994.

A Proof cases for inference of specification constraints without Fake

Let $\Gamma \equiv \langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$ denote a Hoare context. We use induction on the structure of S (Note that $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ stands for $((P_1 \wedge P_2) \wedge \ldots \wedge P_n)$ and not $P_1 \wedge (P_2 \wedge \ldots \wedge P_n)$ etc.). We only show the cases for *If* and *Comp*:

case $S \equiv$ **if** g **then** S_1 **else** S_2 **fi:** By induction we compute for S_1 and S_2 the triples $T_1 = \langle \phi_1, \mathcal{Q}_1^s, C_1^s \rangle$ and $T_2 = \langle \phi_2, \mathcal{Q}_2^s, C_2^s \rangle$ respectively. We choose $T = \langle \phi_1, \mathcal{Q}^s, C^s \rangle$, where $\mathcal{Q}^s = \bigcup_{g' \in G} (\mathcal{Q}_1^s \cap \mathcal{Q}_2^s)(g')$ with $G = \{g' \in \phi_1^{-1}(g = \textbf{true}) \cap \phi_2^{-1}(g = \textbf{false}) \mid g' \text{ satisfies } C_1^s \cup C_2^s\}$ and $C^s = C_1^s \cup C_2^s \cup \{\langle \phi_1, \phi_2 \rangle\}$. Since this is not obvious, we will provide further proof:

Suppose $\Gamma \vdash S{:}P \triangleright Q$. Hence, from the premises of *If* in the Hoare logic we get $\Gamma \vdash S_1{:}P \wedge g = \textbf{true} \triangleright Q$ and $\Gamma \vdash S_2{:}P \wedge g = \textbf{false} \triangleright Q$. Hence, by induction there exist P_1 and P_2 satisfying C_1^s and C_2^s respectively, such that $P \wedge g = \textbf{true} \equiv \phi_1(P_1)$ and $P \wedge g = \textbf{false} \equiv \phi_2(P_2)$ and $Q \in \mathcal{Q}_1^s(P_1)$ and $Q \in \mathcal{Q}_2^s(P_2)$. Hence, P_1 is of the form $P_1' \wedge G_1$ and P_2 is of the form $P_2' \wedge G_2$, such that $P \equiv \phi_1(P_1')$, $P \equiv \phi_2(P_2')$, $g = \textbf{true} \equiv \phi_1(G_1)$ and $g = \textbf{false} \equiv \phi_2(G_2)$. Hence, Q is of the form $P_1' \wedge G_1 \wedge A_1 \wedge \ldots \wedge A_n$ and of the form $P_2' \wedge G_2 \wedge B_1 \wedge \ldots \wedge B_m$. It follows that $m = n$, $P_1' \equiv P_2'$, $G_1 \equiv G_2$ and for all i, $1 \leq i \leq n$ $A_i \equiv B_i$. Hence, $\alpha \wedge A_1 \wedge \ldots \wedge A_n \in \mathcal{Q}_1^s \cap \mathcal{Q}_2^s$ and also $Q \in ((\mathcal{Q}_1^s \cap \mathcal{Q}_2^s)(\alpha \wedge G_1))(P_1')$. $G_1 \equiv G_2$ satisfies both C_1^s and C_2^s. Since $g = \textbf{true} \equiv \phi_1(G_1)$ and $g = \textbf{false} \equiv \phi_2(G_1)$, we have $G_1 \in \phi_1^{-1}(g = \textbf{true}) \cap \phi_2^{-1}(g = \textbf{false})$, hence $Q \in \bigcup_{g \in G} (\mathcal{Q}_1^s \cap \mathcal{Q}_2^s)(\alpha \wedge G_1)(P_1')$. Also, $P_1' \equiv P_2'$, so P_1' satisfies C_1^s and C_2^s and $\phi_1(P_1') \equiv \phi_2(P_1')$, hence P_1' satisfies $C_1^s \cup C_2^s \cup \{\langle \phi_1, \phi_2 \rangle\}$. If follows that there exists a P' satisfying C^s such that $P \equiv \phi_1(P')$ and $Q \in \mathcal{Q}^s(P')$.

Conversely, suppose there exists a P' satisfying C^s. Let Q be an element of $\mathcal{Q}^s(P')$, i.e. Q has the form $P' \wedge G \wedge A_1 \wedge \ldots \wedge A_n$, where $G \in \phi_1^{-1}(g = \textbf{true}) \cup \phi_2^{-1}(g = \textbf{false})$ and $\alpha \wedge A_1 \wedge \ldots \wedge A_n$ is an element of both $\mathcal{Q}_1^s$ and $\mathcal{Q}_2^s$, and where G satisfies C_1^s and C_2^s. Hence, $P' \wedge G$ satisfies C_1^s and C_2^s and hence, by the induction hypothesis, $\Gamma \vdash S_1{:}\phi_1(P' \wedge G) \triangleright Q$ and $\Gamma \vdash S_2{:}\phi_2(P' \wedge G) \triangleright Q$. This is equal to $\Gamma \vdash S_1{:}\phi_1(P') \wedge g = \textbf{true} \triangleright Q$ and $\Gamma \vdash S_2{:}\phi_2(P') \wedge g = \textbf{false} \triangleright Q$. Since P' satisfies $\langle \phi_1, \phi_2 \rangle$ we can now derive $\Gamma \vdash S{:}\phi_1(P) \triangleright Q$ for any $Q \in \mathcal{Q}^s(P')$.

case $S \equiv S_1; S_2$**:** By induction compute $T_1 = \langle \phi_1, \mathcal{Q}_1^s, C_1^s \rangle$ for S_1 and $T_2 = \langle \phi_2, \mathcal{Q}_2^s, C_2^s \rangle$ for S_2. Use $T = \langle \phi_1 \circ \phi_2, \mathcal{Q}^s, C^s \rangle$, where $\mathcal{Q}^s = \mathcal{Q}_2^s(\{Q \mid \phi_2(Q) \in \mathcal{Q}_1^s, Q \text{ satisfies } C_2^s\})$ and $C^s = C_1^s \circ \phi_2 \cup C_2^s$. This requires further proof:

Suppose $\Gamma \vdash S{:}P \triangleright Q$. From the premises of *Comp* we get $\Gamma \vdash S_1{:}P \triangleright R$ and $\Gamma \vdash S_2{:}R \triangleright Q$ for certain R. Hence, there exists a P_2 satisfying C_2^s such that $Q \in \mathcal{Q}_2^s(P_2)$ and $R \equiv \phi_2(P_2)$ and a P_1 satisfying C_1^s such that $R \in \mathcal{Q}_1^s(P_1)$ and $P \equiv \phi_1(P_1)$. Hence, R has the form $P_1 \wedge A_1 \wedge \ldots \wedge A_n$ and since $R \equiv \phi_2(P_2)$, P_2 must have the form $P_1' \wedge A_1' \wedge \ldots \wedge A_n'$ with $P_1 \equiv \phi_2(P_1')$, $A_i \equiv \phi_2(A_i')$ and P_1' and A_i' satisfying C_2^s. Since P_1 satisfies C_1^s, we have that P_1' satisfies $C_1^s \circ \phi_2$, hence P_1' satisfies $C_1^s \circ \phi_2 \cup C_2^s$. Also $P \equiv \phi_1(P_1) \equiv \phi_1(\phi_2(P_1'))$. It remains to show that $Q \in \mathcal{Q}_2^s(\{Q \mid \phi_2(Q) \in \mathcal{Q}_1^s, Q \text{ satisfies } C_2^s\})$. Since P_2 has the form $P_1' \wedge A_1' \wedge \ldots A_n'$ with $A_i \equiv \phi_2(A_i')$ and A_i' satisfying C_2^s, $P_2 \in \{Q \mid \phi_2(Q) \in \mathcal{Q}_1^s, Q \text{ satisfies } C_2^s\}$. From $Q \in \mathcal{Q}_2^s(P_2)$ we get $Q \in \mathcal{Q}_2^s(\{Q \mid \phi_2(Q) \in \mathcal{Q}_1^s, Q \text{ satisfies } C_2^s\})$. Hence, P_1' satisfies C^s, $P \equiv \phi_1 \circ \phi_2(P_1')$ and $Q \in \mathcal{Q}^s(P_1')$.

Conversely, suppose P' satisfies $C_1^s \circ \phi_2 \cup C_2^s$. Let $Q \in \mathcal{Q}^s(P')$. Hence, P' satisfies C_2^s and Q has the form $P' \wedge A_1' \wedge \ldots \wedge A_n' \wedge B_1 \wedge \ldots \wedge B_m$ with A_i' satisfying C_2^s. From T_2 we get $\Gamma \vdash S_2{:}\phi_2(P' \wedge A_1' \wedge \ldots \wedge A_n') \triangleright Q$, i.e. $\Gamma \vdash S_2{:}\phi_2(P') \wedge A_1 \wedge \ldots \wedge A_n \triangleright Q$ with $\alpha \wedge A_1 \wedge \ldots \wedge A_n \in \mathcal{Q}_1^s$. Since P' satisfies $C_1^s \circ \phi_2$, we know $\phi_2(P)$ satisfies C_1^s. Hence, by induction we get $\Gamma \vdash S_1{:}\phi_1 \circ \phi_2(P') \triangleright \phi_2(P') \wedge A_1 \wedge \ldots \wedge A_n$. Using *Comp* we derive $\Gamma \vdash S{:}\phi_1 \circ \phi_2(P) \triangleright Q$.

Since $Q \in \mathcal{Q}_2^s(P_2)$, it has the form $P_2 \wedge B_1 \wedge \ldots \wedge B_m$, which has the form $P_1' \wedge A_1' \wedge \ldots \wedge A_n' \wedge B_1 \wedge \ldots \wedge B_m$. with $\alpha \wedge \phi_2(A_1') \wedge \ldots \wedge \phi_2(A_n') \in \mathcal{Q}_1^s$. Also, for i, $1 \leq i \leq n$ we have that A_i' satisfies C_2^s and $A_i' \in \phi_2^{-1}(A_i)$. Hence, Q has the form $P_1' \wedge A_1' \wedge \ldots \wedge A_n' \in \mathcal{Q}_1^s$

B An example of specification inference without Fake

As an example, consider the following simple program (in this program, g and b are variables of type boolean):

$$S \equiv \textbf{if } g \textbf{ then } b := \textbf{true} \\ \textbf{else } b := \textbf{false} \\ \textbf{fi}$$

For the assignments $b := \textbf{true}$ and $b := \textbf{false}$, the triples $T_1 = \langle \{b \mapsto \textbf{true}\}, \{\alpha\}, \emptyset \rangle$ and $T_2 = \langle \{b \mapsto \textbf{false}\}, \{\alpha\}, \emptyset \rangle$ are computed. Then, for the **if**-statement, the constraint set becomes $\emptyset \cup \emptyset \cup \{\langle \{b \mapsto \textbf{true}\}, \{b \mapsto \textbf{false}\} \rangle\}$. To compute the propositional patterns, we first compute the inverse substitutions $\{e \mapsto \textbf{true}\}^{-1}(g = \textbf{true}) = \{g = \textbf{true}, g = b\}$ and $\{e \mapsto \textbf{false}\}^{-1}(g = \textbf{false}) = \{g = \textbf{false}, g = b\}$. The intersection of these sets is $\{g = b\}$, which trivially satisfies the empty sets of constraints. Hence, the triple for the **if**-statement becomes $T = \langle \{b \mapsto \textbf{true}\}, \{\alpha \wedge g = b\}, \{\langle \{b \mapsto \textbf{true}\}, \{b \mapsto \textbf{false}\} \}\rangle$.

To check the result, consider the proposition $a = \textbf{true}$. It meets all constraints, hence $S : a = \textbf{true} \triangleright a = \textbf{true} \wedge g = b$ must hold, which is easy to derive from the *If* and *Assign* rules of *While*.

Conversely, the proposition $a = b$ does not meet the constraint and hence, $S : a = \textbf{true} \triangleright a = b \wedge g = b$ should not hold. Indeed, the *Assign*-rule of *While* states $b := \textbf{true} : a = \textbf{true} \wedge g = \textbf{true} \triangleright a = b \wedge g = b$ and $b := \textbf{false} : a = \textbf{false} \wedge b = \textbf{false} \triangleright a = b \wedge g = b$. Hence, the *If* rule of *While* cannot be applied.

C Example of a derived program

In this appendix, we show a small example of a program to compute factorials derived in the system we proposed. Since many theorems are needed and used, we cannot give the full derivation. Instead, we will show the final program, including its **fake**-statements and indicate what contexts and proofs are required.

The contexts Γ_1, Γ_2 and Γ_3 be defined as:

$$\begin{array}{lcl} \Gamma_1 & = & \Gamma_{\mathcal{L}}, N : nat \text{ (including natural numbers with addition and muliplication)} \\ \Gamma_2 & = & x : nat \\ \Gamma_3 & = & fac : nat \rightarrow nat, AxFac0 : fac(0) = 1, AxFacN : (\forall n : nat. fac(s(n)) = fac(n) * s(n)) \end{array}$$

A program computing factorials, including the fake-statements is then given by S:

$$\begin{array}{lcl} S & = & \{True\}\textbf{fake } p_0\{fac(0) = 1\}; \\ & & n := 0; x := 1; \{fac(n) = x\} \\ & & \textbf{while } \ n \neq N \textbf{ do} \\ & & \qquad \{fac(n) = x \wedge n \neq N\}\textbf{fake } p_1\{fac(s(n)) = x * (s(n))\}; \\ & & \qquad n := s(n); x := x * n\{fac(n) = x\} \\ & & \textbf{od} \\ & & \{fac(n) = x \wedge \neg(n \neq N)\}\textbf{fake } p_2\{fac(N) = x\} \end{array}$$

where p_0, p_1 and p_2 are λ-terms derivable in $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$, such that

$$\begin{array}{l} \Gamma \vdash p_0{:}True \Rightarrow fac(0) = 1 \\ \Gamma \vdash p_1{:}(fac(n) = x \wedge n \neq N) \Rightarrow fac(s(n)) = x * s(n) \\ \Gamma \vdash p_2{:}(fac(n) = x \wedge \neg(n \neq N)) \Rightarrow fac(N) = x \end{array}$$

These λ-terms are easy to construct; for instance p_0 can be $\lambda H : True.AxFac0$.