RACSAM
Rev. R. Acad. Cien. Serie A. Mat.
VOL. **98 (**1), 2004, pp. 95–111
Ciencias de la Computación / Computational Sciences

Cocktail: A Tool for Deriving Correct Programs

Michael Franssen and Harrie de Swart

Abstract. Cocktail is a tool for deriving correct programs from their specifications. The present version is powerful enough for educational purposes. The tool yields support for many sorted first order predicate logic, formulated in a pure type system with parametric constants (CPTS), as the specification language, a simple *While*- language, a Hoare logic represented in the same CPTS for deriving programs from their specifications and a simple tableau based automated theorem prover for verifying proof obligations.

Cocktail: Una herramienta para obtener programas correctos

Resumen. Cocktail es una herramienta para la obtención de programas correctos a partir de sus especificaciones. La versión actual de la herramienta tiene una potencia suficiente para su uso con fines educativos. La herramienta proporciona soporte a la lógica de predicados de primer orden multivariada, formulada en un sistema puro de tipos con constantes paramétricas (CPTS), como el lenguaje de especificación, un sencillo lenguaje While, una lógica de Hoare representada en el mismo CPTS para la obtención de programas a partir de sus especificaciones, y un demostrador automático de teoremas sencillo, basado en tableaux para la verificación de obligaciones de prueba.

1. Introduction

At present, most tools for correct programming support functional or logical programming paradigms. However, in practice, most programs are written using procedural languages (including object-oriented languages) like Pascal, C/C++ or Java. Tools for procedural languages usually provide only syntactical support.

The essence of algorithms in procedural languages can be expressed in Dijkstra's guarded command language (GCL), for which there is a formal basis that allows semantical support. In fact, programs written in GCL can be *derived* from their specification, as is shown in [6, 13]. So far, there are no tools that support this approach to programming.

The goal of Cocktail is to provide semantical support for deriving programs from their specifications by stepwise refinement, generating proof obligations on the fly. The system is intended for programmers in an educational setting. Cocktail is mainly practice-oriented to increase its usability for the intended users. Also, the system is set up to be extensible, flexible and easy to experiment with.

Cocktail is written entirely in Java and provides a modern graphical user interface. Proofs are graphically represented in Fitch-style natural deduction and can be manipulated by drag-and-drop operations.

Presentado por Luis M. Laita.
Recibido: November 26, 2003. *Aceptado*: October 13, 2004.
Palabras clave / Keywords: program derivation, Pure Type System, theorem proving, Hoare logic
Mathematics Subject Classifications: 68-02, 68-04, 68T15.

1.1. Logical foundation

To ensure a firm logical foundation of the system, we use a typed lambda calculus for the logical parts of our system. Typed lambda calculi are suitable for interactive theorem proving, as is shown by systems like Coq, LEGO and Yarrow (see [5]). Instead of using a fixed calculus, we use a logical framework of typed lambda calculi: the framework of Pure Type Systems (PTSs). This approach has several advantages. Firstly, it allows us to easily increase the expressive power of the logical foundation of the system at a later stage, without having to re-implement the entire logic. Secondly, we can use a type checking algorithm to verify the correctness of constructed proofs, thereby avoiding errors being introduced when the theorem prover gets large. Thirdly, we can communicate our proofs to other proof systems, since lambda terms are standardized proof objects. So, the requirement of communicable proofs, called the De Bruijn criterion, is satisfied.

On the other hand, typed lambda calculi are usually used for higher order logics, which are not (yet) needed for our system. During the derivation of a program, many first order theorems have to be proved, most of which are relatively simple. Since for first order theorems good automation is possible, it is desirable to include an automated theorem prover (ATP) in the system. However, ATPs are usually based on other formalisms and use other techniques to prove theorems than interactive theorem provers.

1.2. Embedding first order tableaux in a pure type system

In Cocktail, automated and interactive theorem proving are combined as follows. Firstly, an extension, introduced in [14], to the standard PTSs is used to faithfully model first order logic. Secondly, a tableaux based theorem prover is implemented that directly uses the formulas from the extended PTS. Since we use a faithful model of first order logic, no problems will occur during the construction of the semantic tableau. Thirdly, the tableau constructed by the ATP is translated into a lambda-term of the underlying logic. A precise description of the translation algorithm and the resulting system can be found in [10, 11].

1.3. Linking Hoare logic with a pure type system

Our aim was to build a tool that assists programmers in constructing correct programs. However, to decide whether a program is correct, the programmer must provide a formal specification of what the program is supposed to do. The programmer will be interested mainly in the program itself and less in its correctness proof. To accommodate the programmer, the tool needs to represent programs as directly as possible and has to construct proofs automatically whenever possible.

Roughly speaking, there are three approaches to correct programming:

- Constructing the program first. In this approach, the correctness of the program is verified after it has been written.
- Constructing the proof first. In this method, the programmer first has to prove constructively a theorem which indirectly states that there exists a program that meets the specification. The program is then extracted from the proof of the theorem. See e.g. [17].
- Simultaneous construction of program and proof. In this method, the program and its correctness proof are constructed hand in hand. It is based on a Hoare logic, which directly links the program to its semantics. From Dijkstra-Hoare calculus and the works of Gries, Feijen and Kaldewaij ([12, 7, 13]) it becomes clear that programs can actually be *derived* from their specifications. When finished, the program and its correctness proof are both available.

In the Hoare logic or axiomatic semantics of the guarded command language supported by Cocktail, programs are annotated with proofs and specifications. The proofs in the annotation provide enough information to check a posteriori if a constructed program does indeed meet its specification. Just as a simple type checking algorithm can ensure that a constructed proof (lambda-term) is correct, a simple program

checking algorithm ensures an annotated program is correct. In this way the soundness of Cocktail depends only on this checking algorithm, and not on the whole system, which may get quite large.

1.4. Result

Cocktail is an experimental system for deriving correct programs, aimed at students. The simultaneous construction of a program and its correctness proof is based on the Dijkstra-Hoare calculus (see [6]). The main advantage of this approach is that the programmer is guided to a correct program by the specification during the program's construction. The underlying logic can easily be extended and changed, since a logical framework - a pure type system - is used rather than a fixed logic. Many of the simple theorems are proved automatically. The graphical user interface provides a clear view of the proofs and allows intuitive construction of proofs.

1.5. Design

The Cocktail tool has been designed roughly in three parts, each extending the previous one.

- The symbolic engine, consisting of the representation of terms of the different formal systems and the type checkers. For (many-sorted) first order logic we use a simple Pure Type System (PTS). To safely support Hoare logic, we design a specific version that conforms to the De Bruijn criterion. Also, we design this Hoare logic in such a way that it has the same structure and properties as the PTS. This allows us to re-use code of the PTS type checker for the program checker. Hence, the symbolic engine remains small, comprehensible and can therefore be trusted to be correct. Also, having both systems - first order logic and the Hoare logic - represented by a uniform structure is more satisfying aesthetically.

- The tactic system, being a layer on top of the symbolic engine, which enables the user to use larger steps in the proof than those allowed by the formal system. We want the tactic system to support at least the following: backward reasoning, forward reasoning, Automated Theorem Proving (ATP), and equational reasoning.

- The user interface, which enables the user to interact with the system by sending commands to the tactic system.

2. Many sorted first order predicate logic

A weakness of the familiar definition of terms of first order logic in the literature is that all terms are treated equally. In practice, we often want to distinguish between terms of different types (for instance, booleans and integers). Therefore we introduce a many sorted first order logic.

Definition: Many sorted first order formulas
Let $\mathcal{F}$ be a set of function symbols, each with a fixed arity ≥ 0. Furthermore, let $\mathcal{P}$ be a set of predicate symbols, each with a fixed arity ≥ 0. For convenience, we assume a special predicate symbol *Falsum* to exist in $\mathcal{P}$ with arity 0. In addition to $\mathcal{F}$ and $\mathcal{P}$ we have a parameter Set that represents a set of basic types. Finally, we assume the existence of an infinite set V of variables. The sets $\mathcal{F}$, $\mathcal{P}$, Set and V are disjoint. With every function symbol $f \in \mathcal{F}$ with arity n, we associate a unique tuple of types $(U_1, \ldots, U_n, U)$, where $U_1, \ldots U_n$ and U are elements of Set. We denote this as $f : (U_1, \ldots U_n, U) \in \mathcal{F}$. With every predicate symbol $P \in \mathcal{P}$ with arity n, we associate a unique tuple of types $(U_1, \ldots, U_n)$, where $U_1, \ldots, U_n \in Set$. This is denoted as $P : (U_1, \ldots, U_n) \in \mathcal{P}$. Since the arity of function and predicate symbols can now be derived from its unique associated tuple of types it will no longer be stated explicitly. The set V of variables in the extended framework contains variables which each have a unique associated type U, where $U \in Set$.

We assume that for every type there are infinitely many variables. The definition of the set T of typed terms is:

1. $a : U \in T$ for every variable a with associated type U.
2. If $f : (U_1, \ldots, U_n, U) \in \mathcal{F}$ and $t_1 : U_1, \ldots, t_n : U_n \in T$, then $(f t_1 \ldots t_n) : U \in T$.

The set Prop of formulas for many sorted first order logic is now defined as:

1. If $P : (U_1, \ldots, U_n) \in \mathcal{P}$ and $t_1 : U_1, \ldots, t_n : U_n \in T$, then $(P t_1 \ldots t_n) \in$ Prop.
2. If $P, Q \in$ Prop, then $(P \wedge Q) \in$ Prop, $(P \vee Q) \in$ Prop and $(P \Rightarrow Q) \in$ Prop.
3. If $P \in$ Prop, then $(\neg P) \in$ Prop.
4. If $P \in$ Prop and $x : U \in V$, then $(\forall x : U.P) \in$ Prop and $(\exists x : U.P) \in$ Prop.

We will only consider substitutions of terms for variables which have the same associated type.

3. The pure type system $\lambda P-$

The PTS framework is used to describe several different logics in a uniform way. We use such a PTS to construct correctness proofs for imperative programs. By using a PTS as the basis of our tool instead of a single fixed logic, we enable future extensions of the logical system. Also, PTSs have proved suitable for constructing interactive theorem provers like COQ [5] and Yarrow [18].

$\lambda P-$ is a Pure Type System with parametric constants (CPTS) that exactly models many sorted first order predicate logic (see [14]).

Definition: $\lambda P-$
$\lambda P-$ is the CPTS specified by:

$$\begin{array}{lcl} \mathcal{S} & = & \{*_s, *_p, \Box_s, \Box_p\} \text{ is the set of sorts} \\ \mathcal{A} & = & \{(*_s, \Box_s), (*_p, \Box_p)\} \text{ is the set of axioms} \\ \mathcal{R} & = & \{(*_p, *_p, *_p), (*_s, *_p, *_p)\} \text{ is the set of rules} \\ \mathcal{PR} & = & \{(*_s, *_s), (*_s, \Box_p)\} \text{ is the set of parametric rules} \end{array}$$

A regular Pure Type System (PTS) has no parametric rules. Given a specification of a CPTS, say $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{PR})$, the terms, contexts and type judgement relation of the CPTS are defined as follows.

Definition: Parametric Terms
Given a set V of variables and a set C of constants (which is empty in the case of a regular PTS), the set T_C of CPTS terms is defined by the following abstract syntax:

$$\begin{array}{lcl} T_C & ::= & \mathcal{S} \mid V \mid \lambda V : T_C.T_C \mid \Pi V : T_C.T_C \mid T_C T_C \mid C(L_C) \\ L_C & ::= & \varepsilon \mid < L_C, T_C > \end{array}$$

The lists of terms produced by L_C are usually denoted as $< A_1, \ldots, A_n >$ or $A_1, \ldots, A_n$ instead of $< \ldots << \varepsilon, A_1 >, A_2 > \ldots A_n >$.

Definition: Contexts of CPTSs
A context is a list of the form $x_1 : A_1, \ldots, x_n : A_n$, such that every A_i is a term and either $x_i \in V$ or x_i has the form $c(y_1 : B_1, \ldots, y_m : B_m)$, where $c \in C$, $y_1, \ldots, y_m \in V$ and $B_1, \ldots, B_m$ are terms. A constant c is called Γ-fresh if it does not occur in Γ.

Definition: β**-reduction**
On the terms of PTSs, we define a reduction relation $\rightarrow_\beta \subseteq T \times T$ as the smallest relation such that

$$(\lambda x : A.b)c \rightarrow_\beta b[x := c]$$

start	$\quad <> \vdash s1{:}s2$	$(s1, s2) \in \mathcal{A}$
intro	$\dfrac{\Gamma \vdash A{:}s}{\Gamma, x{:}A \vdash x{:}A}$	x is Γ-fresh
weaken	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A{:}B}$	x is Γ-fresh
Π-*form*	$\dfrac{\Gamma \vdash A{:}s1 \quad \Gamma, x{:}A \vdash B{:}s2}{\Gamma \vdash (\Pi x{:}A.\ B){:}s3}$	$(s1, s2, s3) \in \mathcal{R}$
Π-*intro*	$\dfrac{\Gamma, x{:}A \vdash b{:}B \quad \Gamma \vdash (\Pi x{:}A.\ B){:}s}{\Gamma \vdash (\lambda x{:}A.\ b){:}(\Pi x{:}A.\ B)}$	
Π-*elim*	$\dfrac{\Gamma \vdash F{:}(\Pi x{:}A.\ B) \quad \Gamma \vdash a{:}A}{\Gamma \vdash\ Fa{:}B[x := a]}$	
conversion	$\dfrac{\Gamma \vdash A{:}B \quad \Gamma \vdash B'{:}s \quad B{=}_\beta B'}{\Gamma \vdash A{:}B'}$	

Figure 1. The type judgment derivation rules of a PTS.

and that is closed under

$$\begin{array}{lll} \text{if} \quad b \to_\beta b' & \text{then} & (\lambda x : b.c) \to_\beta (\lambda x : b'.c) \\ & & (\lambda x : a.b) \to_\beta (\lambda x : a.b'), \\ & & (\Pi x : b.c) \to_\beta (\Pi x : b'.c), \\ & & (\Pi x : a.b) \to_\beta (\Pi x : a.b'), \\ & & (bc) \to_\beta (b'c) \\ & \text{and} & (ab) \to_\beta (ab') \end{array}$$

$\Rightarrow_\beta$ denotes the reflexive and transitive closure of $\to_\beta$. $=_\beta$ denotes the symmetric, reflexive and transitive closure of $\to_\beta$. $B =_\beta B'$ is read as "B is β-equal to B'", which means that there exists a B'' such that B and B' can both be reduced to B'' by β-reduction.

Definition: Type judgment relation of a regular PTS
The type judgment relation describes the actual PTS. A judgment always has the form $\Gamma \vdash A : B$, where A and B are terms and Γ is a context. $\Gamma \vdash A : B$ should be read as: 'A has type B in context Γ'. The type judgment relation $\vdash$ is defined by the rules in figure 1.

We give a brief description of each type judgment rule in figure 1:

$start$ This is the only rule without premises in a PTS. It supplies, starting from the axioms in $\mathcal{A}$, basic typing judgments from which all the other typing judgments are derived.

$intro$ Intro is used in a much more general sense than the $intro$-rule in natural deduction. In natural deduction intro allows one to add assumptions to the context. In a PTS intro allows one to add assumptions, constants (which in a PTS are equal to variables), functions and propositional variables (including predicates) to the context. This depends on the form of A. The type of the introduced item x depends on s, which is the type of the type of x.

$weaken$ Weaken is needed to preserve existing derivations in extended contexts. It states that everything that can be derived in a certain context can also be derived in a more extended context.

Π-$form$ This rule allows the construction of function types, predicates, universal quantifications, etc. The set of rules $\mathcal{R}$ of a PTS determines the ways in which Π-$form$ can be used. Actually, the set $\mathcal{R}$ states which abstractions are allowed.

Π-$intro$ One needs this rule to actually construct terms of a type built with the previous rule. Without this rule, we could only assume that there are terms of this type by using $intro$.

Π-$elim$ Once a term with a Π-type is constructed or assumed, it can be used to create a term with a more concrete type. The Π-$elim$ rule, also referred to as the application rule, instantiates the body of an abstract Π-type by substituting a term for the bound abstract variable.

$Conversion$ This rule states that we don't distinguish β-equal types. In several PTSs a term A can have type B where B can be rewritten to B' by β-reduction. In the propositions-as-types isomorphism, B and B' then represent the same propositional formula (we will come back to this in our example below) and hence, A is a proof of B' just as well as it is a proof of B. To support this switch of representation the $conversion$ rule is needed.

Definition: Valid (or legal) contexts
A context is called valid or legal, if it can occur in a derivation using only the axioms of the Pure Type System. This notion is also used for extended PTSs, where additional rules may also be used.

For many PTSs one can automatically compute an entire derivation of $\Gamma \vdash p : P$ for a given context Γ, a proof term p and a formula P provided that at least one derivation exists (see e.g. [1]). Hence the proof-term can be checked for correctness (type-checked). This has two advantages:

1. Even if a large tool is used to construct a proof-term p, correctness of the proof is assured by type-checking. This algorithm is relatively simple and can be proved to be correct.
2. Communicating proofs corresponds to communicating a syntactical proof term. This proof term can then be checked by another proof system based on λ-calculus.

Definition: Type judgment relation of a CPTS
The type judgment relation of a CPTS consists of all rules of a regular PTS (see figure 1) and two additional rules to make use of parametric constants. Let Δ denote $x_1 : B_1, \ldots, x_n : B_n$ and let Δ_i denote $x_1 : B_1, \ldots, x_{i-1} : B_{i-1}$. Then the additional rules are:

$$\textit{C-weaken}\quad \frac{\begin{array}{lll}\Gamma \vdash b{:}B & & \\ \Gamma, \Delta_i \vdash B_i{:}s_i & \text{for } i = 1, \ldots, n & (s_i, s) \in \mathcal{P} \\ \Gamma, \Delta \vdash A{:}s & & c \text{ is } \Gamma\text{-fresh}\end{array}}{\Gamma, c(\Delta){:}A \vdash b{:}B}$$

$$\textit{C-application}\quad \frac{\begin{array}{ll}\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash b_i{:}B_i[x_j := b_j]_{j=1}^{i-1} & \text{for } i = 1, \ldots, n \\ \Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash A{:}s & \text{if } n = 0\end{array}}{\Gamma_1, c(\Delta){:}A, \Gamma_2 \vdash c(b_1, \ldots, b_n){:}A[x_j := b_j]_{j=1}^{n}}$$

We give a brief description of the additional rules:

C-$weaken$ The C-$weaken$ rule allows us to add a parametric constant to the context. In contrast to other extensions of the context this rule does not allow us to type the parametric constant itself, while the intro-rule (used for regular extensions of the context) allows the typing of every newly added item.

C-$application$ Since a parametric constant itself cannot be typed in a CPTS it cannot be used with the usual Π-$elim$ (sometimes called *application*) rule. The rule C-$application$ allows us to use a parametric constant, but only if we supply all the required arguments at once. This corresponds to the use

of functions and predicates in first order logic: these too can only be used after all the arguments have been supplied. The special premise for the case $n = 0$ is needed to assure that the context $\Gamma_1, c(\Delta) : A, \Gamma_2$ is a valid one.

For properties of CPTSs see [14].

4. Faithful representation of logic in $\lambda P-$

Functions and predicates are now added to the context using the new rule $C\text{-}weaken$, using parametric rule $(*_s, *_s)$ for functions and $(*_s, \Box_p)$ for predicates. A function or a predicate can only be used to form a proposition using the rule $C\text{-}application$. For instance, a function of arity 2 can only be used when it is applied to 2 arguments at once.

As mentioned in the handbook of H.P. Barendregt, Berardi gave a representation of first order logic in a regular PTS without parameters. However, $\lambda P-$ corresponds more closely to first order logic:

1. Constants are now modelled by a parametric constant with zero parameters. For instance, the natural number 0 is modelled in a context as $\Gamma_1, Nat : *_s, 0() : Nat, \Gamma_2$. Since the 0 is now a constant from C, it cannot be confused with a variable from V, like in Berardi's representation, since it is not possible to build a term like $(\lambda 0() : Nat.X)$.

2. Functions themselves do not have types, while they do have a type in Berardi's representation. A binary function f with arguments from sets A and B yielding a value from C occurs in the context as $f(x : A, y : B) : C$. Since f is a parametric constant with 2 arguments it cannot be applied to a single argument $a : A$, as can be done in Berardi's representation. The same holds for predicates.

3. A single proposition corresponds to a single type, while in Berardi's representation a single proposition corresponds to several types. The rule $(*_s, \Box_p, \Box_p)$ allowing the typing of lambda terms representing predicates is no longer available. Therefore, a predicate P is no longer represented by $(\lambda x : U.P)a$, where U corresponds to a set of first order logic and $a : U$. Consequently, the rule $conversion$ is no longer needed, allowing a simpler and faster implementation.

In order to show that the conversion rule is superfluous in $\lambda P-$, we formally define what a 'type' is and then show that all types in $\lambda P-$ are in β-normal form. The proofs can be found in [15].

Definition: Let Γ be a context. A is a *type* in Γ if $A \in \mathcal{S}$ or there is $s \in \mathcal{S}$ such that $\Gamma \vdash A : s$.

Lemma: If $\Gamma \vdash P : Q$ and $\Gamma \vdash Q : *_s$ then P is in β-normal form.

This lemma shows that the terms that represent objects are always in β-normal form.

Theorem: If P is a type in a context Γ then P is in β-normal form.

The context, containing the set-, function- and predicate symbols of the logic, is defined as follows:

Definition: $\Gamma_{\mathcal{L}}$
Let $\mathcal{L}$ be a logic with set symbols $U_1, \ldots, U_k$, function symbols $f_1, \ldots, f_p$ and predicate symbols $P_1, \ldots, P_n$. Furthermore, let $V_{i,j}$ denote the set symbol representing the type of the j'th argument of function f_i and let V_i denote the set symbol representing the type of the result of function f_i. Finally, let $T_{i,j}$ denote the set symbol corresponding to the type of the j'th argument of predicate P_i. Then the context $\Gamma_{\mathcal{L}}$, modelling this first order logic in $\lambda P-$, is defined as:

$$U_1 : *_s, \ldots, U_k : *_s,$$
$$f_1(x_1 : V_{1,1}, \ldots, x_{s_1} : V_{1,s_1}) : V_1, \ldots, f_p(x_1 : V_{p,1}, \ldots, x_{s_p} : V_{p,s_p}) : V_p,$$
$$P_1(x_1 : T_{1,1}, \ldots, x_{r_1} : T_{1,r_1}) : *_p, \ldots, P_n(x_1 : T_{n,1}, \ldots, x_{r_n} : T_{n,r_n}) : *_p$$

s_i and r_j are the arities of f_i and P_j respectively.

The close correspondence of logic $\mathcal{L}$ to $\lambda P-$ with context $\Gamma_{\mathcal{L}}$ is given by the following theorems:

Theorem: $\Gamma_{\mathcal{L}} \vdash U : *_s$ if and only if U is a set symbol of $\mathcal{L}$.

Theorem: For any set symbol U of $\mathcal{L}$ we have $\Gamma_{\mathcal{L}} \vdash t : U$ if and only if t is a term in $\mathcal{L}$ whose type is represented by set symbol U.

Theorem: $\Gamma_{\mathcal{L}} \vdash P : *_p$ if and only if P is a proposition of $\mathcal{L}$.

Theorem: For any proposition P of $\mathcal{L}$ we have $\Gamma_{\mathcal{L}} \vdash p : P$ for some term p if and only if $\models_{\mathcal{L}} P$.

The converse is also true: if Γ is a valid context of $\lambda P-$, then there exists a logic $\mathcal{L}$ such that the theorems above with $\Gamma_{\mathcal{L}}$ replaced by Γ hold. Hence, $\lambda P-$ has a one-to-one correspondence with many-sorted first-order predicate logic (for a proof see [15]).

5. The While programming language

In this section we define a simple imperative language called $While$ (see also [16]).

Definition: $While$
Let Set, V and F be a set of type symbols, variables and function symbols, respectively. Assume that a special symbol **bool** exists in Set. Let T be the set of terms of many sorted predicate logic. The set H of pseudo-programs is defined by the following abstract syntax:

$$\begin{array}{rcl} H & ::= & \mathbf{skip} \mid V := T \mid \mathbf{if}\ T\ \mathbf{then}\ H\ \mathbf{else}\ H\ \mathbf{fi} \mid \mathbf{while}\ T\ \mathbf{do}\ H\ \mathbf{od} \\ & & \mid \quad \mid [\mathbf{var}\ V : Set \bullet H\,] \mid \quad \mid H; H \end{array}$$

A pseudo-program S is well-formed if and only if:

1. For every subprogram $v := e$ occurring in S, the associated types of v and e are equal.

2. For every subprogram **if** g **then** S_1 **else** S_2 **fi** and **while** g **do** S_3 **od** occurring in S, the associated type of g is **bool**.

The language $While$ consists of all well-formed programs in H.

$PV(S)$ denotes the set of program variables of program S (i.e., the variables that are possibly altered during execution of the program, excluding locally defined variables).

Definition: Program Variables
The definition of PV is given by:

$$\begin{array}{lcl} PV(\mathbf{skip}) & = & \emptyset \\ PV(x := e) & = & \{x\} \\ PV(\mathbf{if}\ G\ \mathbf{then}\ S1\ \mathbf{else}\ S2\ \mathbf{fi}) & = & PV(S1) \cup PV(S2) \\ PV(\mathbf{while}\ G\ \mathbf{do}\ S\ \mathbf{od}) & = & PV(S) \\ PV(|[\mathbf{var}\ x : U \bullet S]|) & = & PV(S) \setminus \{x\} \\ PV(S1; S2) & = & PV(S1) \cup PV(S2) \end{array}$$

To define what it means for a well-formed program to be executed, we need the concept of a state.

A state is a mapping from program variables to values, just like the mapping of variables to values in an interpretation of a logical language. Thus, we consider a state to be a part of an interpretation of a logic.

Definition: State
Let $I = (\mathcal{D}, s)$ be an interpretation of the logic used to define $While$. Then s restricted to application on program variables is called a state.

The definition of While as given above differs a bit from usual language definitions in the literature. Usually, expressions and types of variables are defined explicitly and not, as above, defined using expressions and types of a logic. However, by linking the programming language to the logical language we avoid problems with expressibility (any expression in the programming language must have a corresponding expression in the specification language). Also, if we use a more powerful logic, we get a more powerful language automatically.

In [10], section 8.5, two important extensions of the language *While* are introduced. The first extension, arrays, allows new types to be constructed by the user. The second extension, procedures, allows the user to introduce (parametric) macros, which can be used in the actual program.

6. Hoare logic

Hoare triples express claims about the final state of a program related to the initial state of these programs. These claims are formulated by logical formulas rather than explicit states and hence, they deal with groups of states rather than single states.

The formula describing (properties of) the initial state is called the *precondition*. The formula describing (properties of) the final state is called the *postcondition*.

Given a precondition, a postcondition and a program we can denote two kinds of Hoare triples: those for partial correctness and those for total correctness. The difference is that in total correctness termination of the program is guaranteed, while partially correct programs may not terminate. We will only consider Hoare triples for partial correctness.

Definition: Hoare Triples for Partial Correctness
Let $\mathcal{L}$ be a logic with the sets Set, V and F as used to define *While*. Let P and Q be formulas of this logic and let S be a *While*-program. Then $\{P\}S\{Q\}$ is a Hoare triple expressing the partial correctness of S with respect to P and Q.

This kind of Hoare triple should be read as: if P holds in a state s and executing S in s yields s', then Q will hold in s'. Note that it is not claimed that a suitable s' exists, i.e. it is not claimed that S terminates.

A Hoare logic is an axiomatic derivation system to prove the validity of Hoare triples. To present the Hoare logic for *While*, we need Leibniz equality. Moreover, we need to express whether a boolean expression is mapped to *True* or *False* in an interpretation. Therefore, we assume that there are two constants **true** and **false** with s(**true**) = *True* and s(**false**) = *False*. The Hoare logic for *While* then consists of the following derivation rules:

Definition: Hoare Logic for While
The Hoare logic for *While* is defined by the derivation rules given in figure 2.
We briefly describe these rules:

skip Since **skip** does not change the state, the state after termination is equal to the one the execution started in. Hence, the same propositions will hold.

assign If P has to hold after the value of x is changed to the value of e, then $P[x := e]$ had to hold before this assignment was performed. One is often tempted to write this axiom as $\{P\}x := e\{P[x := e]\}$; but then $\{x = 0\}x := 1\{1 = 0\}$ and $\{x < 5\}x := x + 1\{x + 1 < 5\}$ would hold.

[skip] $$\{P\}\textbf{skip}\{P\}$$

[assign] $$\{P[x := e]\}x := e\{P\}$$

[if] $$\frac{\{P \wedge e = \textbf{true}\}S_1\{Q\} \quad \{P \wedge e = \textbf{false}\}S_2\{Q\}}{\{P\}\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\{Q\}}$$

[while] $$\frac{\{P \wedge e = \textbf{true}\}S\{P\}}{\{P\}\textbf{while } e \textbf{ do } S \textbf{ od}\{P \wedge e = \textbf{false}\}}$$

[block] $$\frac{\{P\}S\{Q\}}{\{P\}|[\textbf{var } x : U \bullet S]|\{Q\}} \quad \text{if } x \notin FV(P, Q)$$

[comp] $$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

[cons] $$\frac{\models P' \Rightarrow P \quad \{P\}S\{Q\} \quad \models Q \Rightarrow Q'}{\{P'\}S\{Q'\}}$$

Figure 2. The derivation rules of the Hoare logic for *While*.

if In any state in which P holds, e evaluates either to *True* of *False*. Depending on this, S_1 or S_2 will be executed respectively. From the premises we get that if S_1 is executed in a state in which P holds and e evaluates to *True*, then Q will hold in the resulting state. Similarly, Q will hold after executing S_2 from a state in which P holds and e evaluates to *False*. Hence, regardless of the value of e, Q will hold after execution of the if-statement.

while Operationally, the body S of the while-loop is executed as long as evaluation of the guard e yields *True*. The premise of this rule states that if S is executed in a state in which P holds and $e = \textbf{true}$, that P will still hold upon termination of S. Hence, P remains true, regardless of the number of executions of S. The while-loop ends when evaluating e yields *False*. Hence, upon termination we have both P and $e = \textbf{false}$.

block The block-statement introduces a local variable x. This variable can be used within S as an auxiliary variable, but is not used in the specification of S. In case the variable is ill-named one can use the following (provable) property of programs called α-conversion:

$$|[\textbf{var } x : U \bullet S]| \equiv |[\textbf{var } y : U \bullet S[x := y]]|$$

for any variable $y : U \notin FV(S)$.

comp The composition statement first executes S_1 and after that executes S_2. If Q holds after executing S_1 from a state in which P holds and R holds after executing S_2 from a state in which Q holds (i.e. the final state of S_1), then R holds after executing $S_1; S_2$ in a state in which P holds.

cons The logical premises claim that P holds in all states in which P' holds and that Q' holds in all states in which Q holds. Since Q holds after executing S from a state in which P holds, Q' will also hold after executing S from a state in which P holds. Hence, Q' will also hold after executing S from a state in which P' holds. This rule is known as 'the rule of consequence'.

A proof of the soundness of these rules can be found in [16].

7. Combining Hoare logic and $\lambda P-$

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

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

Stated this way, one assumes $\models$ to denote semantical 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. In case of first-order logic, one can use $\lambda P-$ to construct the required proofs. The rule of consequence then becomes:

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

where $\mathcal{L}$ is the logic used to define *While* and $\Gamma_{\mathcal{L}}$ is the corresponding context for $\lambda P-$ as defined earlier.

The advantages of this approach are:

- Programs are directly accessible by tools, since they are syntactical terms themselves.
- 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, since it is a propositional formula, not a boolean).
- The 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 complete Hoare logic, it still suffers from several drawbacks we encountered in the embeddings:

- Programs cannot be checked after they have been constructed. 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, $x := \mathit{fib}\,(N)$ is a valid, although undesirable, program.

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

Programs cannot be checked for correctness after they have been constructed, because the proofs used for application of the rule of consequence are usually not stored within the program and cannot be (re)constructed automatically. 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:

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

However, during program derivation, one usually alters only the precondition or the postcondition, not both at once. Also, since program S now has become embedded in $cons\ p\ S\ q$, it is less accessible to the tool. If one regards the change of P to P' as a re-formulation of a state-property, one could consider 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** p, which has the

same denotational semantics as **skip**, since the state does not change. The rule of consequence can now be replaced by the simpler rule:

$$\frac{\Gamma_{\mathcal{L}} \vdash p{:}P \Rightarrow Q}{\{P\}\textbf{fake}\ p\{Q\}}$$

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

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

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 be checked once they are constructed. This is nearly a trivial matter, since every statement can only be derived by a single rule, including the fake-statement. The premise of the fake-statement could also read $\Gamma \vdash p{:}P \Rightarrow Q$ for any other PTS, as long as this type judgment can be checked automatically.

However, this "type-checking" for programs has limited applicability: One could for instance derive a program

$$\{a = 4\}\textbf{fake}\ p;\ a := a + 1\{a = 5\},$$

where p is a proof of $(a = 4) \Rightarrow (a + 1 = 5)$. However, checking

$$\{a \geq 0\}\textbf{fake}\ p;\ a := a + 1\{a \geq 1\}$$

would fail, since the proof p stored in the fake-statement has the wrong type; instead we need a proof q of $(a \geq 0) \Rightarrow (a + 1 \geq 1)$.

Since our tool only needs to check if programs meet the specification for which they were derived, this is an acceptable restriction.

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 PTS 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: 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 $x := fib(N)$ are allowed, is that programs 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 PTS like $\lambda P-$, all function symbols of a logic L 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 accessible from the program should always be part of the context accessible from the specification though, since we might get expressibility problems otherwise. For instance, consider a function sqr, computing the square of a natural number, that only exists in the programming language context and not in the specification language context. Then the precondition of the assignment $x := sqr(2)$ with respect to the postcondition $x = 4$ reads $sqr(2) = 4$, but cannot be expressed in the specification.

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 (locally) defined program variables that can be altered by programs [1]. This 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 postconditions $x = \mathit{fib}(N)$. 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. $x = \mathit{fib}(N)$, where fib 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.

Hence, we will add triples of contexts to Hoare triples in the following manner:

Definition: Hoare Contexts
Let Γ_1, Γ_2 and Γ_3 be contexts of a CPTS as described earlier, then $\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$ is a Hoare context.

Note the following important differences:

1. $\langle \Gamma_1, \Gamma_2, \Gamma_3 \rangle$ is a triple of contexts, hence a Hoare context.

2. Since Γ_1, Γ_2 and Γ_3 are contexts of CPTSs, Γ_1, Γ_2 (the concatenation of Γ_1 and Γ_2, denoted without the triple-brackets) and $\Gamma_1, \Gamma_2, \Gamma_3$ (concatenation again) are CPTS-contexts too.

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

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

Spec This rule repeats the definition of a specification. It could be eliminated, but is used for consistency with the formal definition of PTSs. Note that the pre- and postcondition may depend on the entire context.

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 set-type as variable x. Moreover, x must occur in the program context and e may only depend on the language and program context.

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*.

Block Contexts play a main role here. The first premise claims that U is a set-type available to the program (i.e. U is derived from only language and program context). $P \triangleright Q$ must be a valid specification in the full context; but without the fresh variable x. S is a program, which may use a fresh variable x in its program context and which satisfies $P \triangleright Q$.

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

[1] The second context can also contain constants, function symbols etc, but by the definition of the abstract syntax of programs, these can never be altered by the program.

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

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

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

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

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

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

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

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

Figure 3. A Hoare logic with explicit contexts.

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.

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 the PTSs: programs can be checked for correctness once they are completed. This also enables the communication of those programs: programs which include **fake**-statement are self-contained and require no further proof of correctness.

Definition: 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 concepts:

Program synthesis: Given precondition P and postcondition Q, automatically find a program S, such that $\Gamma \vdash S{:}P \triangleright Q$.

Backward inference: Given a program S and a postcondition Q, automatically find a precondition P, such

that $\Gamma \vdash S{:}P \triangleright Q$.

Forward inference: Given a program S and a precondition P, automatically find a postcondition Q, such that $\Gamma \vdash S{:}P \triangleright Q$.

Specification inference: Given a program S, automatically find a precondition P and a postcondition Q, such that $\Gamma \vdash S{:}P \triangleright Q$.

Program checking: Given precondition P, postcondition Q and program S, automatically verify if $\Gamma \vdash S{:}P \triangleright Q$.

Theorem: Program synthesis, Backward inference, Forward inference, Specification inference and Program checking are decidable.

For a proof see [10], section 10.

8. Combining tableaux and $\lambda P-$

In [10], section 9, an algorithm is given to convert closed tableaux, generated by a tableau based Automated Theorem Prover (ATP), into λ-terms of $\lambda P-$. In turn, these λ-terms can easily be transformed into λ-terms of other PTSs, provided that these other PTSs are powerful enough. The closed tableau may be produced by any tableau-based theorem prover. This gives us the capability to use existing theorem provers as a module in an implementation of $\lambda P-$ and thereby adding powerful automated theorem proving to an interactive proof system, without the danger of extending our logic in an unforeseen way. If there is enough trust in the correctness of the implementation of the automatic theorem prover we can also use a special token to encode that the proof can be constructed using the ATP. We then do not have to actually convert the tableau and store the large λ-term that is the result of converting the tableau. The ATP can then reconstruct the tableau and convert it into a λ-term on request; for instance, if we want to communicate our proof to somebody using a different theorem prover based on λ-calculus. The translation of tableaux into λ-terms is also described in [11].

9. Summary

Because of practical reasons, the present Cocktail tool is not (yet) fully featured. However, it is powerful enough for educational purposes. The present Cocktail tool yields support for

- Many sorted first order predicate logic, represented by the Pure Type System with parametric Constants (CPTS) $\lambda P-$ as the specification language. This logic is well known amongst programmers and the PTS formalism allows future extensions. Also, in PTSs, proofs can be verified through type checking and the system will conform to the De Bruijn criterion.
- A simple $While$-language instead of a full Pascal-like language. This will suffice for educational purposes.
- Hoare logic, represented as a PTS, in order to be able to derive programs from their specifications.
- A simple tableau based automated theorem prover for verifying proof obligations. In order to achieve this, we convert closed tableaux into λ-terms of the PTS to allow them to be checked.

To keep the tool extendable, we have to implement it in a transparent manner. Therefore, we have chosen to design the tool modularly, such that parts of it can be studied, maintained and replaced independently of each other.

We now have a formal system combining interactive theorem proving with both automated theorem proving and program derivation. Moreover, the entire system is based on the semantics of first-order logic. Also, the correctness of both proofs and programs can be verified, even if parts of the proof were generated automatically.

$\lambda P-$ was designed to accurately describe first-order logic in a PTS, hence enabling the combination of interactive and automated theorem proving. Yet, since it is a PTS, proofs can be communicated to other PTS-based systems. Even the additional features are easily converted, provided that the target PTS is powerful enough to express the features axiomatically.

Translating tableaux into λ-terms showed some advantages: the automated theorem prover can be extended without extending the logic unexpectedly, provided that its result will remain an ordinary closed tableau. However, to incorporate Leibniz-equality, more work needs to be done. Although tableau methods dealing with equality are known (see [9, 8, 3]), we have not yet investigated how these tableaux can be translated into λ-terms.

The Hoare logic was designed to have properties similar to those of a PTS, which eased the combination of the two formalisms. Also, this enabled us to use a simple logic, rather than the higher order logics required to embed the entire Hoare logic. However, it is desirable to extend the Hoare logic with more advanced features like records, sub-typing, classes and pointers. Some of these features require an extension of the logic and hence, of $\lambda P-$. For instance, in [19] Jan Zwanenburg discussed PTSs with records and sub-typing. Richard Bornat is currently using JAPE to verify pointer semantics for Hoare Logic based on an idea of Rod Burstal (see [2, 4]).

Our goal was to create an educational tool as a proof of concept. Hence, we will not discuss further extensions of the formal basis of the system.

For the actual design and implementation of Cocktail we refer to part III of [10].

References

[1] Benthem Jutting, L.S. van and McKinna, J. and Pollack, R. 1994, *Checking Algorithms for Pure Type Systems*. In: Barendregt, H. and Nipkow, T. (eds.), Types for Proofs and Propositions: International Workshop TYPES'93, Springer, LNCS 806, 19–61.

[2] Bornat, R., 2000, *Proving Pointer Programs in Hoare Logic*. In: Proceedings of the fifth international conference on the Mathematics in Program Construction 2000, Springer, to appear.

[3] Bibel, W. and Schmitt, P. H. (eds.), 1998, *Automated Deduction - A Basis for Applications*, Kluwer Academic Publishers, Volume I : Foundations - Calculi and Methods, Applied Logic Series.

[4] Burstall, R.M., 1972, *Some techniques for proving correctness of programs which alter data structures*, Machine Intelligence **7**, 23–50.

[5] Coq, 1997, *The Coq Proof Assistant*, URL: `http://coq.inria.fr/`

[6] Dijkstra, Edsger W., 1976, *A Discipline of Programming*, Prentice-Hall International.

[7] Dijkstra, Edsger W. and Feijen, W.H.J., 1998, *A Method of Programming*, Addison-Wesley.

[8] D'Agostino, M. and Gabbay, D. and Hähnle, R. and Posegga, J. (eds.), 1999, *Handbook of Tableau Methods*, Kluwer Academic Publishers.

[9] De Kogel, E., 1995, *Equational Proofs in Tableaux and Logic Programming*, Ph.D.thesis, Tilburg University.

[10] Franssen, M., 2000, *Cocktail: A Tool for Deriving Correct Programs*, Ph.D. thesis, Eindhoven University of Technology.

[11] Michael Franssen, 2000, *Embedding First-Order Tableaux into a Pure Type System*. In: Galmiche, D., (ed.), Electronic Notes in Theoretical Computer Science, **17**, Elsevier Science Publishers.

[12] Gries, D., 1981, *The Science of Programming*, Springer.

[13] Kaldewaij, A., 1990, *Programming: the derivation of algorithms*, Prentice-Hall international series in Computer Science.

[14] Laan, T., 1997, *The Evolution of Type Theory in Logic and Mathematics*, Ph.D. thesis, Eindhoven University of Technology.

[15] Laan, T., and Franssen, M., 2001, *Embedding First-Order Logic in a Pure Type System with Parameters*, Journal of Logic and computation, **11**, 545–557.

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

[17] Paulin-Mohring, C., 1989, *Extracting F_ω's Programs from Proofs in the Calculus of Constructions*. In: Sixteenth Annual ACM symposium on Principles of Programming Languages, ACM press, Austin, Texas, 89–104.

[18] Zwanenburg, J., 1997, *The (Y)arrow Home Page*, URL: `http://www.cs.kun.nl/~janz/yarrow/`

[19] Zwanenburg, J., 1999, *Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow*, Ph.D. thesis, Eindhoven University of Technology.

Michael Franssen
Department of Computer Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven
The Netherlands
`m.franssen@tue.nl`

Harrie de Swart
Faculty of Philosophy
Tilburg University
P.O. Box 90153
5000 LE Tilburg
The Netherlands
`H.C.M.deSwart@uvt.nl`