Towards program development, specification and verification with Isabelle[†]

Marek A. Bednarczyk[*] Tomasz Borzyszkowski[**]

draft of August 18, 1995

Abstract

The purpose of this paper is to report on our experiments to use Isabelle — a generic theorem prover — as a universal environment within which specification, development and verification of imperative programs can be performed.

The use of a theorem prover for the programming tasks is most appropriate when the processes of program specification, development and verification can be presented as *logical activities*. In our case this is achieved by adopting pLSD — a novel programming logic.

Introduction

Hoare logic and its variants put several burdens on those who plan to use it for software specification, development and verification. Let us discuss the problems.

Example software task

There are three major programming tasks: specification, development and verification. Program specification task is when given a program we are faced with the problem of finding a specification for it. Program development task is when one is asked to find a program which satisfies a given specification. We talk about verification task when both, a program and a specification are given and the problem is to check if the former satisfies the latter.

Let us suppose that the task is to write a program $?p$ which computes multiplication of two integer values stored in variables x and y. The result should be stored in variable z.

Isabelle's notation $?p$ was used above to introduce a *metavariable* (schematic variable) which ranges over programs. We adopt the convention also to other syntactic classes.

[†]Research sponsored by Polish Committee for Scientific Research, Grant 1312/P3/92/02.

[*]Institute of Computer Science, P.A.S, Gdańsk Div., Abrahama 18, 81–825 Sopot, POLAND, tel./fax (58) 51 49 71, e-mail: M.Bednarczyk@ipipan.gda.pl

[**]Institute of Mathematics, Gdańsk University, Wita Stwosza 57, 80–308 Gdańsk, POLAND, e-mail mattb@halina.univ.gda.pl

Hoare logic In Hoare logic a program π is specified by two first order formulae called *assertions*, viz.: $\{\varphi\}\,\pi\,\{\psi\}$. The idea is that π started in a state satisfying φ should, whenever it terminates, return a state satisfying ψ. Thus, we concentrate on *partial correctness* interpretation of Hoare triples. The issue of *termination* is being considered a separate quality of programs, on par with program complexity, length, etc.

In Hoare logic one might try to write the following specification of the task:

$$\{\mathbf{tt}\}?p\,\{z = x * y\} \tag{1}$$

Above, $\mathbf{tt}$ stands for the always satisfied assertion. Thus, the constraint put by (1) on a candidate solution, call it π, is that π started in an arbitrary initial state, whenever terminates, should do so in a state in which the value of z equals the value of x multiplied by the value of y in that terminal state.

There are two problems with specification (1).

Firstly, it relates the value of z in a terminal state to the values of x and y in that state. But nothing prevents π from changing x or y. This allows 'clever' solutions, e.g., $x := 0; z := 0$. which contradict the intuition that (1) was meant to express the relationship between the initial values of x and y and the value stored in z upon termination.

Secondly, (1) seems *too* liberal. Knowledge that a given program π satisfies (1) gives not a single clue as to its effect upon variables other than z, and upon x and y in particular. Often one would like to consider program as a *black box*, and its specification as the only means of manipulation of its contents. Then, the less one knows about what is in the box, the less often one can use its contents.

One way to solve both problems is to use auxiliary *logical* variables, separate from program variables, to 'store' the initial values. Let capital letters range over the logical variables. The following specification says that the program of interest should not only change the value of z to the multiplication of the initial values of x and y, but also that the value stored in x does not change:

$$\{x = M \wedge y = N\}?p\,\{z = M * N \wedge x = M\} \tag{2}$$

Note, however, that (2) says nothing about y, nor about any other program variable appart from those two mentioned explicitly in the postcondition.

Commands-as-predicates approach Another method is to equip the language of assertions with the ability to refer to the values stored in a program variable in the initial and in a final state of computation. Hehner, cf. [23], would write $\grave{x}$ and $\acute{x}$, for the values stored in x initially and upon termination, respectively. Another convention is to use x and x', respectively. The latter convention is used within the Z! community. It is also advocated by Lamport, see [29], and adopted by Hehner, cf. [24].

In commands-as-predicates approach, and in accord to the latter convention, (2) is captured by:

$$\{\mathbf{tt}\}?p\,\{z' = x * y \wedge x' = x\} \tag{3}$$

Actually, Hehner would not write (3) as a formalisation of our programming task. Hehner, and also Hoare ([27]), investigated a system called *predicative programming*. In this approach each command is identified with a predicate of the extended language. *Programming* in this approach consists of logical manipulation of the specification so long that it becomes an image of a command.

The subtlety of the embedding of commands into predicates is that it takes the programing *environment* as a parameter. Clearly, each assertion ascribed to a command via the embedding should capture the *whole* meaning of the command. For example, assignment $x := x+1$ changes the value of variable x and leaves all the other intact. Thus, in an environment consisting of variables x, y and z, the command corresponds to predicate $x' = x + 1 \wedge y' = y \wedge z' = z$.

To avoid such cumbersome details the commands-as-predicates approach often goes with convention that all variables not mentioned in a specification do not change their values. The convention allows to hide the context. The price to pay is that now the logical formulae also have hidden agenda, e.g., $x' = x'$ is *not* equivalent to $\mathbf{tt}$.

This is the reason why the convention is not used by Lamport in his *Temporal Logic of Actions* — a recent ramification of commands-as-predicates approach to cope with reactive systems, cf. [29]. It should be mentioned that Lamport also uses logical variables, which he calls *rigid* in [29].

The remaining part of paper is organised as follows. Section 1 explains the sense in which substitutions can be used as logical formulae, and presents such a logic called PL_σ. Section 2 introduces logic pLSD in which formulae from PL_σ are used as specifications of imperative programs. Finally, Section 3 shortly describes our interpretation of pLSD in Isabelle.

1 Logic of predicates with explicit substitutions

Substitution plays a prominent role in Hoare logic. Its assignment axiom schema

$$\left\{\varphi\left[\frac{e}{x}\right]\right\} x := e \,\{\varphi\}$$

says that for φ to be a valid postassertion for assignment $x := e$ one should consider as a preassertion the result of performing substitution corresponding to the assignment on φ. From the consequence rule it follows that $\varphi\left[\frac{e}{x}\right]$ is the best preassertion.

In Hoare's axiom the schematic variable φ plays the role of a placeholder, a dummy. Intuitively, the assignment command is completely characterised by the substitution that corresponds to it. If substitutions were admitted as specifications we could write

$$x := e \;\; \textbf{\textit{sat}} \;\; \left[\frac{e}{x}\right].$$

and thereby get rid of the dummy.

We have argued in the introduction that one would want to write *simple* formulae which captures the meaning of a program completely, and independently of the programming context. We have also argued that a logical infrastructure is needed to express that changes of values of certain variables are restricted to a specified set of variables. It would be nice, in particular, if there existed a formula which captures that *nothing happens* to the state, i.e., that *all* variables retain their values.

Extending logic of predicates with *explicit substitutions* allows to achieve these aims.

1.1 Logic of predicates with explicit substitutions as a non-commutative substructural logic

Substitution is normally considered as part of the methatheory. This applies not only to logics, but to λ-calculae and type theories as well. It has been recently realized that more efficient implementations of functional languages can be achieved if one controls the process of performing a substitution. This calls for theories with substitution as a primitive operation.

Indeed, a variety of λ-calculae with explicit substitutions have already been considered, cf [1, 30]. All of them are 2-sorted — there are two syntactic classes: the old class of λ-*terms*, and the new class of *substitutions*.

In case of predicate logic there already are two sorts: *terms* and *formulae*. From the discussion above it follows that we want to consider substitutions as specifications. Thus, while extending the logic of predicates with explicit substitutions we do not add a third sort for them. Instead, substitutions are considered as a new class of atomic formulae. From now on the old

atoms, i.e., the predicates, are called *Platonic*, while substitutions are referred to as *dynamic* atoms.

With predicates and substitutions considered as atoms a new logical connective is needed to express the process of application of a substitution to a predicate. Let us use the tensor symbol $\otimes$ to denote the new connective. Now we can write $\sigma \otimes \sigma'$ to express (pending) composition of substitutions σ and σ'.

We can, it turns out, consider $\otimes$ as a *non-commutative* multiplicative conjunction. Having said that we can also add the multiplicative truth, denoted I. It turns out that I captures the idea that "nothing happens". Thus, I is a good specification for a "do nothing" program.

Altogether, we are led to take as the logic of specifications a fragment of *intuitionistic non-commutative linear logic*. This idea was first put forward in [7]. The logic is referred to as PL_σ. Formally, formulae of PL_σ, called *specifications*, are given by the following grammar.

A	$= \sigma$		substitutions, the *dynamic* atomic formulae
	$\mid a$		predicates (including equality), the *Platonic* atomic formulae
	$\mid A \otimes A$	$\mid I$	multiplicative conjunction and truth
	$\mid A \wedge A$	$\mid \top$	additive conjunction and truth
	$\mid A \vee A$	$\mid \bot$	additive disjunction and false

A sequent-style presentation of the logic is given in Table 1.

$$\text{(Refl)}\ \frac{}{A \vdash A} \qquad \text{(Cut)}\ \frac{\Gamma \vdash A \qquad \Delta, A, \Delta' \vdash B}{\Delta, \Gamma, \Delta' \vdash B}$$

$$\text{(L}I\text{)}\ \frac{\Gamma, \Delta \vdash A}{\Gamma, I, \Delta \vdash A} \qquad \text{(R}I\text{)}\ \frac{}{\vdash I}$$

$$\text{(L}\otimes\text{)}\ \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \otimes B, \Delta \vdash C} \qquad \text{(R}\otimes\text{)}\ \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}$$

$$\text{(L}\wedge\text{-L)}\ \frac{\Gamma, A, \Delta \vdash C}{\Gamma, A \wedge B, \Delta \vdash C} \qquad (\top)\ \frac{}{\Gamma \vdash \top}$$

$$\text{(L}\wedge\text{-R)}\ \frac{\Gamma, B, \Delta \vdash C}{\Gamma, A \wedge B, \Delta \vdash C} \qquad \text{(R}\wedge\text{)}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

$$(\bot)\ \frac{}{\bot, \Gamma \vdash A} \qquad \text{(L}\vee\text{)}\ \frac{\Gamma, A, \Delta \vdash C \qquad \Gamma, B, \Delta \vdash C}{\Gamma, A \vee B, \Delta \vdash C}$$

$$\text{(R}\vee\text{-L)}\ \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \text{(R}\vee\text{-R)}\ \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

Table 1: A sequent system $\mathcal{L}$ for PL_σ.

With one exception, the rules in Table 1 are the natural generalisations of the rules suggested by Girard for the *commutative* intuitionistic linear logic, cf. [19] and [20, 21], to the non-commutative case, cf [14, 15].

Axiom $(\bot)$ is the only exception — the expected generalisation is $\Gamma, \bot, \Delta \vdash A$, as in [14, 15]. However, the stronger axiom is not valid in our intended interpretation in *quasi quantales*, cf. [8].

Actually, quasi quantales were invented by Blikle in the 70's. Firstly, see [10], under the name of *nets* Blike reinvented quantales to study mathematical behaviour of commands. Subsequently, cf. [11], he begun investigation of *quasi nets*, i.e., our quasi quantales. In an attempt to give denotational account of divergence, as opposed to failure, he allowed the composition relation to

be non-strict on the right. The most comprehensive account on quasi nets and their applications, including solving recursive equations in nets, can be found in [12].

1.2 PL_σ — a linear theory

There are numerous reasons which force us to work with theories rather than the pure logic generated by system $\mathcal{L}$.

1.2.1 Platonic sublogic

Programs are supposed to manipulate data which come from a specific domain. Hence, we must cope with axioms which charaterise the domain of data. This calls for describing the way in which the usual logic of predicates can be embedded into PL_σ.

The reader clearly noticed the lack of non-monotone connectives like implication and negation in PL_σ. Since tensor is non-commutative there could be two implications in principle. Even if both existed[1] none of them would semantically correspond to the usual implication. A similar argument applies to the Platonic negation.

But then the problem is that without negation a lot of expressive power is lost. The usual way around the problem is to consider affirmative/positive and refutative/negative predicates for each predicate symbol. More specifically, given a predicate symbol R of arity k we consider two predicates. The first affirmative, $R^+(e_1,\ldots,e_k)$, affirms that R holds on e_1, ..., e_k. The other refutative, $R^-(e_1,\ldots,e_k)$, states the converse. In case of equality predicate this is usually written: $e_1 = e_2$ –an affirmative atom, and $e_1 \neq e_2$ –the corresponding refutative atom.

Now negation, and hence implication as well, becomes operation definiable on the *Platonic sublogic* of PL_σ, i.e., on formulae built without $\otimes$ and I.

$$\begin{array}{rclcrcl} \neg R^+(e_1,\ldots,e_k) & = & R^-(e_1,\ldots,e_k) & \quad & \neg R^-(e_1,\ldots,e_k) & = & R^+(e_1,\ldots,e_k) \\ \neg(\phi \wedge \psi) & = & (\neg\phi) \vee (\neg\psi) & & \neg\top & = & \bot \\ \neg(\phi \vee \psi) & = & (\neg\phi) \wedge (\neg\psi) & & \neg\bot & = & \top \end{array}$$

Missing are the following axioms which state that the affirmative is an opposite of the refutative, and vice versa. In $R^*(e_1,\ldots,e_k)$ bellow $*$ stands either for $+$ or for $-$.

$$R^*(e_1,\ldots,e_k) \wedge \neg R^*(e_1,\ldots,e_k) \vdash \bot \quad \textit{and} \quad \top \vdash R^*(e_1,\ldots,e_k) \vee \neg R^*(e_1,\ldots,e_k)$$

Thus, the Platonic sublogic is classical.

In first order logic the conjunction distributes over disjunction, and vice versa. Since the additive conjunction — the one corresponding to the conjunction of the predicate logic — does not, in general, distribute over additive disjunction, we have to add the missing axioms:

$$A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C) \qquad A \vee (B \wedge C) \dashv (A \vee B) \wedge (A \vee C)$$

The translation of the reflexivity of equality axiom schema to PL_σ is straightforward.

$$\top \vdash e = e$$

Xxioms characterising a particular domain of data are translated in the same straightforward way. For instance, to state that the data are strictly linearly ordered by $<$ one would accept the following axioms.

$$x \neq y \vdash x < y \vee y < x \quad \textit{and} \quad x < y \vee y < x \vdash x \neq y$$

[1] One of the implications is not definiable in the intended model, cf. [8].

1.2.2 Dynamic axioms

The other facet of equality, that equals can be substituted for equals, brings us to the dynamic part of PL_σ. *Finitary simultaneous substitutions*, ranged over by σ and ς, have the form

$$\left[\frac{e_1}{x_1}, \ldots, \frac{e_n}{x_n}\right]$$

The main idea behind tensor is that it codes substitution on a predicate

$$\sigma \otimes R^\star(e_1, \ldots, e_k) \dashv\vdash R^\star(e_1\sigma, \ldots, e_k\sigma)$$

and also composition of two substitutions,

$$\sigma \otimes \varsigma \dashv\vdash \sigma \circ \varsigma$$

where $\sigma \circ \varsigma$ denotes the result of composition of the two substitutions.

We also declare that the identity substitutions are multiplicative truth.

$$\left[\frac{x}{x}\right] \dashv\vdash I \quad \textit{and} \quad I \dashv\vdash [\,]$$

Next, we need a number of axioms to ensure that, effectively, substitution distributes over all finite (equivalently, over all binary and nullary) additive conjunctions and disjunctions. Some of the distributivity properties follow from the logical rules. Those missing are listed below.

$$\top \vdash \sigma \otimes \top \quad \textit{and} \quad \sigma \otimes \bot \vdash \bot$$

$$(\sigma \otimes A) \wedge (\sigma \otimes B) \vdash \sigma \otimes (A \wedge B) \quad \textit{and} \quad (A \otimes C) \wedge (B \otimes C) \vdash (A \wedge B) \otimes C$$

Thus, the axioms ensure that $\sigma \otimes \varphi$, with φ Platonic, is equivalent to a Platonic formula.

The question that remains is what it means for a predicate to *act* upon a formula like in $(x = y) \otimes B$. The answer is that it is catastrophic for B: its role is totally neglected. Namely, for any Platonic φ and any B we have

$$\varphi \dashv\vdash \varphi \otimes B.$$

This is a consequence of accepting the following axioms.

$$R^*(e_1, \ldots, e_k) \vdash R^*(e_1, \ldots, e_k) \otimes \bot \quad \textit{and} \quad \top \vdash \top \otimes \bot$$

Semantically, the formulae of PL_σ are interpreted as *predicate transformers*, cf. [8]. The formulae of the Platonic sublanguage are interpreted as constant predicate transformers.

Finally, we accept an axiom which explicitly expresses the meta-property of substitutivity: equals may be substituted for equals.

$$e_1 = e_2 \wedge \left[\frac{e_1}{x}\right] \vdash \left[\frac{e_2}{x}\right]$$

2 pLSD — a logic for software specification, development and verification

Let us turn now to our logic pLSD. Its connections with Hoare logic are described elsewhere, cf. [9]. Here, we concentrate on its utility for program development.

pLSD is a Labelled Deductive System, cf. [18], built over PL_σ. Declarative units (judgements) of pLSD are of the form

$$\pi \ \textbf{\textit{sat}} \ A$$

where π is an imperative program while A is a PL_σ formula.

Atomic programs include: *indeterminate programs*, denoted p, q; *assignments*, denoted α; and "do-nothing" program **skip**. More complex programs can be built from already constructed programs π and ϖ using one of the following operations: binary *sequential composition*, $\pi;\varpi$; binary *conditional composition*, $\mathsf{if}\ b\ \mathsf{then}\ \pi\ \mathsf{else}\ \varpi\ \mathsf{fi}$; and unary *loop*, $\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}$. The boolean conditions b mentioned above are Platonic formulae.

The tarskian consequence relation $\models$ of pLSD has sequents of the form

$$\mathsf{p}_1\ \boldsymbol{sat}\ A_1, \ldots, \mathsf{p}_n\ \boldsymbol{sat}\ A_n \models \pi\ \boldsymbol{sat}\ A$$

where p_i's are pairwise different. The intended meaning of the sequent is that if all indeterminate programs $\mathsf{p}_1, \ldots, \mathsf{p}_n$ satisfy their respective specifications, then π satisfies A.

In the following sequent style presentation of pLSD metavariables $\mathcal{C}$ and $\mathcal{D}$ range over sequences of satisfaction assumptions about pairwise different indeterminate programs. $\mathsf{PI}(\mathcal{C})$ is used to denote the set of all indeterminate programs in $\mathcal{C}$.

Structural Rules

Reflexivity

$$\frac{}{\mathsf{p}\ \boldsymbol{sat}\ A \models \mathsf{p}\ \boldsymbol{sat}\ A}$$

CUT

$$\frac{\mathcal{C} \models \pi\ \boldsymbol{sat}\ B \quad \mathsf{p}\ \boldsymbol{sat}\ B, \mathcal{D} \models \varpi\ \boldsymbol{sat}\ A}{\mathcal{C}, \mathcal{D} \models \varpi[\pi/\mathsf{p}]\ \boldsymbol{sat}\ A}\ \mathsf{PI}(\mathcal{C}) \cap \mathsf{PI}(\mathcal{D}) = \emptyset$$

Exchange

$$\frac{\mathcal{C}, \mathsf{p}\ \boldsymbol{sat}\ A, \mathsf{q}\ \boldsymbol{sat}\ B, \mathcal{D} \models \pi\ \boldsymbol{sat}\ C}{\mathcal{C}, \mathsf{q}\ \boldsymbol{sat}\ B, \mathsf{p}\ \boldsymbol{sat}\ A, \mathcal{D} \models \pi\ \boldsymbol{sat}\ C}$$

Weakening

$$\frac{\mathcal{C} \models \pi\ \boldsymbol{sat}\ B}{\mathcal{C}, \mathsf{p}\ \boldsymbol{sat}\ A \models \pi\ \boldsymbol{sat}\ B}\ \mathsf{p} \notin \mathsf{PI}(\mathcal{C})$$

Contraction

$$\frac{\mathcal{C}, \mathsf{p}_1\ \boldsymbol{sat}\ A, \mathsf{p}_2\ \boldsymbol{sat}\ A \models \pi\ \boldsymbol{sat}\ B}{\mathcal{C}, \mathsf{p}\ \boldsymbol{sat}\ A \models \pi[\mathsf{p}/\mathsf{p}_1][\mathsf{p}/\mathsf{p}_2]\ \boldsymbol{sat}\ B}\ \mathsf{p} \notin \mathsf{PI}(\mathcal{C})$$

Interaction Rule

Consequence

$$\frac{\mathcal{C} \models \pi\ \boldsymbol{sat}\ A \qquad A \dashv B}{\mathcal{C} \models \pi\ \boldsymbol{sat}\ B}$$

Connective Rules

Assignment

$$\frac{}{\models \alpha\ \boldsymbol{sat}\ [\alpha]}$$

Skip

$$\frac{}{\models \mathsf{skip}\ \boldsymbol{sat}\ I}$$

Sequential composition

$$\frac{\mathcal{C} \models \pi\ \boldsymbol{sat}\ A \qquad \mathcal{C} \models \varpi\ \boldsymbol{sat}\ B}{\mathcal{C} \models \pi;\varpi\ \boldsymbol{sat}\ A \otimes B}$$

Conditional

$$\frac{\mathcal{C} \models \pi\ \boldsymbol{sat}\ A \qquad \mathcal{C} \models \varpi\ \boldsymbol{sat}\ B}{\mathcal{C} \models \mathsf{if}\ b\ \mathsf{then}\ \pi\ \mathsf{else}\ \varpi\ \mathsf{fi}\ \ \boldsymbol{sat}\ (b \wedge A) \vee (\neg b \wedge B)}$$

Loop

$$\frac{\mathcal{C} \Vdash \pi \ \textbf{\textit{sat}}\ B \qquad b \wedge A \vdash B \otimes A \qquad \neg b \wedge A \vdash I}{\mathcal{C} \Vdash \textsf{while}\ b\ \textsf{do}\ \pi\ \textsf{od}\ \textbf{\textit{sat}}\ A}$$

2.1 pLSD — a programming logic over PL_σ

How formulae of PL_σ, and explicit substitutions in particular, can be used as specifications? To answer that question let us turn to the programming task considered in the introduction. It can be now specified as follows.

$$?p \ \textbf{\textit{sat}} \ \left[\frac{?t}{y}, \frac{x * y}{z}\right] \tag{4}$$

The task is to find a program which, whenever terminates, sets the values of variables according to substitution $\left[\frac{?t}{y}, \frac{x*y}{z}\right]$. More specifically it says the following.

- The terminal value of z is equal to the multiplication of the initial values of x and of y.
- The value stored in y may get changed. The metavariable $?t$ ranging over terms can be replaced by an arbitrary term. Thus, we do not care about the final value of y.
- Variables other than z and y should *not* change their values.

There is an obvious similarity between (3) and (4). The use of primes in (3) is compensated by the use of explicit substitutions in (4). But we do not need to talk about two different incarnations of each variable.

Note also that in (4) it must be explicitly mentioned that we 'do not care' how variable y is set. In (3) this is implicit.

Other fundamental difference is revealed when it comes to consider the ways in which the different specification languages are used. A key question: Given a specification, how to find a program that fulfills it? Our aim is to show that program development is a *logical* activity governed by pLSD.

2.1.1 Trivial solution — assignment axiom

The key idea is that each assignment command corresponds to a substitution. If notational differences are neglected one can simply identify assignments, ranged over by α, with a subclass of substitutions, ranged over by σ. Let $[\alpha]$ denote the substitution corresponding to assignment α. Thus, $[x := 0]$ equals $\left[\frac{0}{x}\right]$.

Now, (4) can be solved by an assignment command provided substitution $\left[\frac{?t}{y}, \frac{x*y}{z}\right]$ is an assignment expressible in the programming language. This would be the case if the programming language allows simultaneous assignments, and if expressions of the form $x * y$ are expressible. Then one can take program $(y, z) := (?t, x * y)$. The corresponding (instance of) the pLSD assignment axiom is

$$\frac{}{\Vdash \ (y, z) := (?t, x * y) \ \textbf{\textit{sat}} \ \left[\frac{?t}{y}, \frac{x*y}{z}\right]}.$$

Let us compare the pLSD assignment axiom (left) with its counterpart in Hoare logic (right).

$$\frac{}{\Vdash \ \alpha \ \textbf{\textit{sat}} \ [\alpha]} \qquad\qquad \frac{}{\{\varphi\,[\alpha]\}\,\alpha\,\{\varphi\}}$$

The analogy should be clear. Hoare characterises assignment α by the substitution $[\alpha]$ performed on a dummy assertion φ. Our axiom is obtained by getting rid of the dummy.

2.1.2 Simple solution — sequential composition and consequence rules

Programming tools are constructive. Those available in specifications need not be constructive. Specifications are always more expressive — otherwise there would be no sense in separating program development phase from program specification phase. Hence, programming is an art of fulfilling specifications with limited programming tools.

Consequently, let us assume that our programming language is limited in that it disallows simultaneous assignments. Assumption that multiplication can be used to build valid expression of the language still holds. Then a slightly more complex program, namely $z := x * y; y :=?s$, adheres to the restriction, and is a solution of our task.

Indeed, assignment axiom and sequential composition rule applied to the program give the following derivation.

$$\frac{\overline{\Vdash\ z := x * y\ \ \boldsymbol{sat}\ \left[\frac{x*y}{z}\right]} \qquad \overline{\Vdash\ y :=?s\ \ \boldsymbol{sat}\ \left[\frac{?s}{y}\right]}}{\Vdash\ z := x * y; y :=?s\ \ \boldsymbol{sat}\ \left[\frac{x*y}{z}\right] \otimes \left[\frac{?s}{y}\right]} \tag{5}$$

Let us compare the pLSD sequential composition rule with its counterpart in Hoare logic.

$$\frac{\mathcal{C} \Vdash \pi\ \ \boldsymbol{sat}\ A \qquad \mathcal{C} \Vdash \varpi\ \ \boldsymbol{sat}\ B}{\mathcal{C} \Vdash \pi; \varpi\ \ \boldsymbol{sat}\ A \otimes B} \qquad\qquad \frac{\{\varphi\}\,\pi\,\{\phi\} \qquad \{\phi\}\,\varpi\,\{\psi\}}{\{\varphi\}\,\pi; \varpi\,\{\psi\}}$$

Sequential composition rule in Hoare logic has serious deficiency. In backward proofs it requires, just like CUT, inventing a lemma (assertion ϕ above). In forward proofs it imposes restriction that the postassertion of π matches preassertion of ϖ. In pLSD sequential composition of programs is characterised by multiplicative conjunction of their specifications.

We are not done yet. The consequence rule (6) introduces *proof obligations* to pLSD deductions. Thereby it allows to manipulate the specifications on PL_σ-level.

$$\frac{\mathcal{C} \Vdash \pi\ \ \boldsymbol{sat}\ A \qquad A \dashv B}{\mathcal{C} \Vdash \pi\ \ \boldsymbol{sat}\ B} \tag{6}$$

Notice the contravariance beteen pLSD and PL_σ in (6). It comes about from the fact that pLSD is based on weakest precondition rather than strongest postcondition predicate transformers.

It has been argued in section 1 that tensor applied to two substitutions corresponds to their composition. Thus, by the relevant PL_σ axiom one obtains the following.

$$\overline{\left[\frac{?s\left[\frac{x*y}{z}\right]}{y}, \frac{x*y}{z}\right] \vdash \left[\frac{x*y}{z}\right] \otimes \left[\frac{?t}{y}\right]}$$

So, consequence rule (6) applied to derivation (5) gives finally the following.

$$\frac{(5) \qquad \overline{\left[\frac{?s\left[\frac{x*y}{z}\right]}{y}, \frac{x*y}{z}\right] \vdash \left[\frac{x*y}{z}\right] \otimes \left[\frac{?s}{y}\right]}}{\Vdash\ z := x * y; y :=?s\ \ \boldsymbol{sat}\ \left[\frac{?s\left[\frac{x*y}{z}\right]}{y}, \frac{x*y}{z}\right]}$$

Thus $z := x * y; y :=?s$ satisfies $\left[\frac{?t}{y}, \frac{x*y}{z}\right]$ for $?t =?s\left[\frac{x*y}{z}\right]$, as required.

2.1.3 Problem decomposition — CUT rule

In general the programming language does not support directly the operations required by the specification. In terms of our programming task (4) this would mean that the language neither allows simultaneous assignments, nor expressions of the form $x * y$. Assumption that

multiplication is not defined may seem funny, but it serves as an example here. Also, on appropriatly low level one has to implement multiplication in terms of more primitive operations. Consequently, let us assume that the language allows expressions of the form $x + y$, $x - y$ and $-x$ with their usual meaning.

At first there is nothing we could do. Then comes an idea that perhaps we could solve (4) under additional assumptions. Then, perhaps, we could somehow meet the assumptions. All this is rather vague. More formal explanation of this vague idea is this.

Given a programming task of the form

$$\Vdash ?p \ \textbf{\textit{sat}} \ A$$

do the following.

1. *Guess* another specification B, somehow related to the original problem A.
2. Then, separately, and independently:
   (a) find in a library or construct a program π that satisfies B;
   (b) working under assumption that $\mathsf{p} \ \textbf{\textit{sat}} \ B$, find in a library or construct a program ϖ (over p) which satisfies the original specification A;
3. Build a program that satisfies the original specification by replacing all occurrences of p in ϖ by π.

Diagramatically the above can be expressed in the form of the following instance of CUT.

$$\frac{\Vdash \ \pi \ \textbf{\textit{sat}} \ B \qquad \mathsf{p} \ \textbf{\textit{sat}} \ B \Vdash \varpi \ \textbf{\textit{sat}} \ A}{\Vdash \ \varpi\,[\pi/\mathsf{p}] \ \textbf{\textit{sat}} \ A}$$

Solving the derived task $\mathsf{p} \ \textbf{\textit{sat}} \ B \Vdash \varpi \ \textbf{\textit{sat}} \ A$ might require another decomposition. Thus, it is only natural to accept CUT in full generality.

$$\frac{\mathcal{C} \Vdash \ \pi \ \textbf{\textit{sat}} \ B \qquad \mathsf{p} \ \textbf{\textit{sat}} \ B, \mathcal{D} \Vdash \varpi \ \textbf{\textit{sat}} \ A}{\mathcal{C}, \mathcal{D} \Vdash \ \varpi\,[\pi/\mathsf{p}] \ \textbf{\textit{sat}} \ A}$$

This step turns pLSD into a logic, i.e., into a consequence relation. Now, the following *logical* characterisation of the nature of *modularity* in program development can be offered.

$$\boxed{\boxed{\text{CUT} = \text{MODULARITY}}}$$

It is fair to say, it seems, that neither Hoare logic, nor any of its variants has addressed the issue of modularity of program development in the way just described.

Let us decompose our programming task. The idea is that the solution may be found more easily if we assume that $y \geq 0$. Thus, let us take as the lemma the following strengthened version of our specification.

$$L \stackrel{\text{def}}{=} y \geq 0 \wedge \varsigma \qquad \text{where} \qquad \varsigma \stackrel{\text{def}}{=} \left[\frac{?t}{y}, \frac{x * y}{z}\right] \tag{7}$$

2.1.4 Solving the parametric subproblem — reflexivity axiom and conditional rule

Now let us move to realize the step 2b of the decomposition scenario. In particular we have to find a program ϖ (parametric over p) which satisfies:

$$\mathsf{p} \ \textbf{\textit{sat}} \ y \geq 0 \wedge \varsigma \Vdash ?q \ \textbf{\textit{sat}} \ \varsigma \tag{8}$$

Either $y \geq 0$ or $y \not\geq 0$ always holds. Thus, $\wedge\vee$-distributivity gives the following PL_σ equivalence.

$$\varsigma \dashv\vdash (y \geq 0 \wedge \varsigma) \vee (y \not\geq 0 \wedge \varsigma) \dashv\vdash (y \geq 0 \wedge y \geq 0 \wedge \varsigma) \vee (y \not\geq 0 \wedge y \not\geq 0 \wedge \varsigma) \tag{9}$$

The conditional command is characterised by the following rule.

$$\frac{\mathcal{C} \Vdash \pi \ \boldsymbol{sat}\ A \qquad \mathcal{C} \Vdash \varpi \ \boldsymbol{sat}\ B}{\mathcal{C} \Vdash \mathsf{if}\ b\ \mathsf{then}\ \pi\ \mathsf{else}\ \varpi\ \mathsf{fi}\ \ \boldsymbol{sat}\ (b \wedge A) \vee (\neg b \wedge B)} \tag{10}$$

Thus, an application of consequence and conditional rules gives the following derivation.

$$\frac{\dfrac{\mathsf{p}\ \boldsymbol{sat}\ L \Vdash ?q_1\ \boldsymbol{sat}\ L \qquad \mathsf{p}\ \boldsymbol{sat}\ L \Vdash ?q_2\ \boldsymbol{sat}\ y \not\geq 0 \wedge \varsigma}{\mathsf{p}\ \boldsymbol{sat}\ L \Vdash \mathsf{if}\ y \geq 0\ \mathsf{then}\ ?q_1\ \mathsf{else}\ ?q_2\ \mathsf{fi}\ \ \boldsymbol{sat}\ (y \geq 0 \wedge L) \vee (y \not\geq 0 \wedge y \not\geq 0 \wedge \varsigma)}\ (9)}{\mathsf{p}\ \boldsymbol{sat}\ L \Vdash \mathsf{if}\ y \geq 0\ \mathsf{then}\ ?q_1\ \mathsf{else}\ ?q_2\ \mathsf{fi}\ \ \boldsymbol{sat}\ \varsigma} \tag{11}$$

At this point the task is reduced to two subgoals.

The first one, $\mathsf{p}\ \boldsymbol{sat}\ L \Vdash ?q_1\ \boldsymbol{sat}\ L$, is solved by taking the hypothetical p for $?q_1$. This makes the subgoal an instance of pLSD's reflexivity axiom.

$$\mathsf{p}\ \boldsymbol{sat}\ A \Vdash \mathsf{p}\ \boldsymbol{sat}\ A$$

The other subgoal, $\mathsf{p}\ \boldsymbol{sat}\ L \Vdash ?q_2\ \boldsymbol{sat}\ y \not\geq 0 \wedge \varsigma$, requires some effort. If one negates the value of y and applies p then it remains to modify the output stored in z. Without going into details we state the following lemma.

Lemma 1 *Program* $\pi_2 \stackrel{\text{def}}{=} y := -y; \mathsf{p}; z := -z$ *satisfies parametric specification*

$$\mathsf{p}\ \boldsymbol{sat}\ y \geq 0 \wedge \left[\frac{?t}{y}, \frac{x * y}{z}\right] \Vdash ?q_2\ \boldsymbol{sat}\ y \not\geq 0 \wedge \left[\frac{?t}{y}, \frac{x * y}{z}\right]$$

provided $?t = ?t\left[\frac{-y}{y}\right]$.

The little constraint $?t = ?t\left[\frac{-y}{y}\right]$ has to be met by the other thread of our program development. Conversely, this solution has to meet all constraints developed in the other thread.

2.1.5 Proving the lemma — loop rule

Now, with accord to step 2a, we have to find a program π which fulfills the following task.

$$\Vdash ?p_1\ \boldsymbol{sat}\ y \geq 0 \wedge \left[\frac{?t}{y}, \frac{x * y}{z}\right]$$

The idea is to implement multiplication by y-times iterated addition of x to z. The value of z should, of course, be initialized to the neutral element of addition. The program described above may look as follows.

$$z := 0; \mathsf{while}\ y > 0\ \mathsf{do}\ z := x + z; y := y - 1\ \mathsf{od}$$

Dealing with a loop calls for an application of pLSD's loop rule. Below, it is compared with its counterpart in Hoare logic.

$$\frac{\mathcal{C} \Vdash \pi\ \boldsymbol{sat}\ B \qquad b \wedge A \vdash B \otimes A \qquad \neg b \wedge A \vdash I}{\mathcal{C} \Vdash \mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}\ \ \boldsymbol{sat}\ A} \qquad \frac{\{b \wedge \varphi\}\,\pi\,\{\varphi\}}{\{\varphi\}\,\mathsf{while}\ b\ \mathsf{do}\ \pi\ \mathsf{od}\,\{\neg b \wedge \varphi\}} \tag{12}$$

pLSD's loop rule has one aspect similar to pLSD's consequence rule — both have PL_σ sequents as premisses. In the consequence rule the PL_σ sequents allowed to manipulate the specification. This time, however, the PL_σ sequents are best considered *proof obligations*.

There is an important pragmatic difference between the rools. The one proposed by Hoare requires to find an *invariant* of the body of the loop. Its pLSD counterpart is more elaborate. Invariance of A with respect to the body π is split into two subtasks: finding a specification B of the body and proving A's invariance with respect to B under assumption $\neg b$. Another subgoal is to show *invariability* of A under assumption $\neg b$.

The following result demonstrates that the loop sugested above satisfies an appropriate invariant. Notice that the invariablity obligation generates constraint $?t = 0$.

Theorem 2

$$\Vdash \textsf{while } y > 0 \textsf{ do } z := x + z; y := y - 1 \textsf{ od } \textbf{\textit{sat}}\ y \geq 0 \wedge \left[\frac{0}{y}, \frac{x * y + z}{z}\right]$$

From the Theorem 2, pLSD's sequential composition rule and consequence rule it follows that the following can be derived.

$$\Vdash\ z :=?t; \textsf{while } y > 0 \textsf{ do } z := x + z; y := y - 1 \textsf{ od } \textbf{\textit{sat}}\ y \geq 0 \wedge \left[\frac{0}{y}, \frac{x * y}{z}\right]$$

2.1.6 CUT, ... then paste

The final step is to combine the results obtained in both threads and to create solution of the initial task. At this stage constraints collected in the two threads have to be verified. The second thread requires $?t$ to be equal to 0. Luckily, this constraint is in no conflict with constraint of the first thread since $0 = 0\left[\frac{-y}{y}\right]$.

Thus, pLSD verifies that there exists a solution to the original specification (4).

$$\begin{array}{l}\textsf{if } y \geq 0 \textsf{ then } z := 0; \\ \qquad\qquad \textsf{while } y > 0 \textsf{ do } (z := x + z; y := y - 1) \textsf{ od} \\ \textsf{else } (y := -y; z := 0; \\ \quad \textsf{while } y > 0 \textsf{ do } (z := x + z; y := y - 1) \textsf{ od}; z := -z) \\ \textsf{fi}\end{array} \qquad \textbf{\textit{sat}}\ \left[\frac{0}{y}, \frac{x * y}{z}\right]$$

2.2 pLSD — summary of main features

Let us conclude this section by recalling the two main features that make pLSD different from other variants of Hoare logic.

First, in pLSD the meaning of basic programs is fully captured by a single specification which is independent of the programming environment.

Second, pLSD is a logic whose construction is parameterised with a logic of specifications. The logic of specifications is responsible for manipulation of specifications. At the level of pLSD we are talking about the *logic of program development*.

This feature can also be found in Abramsky's approach to programming logics, see [3], and its specialisations, see e.g., [2, 13, 25]. But, for obvious reasons, pLSD cannot be seen as a specialisation of Abramsky's framework: the later was formulated for propositional logics and it does not allow linear theories on any level.

3 Towards an implementation of pLSD in Isabelle

Why Isabelle? As a general purpose theorem prover Isabelle provides a promissing framework for implementation of pLSD.

Firstly, it supports backward proof method which seems most natural method of program specification, verification and development.

Secondly, it allows to store the results of deriving parametric programs as theorems of the form $\mathtt{p}_1$ ***sat*** $A_1, \ldots, \mathtt{p}_n$ ***sat*** $A_n \Vdash \pi$ ***sat*** A. Therefore forward proof method can also be applied, and ieven used in conjunction with backward proofs.

Thirdly, it allows quite natural interpretation of the many constituents (logics, axiomatizations) of pLSD. The only minor (?) exception to this picture is posed by substitution.

Finally, it allows to specialize logical tools to each theory contributing to interpretation of pLSD in Isabelle. We have not made much progress in that direction yet.

3.1 The shape of theories

The following graph presents the hierarchy of our theories which constitute interpretation of pLSD in Isabelle.

$$\texttt{Pure} \rightarrow \texttt{linlog} \rightarrow \texttt{lltheory} \rightarrow \texttt{numll} \rightarrow \texttt{progll}$$

The `Pure` theory is an Isabelle's basic theory and contains only the meta-logic. The rest of the theories, i.e. `linlog`, `lltheory`, `numll`, `progll`, implement pLSD piece by piece.

3.1.1 `linlog`

The `linlog` theory implements sequent system $\mathcal{L}$ for PL_σ described in section 1 by adding a new constant `|-` which builds these sequents. The theory also introduces:

- the new class `llform` which is the class of all our target linear logic formulae;
- two new types in this class:

`pll`	–	the type of Platonic formulae
`dll`	–	the type of dynamic formulae

- the following constants:

`><`, `I`	–	multiplicative conjuction and truth
`&&`, `tt`	–	additive conjuction and truth
`++`, `ff`	–	additive disjuction and false.

  Both `tt` and `ff` are constants of type `pll`, whereas `I` is assigned to `dll`.

- some other types and constants which are related to representation of sequents of PL_σ. Implementation of sequents of PL_σ and pLSD was based on implementation of sequents in Isabelle's LK logic.

Details can be found in Appendix A.

Examples of theorems in the `linlog` theory[2]:

1. `goal linlog.thy "(A && B) && C |- A && (B && C)";`
2. `goal linlog.thy "A, ff |- A >< B";`

[2] All examples presented were proved with Isabelle.

3.1.2 `lltheory`

The `lltheory` is an extension of `linlog` theory. It adds to the the `linlog` a new class `term` and one type of this class: `num`. The type `num` is the type of objects which are used in programs e.g. integers, reals, booleans etc. The `lltheory` introduces also new constants:

`za`	::	$\texttt{num} \times \texttt{num} \longrightarrow \texttt{dll}$	builds dynamic atoms, i.e., single substitutions
$\sim$	::	$\texttt{pll} \longrightarrow \texttt{pll}$	negation over platonic formulae
`->`	::	$\texttt{pll} \times \texttt{pll} \longrightarrow \texttt{pll}$	implication over platonic formulae
`=`	::	$\texttt{num} \times \texttt{num} \longrightarrow \texttt{pll}$	equality over elements of type `num`.

This theory contains the axioms discussed in section 1. The only difference is that at present only single substitutions are implemented. See Appendix B for details.

Formula `e za x` codes substitution of `e` for `x`[3]. Notice also that negation and implication were defined to operate on Platonic formulae only.

Examples of theorems in the `lltheory`:

1. `goal lltheory.thy "t1 = t2 |- t2 = t1";`
2. `goal lltheory.thy "tt |- e1 = e2 ==> (e1 za x) |- (e2 za x)";`
3. `goal lltheory.thy "(B ++ C) && A |- (B && A) ++ (C && A)";`

3.1.3 `numll`

The `numll` theory implements the first order theory which is used in programs. In our case it is (part of) the arithmetic of integers. By varying this theory we can go from integers to reals, etc.

The theory is an extension of `lltheory` and adds to it the following constans:

`0, 1`	::	`num`	zero and one
`+, *`	::	$\texttt{num} \times \texttt{num} \longrightarrow \texttt{num}$	plus and times
`-`	::	$\texttt{num} \longrightarrow \texttt{num}$	minus
`<`	::	$\texttt{num} \times \texttt{num} \longrightarrow \texttt{pll}$	order (less) on integers

It also contains the usual rules which implement the usual axioms of integer's arithmetic. Apart from arithmetic axioms, in the `numll` theory there are two axioms which tell us how to make substitutions on our new platonic formulae of the form $m < n$, where m, n :: `num`. Appendix C has the details.

Examples of theorems in the `numll` theory:

1. `goal numll.thy "tt |- a * 0 = 0";`
2. `goal numll.thy "(y < 0) |- (0 < y + - 1) ++ (0 = y + - 1)";`
3. `goal numll.thy "tt |- -(x * y) = x * (-y)";`

3.1.4 `progll`

The `progll` theory extends the `numll` theory to the pLSD logic by adding to the `numll`:

- two new types of class `term`: `progr` and `pLSDform`;

[3] "za" means "for" in Polish.

- new constants for program's constructs:

`:=`	::	`num` × `num` ⟶ `progr`	for assignment,
`SKIP`	::	`progr`	for an empty programm,
`;;`	::	`progr` × `progr` ⟶ `progr`	for sequential composition,
`IF`	::	`pll` × `progr` × `progr` ⟶ `progr`	for conditional,
`WHILE`	::	`pll` × `progr` ⟶ `progr`	for loop;

- the constant `sat` :: `progr`×`(a' ::  llform)` ⟶ `pLSDform` which encodes the satisfaction relation;
- the constant `||-` and other machinery needed to implement sequents of pLSD.

Rules of the `progll` theory can be found in Appendix D.
Examples of theorems in the `progll` theory:

1. `goal progll.thy "||- z := x * y ;; y := 0 sat (x * y za z) >< (0 za y)";`
2. ```
   goal progll.thy "||- IF (0 < y ++ 0 = y,
                   (z := 0) ;; WHILE(0 < y, (z := x + z) ;; y := y + - 1),
                   (y := - y) ;;
                   ((z := 0) ;; WHILE(0 < y, (z := x + z) ;; y := y + - 1)) ;;
                   z := - z) sat  (x * y za z) >< (0 za y)";
   ```

The proof constructed to solve goal 1 above also works for the following development task:

```
goal progll.thy "||- ?p sat (x * y za z) >< (0 za y)";
```

where `?p` is Isabelle's metavariable ranged over the type `progr`. As a result we obtain the same program like in example 1.

3.2 Representation of explicit substitutions

Our implementation is strightforward, and poses no questions as far as adequacy is concerned. It is also faithfull. Well, almost.

There are two sources of the apparent lack of faithfulness. One of them has to do with substitutions, the other with pLSD sequents.

Consider the following pLSD sequent

$$\texttt{p}\ \boldsymbol{sat}\ A \Vdash \pi\ \boldsymbol{sat}\ B$$

We insist that `p` above be an indeterminate program. However, we could not find any simple way of putting this restriction into representation. Consequently, as a result of some unifications Isabelle puts code into assumptions. So, it is up to the user to reject such hints.

The same story goes for substitution. `za` has been declared to have type `num` × `num` ⟶ `dll`. Now we can implement the result of applying the substitution `(e za x)` on the term `t(x)`, which depends on `x`, by the λ-term `(%x.t(x))e`. This is possible because the variable `x`, from the substitution `(e za x)`, has the same type as the expresion `e`. But there are also some troubles. One can write `(1 za 2 + 2)` which is type correct, but doesn't make any sense. Even more, we can prove in the `progll` theory that

```
||- 2 + 2 := 1 sat (1 za 2 + 2).
```

So again, it is up to the user to use Isabelle's metavariables ranging over `num` as the second argument of `za` and for the first argument of assignments. Again, steps in proof in which
```

Isabelle uses must be watched for unwanted unifications, but then again we can use the `back()` command to refuse wrong guesses.

In all encodings of programming logics that we are aware of one introduces a new syntactic class of, say, integer variables, together with explicit coercion function from variables to expressions. In Isabelle this would look like

$$\texttt{!} :: \texttt{numVar} \longrightarrow \texttt{num}$$

Then `za` could be given type $\texttt{num} \times \texttt{numVar} \longrightarrow \texttt{dll}$, where `numVar` is the syntactic type of variables over `num`. In this way the problem with unfaithful representation of substitutions dissapears. But at the same time the simple implementation of substitution as function application is lost. For instance, let $\texttt{t(x)} \equiv \texttt{!x + 1}$. Then `(%x. !x + 1)2` is not type correct because $\texttt{x} :: \texttt{numVar}$ while $\texttt{2} :: \texttt{num}$. Indeed, textual substitution would give something like `!2 + 1`.

Thus, at present, we are happy with our solution. It works. Also, avoiding explicit coercions makes the syntax much more readable. The price for the user is not very high.

The drowback of the simple solution is that it does not generalise to simultanous substitutions.

Among other things to be done one task is particularly important. Namely, proof in PL_σ and in the logics of data need to be automated as much as possible. Our experiments show that these are the most boring parts of formal development/verification with pLSD.

References

[1] Abadi, M., Cardelli, L., Curien, P.-L. and J-J. Lévy. Explicit Substitutions. ACM Conf. on Principles of Programming Languages, San Francisco, 1990.

[2] Abramsky, S. The lazy λ-calculus. In D. Turner, (Ed), *Research Topics in Functional Programming*, pp.:65–117, 1990.

[3] Abramsky, S. Domain theory in logical form. *Ann. Pure and Appl. Logic*, 51, pp.:1–77, 1991.

[4] Apt, K. Ten Years of Hoare's Logic: A Survey — Part I. *ACM Transactions on Programming Languages & Systems*, Vol 3, No 4, pp. 431–483, October 1981.

[5] Back, R. J.-R. *Correctness preserving program refinements: Proof theory and applications.* Tract 131, Mathematsch Centrum, Amsterdam, 1980.

[6] Bednarczyk, M. A. Sokołowski Logic — Its Semantics, Soundness and Relative Completeness. GDM Research Report, ICS PAS, 1993.

[7] Bednarczyk, M. A. A translation of Sokołowski Logic to Linear Logic. GDM Research Report, ICS PAS, 1993.

[8] Bednarczyk, M. A. Quasi quantales. GDM Research Report, ICS PAS, 1994.

[9] Bednarczyk, M. A. A labelled deductive system for program development. To be presented at ADT'95, Oslo, 1995.

[10] Blikle, A. Nets; complete lattices with a composition. *Bull. Acad. Polon. Sci., Sér. Sci. Math. Astronom. Phys.* 19, pp. 1123–1127, 1971.

[11] Blikle, A. An extended approach to mathematical analysis of programs. CC PAS Reports 169, 1974.

[12] Blikle, A. An analysis of programs by algebraic means. *Banach Center Publications* 2, Polish Scientific Publishers, Warsaw, pp. 167–213, 1977.

[13] Boudol, G. λ-calculi for (strict) parallel functions. Technical Report 1387, INRIA Sophia-Antipolis, 1991.

[14] Brown, C. and D. Gurr. A representation theorem for quantales. *J. of Pure and Applied Algebra* 85, pp. 27–42, 1993.

[15] Brown, C. and D. Gurr. Relations and non-commutative linear logic. To appear in *J. of Pure and Applied Algebra*

[16] Hahl, O.-J., Dijkstra, E. W. and C.A.R. Hoare. *Structured Programming.* Academic Press, 1972.

[17] Dijkstra, E. W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.

[18] Gabbay, D. Labelled Deductive Systems. Part 1 — Foundations Max-Planck-Institut für Informatik, Research Report MPI-I-223-94,1994.

[19] Girard, J.-Y. Linear Logic. *Theoretical Computer Science*, 50, pp 1–102, 1987.

[20] Girard, J.-Y. and Y. Lafont. Linear Logic and Lazy Computation. In: *Proc. TAPSOFT'87 (Pisa), vol. 2*, LNCS 250, Springer Verlag, pp 52–66, 1987.

[21] Girard, J.-Y., Lafont, Y. and P. Taylor. *Proofs and Types.* volume 7 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1989.

[22] Gries, D. *The Science of Programming.* Springer-Verlag 1981.

[23] Hehner, E.C.R. Predicative programming. *Communications of the ACM*, 27(2):134–151, February 1984.

[24] Hehner, E.C.R. *A practical theory of programming.* Springer-Verlag, 1993.

[25] Hennesy, M. A Denotational Model for Higher-Order Processes. University of Sussex Computer Science Report, November 1992.

[26] Hoare, C.A.R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[27] Hoare, C.A.R. Programs as predicates. In: *Mathematical Logic and Programming Languages*, Eds: C.A.R. Hoare and J.C. Sheperdson, Prentice Hall Int., 1985.

[28] Lamport, L. Specifying concurrent programs modules *ACM Trans., on Programming Languages and System*, 5(2), pp 190–222, April 1983.

[29] Lamport, L. The Temporal Logic of Actions. Digital SRC Research Report 79, December 1991.

[30] Lescane, P. From $\lambda\sigma$ to λv a journej through calculi of explicit substitutions. ACM Conf. on Principles of Programming Languages, San Portland, 1994.

[31] Morgan, C. and T. Vickers (Eds). *On the refinement calculus.* Springer-Verlag, 1994.

[32] Morris, J. M. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3), pp.:287–306, December 1987.

[33] Nelson, G. A generalization of Dijkstra's calculus. *ACM Trans. on Progr. Lang. & Systems*, 11(4), pp.:517–561, October 1989.

[34] Sokołowski, S. Partial correctness as a specification method. A contribution to GDM. Research Report, Institute of Computer Science, P.A.S., January 1993.

[35] Spivey, J. M. *Understanding Z: a Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

A Appendix: The `linlog` theory

In following rules the `$` sign is conctructor of sequents which elements has the type `a'::  llform`. E.g. `$G` means: any sequents of a proper formulae (even empty) which is called `G`.

```
linlog =  Pure +
classes llform < logic
default llform
types
   dll 0
   pll 0
   sequence, seqobj, seqcont, sequ, sobj 0
arities
   dll :: llform
   pll :: llform
consts
  "><"   :: "['a, 'b] => 'c"   (infixr 37)
  "&&"   :: "['a, 'b] => 'c"   (infixr 36)
  "++"   :: "['a, 'b] => 'c"   (infixr 35)
  ident  :: "dll" ("I")
  bottom :: "pll" ("ff")
  top    :: "pll" ("tt")
rules
  (*Structural rules*)
 refl    "P |- P"
 cut     "[| $G |- A ; $F, A, $H |- B |] ==> $F, $G, $H |- B"
 li      "$G, $D |- A ==> $G, I, $D |- A"
 ri      "|- I"
 lt      "$G, A, B, $D |- C ==> $G, A >< B, $D |- C"
 rt      "[| $G |- A ; $D |- B|]==> $G , $D |- A >< B"
 landl   "$G, A, $D |- C ==> $G, A && B, $D |- C"
 landr   "$G, B, $D |- C ==> $G, A && B, $D |- C"
 rand    "[| $G |- A ; $G |- B|]==> $G  |- A && B"
 topr    "$G |- tt"
 bottoml "ff, $H |- A"
 lor     "[| $G, A, $D |- C ; $G, B, $D |- C |] ==> $G, A ++ B, $D |- C"
 rorl    "$G |- A ==> $G |- A ++ B"
 rorr    "$G |- B ==> $G |- A ++ B"
end

ML
```

B Appendix: The `lltheory`

```
lltheory = linlog +
classes
   term < logic
types
   num 0
arities
   num :: term
consts
  "za" :: "[num, num] => dll"    (infixr  35)
  "~"   ::  "pll => pll"             ("~ _"[41] 40)
  "->"  ::  "[pll, pll] => pll"     (infixl 35)
  "="  :: "[num, num] => pll"  (infixr 50)
rules
    (* axiom for the equality *)
 equal_num      "tt |- P = P"
 eq_dll         "(E = F) && (E za X ) |- (F za X )"
(* platonic axioms *)
 r_imp_def     "~(P) ++ Q |- P -> Q"
 l_imp_def     "P -> Q |- ~(P) ++ Q"
 l_demorg_and  "~(P && Q) |- (~P) ++ (~Q)"
 r_demorg_and  "(~P) ++ (~Q) |- ~(P && Q)"
 l_demorg_or   "~(P ++ Q) |- (~P) && (~Q)"
 r_demorg_or   "(~P) && (~Q) |- ~(P ++ Q)"
 l_not_tt      "~tt |- ff"
 r_not_tt      "ff |- ~tt"
 l_not_ff      "~ff |- tt"
 r_not_ff      "tt |- ~ff"
 and_not_1     "A && (~A) |- ff"
 and_not_2     "(~A) && A |- ff"
 or_not        "tt |- A ++ (~A)"
 or_and        "A && (B ++ C) |- (A && B) ++ (A && C)"
 and_or        "(A ++ B) && (A ++ C) |- A ++ (B && C)"
 pl_times_ff   "(A :: pll) |- A >< ff"
 tt_times_ff   "tt |- tt >< ff"
 plat_times    "A::pll |- A >< B"
 times_plat    "(A::pll) >< B |- A"
(* dynamic axioms *)
 id_1        "(X za X) |- I"
 id_2        "I |- (X za X)"
 tt_dll      "tt |- (E za X) >< tt"
 ff_dll      "(E za X) >< ff |- ff"
 times_and   "((E za X) >< A) && ((E za X) >< B) |- (E za X) >< (A && B)"
 and_times_r "(A >< C) && (B >< C) |- (A && B) >< C"
 and_times_l "A && (B >< C) |- (A && B) >< (A && C)"
 l_subs_subs "(E1 za X) >< (E2(X) za X) |- ((% X. E2(X)) (E1) za X)"
 r_subs_subs "(((%X.E2(X)) (E1))   za X) |- (E1 za X) >< (E2(X) za X)"
 l_eq_subs   "((E za X) >< (P(X) = Q(X))) |- \
\             ((((% X.P(X)) (E)) = ((% X.Q(X)) (E))))"
 r_eq_subs   "(% X.P(X)) (E) = (% X.Q(X)) (E) |- (E za X) >< (P(X) = Q(X))"
```

```
 l_neg_subs  "(E za X) >< (~ P(X)) |- ~((%X.P(X))(E))"
 r_neg_subs  "~((%X.P(X))(E)) |- (E za X) >< (~ P(X))"
 l_imp_subs  "(E za X) >< (P(X) -> Q(X)) |- ((%X.P(X))(E)) -> ((%X.Q(X))(E))"
 r_imp_subs  "((%X.P(X))(E)) -> ((%X.Q(X))(E)) |- (E za X) >< (P(X) -> Q(X))"
 change_subst "( |- ~( X = Y)) ==> \
\                (G = (%X.F(X))(E)) && (E = (%Y.H(Y))(G)) && \
\                ((E za X) >< (F(X) za Y) ) |- (G za Y) >< (H(Y) za X)"
end
```

C Appendix: The numll theory

```
numll = lltheory +
consts
 "<"  :: "[num, num] => pll"  (infixr 50)
 "0"  :: "num"                ("0")
 "1"  :: "num"                ("1")
 "+"  :: "[num, num] => num"  (infixr 70)
 "*"  :: "[num, num] => num"  (infixr 80)
 "-"  :: "num => num"         ("- _" [91] 90)
rules
    (* arithmetic's axioms *)
 plus_ass    "tt |- (x + y) + z = x + (y + z)"
 times_ass   "tt |- (x * y) * z = x * (y * z)"
 plus_comm   "tt |- x + y = y + x"
 times_comm  "tt |- x * y = y * x"
 un_0        "tt |- 0 + x = x"
 un_1        "tt |- 1 * x = x"
 opp_plus    "tt |- x + (- x) = 0"
 distr       "tt |- x * (y + z) = (x * y) + (x * z)"
 trich_1_l   "~(x = y) |- ((x < y) ++ (y < x))"
 trich_1_r   "((x < y) ++ (y < x)) |- ~(x = y)"
 trich_2_l   "~(x < y) |- ((x = y) ++ (y < x))"
 trich_2_r   "((x = y) ++ (y < x)) |- ~(x < y)"
 trans       "((x < y) && (y < z)) |- (x < z)"
 le_plus_r   "(x < y) |- (x + z < y + z)"
 le_plus_l   " (x + z < y + z) |- (x < y)"
 le_times_   "((0 < z) && (x < y)) |- (x * z < y * z)"
 le_times_   "((0 < z) && (x * z < y * z)) |- (x < y)"
 pl_congru   "(a = y) && (b = z) |- (a + b) = (y + z)"
 tm_congru   "(a = y) && (b = z) |- (a * b) = (y * z)"
 eq_le_1     "(g = e) && (e < f) |- (g < f)"
 eq_le_2     "(f < g) && (g = e) |- (f < e)"
 dyskr       "(x < y + 1) |- ~(y < x)"
(* substitutions *)
 l_le_subs   "((E za X) >< (P(X) < Q(X))) |- \
 \            ((((% X.P(X)) (E)) < ((% X.Q(X)) (E))))"
 r_le_subs   "(% X.P(X)) (E) < (% X.Q(X)) (E) |- \
 \            (E za X) >< (P(X) < Q(X))"
end
```

D Appendix: The progll theory

```
progll = numll +
types
    progr, pLSDform 0
arities
    progr, pLSDform :: term
consts
    (* programs *)
 ":="    :: "[num, num] => progr"              ("_ := _" [28, 29] 20)
 "SKIP"  :: "progr"                            ("SKIP")
 ";;"    :: "[progr, progr] => progr"          (infixr 20)
 IF      :: "[pll, progr, progr] => progr"
 WHILE   :: "[pll, progr] => progr"
 "sat"   :: "[progr, 'b :: llform] => pLSDform" ("((_)/ sat  (_))" [19, 19] 18)
rules
  (* program rules *)
 ass    "||- (x := e) sat (e za x )"
 skip   "||- SKIP sat I"
 comp   "[|$C ||- P sat A; $C ||- Q sat B|] ==> \
 \         $C ||- (P ;; Q) sat (A >< B)"
 cond   "[|$C ||- P sat A; $C ||- Q sat B|] ==>\
\          $C ||- IF(d,P,Q) sat ((d && A) ++ ((~d) && B))"
 loop   "[|$C ||- P sat B ; (d :: pll) && A |- B >< A; \
\                           (~ d) && A |- I|] ==>      \
\          $C ||- WHILE(d,P) sat A"
  (* Structural rules  *)
  (* !!! NOTE: without side conditions !!! *)
 p_refl       "P sat A ||- P sat A"
 cut_p        "[|$C ||- P sat B; $E, (Q sat B), $D ||- R(Q) sat A|] ==>\
\               $E, $C, $D ||- ( (%Q.R(Q)) (P) ) sat A"
 exchange     "$C, (P sat A), (Q sat B), $D ||- R sat E ==>\
\              $C, (Q sat B), (P sat A), $D ||- R sat E"
 weakening      "$C ||- P sat B ==> $C, (Q sat A) ||- P sat B"
 contraction  "$C, (Q sat A), (R sat A) ||- P(Q,R) sat B ==>\
\              $C, (T sat A) ||- ((%R Q.P(Q,R))(T))(T) sat B"
 consequence  "[|$C ||- P sat A; B |- A|] ==> $C ||- P sat B"
end
```