1991

A.S. Klusener

An executable semantics for a subset of COLD

Computer Science/Department of Software Technology Report CS-R9145 October

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

An Executable Semantics for a Subset of COLD

Steven Klusener

Department of Software Technology, CWI

P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

e-mail: stevenk@cwi.nl

Abstract

In order to develop an interpreter for the specification language COLD an executable subset of this language is introduced, to which will be referred to as PROTOCOLD. A semantics for PROTOCOLD is given, which is easier to understand than the relational semantics for the full language COLD. The semantics we discuss here describes the behavior of the language constructs by inference rules. The starting point is an *abstract* semantics and the endpoint is a more *executable* one from which it is only a small step towards a code generator.

1985 Mathematics Subject Classification: 68B10, 68F20.
1982 CR Categories: D.2, D.3, F.3
Key Words : Specification Languages, Natural Semantics, Structured Operational Semantics, Automatic Program Construction.
Note: This work is sponsored by the ESPRIT II project Atmosphere

1 Introduction

1.1 PROTOCOLD; an Executable subset of COLD

In the recent years the wide spectrum specification COLD has been developed at the Research Centre of Philips in Eindhoven. This specification language supports the use of formal methods in Software Engineering since it has a formal semantics [LFdL87]. For an introduction of COLD we refer to [Jon89]. However, due to its generality, it is not possible to obtain automatically an implementation in a programming language like 'C' from a specification in COLD. A COLD specification has to be translated manually into 'C', which is of course error prone and not fully in agreement with the idea of formal methods. Furthermore, one ends up with two documents and in case of a change in the specification two documents have to be maintained in parallel. This situation is not ideal, especially when prototyping a system.

Therefore, Hans Jonkers of the Research Centre of Philips in Eindhoven has defined PROTOCOLD, which is an executable subset of COLD. The idea is that if one is building a system the prototyping is done only at the highest level. The highest level, or system level, does nothing more than combining library components with each other. PROTOCOLD is expressive enough to express this level; it enables to combine library components in all possible ways. Moreover, it features the several non deterministic composition operators.

1.2 A Semantics for PROTOCOLD

In this paper we discuss several formal semantics for PROTOCOLD. A formal semantics enables statements about software systems which can be proven formally. Without a formal semantics it is not possible to guarantee that supporting software tools such as an interpreter are correct.

Report CS-R9203
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In [LFdL87] a relational semantics is given already for the full language. However, this semantics uses various theories which are not so easily accessible in Computer Science such as admissible ordinals and infinitary logic.

Here, we give a semantics for PROTOCOLD, which is easier to understand than the relational one for the full language. The semantics we discuss here is a transition relation; each pair of a PROTOCOLD expression and a start state is mapped onto (possibly more) resulting states. This transition relation is defined by several inference rules. This way of giving semantics is familiar to *natural semantics* ([Kahn87]) and *Structured Operational Semantics* ([Plo81]).

Our point of departure is an *intuitive* semantics for PROTOCOLD which corresponds with the standard relational semantics of COLD. This intuitive semantics is non deterministic. Via an intermediate *backtracking* semantics, obtained by incorporating backtracking techniques into the *intuitive* semantics, we come to our final *sequentialized* semantics. The *sequentialized* semantics corresponds with a deterministic interpreter; either an inference rule can be applied unambiguously or an atomic construct is encountered which can be evaluated directly.

1.3 Contents of the Paper

In the first section we start with a toy language and the three semantics are explained and their correspondence is shown. Moreover two "cut" operators are introduced, which enable to control the backtracking mechanism. Due to this cut operations the backtrack techniques of running versions, such as PROTOCOLD 1.1 ([Jon91]), can be expressed.

In the second section the language PROTOCOLD is introduced and the notion of a state is defined. It is discussed that *library constructs* are *atomic* from the point of view of PROTOCOLD and how they can be evaluated in a state.

In the third section the three semantics are discussed for PROTOCOLD. A relation $\rightsquigarrow$ is introduced which reduces a construct to a simpler but semantically equivalent construct. This relation avoids a great amount of inference rules, since inference rules are needed only for the normal forms of $\rightsquigarrow$. Moreover, reduction by $\rightsquigarrow$ guarantees that assertions behave as expected, for example $NOT(\alpha \vee \beta) \rightsquigarrow NOT(\alpha) \wedge NOT(\beta)$.

The inference rules of the semantics express the behavior of a configuration in terms of other (simpler) configurations until atomic ones are reached. The interpretation of atomic configurations is done by the function *evaluate* which may update the state. The *evaluate* function is discussed in the fourth section. Several versions of this function will be given. The version which corresponds with the COLD semantics is undecidable since it obeys the underlying theory of the library. Other, more operational ones, are introduced as well.

This *evaluate* function is an example of a *parameter* of the semantics. Other parameters will be recognized, these parameters enable to tailor PROTOCOLD to ones needs and limitations. For example, the *sequentialized* semantics assume that every library procedure has an inverse. Assume an action *Abort* which forces unsuccessful termination. By taking *Abort* as the inverse action for each library procedure it is expressed that backtracking is not possible over those procedures (e.g. leads to unsuccessful termination).

Finally, we take PROTOCOLD 1.1 and provide it with a sequentialized semantics. The parameters are instantiated and since it is a more restricted language simplifications in the inference rules are made. Moreover, the backtracking mechanism of PROTOCOLD 1.1 is expressed by application of the two cut operators.

Contents

2 Three Equivalent Semantics

In this section we introduce a simple toy-language and we give three semantics for it. This toy-language features sequential composition and alternative composition. The first semantics is the most intuitive one; it reflexes the alternative composition non deterministically. Via an intermediate semantics we come to a semantics which can be read as an non deterministic interpretator; it deals with a recursive datastructure which captures all intermediate choices, while the state is kept outside this datastructure.

The language E, with typical elements e, e_1, e_2 is defined by the following BNF rule:

$$e ::= a \mid e_1 \cdot e_2 \mid e_1 + e_2$$

where a is an atomic action, taken from a set A. We assume a set of states, denoted by Γ, with typical elements γ, γ'. In this section we will not specify the notion of a state any further. There is a special state $FAIL$. An atomic action a can be evaluated in a state γ. We assume that this evaluation is deterministic and total, expressed by the (total) function

$$evaluate \ : \ A \times \Gamma \to \Gamma$$

2.1 An Intuitive Semantics

We define a transition relation

$$\to_i \ \subset \ (E \times \Gamma) \times \Gamma$$

where $((e, \gamma), \gamma') \in \to_i$ is abbreviated by $e, \gamma \to_i \gamma'$. In the sequel two other transition relations will be introduced, hence we need the subscript i to refer to the intuitive semantics. When it is clear from the context which transition relation is used, the subscript will be omitted. The relation $\to_i$ is defined as the least relation satisfying the rules of Table 1. The relation $\to_i$ is defined by applying the inference rules of Table 1 simultaneously inductively.

Since we are not interested in the intermediate states of a computation, we design our semantics such that we obtain the final state immediate. Hence, if the evaluation of e in the state γ results in the state γ' then this is expressed by $e, \gamma \to \gamma'$. Note that this is opposite to the so called "uniform" languages (as studied for example in ([dB88]), in which the transition systems are labeled and in which the transition system is a relation between terms where no states are involved. So we have the following rule for sequential composition:

$$\frac{e_1, \gamma \to \gamma' \quad e_2, \gamma' \to \gamma''}{e_1 \cdot e_2, \gamma \to \gamma''} \quad \text{and not} \quad \frac{e_1, \gamma \to e_1', \gamma'}{e_1 \cdot e_2, \gamma \to e_1' \cdot e_2, \gamma}$$

Note that if we would prefer the right hand side rule, we would need an "end"-marker for every expression.

An atomic action can only be evaluated in a state if it does not fail.

$\dfrac{e_1, \gamma \to \gamma' \quad e_2, \gamma' \to \gamma''}{e_1 \cdot e_2, \gamma \to \gamma''}$	$\dfrac{i = 1 \ \vee \ i = 2 \quad e_i, \gamma \to \gamma'}{e_1 + e_2, \gamma \to \gamma'}$
$\dfrac{\gamma' = evaluate(a, \gamma) \neq FAIL}{a, \gamma \to \gamma'}$	

Table 1: Rules for an Intuitive Semantics

2.2 An Semantics with Backtracking

The semantics of the previous subsection is close to our intuition, but since it supports non-determinism fully, the reduction of an expression in a state may end in several different states. Since non-determinism is not implementable directly we have to give another semantics in which a backtracking mechanism is encoded into the rules.

We define the set *stack*, with typical elements s. The empty stack is denoted by δ. Elements of a stack s are *possibilities*, since they capture one or more final states. The sets *stack* and *possibility*, with typical elements p, are defined simultaneously by

$$\begin{array}{lll} \textit{possibility} & p & ::= \ \gamma \mid e, \gamma \mid s \cdot e \\ \textit{stack} & s & ::= \ \delta \mid (p \ : \ s) \end{array}$$

We will not always write the elements of the stack in between brackets. A *possibility* of the form γ corresponds with a final *possibility* which does not have to be reduced further.

We define a transition relation

$$\to_b \ \subset \ \textit{stack} \times \textit{stack}$$

The transition $e, \gamma \ : \ \delta \to \gamma' \ : \ s$ expresses that the evaluation of expression e in state γ yields γ' as first alternative. If the context in which e is being evaluated fails in γ', then the next alternative γ'' (if it exists) is obtained by $s \to \gamma'' \ : \ s'$. Again $(s, s') \in \to_b$ is denoted by $s \to s'$ where to s' is referred to as the resulting state. The transition relation $\to_b$ is defined such that a resulting state is either δ or of the form $\gamma \ : \ s$. So, either a transition fails or it delivers a state on top of the resulting stack.

If a choice is encountered it is split in its components and these two components are pushed on the stack; each as a separate element. Note that the state in which the choice is made is copied in each element of the new stack:

$$\frac{(\ (e_1, \gamma) \ : \ (e_2, \gamma \ : \ s)\ ) \to s'}{(\ (e_1 + e_2, \gamma) \ : \ s\ ) \to s'} \text{ or simply } \frac{e_1, \gamma \ : \ e_2, \gamma \ : \ s \to s'}{e_1 + e_2, \gamma \ : \ s \to s'}$$

If the top element fails then it is popped and the other component will become the top element:

$$\frac{\begin{array}{l} e \text{ fails in } \gamma \\ s \to s' \end{array}}{((e, \gamma) \ : \ s) \to s'}$$

These backtracking techniques can be found in [dB86], [dB88] and [Eli91]. However, the approach is not fully satisfying for our case, since it does not give a solution for an expression like $(e_1 + e_2) \cdot e$.

We could try a rule like

$$\frac{e_1 \cdot e, \gamma \ : \ e_2 \cdot e, \gamma \ : \ s \rightarrow s'}{(e_1 + e_2) \cdot e, \gamma \ : \ s \rightarrow s'}$$

But it is not always easy, and certainly not efficient, to gather all possible choices captured by the first component of a sequential composition, especially if we would add recursion to the language. Assume $evaluate(e_1, \gamma) = \gamma_{e_1}$ and $evaluate(e_2, \gamma) = \gamma_{e_2}$. We say that both γ_{e_1} and γ_{e_2} are *captured* by $e_1 + e_2$ in γ. Examples of *stacks* associated to $e_1 + e_2$ in γ are (note that each of them capture γ_{e_1} and γ_{e_2} as well):

$$\begin{array}{l} ((e_1 + e_2), \gamma) \ : \ \delta \quad , \\ (e_1, \gamma) \ : \ (e_2, \gamma) \ : \ \delta \quad \text{and} \\ \gamma_{e_1} \ : \ (e_2, \gamma) \ : \ \delta \end{array}$$

The solution to the problem above is to allow a *stack* on the position of the first element of a sequential composition.

We define a mapping [.] onto $\mathcal{P}(\Gamma)$ to define which final states are captured by a *possibility* resp. *stack*:

$$\begin{array}{lcl} [e_1 \cdot e_2, \gamma] & = & [(e_1, \gamma \ : \ \delta) \cdot e_2] \\ [e_1 + e_2, \gamma] & = & [e_1, \gamma] \cup [e_2, \gamma] \\ [a, \gamma] & = & \begin{cases} \{evaluate(a, \gamma)\} & if \quad evaluate(a, \gamma) \neq FAIL \\ \emptyset & otherwise \end{cases} \\ [\gamma] & = & \{\gamma\} \\ [s \cdot e] & = & \{\gamma' \mid \exists \gamma \in [s] \ : \ \gamma' \in [e, \gamma]\} \\ \\ [\delta] & = & \emptyset \\ [p \ : \ s] & = & [p] \cup [s] \end{array}$$

The reduction of $(e_1 + e_2) \cdot e, \gamma$ is now turned into $(e_1 + e_2, \gamma \ : \ \delta) \cdot e$, thus the state is moved into the the first element of a sequential composition. Let's assume that the reduction of the *stack* $e_1 + e_2, \gamma \ : \ \delta$ results $\gamma_{e_1} \ : \ (e_2, \gamma) \ : \ \delta$, denoting that the first possibility is γ_{e_1} and that the other possibilities (in this case only γ_{e_2}) are still kept in the stack below. Now, the reduction of $(e_1 + e_2, \gamma \ : \ \delta) \cdot e$ may continue with the reduction of $e, \gamma_{e_1} \ : \ (e_2, \gamma \ : \ \delta) \cdot e$, more formally:

$$\dfrac{\dfrac{\dfrac{evaluate(e_1, \gamma) = \gamma_{e_1}}{(e_1, \gamma) \ : \ (e_2, \gamma) \ : \ \delta \rightarrow \gamma_{e_1} \ : \ (e_2, \gamma) \ : \ \delta} \quad \dfrac{}{\ldots}}{(e_1 + e_2, \gamma) \ : \ \delta \rightarrow \gamma_{e_1} \ : \ (e_2, \gamma) \ : \ \delta \qquad e, \gamma_{e_1} \ : \ (e_2, \gamma \ : \ \delta) \cdot e \ : \ s \rightarrow s'}}{(e_1 + e_2, \gamma \ : \ \delta) \cdot e \ : \ s \rightarrow s'}}{(e_1 + e_2) \cdot e, \gamma \ : \ s \rightarrow s'}$$

If e fails in γ_{e_1} then it is popped from the stack which becomes $(e_2, \gamma \ : \ \delta) \cdot e \ : \ s$, in which case the evaluation continues with the evaluation of e_2 in γ.

The empty stack δ reduces to δ. Whenever a *stack* reduces to δ, then this corresponds with a failing reduction.

There are three rules for the sequential composition, see Table 2. The first one is the so-called *initializing* rule. If the sequential composition $e_1 \cdot e_2$ is encountered in the state γ, then we initialize a *stack* of the form $e_1, \gamma \ : \ \delta$ on the first position of the sequential composition. Next, we need two rules for a sequential composition of a the form $s \cdot e$. The first of these two deals with the situation where the *stack* s reduces to $\gamma \ : \ s''$, denoting that the first possibility is γ and that the rest of the possibilities are still kept in s''. The second of these two rules covers the situation where s fails, in which case the evaluation continues with the rest of the stack.

To conclude this section we give an definition and a characterizing theorem. By definition of the mapping [.] we have the following lemma

$\delta \rightarrow \delta$		
$\dfrac{(e_1, \gamma : \delta) \cdot e_2 \ : \ s \rightarrow s'}{e_1 \cdot e_2, \gamma \ : \ s \rightarrow s'}$ $\dfrac{e_1, \gamma \ : \ e_2, \gamma \ : \ s \rightarrow s'}{e_1 + e_2, \gamma \ : \ s \rightarrow s'}$	$\dfrac{s \rightarrow \gamma \ : \ s'' \quad e, \gamma \ : \ s'' \cdot e \ : \ s' \rightarrow s'''}{s \cdot e \ : \ s' \rightarrow s'''}$	$\dfrac{s \rightarrow \delta \quad s' \rightarrow s''}{s \cdot e \ : \ s' \rightarrow s''}$
$\dfrac{\gamma' = evaluate(a, \gamma) \neq FAIL}{a, \gamma \ : \ s \rightarrow \gamma' \ : \ s}$	$\dfrac{evaluate(a, \gamma) = FAIL \quad s \rightarrow s'}{a, \gamma \ : \ s \rightarrow s'}$	

Table 2: Rules for a Backtracking Semantics

Lemma 2.2.1

$$s \rightarrow s' \quad \Longrightarrow \quad [s] = [s']$$

Proof. By induction on the length of the derivation. □

And we can state the correspondence between the two semantics:

Theorem 2.2.2

$$e, \gamma \rightarrow_i \gamma' \quad \Longleftrightarrow \quad e, \gamma \ : \ \delta \rightarrow_b s \ \wedge \ \gamma' \in [s]$$

Proof. By induction on the structure of e. □

2.3 A Sequentialized Semantics

The backtracking semantics of the previous section is deterministic and for every configuration exactly one rule is applicable. However, this backtracking semantics is not yet that close to an interpreter, since it is necessary to maintain a state in each element of the stack.

In this section we give a semantics without a state in each element of the stack structure. In case of a change of state, an inverse action to "undo" the change of state is pushed on the stack. When a backtracking is needed, first the inverse action is passed, which turns the state back into the original one.

Let I be the set of inverse actions, with typical element i. Here, the function *evaluate* takes a pair (a, γ). Besides the resulting state γ' it does also result the *inverse* action i such that the evaluation can turn the change of γ' back to γ. We have a function $evaluate \ : \ I \times \Gamma \rightarrow \Gamma_{\setminus \{FAIL\}}$. Initially we assume the following

$$a \in A \ : \ evaluate(a, \gamma) = (i, \gamma') \Longrightarrow evaluate(i, \gamma') = \gamma$$

We have to redefine the sets *stack* and *possibility*, by removing the states out of the previous definitions:

$$\begin{array}{lll} \textit{possiblity} & p & ::= e \mid i \mid s \cdot e \\ \textit{stack} & s & ::= \delta \mid p \ : \ s \end{array}$$

Again, we define a mapping [.]. on *possibilities* and *stacks*, now interpreted in a state:

$$\begin{array}{lll} & [e_1 \cdot e_2]\gamma & = [(e_1 \ : \ \delta) \cdot e_2]\gamma \\ & [e_1 + e_2]\gamma & = [e_1]\gamma \cup [e_2]\gamma \\ & [a]\gamma & = \begin{cases} \{\gamma'\} & \textit{if} \quad \textit{evaluate}(a,\gamma) = (i,\gamma') \neq \textit{FAIL} \\ \emptyset & \textit{otherwise} \end{cases} \\ & [i]\gamma & = \textit{evaluate}(i,\gamma) \\ & [s \cdot e]\gamma & = \{\gamma'' \mid \exists \gamma' \in [s]\gamma \ : \ \wedge \ \gamma'' \in [e]\gamma'\} \\ \\ & [\delta] & = \emptyset \\ p \neq i & [p \ : \ s]\gamma & = [p]\gamma \cup [s]\gamma \\ & [i \ : \ s]\gamma & = [s]\textit{evaluate}(i,\gamma) \end{array}$$

The set $conf$ of configurations is defined as $stack \times \Gamma$. A typical element of $conf$ is denoted by c. The transition relation of the sequentialized semantics denoted by $\rightarrow_s$ is defined as

$$\rightarrow_s \quad \subset \quad conf \times conf$$

where we use the standard abbreviation $s,\gamma \rightarrow_s s,\gamma'$ for $((s,\gamma),(s',\gamma')) \in \rightarrow_s$.

So, we have one rule for the case that an atomic action is evaluated, now its *inverse* action is put on the stack:

$$\frac{(\gamma',i) = \textit{evaluate}(a,\gamma) \neq \textit{FAIL}}{a \ : \ s,\gamma \rightarrow i \ : \ s,\gamma'}$$

If an *inverse* action is encountered, then it is evaluated and one continues with the rest of the stack:

$$\frac{\begin{array}{l} \gamma' = \textit{evaluate}(i,\gamma) \\ s,\gamma' \rightarrow s',\gamma'' \end{array}}{i \ : \ s,\gamma \rightarrow s',\gamma''}$$

The other rules are constructed by simply taking the rules of the previous subsection and removing the state out of the stacks. Note that the set of rules can be considered as a sequential program which manipulates a stack structure. We state the correctness of the transition relation $\rightarrow_s$ w.r.t. the original transition relation $\rightarrow_i$. We have the following lemma

Lemma 2.3.1

$$s,\gamma \rightarrow_s s',\gamma' \quad \Longrightarrow \quad [s]\gamma = \{\gamma'\} \cup [s']\gamma'$$

Proof. By induction on the length of the derivation. □

And the correctness is obtained by the following theorem:

Theorem 2.3.2

$$e,\gamma \rightarrow_i \gamma' \quad \Longleftrightarrow \quad e \ : \ \delta,\gamma \rightarrow_s s,\gamma'' \ \wedge \ \gamma' \in \{\gamma''\} \cup [s]\gamma''$$

Proof. By induction on the structure of e. □

$\delta, \gamma \rightarrow \delta, \gamma$		
$\dfrac{(e_1 : \delta) \cdot e_2 \ : \ s, \gamma \rightarrow c}{e_1 \cdot e_2 \ : \ s, \gamma \rightarrow c}$ $\dfrac{e_1 \ : \ e_2 \ : \ s, \gamma \rightarrow c}{e_1 + e_2 \ : \ s, \gamma \rightarrow c}$	$\dfrac{s, \gamma \rightarrow s'', \gamma' \quad e \ : \ s'' \cdot e \ : \ s', \gamma' \rightarrow c}{s \cdot e \ : \ s', \gamma \rightarrow c}$	$\dfrac{s, \gamma \rightarrow \delta, \gamma' \quad s', \gamma' \rightarrow c}{s \cdot e \ : \ s', \gamma \rightarrow c}$
$\dfrac{(i, \gamma') = evaluate(a, \gamma) \neq FAIL}{a \ : \ s, \gamma \rightarrow i \ : \ s, \gamma'}$	$\dfrac{evaluate(a, \gamma) = FAIL \quad s, \gamma \rightarrow s', \gamma'}{a \ : \ s, \gamma \rightarrow s', \gamma'}$	$\dfrac{\gamma' = evaluate(i, \gamma) \quad s, \gamma' \rightarrow s', \gamma''}{i \ : \ s, \gamma \rightarrow s', \gamma''}$

Table 3: Rules for a Sequentialized Semantics

2.4 Two Cut Operators

In running versions of PROTOCOLD some backtrack mechanism are built in, which differ from the semantics discussed earlier in this paper. Take for example a procedure P defined as

PROC $P : Nat \rightarrow$
IN x
DEF
$x := 1 | x := 2$

Assume a state $\gamma[\perp / x]$ in which x is not yet properly bind. According to the semantics of the previous section the expression $P; x = 2$ interpreted in $\gamma[\perp / x]$ results in $\gamma[2/x]$.

However, in PROTOCOLD 1.1 the expression $P; x = 2$ will fail in $\gamma[\perp / x]$. First the procedure P is evaluated, resulting in $\gamma[1/x]$, when "leaving" the body of P all other alternative (thus $x := 2$) are discarded in the rest of the computation. In the literature operators have been proposed by which backtracking can be controlled. The *soft cut*, here denoted by the postfix operator \$ removes all alternatives of its scope. In PROTOCOLD 1.1 it holds that

$$P = (x := 1 + x := 2)\$$$

Note that

$$\begin{array}{lcl} P; x = 2, \gamma[2/x] & \rightarrow & \gamma[2/x] \\ (x := 1 + x := 2); P; x = 2 \gamma[\perp / x] & \rightarrow & \gamma[2/x] \end{array}$$

Thus the expression $e\$$ applied in a state γ will result at most one state, the other alternatives will not be kept, only the inverse actions are kept. This is expressed by the following rule:

$$\frac{e \ : \ \delta, \gamma \rightarrow s', \gamma'}{e\$ \ : \ s, \gamma \rightarrow sel\text{-}inv(s', s), \gamma'}$$

The application $sel\text{-}inv(s', s)$ *selects* all *inverses* occurring in s' and pushes them on top of s. It is clear that all other alternatives kept by s' are removed. Assume

$$(x := 1 | x := 2) \ : \ \delta, \gamma[\perp / x] \rightarrow clear(x) \ : \ x := 2 \ : \ \delta, \gamma[1/x]$$

where $clear(x)$ is the action to undo the binding $x := 1$, then

$$(x := 1 | x := 2)\$ \ : \ s, \gamma \rightarrow clear(x) \ : \ s, \gamma[1/x]$$

So if we start in a state where x is not yet properly bind then the expression $(x := 1 | x := 2)\$$ has the same resulting state as $x := 1$.

$$\begin{array}{lll} & sel\text{-}inv((i \ : \ s), s') & = \ i \ : \ sel\text{-}inv(s, s') \\ p \notin I & sel\text{-}inv((p \ : \ s), s') & = \ sel\text{-}inv(s, s') \\ & sel\text{-}inv(\delta, s) & = \ s \\ & sel\text{-}inv(\Delta, s) & = \ \Delta \end{array}$$

The following operator is the *hard cut*, denoted by the postfix operator \$\$. The expression $e\$\$$ is similar to $e\$$ with that respect that the failure stack of the context is removed as well. This operator expresses that backtracking is not allowed at all, thus the inverses may be removed as well.

When a stack reduces to the empty stack δ, then no alternatives are kept and the context in which this stack occurs will backtrack whenever possible. However, it is not possible to backtrack over the *hard cut*. Hence, we need a new "base stack", which will be denoted by Δ. We extend the definition of *stacks* by taking Δ as another base stack. Moreover, we extend the set $conf$ with $\perp$.

$$\text{The rule} \quad \frac{e \ : \ \delta, \gamma \rightarrow s', \gamma'}{e\$\$ \ : \ s, \gamma \rightarrow \Delta, \gamma'} \quad \text{together with} \quad \Delta, \gamma \rightarrow \perp$$

express the behavior of the *hard cut*. The first alternative of e in γ is γ', the other final states are captured by s'. As in the case of the *soft cut*, only the first alternative is taken. But in the case of the *hard cut*, also the failure stack of the context (e.g s) has to be removed. Finally, the fact that no backtracking is possible, is expressed by Δ and the second rule; if one tries to backtrack over Δ then the *error* configuration $\perp$ is the result:

$$\frac{\begin{array}{l} s, \gamma \rightarrow \Delta, \gamma' \\ e \ : \ \Delta, \gamma' \rightarrow c \end{array}}{s \cdot e \ : \ s' \rightarrow c} \quad \text{and} \quad \frac{s, \gamma \rightarrow \perp}{s \cdot e, \gamma \rightarrow \perp}$$

3 The Language PROTOCOLD

In the previous section we have introduced a toy-language; only sequential composition and alternative composition were mentioned, moreover states were kept abstract. In this section we add all features of PROTOCOLD.

3.1 The *System* Level and the *Library*

PROTOCOLD is parameterized by its *library*; the *library* offers procedures, predicates and functions of which the semantics is given already. Within PROTOCOLD it is possible to define procedures, predicates and functions on the *system* level. Their semantics can be given in terms of the language constructs of PROTOCOLD and the semantics of the *library* components. A typical procedure (either a *system* or a *library*) is denoted by p; a typical *library* procedure is denoted by p_{lib} and a typical *system* procedure is denoted by p_{sys}. Similarly, we have a typical predicate r and a typical function f with subscripts $_{lib}$ or $_{sys}$ when needed.

3.2 Expressions, Assertions and Terms

The set of expressions, with typical element e, is denoted by $\mathcal{E}$, the set of assertions, with typical element α, is denoted by $\mathcal{A}$ and the set of terms, with typical element t, is denoted by $\mathcal{T}$. The subset of $\mathcal{T}$ of *atomic* terms, is denoted by $\mathcal{T}^{at}$. A term is *atomic* if it is constructed from *library* components and variables only, thus no guards, bindings and choice operators or *system* functions occur in an *atomic* term.

$$e ::= \alpha? \mid x := t \mid e_1; e_2 \mid e_1 | e_2 \mid p(\bar{t})$$

$$\alpha ::= \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid NOT(\alpha) \mid TRUE \mid FALSE \mid r(\bar{t}) \mid t_1 = t_2 \mid t_1 \neq t_2 \mid t!$$

$$t ::= t \in \mathcal{T}^{at} \mid \alpha?; t \mid x := t; t' \mid t_1 | t_2 \mid f_{sys}(\bar{t}) \mid t!$$

The operator $\vee$ is the alternative operator on assertions. The operator $|$ is the alternative composition on expressions resp. terms. Hence, terms are non deterministic in general. However, in some contexts such as a function body we require that the term is uniquely defined. This is expressed by the assertion $t!$ which holds whenever t is uniquely defined. As term $t!$ denotes the unique value in case the term is uniquely defined otherwise it is undefined. The *uniqueness requirement* does also hold for the parameters of a function application. For example $f(1|2)$ is not defined, whereas a procedure application $p(1|2)$ equals $p(1)|p(2)$.

3.2.1 Recursion at the *System* Level

The definition of a procedure level is of the following form

PROC $p_{sys} : V_1 \#...\# V_n \rightarrow$
IN $x_1, ..., x_n$
DEF
LET $y_1 : W_1, ..., y_m : W_m;$
B

Where $B \in \mathcal{E}$ is the body. The variables x_i and y_j are "symbolic" variables in contrary to the concrete variables from the set of variables VAR which will be discussed later.

Since the semantics of the procedures, predicated and functions are defined simultaneously inductively we require *guardedness* on the set of procedures, predicated and functions. This requirement excludes the following specification which would lead to an inconsistency when the natural numbers are defined as usually.

PRED $R : NAT$	**FUNC** $F : NAT -> NAT$
IN x	**IN** x
DEF	**DEF**
$F(x) = x$	$R(x)?; 2 \quad \mid \quad NOT(R(x))?; 1$

3.2.2 The Auxiliary Operation $\neg$

When verifying $NOT(r(\bar{t}))$ there are two options. Either we verify $NOT(\bar{t}!)$ or we assume $\bar{t}!$ in which case we have to verify $NOT(p(\bar{t}!))$. We prefer to introduce a new operator $\neg$, which assumes that all terms involved are uniquely defined. Now $NOT(p(\bar{t}))$ can be reduced to $NOT(\bar{t}!) \vee \neg(p(\bar{t}))$. If we would reduce it to $NOT(\bar{t}!) \vee NOT(p(\bar{t}!))$ we would have cycles in the reduction. This operator $\neg$ will only occur in assertions of the following forms for which the so called *uniqueness requirement* holds.

$$t_1 \neg = t_2 \mid \neg r(\bar{t}) \mid \neg f(\bar{t})!$$

3.2.3 Atomic Assertions

We define a subset of $\mathcal{A}$, denoted by $\mathcal{A}^{at}$, where $\bar{t}$ is now a subset of $\mathcal{T}^{at}$.

$$\mathcal{A}^{at} \ : \ t_1 = t_2 \mid t_1 \neg = t_2 \mid r_{lib}(\bar{t}) \mid \neg(r_{lib}(\bar{t})) \mid f_{lib}(\bar{t})! \mid \neg(f_{lib}(\bar{t})!) \mid TRUE \mid FALSE$$

If $\alpha \in \mathcal{A}^{at}$ then we may write $\alpha(\bar{t})$ to make explicit on which terms it depends.

Furthermore, we write $\theta \in [\![\alpha]\!]$ if θ is a *binding function* which validates the assertion α. (A *binding function* is a function which assigns to each variable in VAR a value in the domain, this notion will be discussed in the sequel.)

If $t \in \mathcal{T}^{at}$ occurs in σ (where $\sigma \in \mathcal{P}^f(\mathcal{A})$, i.e. t occurs in one of the atomic assertions in σ), then the substitution of t by the variable z is denoted by $\sigma[z/t]$. If σ is the sequence $\{\alpha_1, ..., \alpha_n\}$ then we abbreviate $\cap_{i \in \{1,..,n\}} [\![\alpha_i]\!]$ by $[\![\sigma]\!]$. Furthermore $\{\alpha\} \cup \sigma$ is abbreviated by (α, σ).

A vector of terms $t_1, .., t_n$ may be abbreviated by $\bar{t}$. We allow ourselves the freedom to consider $\bar{t}$ as a set of terms, thus allowing $\bar{t} \subset \mathcal{T}^{at}$. Moreover $\bar{t}!$ abbreviates $(t_1!, ..., t_n!)$.

3.3 The Notion of a State

In the previous section we introduced the set of terms $\mathcal{T}$ and its subset of atomic terms $\mathcal{T}^{at}$. In this section we define the notion of a state. The set of states is denoted by Γ, with typical elements γ. Every γ is a pair (θ, σ) where θ is a *binding function* interpreting *atomic* terms. The second component σ is a finite subset of $\mathcal{A}^{at}$ and it contains the encountered atomic assertions in which not yet properly defined variables occur. This σ can be considered as the *assumption* on not yet properly defined variables, it restricts the possible future states.

When instantiating a body, *free* concrete variables from the set of variables VAR are taken for the local variables. Initially they are not yet properly defined. We allow that local variables are used "before" the have a properly value bind to it. For example:

PRED *skip* :	**PRED** *fail* :
DEF	**DEF**
LET $x; y : Nat$	**LET** $x; y : Nat$
$x = y;$	$x = y;$
$x = 1;$	$x = 1;$
$y = 1;$	$y = 2;$

These examples show the need to store the encountered assertions in which not yet properly defined variables occur.

Probably this introduction will suffice for the rest of the paper. On a first reading the reader is suggested to skip the rest of this section.

3.3.1 The *library* Components in the set of Terms

The *library* defines a signature Σ and an associated Σ-algebra $\mathcal{ALG}$ and a set of (partial) interpretations $\Theta^{\Sigma} \ : \ T(\Sigma) \rightarrow \mathcal{ALG}$, with typical elements θ^{Σ}. Θ^{Σ} is the set of terms over the signature Σ, thus it is the set of terms provided by the *library*. Each θ^{Σ} can be considered as a function which assigns to each constant of a sort s an element in the carrier of s (denoted by D_s) and which assigns to each function symbol the associated operator in $\mathcal{ALG}$. Furthermore, we assume that the carrier of sort s has a bottom element, denoted by $\perp_s$. We omit the sort of the bottom element if it is clear from the context which bottom element is meant.

3.3.2 The Set of Variables *VAR*

VAR is a so called "ground"-signature, a signature containing only constants, where VAR has the same sorts as Σ and every sort has countable many constants in VAR. In the sequel a constant from

VAR is called a variable. The set of (partial) functions from $T(VAR)$ to A is referred to as the set of *binding functions* Θ^V. We use the following notations:

$$\begin{array}{lcllllll} \theta^V[u/x] & = & \lambda y. & \text{if} & y = x & \text{then} & u & \text{else} \quad \theta^V(y) \\ \theta^V \backslash x & = & \lambda y. & \text{if} & y = x & \text{then} & undefined & \text{else} \quad \theta^V(y) \end{array}$$

Note the difference between $\theta_1^V(x) = \perp$ (hence $x \in dom(\theta_1^V)$) and $x \notin dom(\theta_2^V)$, since $\perp$ is an element of the associated carrier.

3.3.3 Interpreting Atomic Terms

The set of *atomic* terms T^{at} coincides with the set $T(\Sigma(VAR))$. $T(\Sigma(VAR))$ is the set of terms build from constants and functions from Σ and the variables from VAR. An interpretation of $T(\Sigma(VAR))$ into $\mathcal{ALG}$ is a pair $\theta = (\theta^\Sigma, \theta^V) \in \Theta$. On Θ, two projection functions are defined, Σ and V:

$$\begin{array}{lcl} \Sigma((\theta^\Sigma, \theta^V)) & = & \theta^\Sigma \\ V((\theta^\Sigma, \theta^V)) & = & \theta^V \end{array}$$

Each $\theta \in \Theta$ is an interpretation from $T(\Sigma(VAR))$ in $\mathcal{ALG}$:

$$\begin{array}{lcll} f(t_1, .., t_n) \in T(\Sigma(VAR)) & : & \theta(f(t_1, .., t_n)) & = & (f^{(\theta)})(\theta(t_1), .., \theta(t_n)) \\ f \text{ is a functionsymbol from } \Sigma & : & f^\theta & = & f^{\Sigma(\theta)} \\ c \text{ is a constant from } \Sigma & : & \theta(c) & = & \Sigma(\theta)(c) \\ x \text{ is a variable from } VAR & : & \theta(x) & = & V(\theta)(x) \end{array}$$

If $x \in dom(V(\theta))$ such that $\theta(x) \neq \perp$ then we say that x is properly defined in θ. If $t \in T(\Sigma(VAR))$ then we denote the set of variables occurring in t by $var(t)$, furthermore we write $t \in dom(\theta)$ if all function symbols and constant symbols occurring in t are in $dom(\Sigma(\theta))$ and all variables occurring in t are in $dom(V(\theta))$. A term $t \in T(\Sigma(VAR))$ is properly defined in θ if $t \in dom(\theta)$ and all variables in $var(t)$ are properly defined. An element of Θ is called a *binding function.*

3.3.4 Extension of a State

A state can be *extended* by binding the $\perp$-value to a *free* variable or by binding a proper value to a variable which is not yet properly defined.

In a computation (e.g. a construction of a proof tree according to the rules) every *binding function* of a state is an extension of a *binding function* of a previous state. A state (θ, σ) means that there will be no extensions of θ later in the computation which do not obey the *assumption* σ.

For each $\theta \in \Theta$ we define a set of possible extensions. θ' is an *extension* of θ if θ' can be written as $\theta[u_1/x_1]...[u_n/x_n]$ with $n \geq 0$ and $x_i \in VAR$ such that $\theta(x_i) = \perp$ or $x_i \notin dom(\theta)$.

$ext : \Theta \rightarrow \mathcal{P}(\Theta)$
is defined as the smallest set satisfying
$\theta \in ext(\theta)$
$\theta' \in ext(\theta) \wedge (v \notin dom(\theta')) \implies \theta'[\perp /v] \in ext(\theta)$
$\theta' \in ext(\theta) \wedge (\theta'(v) = \perp) \implies \theta'[u/v] \in ext(\theta)$

If $\sigma \in \mathcal{P}^f(\mathcal{A}^{at})$ and $\theta \in \Theta$ then we define:

$$\begin{array}{lcl} \sigma \text{ is possible in } \theta & \overset{def}{\iff} & ext(\theta) \cap [\![\sigma]\!] \neq \emptyset \\ \sigma \text{ is not inconsistent in } \theta & \overset{def}{\iff} & \theta(\bigwedge_{\alpha_i \in \sigma} \alpha_i) \neq FALSE \end{array}$$

Similarly we can define when σ is impossible or consistent in a state θ. Note that if all variables occurring in $\alpha \in \mathcal{A}^{at}$ are properly defined in θ then $\theta(\alpha)$ reduces to either $TRUE$ or $FALSE$ depending

whether $\theta \in [\![\alpha]\!]$ or $\theta \notin [\![\alpha]\!]$. Moreover, $\alpha_1 \wedge \ldots \wedge \alpha_n = FALSE$ iff there is an $\alpha_i = FALSE$. We say "σ is not inconsistent in θ" instead of "σ consistent in θ", since it does not make sense to say that $\{x = y, grt(x,y)\}$ is consistent in $\gamma[\perp /x], [1/y]$. The inconsistency will appear as soon as a properly defined value is bind to x. For example

$$\begin{array}{lll} \{(x=y),(grt(x,y))\} & \text{is inconsistent in} & \theta[1/x][2/y] \\ \{(x=y),(grt(x,y))\} & \text{is not possible in} & \theta[1/x][2/y] \\ \{(x=y),(grt(x,y))\} & \text{is not inconsistent in} & \theta[1/x][\perp /y] \\ \{(x=y),(grt(x,y))\} & \text{is not possible in} & \theta[1/x][\perp /y] \end{array}$$

Note that

$$\sigma \text{ is possible in } \theta \quad \Longrightarrow \quad \sigma \text{ is not inconsistent in } \theta$$

The converse is of course not the case. The point is that it is in general undecidable whether σ is possible in θ, while the question whether σ is inconsistent in θ can be solved by simple instantiating.

3.3.5 Free Variables and Local Variables

Previously we introduced an infinite, but countable, set of variables denoted by VAR. A variable v is *free* in a state (θ, σ) is $v \notin dom(\theta)$. It seems counter intuitive to introduce an infinite set of variables, but only the variables which are not *free* must be stored actually. Since every finite computation will deal with only finitely many not *free* variables, no counter intuitive situations occur.

There is a predicate $free \subseteq VAR \times \Theta$:

$$free(v,\theta) \stackrel{def}{\Longleftrightarrow} v \notin dom(V(\theta))$$

We abbreviate $free(v_1,\theta) \wedge .. \wedge free(v_n,\theta)$ by $free(\overline{v})$. Also we simply say "$\overline{w}$ is free in θ". $\theta \backslash x$ is the function θ without x in its domain. By definition we know that x is free in $\theta \backslash x$.

The body of a procedure definition at *system*-level is preceded by a LET-statement, which declares the local variables. These local variables are given the value $\perp$ initially. Between this LET-statement and an associated binding expression (or a binding assertion) other assertions on this variable may occur. For example the fragment

$$LETx, y : Nat; \quad x = y?; x := 1; (y = 1?; e_1 | y = 2?; e_2)$$

can considered to be equal to

$$LETx, y : Nat; \quad x = y?; x := 1; y = 1?; e_1$$

or simply

$$LETx, y : Nat; \quad x = y?; x := 1; e_1$$

Initially the (atomic) assertion $x = y$ is added to the current *assumption*, when encountering the binding $x := 1$ first 1 is bounded to 1 then 1 is bounded to y as well since $x = y$ is part of the current *assumption*. Whenever a binding is encountered, it may not be inconsistent with the current assumption. Eventually the *assumption* may be reduced when variables have become properly defined. Hence, the set of states Γ, is the set of pairs of ***binding functions*** and ***assumptions***

$$\Gamma = \Theta \times \mathcal{P}^f(\mathcal{A}^{at})$$

There is a special element in Γ denoted by $FAIL$, which serves as a bottom element of Γ; for example one could take $FAIL = (\theta, \{FALSE\})$ for an arbitrary θ.

3.4 Procedures, Predicates and Functions

A procedure defined inside PROTOCOLD is called a *system* procedure, a typical *system* procedure is denoted by p_{sys}. Similarly we have *system* predicates, typical ones denoted by r_{sys}, and *system* functions, typical ones denoted by f_{sys}. Besides *system* procedures, there are *library* procedures, typical ones denoted by p_{lib}. Similarly we have r_{lib} and f_{lib}. We may omit the suffix *sys* or *lib* if we are not interested whether a procedure, predicate or function is defined within PROTOCOLD or the *library*.

3.4.1 Instantiating a *system* Procedure, Predicate or Function

A *system* procedure p_{sys} has the following definition:

$$\begin{array}{l}\textbf{PROC}\ p_{sys} : V_1\ \#...\#\ V_n \rightarrow \\ \textbf{IN}\ x_1, ..., x_n \\ \textbf{DEF} \\ \textbf{LET}\ y_1 : W_1, ..., y_m : W_m; \\ B\end{array}$$

Where $B \in \mathcal{E}$ is the body. The variables x_i and y_j are "symbolic" variables in contrary to the concrete variables from VAR. To obtain an actual body associated with an application $p_{sys}(\overline{t})$ (with $\overline{t} \subset \mathcal{T}^{at}$) we have to substitute every occurrence of x_i in the body B by the actual parameter t_i.

Furthermore, we have to introduce a set of "local" variables according to the LET-statement. So, we take *free* variables $\overline{w}$ of the sorts $W_1, ..., W_n$ and substitute them for $\overline{y}$.

Hence, the actual body associated with an application $p(\overline{t})$ (with $\overline{t} \subset \mathcal{T}^{at}$) in a state γ is denoted by $B_{\overline{t},\overline{w}}$, where $\overline{w}$ must be *free* in γ, which abbreviates:

$$B_{\overline{t},\overline{w}} = B[\overline{t}/\overline{x}][\overline{w}/\overline{y}]$$

The definition of a *system* predicate r_{sys} is similar to the definition of a *system* procedure, where the body B is now taken from $\mathcal{A}$.

$$\begin{array}{l}\textbf{PRED}\ r_{sys} : V_1\ \#...\#\ V_n \\ \textbf{IN}\ x_1, ..., x_n \\ \textbf{DEF} \\ \textbf{LET}\ y_1 : W_1, ..., y_m : W_m; \\ B\end{array}$$

Similarly we have the abbreviation $B_{\overline{t},\overline{w}}$ for the actual body associated to $r_{sys}(\overline{t})$ in γ, where $\overline{w}$ is free in γ. Likewise, we have the same abbreviation for the actual body of a *system* function application $f_{lib}(\overline{t})$. For

$$\alpha(\overline{t})\ \text{in}\ r_{sys}(\overline{t}) \mid \neg(r_{sys}(\overline{t}) \mid f_{sys}(\overline{t})! \mid \neg(f_{sys}(\overline{t})!)$$

we allow ourselves the freedom to use the abbreviation $B^{\alpha(\overline{t})}_{\overline{t},\overline{w}}$, where

$$\begin{array}{lcl} B^{r_{sys}(\overline{t})}_{\overline{t},\overline{w}} & \stackrel{abbr}{=} & B_{\overline{t},\overline{w}} \\ B^{\neg(r_{sys}(\overline{t}))}_{\overline{t},\overline{w}} & \stackrel{abbr}{=} & NOT(B_{\overline{t},\overline{w}}) \\ B^{f_{sys}(\overline{t})!}_{\overline{t},\overline{w}} & \stackrel{abbr}{=} & (B_{\overline{t},\overline{w}})! \\ B^{\neg(f_{sys}(\overline{t})!)}_{\overline{t},\overline{w}} & \stackrel{abbr}{=} & NOT((B_{\overline{t},\overline{w}})!) \end{array}$$

3.4.2 The Meaning of an Instantiated *library* Procedure

For each *library* procedure p_{lib} on $S_1 \times ... \times S_n$ we assume a transition function:

$$[\![p_{lib}]\!] \; : \; S_1 \times ... \times S_n \times \Theta^{\Sigma} \rightarrow \Theta^{\Sigma}$$

Thus we assume that *library* procedures are deterministic. We abbreviate $[\![p_{lib}]\!](t_1, ..., t_n, \theta)$ by $[\![p_{lib}(\bar{t})]\!]\theta$. Next, we define for each $\bar{t} \subset \mathcal{T}^{at}$ the transition function on states (note that a state is a pair of an interpretation and a sequence of atomic assertions) $[\![p_{lib}]\!] \; : \; S_1 \times ... \times S_n \times \Gamma \rightarrow \mathcal{P}(\Gamma)$, where we use the same notation as before.

$$[\![p_{lib}(\bar{t})]\!](\theta, \sigma) \;\; = \;\; ([\![p_{lib}(\bar{t})]\!](\theta[\theta(t_i)/z_i]) \; , \; \sigma[z_i/t_i])$$
$$\text{for those } t_i \text{ which occur in } \sigma, \text{ where } z_i \text{ is free in } \theta$$

Some work is needed here to rename terms in σ. Assume $[\![p_{lib}(\bar{t})]\!]$ may change the interpretation of $f(1)$ and assume $\theta(f(1)) = u$. If we are in a state $(\theta, \{grt(x, f(1))\})$ and we encounter the procedure application $[\![p_{lib}(\bar{t})]\!]$ then we allow after the evaluation of $[\![p_{lib}(\bar{t})]\!]$ only bindings of x to a value greater than u. Therefore, we introduce extra variables to keep the "old" values. Thus the application $[\![p_{lib}(\bar{t})]\!]$ is evaluated in the state $(\theta[u/z], \{grt(x, z)\})$.

4 The Semantics of PROTOCOLD

In the first section we have introduced three semantics. The last one could be considered as a sequential interpretator for a language with non determinism.

In the second section we have introduced the notion of a state and the language PROTOCOLD, of which the constructs can modify a state. In this section we combine these previous sections. The rules for the alternative composition, one for expressions, one for assertions and one for terms, will not be discussed anymore. They can be constructed directly conform the rules of the first section. However, some concepts, such as recursion and the reduction on assertions, have to be discussed further.

In the first section we have defined the set *stack* with typical elements s. Since we have to deal now with *expression-stacks*, *assertion-stacks* and *function-stacks*, we add a subscript to the typical stack s, obtaining resp. s_e,s_α and s_t.

4.1 Recursion

If an instantiated *system* procedure $p_{sys}(\bar{t})$ is encountered, then the following must be done

- The input arguments $\bar{t}$ have to be reduced to a corresponding term vector $\bar{u}$ of *atomic* terms by applying the relation $\rightarrow$, thus $\bar{t}, \gamma \rightarrow \bar{u}, \gamma'$. All terms in $\bar{u}$ will have a defined value in γ'.
- A vector $\bar{w} \subset VAR$ of concrete local variables must be chosen, according to the *LET* statement of the procedure definition.
- The body B is instantiated with the *atomic* input arguments $\bar{u}$ and the *free* local variables $\bar{w}$, obtaining $B_{\bar{u},\bar{w}}$.

4.1.1 The Intuitive Semantics

By the consideration given above we obtain the following rule for $\rightarrow_i$:

$$\frac{\begin{array}{l}\bar{t}, \gamma \rightarrow_i \bar{u}, \gamma' \\ \bar{w} \text{ free in } \gamma' \\ B_{\bar{u},\bar{w}}, \gamma'[\underline{\perp}/\bar{w}] \rightarrow_i \gamma''\end{array}}{p_{sys}(\bar{t}), \gamma \rightarrow_i \gamma''}$$

One could be temped to make the local variables $\overline{w}$ *free* again. By introducing the action *free* with

$$[\![free(w)]\!] = \lambda\gamma.\gamma_{\setminus w} \text{ and } [\![free(w_1,...,w_n)]\!] = [\![free(w_1)]\!] \circ ... \circ [\![free(w_n)]\!]$$

one could express this idea by

$$\frac{\begin{array}{l}\overline{t},\gamma \rightarrow_i \overline{u},\gamma' \\ \overline{w} \text{ free in } \gamma' \\ B_{\overline{u},\overline{w}},\gamma'[\overline{\bot}/\overline{w}] \rightarrow_i \gamma'' \\ \gamma''' = [\![free(\overline{w})]\!]\gamma''\end{array}}{p_{sys}(\overline{t}),\gamma \rightarrow_i \gamma'''}$$

But this is in general not possible since the local variables still may occur in the current *assumption*. Assume

PROC $eq : NAT \times NAT$
IN x_1, x_2
LET $y_1, y_2 : NAT$;
DEF
$y_1 := x_1$;
$y_2 := x_2$;
$y_1 = y_2$?

Then

$$eq(v_1,v_2),(\theta[\bot/v_1][\bot/v_2],\emptyset)$$
$$\rightarrow$$
$$(\theta[\bot/v_1][\bot/v_2][\bot/w_1][\bot/w_2],\{(v_1=w_1),(v_2=w_2),(w_1=w_2)\})$$

Now it is not possible to *free* the local variables w_1 and w_2. A compromise is to *free* all local variables which do not occur in the final *assumption*:

$$\frac{\begin{array}{l}\overline{t},\gamma \rightarrow_i \overline{u},\gamma' \\ \overline{w} \text{ free in } \gamma' \\ B_{\overline{u},\overline{w}},\gamma'[\overline{\bot}/\overline{w}] \rightarrow_i (\theta,\sigma) \\ \gamma''' = [\![free(\overline{w})_{\setminus var(\sigma)}]\!](\theta,\sigma)\end{array}}{p_{sys}(\overline{t}),\gamma \rightarrow_i \gamma'''}$$

We will not yet adopt this idea in our semantics, since initially we are not interested in efficiency considerations. When discussing the semantics of a running versions of PROTOCOLD we will use it. For *library* procedures less has to be done; only the input arguments have to be reduced to one of their *atomic* forms, such that the evaluate function can be applied:

$$\frac{\begin{array}{l}\overline{t},\gamma \rightarrow_i \overline{u},\gamma' \\ \gamma'' = evaluate(p_{lib}(\overline{u}),\gamma')\end{array}}{p_{lib}(\overline{t}),\gamma \rightarrow_i \gamma''}$$

4.1.2 The Backtracking Semantics

The reduction of the input arguments to their *atomic* forms is non deterministic, e.g. there are more *atomic* forms possible. For the associate rules for $\rightarrow_b$ we use similar rules as the ones for sequential composition in the first section on the toy language. The first rule *initializes* the evaluation by initializing a *vector-stack* on the position of the first input argument:

$$\frac{\begin{array}{l}\{t_1,...,t_n\} \not\subset \mathcal{T}^{at} \\ p((t_1,\gamma : \delta),t_2,...,t_n) : s_e \rightarrow_b s'_e\end{array}}{p(t_1,...,t_n),\gamma : s_e \rightarrow_b s'_e}$$

In the computation of one of the *atomic* forms we have to deal with vectors with a *term-stack* in stead of a term on one of the positions, such a vector is called a *vector-stack* typically denoted by $s_{\overline{t}}$ thus

$$s_{\overline{t}} \quad = \quad (t_1, ..., s_{\overline{t}}, ..., t_n)$$

We have to cover the case where the argument of the procedure application is such a *vector − stack*.

$$\frac{s_{\overline{t}} \rightarrow_b \overline{t}, \gamma \ : \ \overline{t}'_s \qquad p@(\overline{t}), \gamma \ : \ p(s'_{\overline{t}}) \ : \ s_e \rightarrow_b s'_e}{p(s_{\overline{t}}), \gamma \ : \ s_e \rightarrow_b s'_e} \qquad \frac{\overline{t}_s \rightarrow_b \delta \qquad s_e \rightarrow_b s'_e}{p(\overline{t}_s), \gamma \ : \ s_e \rightarrow_b s'_e}$$

Note that in the premise of the right rule the result vector $\overline{t}$ is a vector of *atomic* terms which are defined in γ. By the marking @ we can remember that the input arguments of the procedure application are already reduced and *atomic*. Procedure applications which are marked can be evaluated by the following rules:

$$\frac{\gamma' = eval(p_{lib}(\overline{t}), \gamma) \neq FAIL}{p_{lib}(\overline{t})@, \gamma \ : \ s_e \rightarrow \gamma' \ : \ s_e} \qquad \frac{eval(p_{lib}(\overline{t}), \gamma) = FAIL \qquad s_e \rightarrow s'_e}{p_{lib}(\overline{t})@, \gamma \ : \ s_e \rightarrow s'_e} \qquad \frac{\overline{w} \text{ free in } \gamma \qquad B_{\overline{t},\overline{w}}, \gamma[\overline{\perp}/\overline{w}] \ : \ s_e \rightarrow s'_e}{p_{sys}(\overline{t})@, \gamma \ : \ s_e \rightarrow s'_e}$$

For $\rightarrow_s$ we have similar rules, which can be obtained by removing the state out of the stack.

An example of the evaluation of the arguments of the procedure application $p((1|2), 3)$ is given below.

$$\frac{\dfrac{1|2, \gamma \ : \rightarrow_b 1, \gamma \ : \ 2, \gamma \ : \ \delta \qquad p(1,3)@, \gamma \ : \ p((2, \gamma \ : \ \delta), 3), \gamma \ : \ s_e \rightarrow s'_e}{p((1|2, \gamma \ : \ \delta), 3) \ : \ s_e \rightarrow s'_e}}{p((1|2), 3), \gamma \ : \ s_e \rightarrow s'_e}$$

The first alternative of $p((1|2), 3)$ is $p(1, 3)$.(Note that the marker @ in $p(1, 3)@$ denotes that all the arguments of the procedure application $p(1, 3)$ have already been reduced to their atomic forms.) When $p(1, 3)$ fails in γ the configuration $p((2, \gamma \ : \ \delta), 3)$ will be evaluated next in γ, which will lead to the application $p(2, 3)$.

4.1.3 Uniqueness of Input Arguments

We don't require the uniqueness property for the input arguments of a procedure application, hence $p(1|2)$ equals $p(1)|p(2)$. For functions, however, we require this property. Thus $f(1|1)$ and $f(\alpha?; 1|NOT(\alpha)?; 2)$ may be defined, while $f(1|2)$ is not defined. Moreover, the body itself must result in a unique term as well. This is expressed by the following rule:

$$\frac{\overline{t}!, \gamma \rightarrow_i \overline{u}, \gamma' \qquad \overline{w} \text{ free in } \gamma' \qquad (B_{\overline{u},\overline{w}})!, \gamma'[\overline{\perp}/\overline{w}] \rightarrow_i t, \gamma''}{f_{sys}(\overline{t}), \gamma \rightarrow_i t, \gamma''} \quad \text{and} \quad \frac{\overline{t}!, \gamma \rightarrow_i \overline{u}, \gamma'}{f_{lib}(\overline{t}), \gamma \rightarrow_i f_{lib}(\overline{u}), \gamma''}$$

Rules for $\rightarrow_b$ and $\rightarrow_s$ can be obtained similarly. Also for predicate application we require the uniqueness property of the input arguments.

In PROTOCOLD 1.1 the well formedness allows only atomic terms as input arguments for procedure, assertion or function application.

4.2 Reduction on Assertions and Terms

When giving rules for assertions (or terms) it is not necessary to state in all the three semantics that two assertions (or terms) behave equally. If we consider for example the NOT-operator on assertions, then we can push it as far as possible inside the assertions. We could express this by the following inference rules for each of the transition relations

$$\frac{NOT(\alpha_1) \wedge NOT(\alpha_2), \gamma \rightarrow_i \gamma'}{NOT(\alpha_1 \vee \alpha_2), \gamma \rightarrow_i \gamma'}$$

$$\frac{NOT(\alpha_1) \wedge NOT(\alpha_2), \gamma : s_\alpha \rightarrow_b s'_\alpha}{NOT(\alpha_1 \vee \alpha_2), \gamma : s_\alpha \rightarrow_b s'_\alpha}$$

$$\frac{NOT(\alpha_1) \wedge NOT(\alpha_2) : s_\alpha, \gamma \rightarrow_s c}{NOT(\alpha_1 \vee \alpha_2) : s_\alpha, \gamma \rightarrow_s c}$$

but this would lead to a great amount of rules. Instead we state this fact only once by defining the relation $\rightsquigarrow$:

$$NOT(\alpha_1 \vee \alpha_2) \quad \rightsquigarrow \quad NOT(\alpha_1) \wedge NOT(\alpha_2)$$

Now it is sufficient to state for all α, α' such that $\alpha \rightsquigarrow \alpha'$ we have

$$\frac{\alpha \rightsquigarrow \alpha' \quad \alpha', \gamma \rightarrow_i \gamma'}{\alpha, \gamma \rightarrow_i \gamma'} \qquad \frac{\alpha \rightsquigarrow \alpha' \quad \alpha', \gamma : s \rightarrow_b s'}{\alpha, \gamma : s \rightarrow_b s'} \qquad \frac{\alpha \rightsquigarrow \alpha' \quad \alpha' : s, \gamma \rightarrow_s s', \gamma'}{\alpha : s, \gamma \rightarrow_s s', \gamma'}$$

Thus the introduction of $\rightsquigarrow$ must be considered as an abbreviation. A total overview of these reduction rules is given in the following table. In the inference rules we have to cover only those cases which can not be reduced by $\rightsquigarrow$. Hence, we need the following fact:

Remark 4.2.1

$$\alpha(\bar{t}) \not\rightsquigarrow \iff \alpha(\bar{t}) \text{ in } \begin{array}{l} t_1 = t_2 \mid t_1 \neg = t_2 \mid r(\bar{t}) \mid \neg r(\bar{t}) \mid (\bar{t} \subset \mathcal{T}^{at})\ \bar{t}! \mid (\bar{t} \subset \mathcal{T}^{at})\ NOT(t!) \mid x := t \\ f_{sys}(\bar{t})! \mid \neg(f_{sys}(\bar{t})!) \mid \alpha_1(\bar{t}) \vee \alpha_2(\bar{t}) \mid \alpha_1 \wedge \alpha_2 \mid TRUE \mid FALSE \end{array}$$

On terms we have to deal with the uniqueness operator, note that $t!$ is defined when t is determined uniquely and undefined otherwise. Note that every term of the form $t!$ can be reduced, hence no inference rules for $t!$ are needed in the semantics.

4.3 The Auxiliary Operation $NOT(.!!)$

A binding $x := t$ is falsified whenever $x = t$ can be falsified or when t is not defined at all (when t can not be reduced to some *atomic* term). The difference between $NOT(t!)$ and $NOT(t!!)$ is that the first case also succeeds when t has more than one atomic form.

4.4 The Auxiliary Operation $\neg$

We introduced the operation $\neg$ to distinguish the cases where an assertion has to be falsified where as its arguments are defined uniquely. Hence, we need the following rules for example for $\neg(r(\bar{t}))$.

$$\frac{\begin{array}{l}\bar{t}!, \gamma \rightarrow_i \bar{u}, \gamma' \\ \bar{w} \text{ free in } \gamma' \\ NOT(B_{\bar{u},\bar{w}}), \gamma'[\underline{\perp}/\bar{w}] \rightarrow_i \gamma''\end{array}}{\neg(r_{sys}(\bar{t})), \gamma \rightarrow_i \gamma''} \qquad \frac{\begin{array}{l}\bar{t}!, \gamma \rightarrow_i \bar{u}, \gamma' \\ FAIL \neq \gamma'' = evaluate(NOT(r_{lib}(\bar{u})), \gamma')\end{array}}{\neg(r_{lib}(\bar{t})), \gamma \rightarrow_i \gamma''}$$

And similar rules for $\rightarrow_b$ and $\rightarrow_s$. The other applications of the $\neg$ operation can be dealt with correspondingly; in the premise one has to require the unique definedness of the arguments.

$NOT(TRUE)$	$\leadsto$	$FALSE$
$NOT(FALSE)$	$\leadsto$	$TRUE$
$NOT(\alpha_1 \vee \alpha_2)$	$\leadsto$	$NOT(\alpha_1) \wedge NOT(\alpha_2)$
$NOT(\alpha_1 \wedge \alpha_2)$	$\leadsto$	$NOT(\alpha_1) \vee NOT(\alpha_2)$
$NOT(t_1 = t_2)$	$\leadsto$	$t_1 \neq t_2$
$NOT(t_1 \neq t_2)$	$\leadsto$	$t_1 = t_2$
$NOT(NOT(\alpha))$	$\leadsto$	α
$NOT(x := t)$	$\leadsto$	$x \neq t \vee NOT(t!!)$
$NOT(r(\bar{t}))$	$\leadsto$	$NOT(\bar{t}!) \vee \neg(r(\bar{t}))$
$t_1 \neq t_2$	$\leadsto$	$NOT(t_1! \wedge t_2!) \vee t_1 \neg = t_2$
$(\alpha?;t)!$	$\leadsto$	$\alpha \wedge t!$
$(t_1 \mid t_2)!$	$\leadsto$	$(t_1! \wedge NOT(t_2!)) \vee (NOT(t_1!) \wedge t_2!)) \vee t_1 = t_2$
$NOT((\alpha?;t)!)$	$\leadsto$	$NOT(\alpha) \vee NOT(t!)$
$NOT((t_1 \mid t_2)!)$	$\leadsto$	$t_1 \neq t_2$
$NOT(f(\bar{t})!)$	$\leadsto$	$NOT(\bar{t}!) \vee \neg(f(\bar{t})!)$
$NOT((t_1,...,t_n)!)$	$\leadsto$	$NOT(t_1!) \vee (t_1! \wedge NOT(t_2,...t_n!) \quad (n > 1)$
$NOT((\alpha?;t)!!)$	$\leadsto$	$NOT(\alpha) \vee NOT(t!!)$
$NOT((t_1 \mid t_2)!!)$	$\leadsto$	$NOT(t_1!!) \wedge NOT(t_2!!)$
$NOT(f(\bar{t})!!)$	$\leadsto$	$NOT(\bar{t}!) \vee \neg(f(\bar{t})!)$

Table 4: Reduction Rules for Assertions

4.5 The Binding Operation :=

The binding $x := t$ in γ reduces t to one of its *atomic* forms and binds the value of the obtained normal form to x whenever x is not yet properly defined. If it is properly defined in γ then it equals to $SKIP$ if one of the *atomic* forms of t have the same value, otherwise it equals a $FAIL$. Note that there is no uniqueness requirement on t, e.g. $x := (1|2)$ equals $(x := 1)|(x := 2)$. We don't have to spell this out, we simply take one *atomic* form u of t in γ and execute the *evaluate* function with $x = u$. Whether a binding is performed, or a $SKIP$ or a $FAIL$ is determined by the *evaluate* function. The rule which expresses this behavior is:

$$\frac{t, \gamma \rightarrow_i u, \gamma' \quad \gamma'' = evaluate(x = u, \gamma') \neq FAIL}{x := t, \gamma \rightarrow \gamma''}$$

5 The *evaluate* Function

5.1 Introduction

The rules of the three semantics generate proof trees where on top of these proof trees only *atomic* terms and *atomic* assertions and applications of *library* procedures on *atomic* terms occur. Thus the

$(x := t; t')!$	$\rightsquigarrow$	$(x := t)?; t'!$
$(\alpha?; t)!$	$\rightsquigarrow$	$\alpha?; (t!)$
$(t_1 \mid t_2)!$	$\rightsquigarrow$	$(t_1 = t_2?; t_1) \mid (t_1! \wedge NOT(t_2!)?; (t_1!)) \mid (NOT(t_1!) \wedge t_2!)?; (t_2!)$
$f(\bar{t})!$	$\rightsquigarrow$	$f(\bar{t})$

Table 5: Reduction Rules for Terms

rules do nothing more than reducing a language construct to one of its *atomic* forms. When this point is reached, the *evaluate* function is applied.

The *evaluate* function on assertions can be separated into three parts:

- First it is *checked* whether no inconsistencies appear.
- Then all possible bindings are performed.
- Finally the *assumption* is *reduced* by removing all assertions which have become tautologies due to the bindings of the previous step.

Each state γ is a pair (θ, σ), where θ is a *binding function* and σ is an *assumption.* If an atomic assertion α is encountered during the evaluation of a PROTOCOLD expression, the first thing to do is to check whether α is consistent with the other assertions in σ. This is done by the operation *check*, which delivers $FAIL$ if α is inconsistent with the state (θ, σ) and otherwise it delivers $(\theta, (\alpha, \sigma))$ thus adding α to the *assumption* σ. For example:

$$check(\ geq(x,y)\ ,\ (\theta[\perp/x][\perp/y]\ ,\ \{leq(x,y)\})\ ) \quad = \quad (\theta[\perp/x][\perp/y]\ ,\ \{geq(x,y), leq(x,y)\})$$

$$check(\ grt(x,y)\ ,\ (\theta[\perp/x][\perp/y]\ ,\ \{leq(x,y)\})\ ) \quad = \quad FAIL$$

The next thing to do is to evaluate all possible bindings caused by α. Therefore we apply the operation *bind*, for example:

$$bind(\theta[\perp/y][1/x], (y = x, \sigma)) = bind(\theta[1/x][1/y], \sigma)$$

At last we may reduce the sequence of atomic assertions by removing all assertions for which all parameters have a properly defined value.

Assume the execution is in the state

$$(\ \theta[\perp/y][1/x][\perp/z], \{z \neq x, y > 0\}\ )$$

and the binding assertion $(x = y)?$ is encountered, then we expect $(\theta[1/y][1/x][\perp/z], \{z \neq x\})$ as final state, as is shown below:

$$\begin{aligned} check(\ (x = y)\ ,\ (\theta[\perp/y][1/x][\perp/z], \{z \neq x, y > 0\})\ ) \quad &= \\ (\theta[\perp/y][1/x][\perp/z], \{x = y, z \neq x, y > 0\}) & \\ bind(\ (\theta[\perp/y][1/x][\perp/z], \{x = y, z \neq x, y > 0\})\ ) \quad &= \\ (\theta[1/y][1/x][\perp/z], \{z \neq x, y > 0\}) & \\ reduce(\ (\theta[1/y][1/x][\perp/z], \{z \neq x, y > 0\})\ ) \quad &= \\ (\theta[1/y][1/x][\perp/z], \{z \neq x\}) & \end{aligned}$$

The operation *evaluate* is the "concatenation" of the three operations *check*, *bind* and *reduce*. More formally:

$$\begin{array}{lcll} \mathit{evaluate} & : & \mathcal{A}^{at} \times \Gamma & \rightarrow \Gamma \\ \\ \mathit{check} & : & \mathcal{A}^{at} \times \Gamma & \rightarrow \Gamma \\ \mathit{bind} & : & \Gamma & \rightarrow \Gamma \\ \mathit{reduce} & : & \Gamma & \rightarrow \Gamma \\ \\ \mathit{evaluate} & = & \mathit{reduce} \circ \mathit{bind} \circ \mathit{check} & \end{array}$$

In tables in the appendices *evaluate* may be abbreviated by *eval*.

5.2 Can *check* be an oracle or does it have to be implemented?

For the *check* mechanism we have several options.

f we require that the resulting transition relation corresponds with the standard COLD semantics we have to assume an *oracle* which is able to solve all questions for us, even undecidable ones. This version is denoted by $check_{orcl}$.

A more practical approach is to add the atomic assertions to the set whenever it is not a binding. When a binding is encountered then it is applied whenever the set of atomic assertions is not inconsistent in the resulting *binding function.*

A third version is the *rigid* one. It allows only assertions which hold unambiguously in a state. Thus when an assertion is encountered with not yet properly defined variables $FAIL$ is returned. The advantage of this version is that the component σ of a state, containing all assertions on not yet undefined variables, remains empty when started with the empty set. So, also *bind* and *reduce* are redundant now; the are the identity on Γ. In Appendix A it is shown that the running version PROTOCOLD 1.1. uses the *rigid* one.

Of course other versions are possible as well.

The formal definitions of the above *check* versions are:

$$\begin{array}{lcll} check_{orcl}(\alpha,(\theta,\sigma)) & = & (\theta,(\alpha,\sigma)) & if\ (\alpha,\theta) \text{ is possible in } \theta \\ check_{orcl}(\alpha,(\theta,\sigma)) & = & FAIL & otherwise \\ check_{orcl}(x := t,(\theta,\sigma)) & = & (\theta[\theta(t)/x],\sigma) & if \sigma \text{ is possible in } \theta[\theta(t)/x] \\ check_{orcl}(x := t,(\theta,\sigma)) & = & FAIL & otherwise \\ \\ check_{imp}(\alpha,(\theta,\sigma)) & = & (\theta,(\alpha,\sigma)) & if\ (\alpha,\theta) \text{ is not inconsistent in } \theta \\ check_{imp}(\alpha,(\theta,\sigma)) & = & FAIL & otherwise \\ check_{imp}(x := t,(\theta,\sigma)) & = & (\theta[\theta(t)],\sigma) & if \theta \text{ is not inconsistent in } \theta[\theta(t)/x] \\ check_{imp}(x := t,(\theta,\sigma)) & = & FAIL & otherwise \\ \\ check_{rigid}(\alpha,(\theta,\sigma)) & = & (\theta,\sigma) & if\ \theta \in [\![\alpha]\!] \\ check_{rigid}(\alpha,(\theta,\sigma)) & = & FAIL & otherwise \\ check_{rigid}(x := t,(\theta,\sigma)) & = & (\theta[\theta(t)/x],\sigma) & if \theta[\theta(t)/x] \in [\![\alpha]\!] \\ check_{rigid}(x := t,(\theta,\sigma)) & = & FAIL & otherwise \end{array}$$

Since we have to deal with well formed fragments only it is guaranteed that $t \in dom(\theta)$ in the bindings $x := t$ above.

5.3 The functions *bind* and *reduce*

Assume we are in the state $(\theta[\perp\ /x][\perp\ /y], \{x = y\})$ and we encounter the binding $x := 1$. Then this binding is executed by *check* obtaining $\theta[1/x][\perp\ /y], \{x = y\}$. Now we may *bind* 1 to y as well

according to the assertion $x = y$, obtaining $\theta[1/x][1/y]$. For example

$$bind(\ \theta[1/n][2/x]\ ,\ \{(n = m), (z = y), (y = x)\}\ ) \quad = \quad \theta[1/n][2/x][m/1][y/1][z/1]$$

The definition below "walks" twice through the set of assertions, from the beginning to the end and back. Finally the *atomic* assertion $x = y$ may be dropped from the state since it is guaranteed by the *binding function* $\theta[1/x][1[y]$. The definitions of the functions *bind* and *reduce* are given below, they assume that the input arguments are already *checked.* Thus cases of the form $bind(\theta[1/x][2/y], \{x = y\})$ do not have to be considered. We use an auxiliary function *add*, which adds an atomic assertion to the set of atomic assertions of a state, hence $add(\alpha, (\theta, \sigma)) = (\theta, (\alpha, \sigma))$.

$$\begin{array}{lcl} bind(FAIL) & = & FAIL \\ bind(\theta, \emptyset) & = & (\theta, \emptyset) \end{array}$$

$$bind(\theta, (x = t, \sigma)) \quad =$$

$$\begin{cases} bind(\theta[\theta(t)/x], \sigma) & if \quad \theta(x) = \perp \neq \theta(t) \\ (\theta'[\theta'(y)/x], \sigma') & if \quad \theta(x) = \perp = \theta(t) \ \wedge \ \theta'(x) = \perp \neq \theta'(t) \quad where\ (\theta', \sigma') = bind(\theta, \sigma) \\ add(x = t, bind(\theta, \sigma)) & otherwise \end{cases}$$

Similar for the case $t = x$, note that in case $x = y$ with $x, y \in VAR$ both cases coincide.

$$\begin{array}{lcl} bind(\theta, (x := t, \sigma)) & = & bind(\theta, (x = t, \sigma)) \\ bind(\theta, (\alpha, \sigma)) & = & add(\alpha, bind(\theta, \sigma)) \quad if\ \alpha \text{ is not of the form } x = t, t = x \text{ or } x := t \end{array}$$

$$\begin{array}{lcl} reduce(FAIL) & = & FAIL \\ reduce(\theta, \emptyset) & = & (\theta, \emptyset) \\ reduce(\theta, (\alpha, \sigma)) & = & \begin{cases} reduce(\theta, \sigma) & if \quad \theta(\alpha) = TRUE \\ add(\alpha, reduce(\theta, \sigma)) & otherwise \end{cases} \end{array}$$

5.4 The *evaluation* function with inverses

In the previous section we saw that $evalate(x := 1, (\theta[\perp /x][\perp /y], \{x = y\})) = (\theta[1/x][1/x], \emptyset)$ However, when formulating the sequentialized semantics, the *evaluate* function must deliver an *inverse* action as well. This enables us to "undo" the change of state in case of backtracking, thus obtaining $(\theta[\perp /x][\perp /y], \{x = y\})$ back again. In this section we develop this notion of *inverse* action, in the example above the *inverse* action would be $clear(x); clear(y); reset(x = y)$ since we must undo the bindings to both x and y and we have to add $x = y$ again to the current *assumption.*

Let I be the set of instantiated inverse actions, with typical element i. I contains atomic actions like $clear(x)$ for each $x \in VAR$, for undoing a binding to x, and actions like $reset(\alpha)$ for each $\alpha \in \mathcal{A}^{at}$ moreover there is a special constant ϵ in I. Each i has a function $[\![i]\!] \in \Gamma \rightarrow \Gamma$ associated with it. For the atomic ones introduced above these functions are:

$$\begin{array}{lcl} [\![clear(x)]\!](\theta, \sigma) & = & (\theta[\perp /x], \sigma) \\ [\![reset(\alpha)]\!](\theta, \sigma) & = & (\theta, (\alpha, \sigma)) \\ [\![remove(\alpha)]\!](\theta, \sigma) & = & (\theta, \sigma_{\backslash \alpha}) \\ [\![\epsilon]\!]\gamma & = & \gamma \end{array}$$

where

$$(\alpha, \sigma)_{\backslash \alpha}) \quad = \quad \sigma$$

$$\alpha' \neq \alpha \quad (\alpha', \sigma)_{\backslash \alpha}) \quad = \quad (\alpha', (\sigma_{\backslash \alpha}))$$

On I we have a sequential composition, denoted by ;. The function $[\![.]\!]$ for $i_1; i_2$ is given by

$$[\![i_1; i_2]\!] = [\![i_1]\!] \circ [\![i_2]\!]$$

We redefine the functions concerning *evaluate*, moreover $FAIL$ is now a special element of $\Gamma \times I$:

$$\begin{array}{lcll} evaluate & \in & \mathcal{A}^{at} \times \Gamma & \rightarrow & I \times \Gamma \\ \\ check & \in & \mathcal{A}^{at} \times \Gamma & \rightarrow & I \times \Gamma \\ bind & \in & I \times \Gamma & \rightarrow & I \times \Gamma \\ reduce & \in & I \times \Gamma & \rightarrow & I \times \Gamma \end{array}$$

$$evaluate = reduce \circ bind \circ check$$

If $evaluate(\alpha, \gamma) = (i, \gamma')$ it implies that i is the action to undo the change of state, i.e. $[\![i]\!]\gamma' = \gamma$. For example:

$$\begin{array}{lcl} evaluate(\ x = 1\ ,\ (\theta[\bot\ /x], \sigma)\ ) & = & (\ clear(x); \epsilon\ ,\ (\theta[1/x], \sigma)\ ) \\ evaluate(\ x = 1\ ,\ (\theta[\bot\ /x][\bot\ /y], \{x = y\}\ ) & = & (\ clear(x); reset(x = y); \epsilon\ ,\ (\theta[1/x][1/y]\ ) \end{array}$$

The function *check* is defined as before, we only give $check_{orcl}$ the other version differ only on the conditions.

$$\begin{array}{lcll} check_{orcl}(\alpha, (\theta, \sigma)) & = & (remove(\alpha), (\theta, (\alpha, \sigma))) & (\alpha, \theta) \text{ is possible in } \theta \\ check_{orcl}(\alpha, (\theta, \sigma)) & = & FAIL & otherwise \\ check_{orcl}(x := t, (\theta, \sigma)) & = & (clear(x), (\theta[\theta(t)/x], \sigma)) & if\ \theta(x) = \bot \neq \theta(t)\ \wedge \\ & & & \sigma \text{ is possible in } \theta[\theta(t)/x] \\ check_{orcl}(x := t, (\theta, \sigma)) & = & (\epsilon, (\theta, \sigma)) & if\ \theta(x) = \theta(t) \neq \bot \\ check_{orcl}(x := t, (\theta, \sigma)) & = & (remove(x = t), (\theta, (x = t, \sigma)) & if\ \theta(x) = \theta(t) = \bot\ \wedge \\ & & & (x = t, \sigma) \text{ is possible in } \theta \\ check_{orcl}(\alpha, (\theta, \sigma)) & = & FAIL & otherwise \end{array}$$

We need two auxiliary functions *add*; one which adds an atomic assertion to the *assumption* of the state argument $add(\alpha, (i, (\theta, \sigma))) = (i, (\theta, (\alpha, \sigma)))$ and one which concatenates an inverse action to the inverse action in the input argument $add(i, (i', (\theta, \sigma))) = (i; i', (\theta, (\sigma)))$.

$$bind(i, (\theta, \emptyset)) = (i, (\theta, \emptyset))$$

$$bind(i, (\theta, (x = t, \sigma))) =$$
$$\begin{cases} add(clear(x), bind(\theta[\theta(t)/x], \sigma)) & if\ \ \theta(x) = \bot \neq \theta(t) \\ add(clear(x), (\theta'[\theta'(y)/x], \sigma')) & if\ \ \theta(x) = \bot = \theta(t)\ \wedge\ \theta'(x) = \bot \neq \theta'(t) \\ & \qquad where\ (\theta', \sigma') = bind(\theta, \sigma) \\ add(x = t, bind(i, (\theta, \sigma))) & otherwise \end{cases}$$

Similar for the case $t = x$, note that for $x = y$ with $x, y \in VAR$ both cases coincide.

$$bind(i, \theta, (\alpha, \sigma)) = add(\alpha, bind(i, (\theta, \sigma)))\quad if\ \alpha \text{ is not of the form } x = t, t = x$$

$$\begin{array}{lcl} reduce(i, (\theta, \emptyset)) & = & (i, (\theta, \emptyset)) \\ reduce(i, (\theta, (\alpha, \sigma))) & = & \begin{cases} add(reset(\alpha), reduce(i, (\theta, \sigma))) & if\ \ \theta \in [\![\alpha]\!] \\ add(\alpha, reduce(i, (\theta, \sigma))) & otherwise \end{cases} \end{array}$$

6 A Small Example

In this section we present a small example. We give the evaluation of the assertion $(x = 1 \ \vee \ x = 2) \ \wedge \ x = 1 + 1$ in the state $\gamma[\perp /x]$. We start with some partial derivations. Since all terms involved are already *atomic* it is not necessary to show the derivation of an assertion $\alpha(\overline{t})$ to a corresponding application $\alpha(\overline{t}')@$ where $\overline{t}'$ is a vector of *atomic* terms associated with $\overline{t}$.

The assertion $(x = 1 \ \vee \ x = 2)$ in $\gamma[\perp /x]$ gives as first alternative $\gamma[1/x]$ with the the *inverse* action $clear(x)$ and the other alternative $x = 2$ in the stack.

$$\frac{evaluate(x = 1, \gamma[\perp /x]) = (clear(x), \gamma[1/x])}{\dfrac{x = 1 \ : \ x = 2 \ : \ s_\alpha \ , \ \gamma[\perp /x] \rightarrow clear(x) \ : \ x = 2 \ : \ s_\alpha \ , \ \gamma[1/x]}{(x = 1 \ \vee \ x = 2) \ : \ s_\alpha \ , \ \gamma[\perp /x] \rightarrow clear(x) \ : \ x = 2 \ : \ s_\alpha \ , \ \gamma[1/x]}} \tag{1}$$

When it is discovered that $\gamma[1/x]$ is not the right alternative, the stack is evaluated.

$$\frac{\begin{array}{c}\gamma[\perp /x] = evaluate(clear(x), \gamma[1/x]) \\ x = 2 \ : \ s_\alpha \ , \ \gamma[\perp /x] \rightarrow clear(x) \ : \ s_\alpha \ , \ \gamma[2/x]\end{array}}{clear(x) \ : \ x = 2 \ : \ s_\alpha \ , \ \gamma[1/x] \rightarrow clear(x) \ : \ s_\alpha \ , \ \gamma[2/x]} \tag{2}$$

When $x = 1 + 1$ is evaluated in a state $\gamma[2/x]$ then it is simply skipped.

$$\frac{evaluate(x = 1 + 1, \gamma[2/x]) = (\epsilon, \gamma[2/x])}{x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[2/x] \rightarrow \epsilon \ : \ s_\alpha \ , \ \gamma[2/x]} \tag{3}$$

When a backtracking is applied, since $x = 1 + 1$ is evaluated in $\gamma[1/x]$ as in the next equation, then the binding to x is undone by applying $clear(x)$ and the other alternative $\gamma[2/x]$ is obtained.

$$\frac{\begin{array}{l}clear(x) \ : \ x = 2 \ : \ \delta \ , \ \gamma[1/x] \overset{(2)}{\rightarrow} clear(x) \ : \ \delta \ , \ \gamma[2/x] \\ x = 1 + 1 \ : \ (clear(x) \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[2/x] \ \overset{(3)}{\rightarrow} \\ \qquad\qquad (clear(x) \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[2/x]\end{array}}{\begin{array}{l}(clear(x) \ : \ x = 2 \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[1/x] \ \rightarrow \\ \qquad\qquad (clear(x) \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[2/x]\end{array}} \tag{4}$$

The previous backtracking is needed when $x = 1 + 1$ is evaluated in $\gamma[1/x]$.

$$\frac{\begin{array}{l}evaluate(x = 1 + 1, \gamma[1/x]) = FAIL \\ (clear(x) \ : \ x = 2 \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[1/x] \ \overset{(4)}{\rightarrow} \\ \qquad\qquad (clear(x) \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[2/x]\end{array}}{\begin{array}{l}x = 1 + 1 \ : \ (clear(x) \ : \ x = 2 \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[1/x] \ \rightarrow \\ \qquad\qquad (clear(x) \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ s_\alpha \ , \ \gamma[2/x]\end{array}} \tag{5}$$

Finally we can put every thing together. The evaluation of $(x = 1 \ \vee \ x = 2) \ \wedge \ x = 1 + 1$ in $\gamma[\perp /x]$ starts in the initial *configuration* $(x = 1 \ \vee \ x = 2) \ \wedge \ x = 1 + 1 \ : \ \delta \ , \ \gamma[\perp /x]$. First $(x = 1 \ \vee \ x = 2)$ is being evaluated, resulting in the replacement of $(x = 1 \ \vee \ x = 2)$ by $(x = 1 \ \vee \ x = 2) \ : \ \delta$, by applying the equations 1 and 5 we come to the final *configuration* $(clear(x) \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ \delta \ , \ \gamma[2/x]$. As expected the resulting state is $\gamma[2/x]$. The stack $(clear(x) \ : \ \delta) \ \wedge \ x = 1 + 1 \ : \ \delta$ expresses that whenever the binding of 2 to x would lead to a $FAIL$ in the context, the next possible alternative is obtained by first evaluating $(clear(x) \ : \ \delta)$. $clear(x)$ will undo the binding of 2 to x, but then it will be discovered that no other alternatives are captured by the first component of the conjunction,

which corresponds with our intuition as well.

$$\frac{\begin{array}{l}(x=1 \vee x=2) : \delta\ ,\ \gamma[\bot/x] \overset{(1)}{\rightarrow} clear(x) : x=2 : \delta\ ,\ \gamma[1/x] \\ x=1+1 : (clear(x) : x=2 : \delta) \wedge x=1+1 : \delta\ ,\ \gamma[1/x] \overset{(5)}{\rightarrow} \\ \qquad (clear(x) : \delta) \wedge x=1+1 : \delta\ ,\ \gamma[2/x]\end{array}}{\begin{array}{l}((x=1 \vee x=2) : \delta) \wedge x=1+1 : \delta\ ,\ \gamma[\bot/x] \rightarrow \\ \qquad (clear(x) : \delta) \wedge x=1+1 : \delta\ ,\ \gamma[2/x]\end{array}} \qquad (6)$$

$$\frac{}{\begin{array}{l}(x=1 \vee x=2) \wedge x=1+1 : \delta\ ,\ \gamma[\bot/x] \rightarrow \\ \qquad (clear(x) : \delta) \wedge x=1+1 : \delta\ ,\ \gamma[2/x]\end{array}}$$

7 Conclusions

This paper discusses in detail the semantics for PROTOCOLD which is an executable subset of the wide spectrum specification language COLD. Our goal was a semantics which is easier to understand than the standard relational semantics for COLD as presented in [LFdL87]. Moreover, it was required that an interpreter for PROTOCOLD is easily derivable from this semantics. Therefore we have defined three semantics, each consisting of a set of inference rules, which are parameterized by certain mechanism such as the evaluation of atomic constructs.

The first semantics is the most intuitive one. Via an intermediate semantics we come to a sequentialized semantics which can be considered as a (non deterministic) interpretator of PROTOCOLD.

When instantiated with the right parameters these semantics correspond with the standard semantics of COLD. Since the application area of a PROTOCOLD version may enforce restrictions on the backtrack mechanism two cut operators are introduced. A PROTOCOLD version may also have extra well formedness conditions, which lead to simplifications in the evaluation mechanisms of the atomic constructs and to simplifications in the inference rules of the semantics. An example can be found in the first appendix where we give a semantics for PROTOCOLD 1.1.

Furthermore, from the parameters it is clear to what extend the semantics of PROTOCOLD corresponds with the standard COLD semantics. In this way different versions of PROTOCOLD can be compared with each other and with standard COLD, which is necessary for managing the different PROTOCOLD versions.

By studying the semantics of this paper one can find limitations on the language which may increase the efficiency of the associated interpreter. For example, when we allow only *atomic* terms in *expressions* and *assertions* a lot of inference rules are not needed anymore. This syntactical limitation is not a semantical one, since for example $p(1|2)$ can always be written as $p(1)|p(2)$. Another possibility is to rewrite before runtime all arguments of *expressions* and *assertions* to their *atomics* terms by pushing the choices and guards "upwards".

Another consideration is that the expression $e_1+(e_2+e_3)$ leads to a smaller stack then $(e_1+e_2)+e_3$. By this rewriting before runtime, the size of the stack can be minimalized.

Acknowledgements

The author wants to thank Aad Droppert, Loe Feijs, Hans Jonkers and Ron Koymans and Thijs Winter of Philips Research Eindhoven for their inspiring discussions. Alban Ponse of the CWI in Amsterdam is thanked for his initial effort and for suggesting literature.

Jaco de Bakker of the CWI is thanked for pointing out that the style of inference rules used in this paper is very close to Kahn's natural semantics.

References

[dB86] A. de Bruin. *Experiments with Continuation Semantics.* PhD thesis, Free University, Amsterdam, 1986.

[dB88] J.W. de Bakker. Comparative semantics for flow of control in logic programming without logic. Technical Report CS-R8840, Centre for Mathematics and Computer Science, 1988.

[Eli91] A. Eliens. *A language for distributed logic programming.* PhD thesis, University of Amsterdam, 1991.

[Jon89] H.B.M. Jonkers. An introduction to COLD-K. In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications, Work shop Passau 1987*, volume 394 of *LNCS*, pages 139–205. Springer-Verlag, 1989.

[Jon91] H.B.M. Jonkers. PROTOCOLD 1.1 user manual. Technical report RWR-513-hj-91080-hj, Philips Research Laboratories, August 1991.

[Kahn87] G. Kahn. Natural Semantics. Report de Recherche No 601, INRIA, Sofia Antipolis, February 1987.

[LFdL87] C.P.J. Koymans L.M.G. Feijs, H.B.M. Jonkers and G.R. Renardel de Lavelette. Formal definition of the design language COLD-K. Technical report, Philips Research Laboratories, April 1987.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

A Running Example; PROTOCOLD 1.1

In this appendix we take the language definition of PROTOCOLD 1.1 and provide it with a *sequentialized* semantics. We will not give a formal definition of the language, this can be found in [Jon91]. Roughly, it is that subset of PROTOCOLD obeying the following requirements:

1. The binding status of a variable must be equivalent in both components of a choice, thus allowing $((x := 1)|(x := 2)); x = 3$ but excluding $((x := 1)|(y := 2)); x = 3$.

2. If a variable x occurs in an assertion α, then there must be a binding before α on x. This avoids dealing with *assumptions*, hence we may simplify the notion of a state to a ***binding function.*** As check function $check_{rigid}$ can be taken. Hence, each state is simply a ***binding function.***

3. If a binding of a variable x is in the scope of the $NOT()$ operator, then the variable x be used afterwards; the fragment $NOT(x := 1); x := 2$ is not allowed.

   This avoids dealing with "negative information", and we can express the behavior of $NOT(\alpha)$ in terms of the behavior of α. It is not needed anymore to push the NOT inside the assertion until atomic assertions are reached.

Moreover PROTOCOLD has a restricted backtrack mechanism:

- *No backtracking is possible over **library** procedures at all. This can be expressed in our framework* by applying the strong cut over every ***library*** procedure application.

$$p_{lib}(\bar{t}) \text{ is rewritten to } (plib(\bar{t}))\$\$$$

- From the outside an *system* procedure is deterministic and only within the evaluation of the body backtracking is possible. This can be expressed by adding the soft cut to each *system* procedure definition.

$$\begin{array}{l}\textbf{PROC}\ p_{sys} : V_1\ \#...\#\ V_n \rightarrow \\ \textbf{IN}\ x_1, ..., x_n \\ \textbf{DEF} \\ \textbf{LET}\ y_1 : W_1, ..., y_m : W_m; \\ B\$\end{array}$$

Below we give some of the rules for a semantics formulated within PROTOCOLD 1.1. Thus no cut operators are involved, their behavior is incorporated in the rules for resp. the *library* procedures and the *system* procedures.

The *evaluate* function is simply defined as

$$\begin{array}{lcll} evaluate(x := t, \gamma) & = & (clear(x), \gamma[\gamma(t), x]) & if\ \gamma(x) = \perp \neq \theta(t) \\ evaluate(x := t, \gamma) & = & (\epsilon, \gamma) & if\ \gamma(x) = \theta(t) \neq \perp \\ evaluate(x := t, \gamma) & = & FAIL & otherwise \\ evaluate(\alpha, \gamma) & = & & if\ \gamma \in [\![\alpha]\!] \end{array}$$

where each state γ is now a *binding function* from Θ.

A.1 The Rules for Expressions

The evaluation of a *library* procedure sets the failure stack on Δ, expressing that no backtracking is possible. The evaluation of a *system* procedure delivers only the first alternative. The rest of the stack is being removed except for the *inverse* actions. Moreover, the local variables $\overline{w}$ may be removed, since no *assumptions* are maintained.

$$\frac{\gamma' = eval(p_{lib}(\overline{t}), \gamma) \neq FAIL}{p_{lib}(\overline{t})@\ :\ s_e, \gamma \rightarrow \Delta, \gamma'} \qquad \frac{eval(p_{lib}(\overline{t}), \gamma) = FAIL \quad s_e, \gamma \rightarrow c}{p_{lib}(\overline{t})@\ :\ s_e, \gamma \rightarrow c}$$

$$\frac{\begin{array}{c}\overline{w}\ \text{free in}\ \gamma \\ B_{\overline{t},\overline{w}}, [\overline{\perp}/\overline{w}]\ :\ \delta, \gamma \rightarrow s'_e, \gamma' \\ \gamma'' = [\![free(\overline{w})]\!]\gamma'\end{array}}{p_{sys}(\overline{t})@\ :\ s_e, \gamma \rightarrow sel\text{-}inv(s'_e, s_e), \gamma''}$$

A.2 The Rules for Assertions

The reduction $\rightsquigarrow$ is not used anymore. We have two simple rules for the $NOT()$. Note that α fails in γ whenever $\alpha\ :\ \delta, \gamma \rightarrow \delta, \gamma'$, where it is guaranteed that $\gamma = \gamma'$.

$$\frac{\alpha\ :\ \delta, \gamma \rightarrow \delta, \gamma'}{NOT(\alpha)\ :\ s_\alpha, \gamma \rightarrow s_\alpha, \gamma'} \qquad \frac{\alpha\ :\ \delta, \gamma \rightarrow s'_\alpha, \gamma' \quad sel\text{-}inv(s'_\alpha, s_\alpha), \gamma' \rightarrow c}{NOT(\alpha)\ :\ s_\alpha, \gamma \rightarrow c}$$

Note that $NOT(x := 3)$ in $\gamma[\perp\ /x]$ results in $\gamma[3/x]$ which is harmless since the well formedness requirements guarantee that x will not be used anymore.

A.3 The Rules for Terms

We still need the reduction relation $\rightsquigarrow$ on terms to reduce all terms of the form $t!$ to a terms without ! operator.

B Abstract Semantics

$$\frac{\alpha \rightsquigarrow \alpha' \quad \alpha', \gamma \to \gamma'}{\alpha, \gamma \to \gamma'} \qquad \frac{\alpha, \gamma \to \gamma'}{\alpha?, \gamma \to \gamma'} \qquad \frac{t, \gamma \to u, \gamma' \quad x = u?, \gamma' \to \gamma''}{x := t, \gamma \to \gamma''} \qquad \frac{e_1, \gamma \to \gamma' \quad e_2, \gamma' \to \gamma''}{e_1; e_2, \gamma \to \gamma''} \qquad \frac{i = 1 \vee i = 2 \quad e_i, \gamma \to \gamma'}{e_1 | e_2, \gamma \to \gamma'}$$

$$\frac{\overline{t}, \gamma \to \overline{u}, \gamma' \quad \gamma'' = eval(p_{lib}(\overline{t}), \gamma') \neq FAIL}{p_{lib}(\overline{t}), \gamma \to \gamma''} \qquad \frac{\overline{t}, \gamma \to \overline{u}, \gamma' \quad \overline{w} \text{ free in } \gamma' \quad B_{\overline{u},\overline{w}}, \gamma'[\overline{\bot}/\overline{w}] \to \gamma''}{p_{sys}(\overline{t}), \gamma \to \gamma''}$$

$$\frac{\alpha \rightsquigarrow \alpha' \quad \alpha', \gamma \to \gamma'}{\alpha, \gamma \to \gamma'} \qquad TRUE, \gamma \to \gamma \qquad \frac{t, \gamma \to u, \gamma' \quad x = u?, \gamma' \to \gamma''}{x := t, \gamma \to \gamma''} \qquad \frac{\alpha_1, \gamma \to \gamma' \quad \alpha_2, \gamma' \to \gamma''}{\alpha_1 \wedge \alpha_2, \gamma \to \gamma''} \qquad \frac{i = 1 \vee i = 2 \quad \alpha_i, \gamma \to \gamma'}{\alpha_1 \vee \alpha_2, \gamma \to \gamma'}$$

$$\alpha(\overline{t}) \text{ in } r_{sys}(\overline{t}) \mid \neg(r_{sys}(\overline{t})) \mid f_{sys}(\overline{t})! \mid \neg(f_{sys}(\overline{t})!)$$

$$\frac{\overline{t}!, \gamma \to \overline{u}, \gamma' \quad \overline{w} \text{ free in } \gamma' \quad B^{\alpha(\overline{t})}_{\overline{u},\overline{w}}, \gamma'[\overline{\bot}/\overline{w}] \to \gamma''}{\alpha(\overline{t}), \gamma \to \gamma''}$$

$$\alpha(\overline{t}) \text{ in } t_1 = t_2 \mid t_1 \neg = t_2 \mid r_{lib}(\overline{t}) \mid \neg(r_{lib}(\overline{t})) \mid t \in \mathcal{T}^{at}\ t! \mid t \in \mathcal{T}^{at}\ NOT(t!)$$

$$\frac{\overline{t} \not\subset \mathcal{T}^{at} \quad \overline{t}!, \gamma \to \overline{u}, \gamma' \quad \alpha(\overline{u}), \gamma' \to \gamma''}{\alpha(\overline{t}), \gamma \to \gamma''} \qquad \frac{\overline{t} \subset \mathcal{T}^{at} \quad FAIL \neq \gamma' = eval(\alpha(\overline{t}), \gamma)}{\alpha(\overline{t}), \gamma \to \gamma'}$$

$$\frac{t \rightsquigarrow t' \quad t', \gamma \to u, \gamma'}{t, \gamma \to u, \gamma'} \qquad \frac{t \in \mathcal{T}^{at} \quad \gamma' = eval(t!, \gamma) \neq FAIL}{t, \gamma \to t, \gamma'} \qquad \frac{\alpha, \gamma \to \gamma' \quad t, \gamma' \to t', \gamma''}{\alpha?; t, \gamma \to t', \gamma''} \qquad \frac{t, \gamma \to u, \gamma' \quad x = u?; t', \gamma' \to t'', \gamma''}{x := t; t' \to t'', \gamma''}$$

$$\frac{i = 1 \vee i = 2 \quad t_i, \gamma \to t, \gamma'}{t_1 | t_2, \gamma \to t, \gamma'} \qquad \frac{\overline{t}!, \gamma \to \overline{u}, \gamma' \quad f_{lib}(\overline{u}), \gamma \to t, \gamma''}{f_{lib}(\overline{t}), \gamma \to t, \gamma''} \qquad \frac{\overline{t}!, \gamma \to \overline{u}, \gamma' \quad \overline{w} \text{ free in } \gamma' \quad (B_{\overline{u},\overline{w}})!, \gamma'[\overline{\bot}/\overline{w}] \to t, \gamma'}{f_{sys}(\overline{t}), \gamma \to t, \gamma'}$$

$$\frac{t_1, \gamma_1 \to u_1, \gamma_2 \quad \ldots \quad t_n, \gamma_n \to u_n, \gamma_{n+1}}{(t_1, \ldots, t_n), \gamma_1 \to (u_1, \ldots, u_n), \gamma_{n+1}}$$

C Backtracking Semantics

C.1 The Rules for Expressions

$\delta \rightarrow \delta$	$\dfrac{e_1, \gamma \ : \ e_2, \gamma \ : \ s_e \rightarrow s'_e}{e_1 \vert e_2, \gamma \ : \ s_e \rightarrow s'_e}$	$\dfrac{(x := t)?, \gamma \ : \ s_e \rightarrow s'_e}{x := t, \gamma \ : \ s_e \rightarrow s'_e}$
$\dfrac{(\alpha, \gamma \ : \ \delta)? \ : \ s_e \rightarrow s'_e}{\alpha?, \gamma \ : \ s_e \rightarrow s'_e}$	$\dfrac{s_\alpha \rightarrow \gamma \ : \ s'_\alpha}{s_\alpha? \ : \ s_e \rightarrow \gamma \ : \ s'_\alpha? \ : \ s_e}$	$\dfrac{\begin{array}{c} s_\alpha \rightarrow \delta \\ s_e \rightarrow s'_e \end{array}}{s_\alpha? \ : \ s_e \rightarrow s'_e}$
$\dfrac{(e_1, \gamma : \delta); e_2 \ : \ s_e \rightarrow s'_e}{e_1; e_2, \gamma \ : \ s_e \rightarrow s'_e}$	$\dfrac{\begin{array}{c} s_e \rightarrow \gamma \ : \ s''_e \\ e, \gamma \ : \ s''_e; e \ : \ s'_e \rightarrow s'''_e \end{array}}{s_e; e \ : \ s'_e \rightarrow s'''_e}$	$\dfrac{\begin{array}{c} s_e \rightarrow \delta \\ s'_e \rightarrow s''_e \end{array}}{s_e; e \ : \ s'_e \rightarrow s''_e}$
$\dfrac{p((t_1, \gamma \ : \ \delta), t_2, ..., t_n) \ : \ s_e \rightarrow s'_e}{p(t_1, ..., t_n), \gamma \ : \ s_e \rightarrow s'_e}$	$\dfrac{\begin{array}{c} s_{\overline{t}} \rightarrow \overline{u}, \gamma \ : \ s'_{\overline{t}} \\ p(\overline{u})@, \gamma \ : \ p(s'_{\overline{t}}) \ : \ s_e \rightarrow s'_e \end{array}}{p(s_{\overline{t}}) \ : \ s_e \rightarrow s'_e}$	$\dfrac{\begin{array}{c} s_{\overline{t}} \rightarrow \delta \\ s_e \rightarrow s'_e \end{array}}{p(s_{\overline{t}}) \ : \ s_e \rightarrow s'_e}$
$\dfrac{\gamma' = eval(p_{lib}(\overline{t}), \gamma) \neq FAIL}{p_{lib}(\overline{t})@, \gamma \ : \ s_e \rightarrow \gamma' \ : \ s_e}$	$\dfrac{\begin{array}{c} eval(p_{lib}(\overline{t}), \gamma) = FAIL \\ s_e \rightarrow s'_e \end{array}}{p_{lib}(\overline{t})@, \gamma \ : \ s_e \rightarrow s'_e}$	$\dfrac{\begin{array}{c} \overline{w} \text{ free in } \gamma \\ B_{\overline{t}, \overline{w}}, \gamma[\overline{\perp}/\overline{w}] \ : \ s_e \rightarrow s'_e \end{array}}{p_{sys}(\overline{t})@, \gamma \ : \ s_e \rightarrow s'_e}$

C.2 The Rules for Assertions

$$\delta \to \delta$$

$$\frac{\alpha \rightsquigarrow \alpha' \quad \alpha', \gamma : s_\alpha \to s'_\alpha}{\alpha, \gamma : s_\alpha \to s'_\alpha}$$

$$TRUE, \gamma : s_\alpha \to \gamma : s_\alpha$$

$$\frac{s_\alpha \to s'_\alpha}{FALSE, \gamma : s_\alpha \to s'_\alpha}$$

$$\frac{\alpha_1, \gamma : \alpha_2, \gamma : s_\alpha \to s'_\alpha}{\alpha_1 \vee \alpha_2, \gamma : s_\alpha \to s'_\alpha}$$

$$\frac{x := (t, \gamma : \delta) : s_\alpha \to s'_\alpha}{x := t, \gamma : s_\alpha \to s'_\alpha}$$

$$\frac{s_t \to t, \gamma : s'_t \quad x = t?, \gamma : x := s'_t : s_\alpha \to s'_\alpha}{x := s_t : s_\alpha \to s'_\alpha}$$

$$\frac{s_t \to \delta \quad s_\alpha \to s'_\alpha}{x := s_t : s_\alpha \to s'_\alpha}$$

$$\frac{(\alpha_1, \gamma : \delta) \wedge \alpha_2 : s_\alpha \to s'_\alpha}{\alpha_1 \wedge \alpha_2, \gamma : s_\alpha \to s'_\alpha}$$

$$\frac{s_\alpha \to \gamma : s''_\alpha \quad \alpha, \gamma : s''_\alpha \wedge \alpha : s'_\alpha \to s'''_\alpha}{s_\alpha \wedge \alpha : s'_\alpha \to s'''_\alpha}$$

$$\frac{s_\alpha \to \delta \quad s'_\alpha \to s''_\alpha}{s_\alpha \wedge \alpha : s'_\alpha \to s''_\alpha}$$

$$\alpha(\overline{t}) \text{ in } t_1 = t_2 \mid t_1 \neg = t_2 \mid r(\overline{t}) \mid \neg(r(\overline{t})) \mid f(\overline{t})! \mid \neg(f(\overline{t})!)$$

$$\frac{\alpha((t_1!, \gamma : \delta), t_2!, \ldots, t_n!) : s_\alpha \to s'_\alpha}{\alpha(t_1, \ldots, t_n), \gamma : s_\alpha \to s'_\alpha}$$

$$\frac{s_{\overline{t}} \to \overline{u}, \gamma : s'_{\overline{t}} \quad \alpha(\overline{u})@, \gamma : \alpha(s'_{\overline{t}}) : s_\alpha \to s'_\alpha}{\alpha(s_{\overline{t}}) : s_\alpha \to s'_\alpha}$$

$$\frac{s_{\overline{t}} \to \delta \quad s_\alpha \to s'_\alpha}{\alpha(s_{\overline{t}}) : s_\alpha \to s'_\alpha}$$

$$\alpha(\overline{t}) \text{ in } t_1 = t_2 \mid t_1 \neg = t_2 \mid r_{lib}(\overline{t}) \mid \neg(r_{lib}(\overline{t})) \mid f_{lib}(\overline{t})! \mid \neg(f_{lib}(\overline{t})!)$$

$$\frac{\gamma' = eval(\alpha(\overline{t}), \gamma) \neq FAIL}{\alpha(\overline{t})@, \gamma : s_\alpha \to \gamma' : s_\alpha}$$

$$\frac{eval(\alpha(\overline{t}), \gamma) = FAIL \quad s_\alpha \to s'_\alpha}{\alpha(\overline{t})@, \gamma : s_\alpha \to \gamma' : s_\alpha}$$

$$\alpha(\overline{t}) \text{ in } r_{sys}(\overline{t}) \mid \neg(r_{sys}(\overline{t}) \mid f_{sys}(\overline{t})! \mid \neg(f_{sys}(\overline{t})!)$$

$$\frac{\overline{w} \text{ free in } \gamma \quad B^{\alpha(\overline{t})}_{\overline{t},\overline{w}}, \gamma[\overline{\perp}/\overline{w}] : s_\alpha \to s'_\alpha}{\alpha(\overline{t})@, \gamma : s_\alpha \to s'_\alpha}$$

C.3 The Rules for Terms

$$\delta \to \delta \qquad \frac{t \rightsquigarrow t' \quad t', \gamma \,:\, s_t \to s'_t}{t, \gamma \,:\, s_t \to s'_t} \qquad \frac{t_1, \gamma \,:\, t_2, \gamma \,:\, s_t \to s'_t}{t_1 | t_2, \gamma \,:\, s_t \to s'_t} \qquad \frac{(x := t)?; t', \gamma \,:\, s_t \to s'_t}{x := t; t', \gamma \,:\, s_t \to s'_t}$$

$$\frac{(\alpha, \gamma \,:\, \delta)?; t \,:\, s_t \to s'_t}{\alpha?; t, \gamma \,:\, s_t \to s'_t} \qquad \frac{s_\alpha \to \gamma \,:\, s'_\alpha \quad t, \gamma \,:\, s'_\alpha?; t \,:\, s_t \to s'_t}{s_\alpha?; t \,:\, s_t \to s'_t} \qquad \frac{s_\alpha \to \delta \quad s_t \to s'_t}{s_\alpha?; t \,:\, s_t \to s'_t}$$

$$\frac{f((t_1, \gamma \,:\, \delta), t_2, ..., t_n) \,:\, s_t \to s'_t}{f(t_1, ..., t_n), \gamma \,:\, s_t \to s'_t} \qquad \frac{s_{\overline{t}} \to \overline{u}, \gamma \,:\, s'_{\overline{t}} \quad f(\overline{u})@, \gamma \,:\, f(s'_{\overline{t}}) \,:\, s_t \to s'_t}{f(s_{\overline{t}}) \,:\, s_t \to s'_t} \qquad \frac{s_{\overline{t}} \to \delta \quad s_t \to s'_t}{f(s_{\overline{t}}) \,:\, s_t \to s'_t}$$

$$\frac{\gamma' = eval(f_{lib}(\overline{t})!, \gamma) \neq FAIL}{f_{lib}(\overline{t})@, \gamma \,:\, s_t \to f_{lib}(\overline{t})@, \gamma' \,:\, s_t} \qquad \frac{eval(f_{lib}(\overline{t})!, \gamma) = FAIL \quad s_t \to s'_t}{f_{lib}(\overline{t})@, \gamma \,:\, s_t \to s'_t} \qquad \frac{\overline{w} \text{ free in } \gamma \quad (B_{\overline{t},\overline{w}})!, \gamma[\overline{\perp}/\overline{w}] \,:\, s_t \to s'_t}{f_{sys}(\overline{t})@, \gamma \,:\, s_t \to s'_t}$$

$$v, \gamma \,:\, s_t \to v, \gamma \,:\, s_t$$

$$\delta \to \delta$$

$$\frac{1 \leq j < n \quad s_t \to t, \gamma \,:\, s'_t \quad t_1, ..., t_{j-1}, t, (t_{j+1}, \gamma \,:\, \delta), ..., t_n \,:\, t_1, ..., t_{j-1}, s'_t, t_{j+1}, ..., t_n \,:\, s_{\overline{t}} \to s'_{\overline{t}}}{t_1, ..., t_{j-1}, s_t, t_{j+1}, ..., t_n \,:\, s_{\overline{t}} \to s'_{\overline{t}}}$$

$$\frac{s_t \to t, \gamma \,:\, s'_t}{t_1, ..., t_{n-1}, s_t \,:\, s_{\overline{t}} \to (t_1, ..., t_{n-1}, t), \gamma \,:\, t_1, ..., t_{n-1}, s'_t \,:\, s_{\overline{t}}}$$

$$\frac{1 \leq j \leq n \quad s_t \to \delta \quad s_{\overline{t}} \to s'_{\overline{t}}}{t_1, ..., t_{j-1}, s_t, t_{j+1}, ..., t_n \,:\, s_{\overline{t}} \to s'_{\overline{t}}}$$

D Sequentialized Semantics

D.1 The Rules for Expressions

$$\delta, \gamma \rightarrow \delta, \gamma$$

$$\frac{e_1 \ : \ e_2 \ : \ s_e, \gamma \rightarrow c}{e_1 | e_2, \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{(x := t)? \ : \ s_e, \gamma \rightarrow c}{x := t \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{(\alpha \ : \ \delta)? \ : \ s_e, \gamma \rightarrow c}{\alpha? \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{s_\alpha, \gamma \rightarrow s'_\alpha, \gamma'}{s_\alpha? \ : \ s_e, \gamma \rightarrow s'_\alpha? \ : \ s_e, \gamma'}$$

$$\frac{s_\alpha, \gamma \rightarrow \delta, \gamma' \quad s_e, \gamma' \rightarrow c}{s_\alpha? \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{(e_1 : \delta); e_2 \ : \ s_e, \gamma \rightarrow c}{e_1; e_2 \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{s_e, \gamma \rightarrow s''_e, \gamma' \quad e \ : \ s''_e; e \ : \ s'_e, \gamma' \rightarrow c}{s_e; e \ : \ s'_e, \gamma \rightarrow c}$$

$$\frac{s_e, \gamma \rightarrow \delta, \gamma' \quad s'_e, \gamma' \rightarrow c}{s_e; e \ : \ s'_e, \gamma \rightarrow c}$$

$$\frac{p((t_1 \ : \ \delta), t_2, ..., t_n) \ : \ s_e, \gamma \rightarrow c}{p(t_1, ..., t_n) \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{s_{\overline{t}}, \gamma \rightarrow \overline{u} \ : \ s'_{\overline{t}}, \gamma' \quad p(\overline{u})@ \ : \ p(s'_{\overline{t}}) \ : \ s_e, \gamma' \rightarrow c}{p(s_{\overline{t}}) \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{s_{\overline{t}}, \gamma \rightarrow \delta, \gamma' \quad s_e, \gamma' \rightarrow c}{p(s_{\overline{t}}) \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{(i, \gamma') = eval(p_{lib}(\overline{t}), \gamma) \neq FAIL}{p_{lib}(\overline{t})@ \ : \ s_e, \gamma \rightarrow i \ : \ s_e, \gamma'}$$

$$\frac{eval(p_{lib}(\overline{t}), \gamma) = FAIL \quad s_e, \gamma \rightarrow c}{p_{lib}(\overline{t})@ \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{\overline{w} \text{ free in } \gamma \quad B_{\overline{t},\overline{w}}, [\overline{\perp}/\overline{w}] \ : \ s_e, \gamma \rightarrow c}{p_{sys}(\overline{t})@ \ : \ s_e, \gamma \rightarrow c}$$

$$\frac{\gamma' = eval(i, \gamma) \quad s_e, \gamma' \rightarrow c}{i \ : \ s_e, \gamma \rightarrow c}$$

D.2 The Rules for Assertions

$$\delta, \gamma \rightarrow \delta, \gamma$$

$$TRUE \ : \ s_\alpha, \gamma \rightarrow \epsilon \ : \ s_\alpha, \gamma$$

$$\frac{\alpha \rightsquigarrow \alpha' \quad \alpha' \ : \ s_\alpha, \gamma \rightarrow c}{\alpha \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{s_\alpha, \gamma \rightarrow c}{FALSE \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{\alpha_1 \ : \ \alpha_2 \ : \ s_\alpha, \gamma \rightarrow c}{\alpha_1 \ \vee \ \alpha_2 \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{x := (t \ : \ \delta) \ : \ s_\alpha, \gamma \rightarrow c}{x := t \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{s_t, \gamma \rightarrow t \ : \ s'_t, \gamma' \quad x = t? \ : \ x := s'_t \ : \ s_\alpha, \gamma' \rightarrow c}{x := s_t \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{s_t, \gamma \rightarrow \delta, \gamma' \quad s_\alpha, \gamma' \rightarrow c}{x := s_t \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{(\alpha_1 \ : \ \delta) \ \wedge \ \alpha_2 \ : \ s_\alpha, \gamma \rightarrow c}{\alpha_1 \ \wedge \ \alpha_2 \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{s_\alpha, \gamma \rightarrow s''_\alpha, \gamma' \quad \alpha \ : \ s''_\alpha \ \wedge \ \alpha \ : \ s'_\alpha, \gamma' \rightarrow c}{s_\alpha \ \wedge \ \alpha \ : \ s'_\alpha, \gamma \rightarrow c}$$

$$\frac{s_\alpha, \gamma \rightarrow \delta, \gamma' \quad s'_\alpha, \gamma' \rightarrow c}{s_\alpha \ \wedge \ \alpha \ : \ s'_\alpha, \gamma \rightarrow c}$$

$\alpha(\overline{t})$ in $t_1 = t_2 \mid t_1 \neg = t_2 \mid r(\overline{t}) \mid \neg(r(\overline{t})) \mid f(\overline{t})! \mid \neg(f(\overline{t})!)$

$$\frac{\alpha((t_1! \ : \ \delta), t_2!, ..., t_n!) \ : \ s_\alpha, \gamma \rightarrow c}{\alpha(t_1, ..., t_n) \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{s_{\overline{t}}, \gamma \rightarrow \overline{u} \ : \ s'_{\overline{t}} \quad \alpha(\overline{u})@ \ : \ \alpha(s'_{\overline{t}}) \ : \ s_\alpha, \gamma \rightarrow c}{\alpha(s_{\overline{t}}) \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{s_{\overline{t}}, \gamma \rightarrow \delta, \gamma' \quad s_\alpha, \gamma' \rightarrow c}{\alpha(s_{\overline{t}}) \ : \ s_\alpha, \gamma \rightarrow c}$$

$\alpha(\overline{t})$ in $t_1 = t_2 \mid t_1 \neg = t_2 \mid r_{lib}(\overline{t}) \mid \neg(r_{lib}(\overline{t})) \mid f_{lib}(\overline{t})! \mid \neg(f_{lib}(\overline{t})!)$

$$\frac{(i, \gamma') = eval(\alpha(\overline{t}), \gamma) \neq FAIL}{\alpha(\overline{t})@ \ : \ s_\alpha, \gamma \rightarrow i \ : \ s_\alpha, \gamma'}$$

$$\frac{eval(\alpha(\overline{t}), \gamma) = FAIL \quad s_\alpha, \gamma \rightarrow c}{\alpha(\overline{t})@ \ : \ s_\alpha, \gamma \rightarrow c}$$

$\alpha(\overline{t})$ in $r_{sys}(\overline{t}) \mid \neg(r_{sys}(\overline{t}) \mid f_{sys}(\overline{t})! \mid \neg(f_{sys}(\overline{t})!)$

$$\frac{\overline{w} \text{ free in } \gamma \quad B^{\alpha(\overline{t})}_{\overline{t},\overline{w}} \ : \ s_\alpha, \gamma[\overline{\perp}/\overline{w}] \rightarrow c}{\alpha(\overline{t})@ \ : \ s_\alpha, \gamma \rightarrow c}$$

$$\frac{\gamma' = eval(i, \gamma) \quad s_\alpha, \gamma' \rightarrow c}{i \ : \ s_\alpha \rightarrow c}$$

D.3 The Rules for Terms

$$\delta, \gamma \to \delta, \gamma$$

$$\frac{t \rightsquigarrow t' \quad t' \ : \ s_t, \gamma \to c}{t \ : \ s_t, \gamma \to c}$$

$$\frac{\gamma' = eval(i, \gamma)}{i \ : \ s_t, \gamma \to c}$$

$$\frac{t_1 \ : \ t_2 \ : \ s_t, \gamma \to c}{t_1 | t_2 \ : \ s_t, \gamma \to c}$$

$$\frac{(x := t)?; t' \ : \ s_t, \gamma \to c}{x := t; t' \ : \ s_t, \gamma \to c}$$

$$\frac{(\alpha \ : \ \delta)?; t \ : \ s_t, \gamma \to c}{\alpha?; t \ : \ s_t, \gamma \to c}$$

$$\frac{s_\alpha, \gamma \to s'_\alpha, \gamma' \quad t \ : \ s'_\alpha?; t \ : \ s_t, \gamma' \to c}{s_\alpha?; t \ : \ s_t, \gamma \to c}$$

$$\frac{s_\alpha, \gamma \to \delta, \gamma' \quad s_t, \gamma \to c}{s_\alpha?; t \ : \ s_t, \gamma \to c}$$

$$\frac{f((t_1 \ : \ \delta), t_2, ..., t_n) \ : \ s_t, \gamma \to c}{f(t_1, ..., t_n) \ : \ s_t, \gamma \to c}$$

$$\frac{s_{\overline{t}}, \gamma \to \overline{u} \ : \ s'_{\overline{t}}, \gamma' \quad f(\overline{u})@ \ : \ f(s'_{\overline{t}}) \ : \ s_t, \gamma' \to c}{f(s_{\overline{t}}) \ : \ s_t, \gamma \to c}$$

$$\frac{s_{\overline{t}}, \gamma \to \delta, \gamma' \quad s_t, \gamma' \to c}{f(s_{\overline{t}}) \ : \ s_t, \gamma \to c}$$

$$\frac{(i, \gamma') = eval(f_{lib}(\overline{t})!, \gamma) \neq FAIL}{f_{lib}(\overline{t})@ \ : \ s_t, \gamma \to f_{lib}(\overline{t}) \ : \ i \ : \ s_t, \gamma'}$$

$$\frac{eval(f_{lib}(\overline{t})!, \gamma) = FAIL \quad s_t, \gamma \to c}{f_{lib}(\overline{t})@ \ : \ s_t, \gamma \to c}$$

$$\frac{\overline{w} \text{ free in } \gamma \quad (B_{\overline{t}, \overline{w}})!, [\overline{\perp}/\overline{w}] \ : \ s_t, \gamma \to c}{f_{sys}(\overline{t})@ \ : \ s_t, \gamma \to c}$$

$$v \ : \ s_t, \gamma \to v \ : \ s_t, \gamma$$

$$\delta, \gamma \to \delta$$

$$\frac{1 \leq j < n \quad s_t, \gamma \to t, \ : \ s'_t \quad t_1, ..., t_{j-1}, t, (t_{j+1}, \ : \ \delta), ..., t_n \ : \ t_1, ..., t_{j-1}, s'_t, t_{j+1}, ..., t_n \ : \ s_{\overline{t}}, \gamma \to s'_{\overline{t}}}{t_1, ..., t_{j-1}, s_t, t_{j+1}, ..., t_n \ : \ s_{\overline{t}}, \gamma \to s'_{\overline{t}}}$$

$$\frac{s_t, \gamma \to t, \ : \ s'_t}{t_1, ..., t_{n-1}, s_t \ : \ s_{\overline{t}}, \gamma \to (t_1, ..., t_{n-1}, t), \ : \ t_1, ..., t_{n-1}, s'_t \ : \ s_{\overline{t}}}$$

$$\frac{1 \leq j \leq n \quad s_t, \gamma \to \delta, \gamma' \quad s_{\overline{t}}, \gamma' \to s'_{\overline{t}}}{t_1, ..., t_{j-1}, s_t, t_{j+1}, ..., t_n \ : \ s_{\overline{t}}, \gamma \to s'_{\overline{t}}}$$