View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Queensland eSpace

Structural Operational Semantics through Context-Dependent Behaviour

Robert Colvin
Ian J. Hayes

February 2010

Technical Report SSE-2010-02

Division of Systems and Software Engineering Research
School of Information Technology and Electrical Engineering
The University of Queensland
QLD, 4072, Australia

http://www.itee.uq.edu.au/∼sse

Structural Operational Semantics through Context-Dependent Behaviour

Robert Colvin[a,b], Ian J. Hayes[a,c]

[a]The University of Queensland
[b]The Queensland Brain Institute
[c] The School of Information Technology and Electrical Engineering

Abstract

We present an operational semantics for an imperative language with concurrency and procedures. The approach is novel because we expose the building block operations – variable assignment and condition checking – in the labels on the transitions; these form the *context-dependent behaviour* of a program. Using this style results in two main advantages over existing formalisms for imperative programming language semantics: firstly, our individual transition rules are less cluttered, and secondly, we are able to more abstractly and intuitively describe the semantics of procedures, including by-value and by-reference parameters. Existing techniques in the literature tend to result in complex and hard-to-read rules for even simple language constructs, or avoid discussion of procedures and parameters entirely. Our semantics for procedures utilises the context-dependent behaviour in the transition label to neatly handle variable name scoping, and defines the semantics of recursion without requiring additional rules. We also demonstrate how the semantics may be extended to handle function calls within expressions, array element and record field updates, and higher-order programming constructs.

1. Introduction

In 1981, Plotkin [1, 2] described the semantics of an imperative programming language using *structural operational semantics* (SOS). This description was considerably simpler than previous formulations, which contained a significant amount of implementation detail that got in the way of a higher-level understanding of the basic constructs. SOS has become one of the preferred methods for describing language semantics, from sequential programming languages [3, 4, 5] to process calculi [6, 7, 8].

In standard, or Plotkin-style, SOS approaches, the value of variables appear as a meta-level construct in all transition rules, and the environment, which includes procedure definitions and constants, is collected starting from the outermost scope, with inner environments overriding the outer. Then the effect of a basic command such as an assignment can be precisely determined because its context is supplied.

In this paper we employ a different approach to SOS, where the transition arrows are labelled by the behaviour associated with the step. We refer to the information in the label as the *context-dependent behaviour* of the command being executed, where the *context* consists of the values of variables and procedure definitions. The context-dependent behaviour of a basic command is nondeterministic, in that different behaviour will occur in different contexts. For instance, the evaluation of expression x has as many possible answers as there are possible values for variable x. However, given a particular context, in which $x = 3$ (say), there is only one evaluation, and this is the behaviour selected by the rules. The nondeterminism is resolved neatly by the style of transition rules found in an operational semantics.

The concept of context-dependent behaviour appearing in transition labels is not new, and is the typical approach to specifying the operational semantics of process calculi [9, 7]. However, in that setting, the context-dependent behaviour of a process, which is often called the *observable* behaviour, is usually an abstract event. In an imperative language setting, the context-dependent "events" are updates and tests of the value of variables.

The paper is organised as follows. In Sect. 2 we give a small-step semantics for expression evaluation. In Sect. 3 we give the dynamic semantics of commands within an imperative language with concurrency. In Sect. 4 we introduce procedures and parameters to the language, giving the semantics of call-by-value and call-by-reference parameter passing, and describing recursion. In Sect. 5 we return to expression evaluation, and provide a semantics for evaluating function calls within expressions. In Sect. 6 we define a notion of equivalence for the language. In Sect. 7 we address several issues identified by Plotkin [2] regarding the use of locations, and sketch how higher-order programming may be accommodated in our framework. In Sect. 8 we compare our approach to that of Plotkin in detail, and discuss other related work.

As the definition of the static semantics (e.g., type correctness) of our language would be no different to those of Plotkin, we do not address static semantics in this paper. By extension, we implicitly assume all commands and expressions appearing in our rules are well-formed. We present our language as untyped, although, as above, we assume that the rules for a typed language are straightforward extensions of those we provide, following similar principles to existing type systems [2, 3, 4].

2. Expression evaluation

In this section we introduce some of our basic syntax, in particular expressions (Sect. 2.1), and describe how they may be incrementally evaluated in a *small-step* style using operational semantics through context-dependent behaviour (Sect. 2.2)[1].

2.1. Basic syntax

Assume a set of values, Val, and a set of identifiers, $Ident$. We assume the set Val contains the standard boolean and integer values, as well as more complex values as required. The set $Ident$ is a set of identifiers, which are used for variable names, and later for procedure names.

A $State$ is a partial mapping from variable identifiers to values, $Ident \nrightarrow Val$. We use the notation $\{x \mapsto v\}$ to represent the state which maps x to the value v. The set of variables declared by a state σ are those in its domain, written dom(σ), e.g., dom($\{x \mapsto v\}$) $= \{x\}$.

Note there are two differences to handling the values of variables in Plotkin-style semantics: firstly, our states are partial, and secondly, we do not (need to) introduce another mapping to handle references to the same variable using locations (the *environment*).

[1]Because we will be describing a concurrent programming language with function calls in expressions, we do not give a big-step expression evaluation semantics. A big-step semantics may be derived from the small-step rules by generalising labels to accept arbitrary predicates – see [10].

The abstract syntax of an expression, $e \in \mathit{Expr}$, is given below, where $v \in \mathit{Val}$, $x \in \mathit{Ident}$, and $\sigma \in \mathit{State}$.

$$e ::= v \mid x \mid (e_1 + e_2) \mid \langle e, \sigma \rangle$$

An expression is either a value, a variable, the addition of two expressions, or $\langle e, \sigma \rangle$, which is an expression occurring within the scope of the state σ. The state may provide values for some of the variables in e, and is (part of) the *context* in which e appears. A more general expression language is given in Sect. 5.

2.2. Small-step evaluation

A small-step evaluation strategy allows a large degree of interleaving in a concurrent language – expressions are evaluated one step at a time, where a 'step' may be looking up the value of a variable or calculating the effect of some operation on values. Implementation languages will differ in the atomicity they allow in their evaluation strategies. Because we will eventually allow function calls to appear in expressions, small-step is the more natural choice than big-step, in which the expression is evaluated without interleaving. Other evaluation strategies with different atomicity may be based on the semantics we give here.

The semantics of expression evaluation is given through a labelled transition relation.

$$\longrightarrow : \mathit{Label} \nrightarrow (\mathit{Expr} \leftrightarrow \mathit{Expr})$$

It is defined as the least relation that satisfies the operational rules. A transition, written $e_1 \xrightarrow{l} e_2$, states that expression e_1 (partially) evaluates to expression e_2, provided that the label l is satisfied. The label l is the context-dependent predicate, that is, the transition occurs provided l holds in context. For expression evaluation, there are two possible forms for a label $l \in \mathit{Label}$: an equality $x = v$, where $x \in \mathit{Ident}$ and $v \in \mathit{Val}$, or true.

$$l ::= \mathit{true} \mid x = v \tag{1}$$

At this stage, a label is a (restricted form of) predicate which must be satisfied by the state. When describing the semantics of commands, the syntax of labels will be extended to also describe pre/post relationships between states.

The rules for defining expression evaluation are given in Fig. 1. A variable x evaluates to a value v provided $x = v$ in context (Rule 1). This transition defines a relationship between every variable and every value. For instance, assume that there are only two variables, $\mathit{Ident} = \{x, y\}$, and two values, $\mathit{Val} = \{0, 1\}$. Then the complete set of possible transitions is as follows.

$$x \xrightarrow{x=0} 0 \qquad x \xrightarrow{x=1} 1 \qquad\qquad y \xrightarrow{y=0} 0 \qquad y \xrightarrow{y=1} 1 \tag{2}$$

The rules for evaluating a binary addition enforce a strict left-to-right evaluation order, where the left-most operand must be fully evaluated to a value before the right-most operand is evaluated[2] (Rule 2(a) and (b)). The context-dependent behaviour of the left-most operand when being evaluated, l, is the context-dependent behaviour of the whole expression when being evaluated, and similarly for the right-most operand.

[2]Replacing both occurrences of v by e_1 in Rule 2(b) would allow any order of evaluation.

$$\mathit{Assume} \quad x \in \mathit{Ident} \quad v, v_i \in \mathit{Val} \quad e, e_i \in \mathit{Expr} \quad l \in \mathit{Label} \quad \sigma \in \mathit{State}$$

Rule 1 (Evaluate variable)

$$x \xrightarrow{x=v} v$$

Rule 2 (Evaluate addition)

$$(a)\ \frac{e_1 \xrightarrow{l} e_1'}{(e_1 + e_2) \xrightarrow{l} (e_1' + e_2)} \qquad (b)\ \frac{e_2 \xrightarrow{l} e_2'}{(v + e_2) \xrightarrow{l} (v + e_2')}$$

$$(c)\ \frac{v_1 + v_2 = v}{(v_1 + v_2) \xrightarrow{\mathit{true}} v}$$

Rule 3 (Evaluate in context)

$$(a)\ \ \langle v, \sigma \rangle \xrightarrow{\mathit{true}} v \qquad (b)\ \frac{e \xrightarrow{\mathit{true}} e'}{\langle e, \sigma \rangle \xrightarrow{\mathit{true}} \langle e', \sigma \rangle}$$

$$(c)\ \frac{e \xrightarrow{x=v} e' \quad x \in \mathit{dom}(\sigma) \quad \sigma(x) = v}{\langle e, \sigma \rangle \xrightarrow{\mathit{true}} \langle e', \sigma \rangle} \qquad (d)\ \frac{e \xrightarrow{x=v} e' \quad x \notin \mathit{dom}(\sigma)}{\langle e, \sigma \rangle \xrightarrow{x=v} \langle e', \sigma \rangle}$$

Figure 1: Small-step expression evaluation

When both have been evaluated to values, the sum is calculated (Rule 2(c)). Note the usual distinction between the symbol '+' on the bottom line, which is a syntactic construct, and the symbol '+' above the line, which denotes the semantics of addition. This form of transition rule may be used to specify the evaluation of other expressions; in particular we assume similar rules exist for calculating subtraction, multiplication, exponentiation, and inequalities.

As an example, consider the evaluation of expression $x + y$ in our state space with only two values. From Rule 2(a), Rule 1 and (2) we have two possible evaluations.

$$x + y \xrightarrow{x=0} 0 + y \qquad x + y \xrightarrow{x=1} 1 + y$$

Let us assume for now that $x = 0$ in context (the mechanics of this are explained below). Then continuing the evaluation via Rule 2(b), Rule 1 and (2):

$$0 + y \xrightarrow{y=0} 0 + 0 \qquad 0 + y \xrightarrow{y=1} 0 + 1$$

Finally, assuming $y = 1$ in context, we have the following final transition from Rule 2(c).

$$0 + 1 \xrightarrow{\mathit{true}} 1$$

The $true$ label on the transition indicates that, as expected, the expression $0+1$ evaluates to 1 in any context. No more evaluation is possible, since we have reduced the expression to the basic value 1.

Now let us consider evaluating an expression in some state σ supplied as a context. Rule 3(a) applies when the expression has already been fully evaluated to v, and therefore the state σ is no longer required and is eliminated. Rule 3(b) applies when the evaluation step has a condition of $true$. Such a step applies in any context, and the state σ is therefore irrelevant. For example, the following two transitions are justified by Rule 2(c) and Rule 3(b), and Rule 3(a), respectively, for any state σ.

$$\langle 0+1, \sigma\rangle \xrightarrow{true} \langle 1, \sigma\rangle \xrightarrow{true} 1$$

Rule 3(c) and Rule 3(d) handle the more interesting case where the label is $x = v$. We require two rules, depending on whether x is contained in the domain of σ or not, that is, whether x is local to the expression.

Let us consider the evaluation of the expression x in a state where $x = 0$. The following transition is justified by Rule 1 and Rule 3(c).

$$\langle x, \{x \mapsto 0\}\rangle \xrightarrow{true} \langle 0, \{x \mapsto 0\}\rangle$$

Although, as shown in (2), there are two possible evaluations of x, to either 0 or 1, when a state σ is provided only the value $\sigma(x)$ is allowed by the premise of Rule 3(c). This constraint eliminates the nondeterminism inherent in Rule 1. The transition $x \xrightarrow{x=1} 1$ cannot be selected once placed in context $\{x \mapsto 0\}$. Note that the transition label becomes $true$ – there is no dependence on an outer context.

When x is not in the domain of σ, as given by Rule 3(d), the label $x = v$ is preserved as it is dependent on an outer context.

As a demonstration, consider the step-by-step evaluation of the expression

$$\langle\langle x+y, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle \tag{3}$$

There are five transitions which are shown below, and the first four of which are justified in Fig. 2. First x is evaluated, then y, then sum is calculated, and finally the two now-redundant states are eliminated.

Notice that in Fig. 2 each transition can be read intuitively by treating the label as the pre-condition for the transition to occur. Other choices can be made when applying Rule 1 at the top of the rule 'pyramid' in (4) and (5), but such choices would not result in a successful application of Rule 3(c), and hence would not form part of an allowable transition.

$$\begin{aligned}
&\quad \langle\langle x+y, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle \\
&\xrightarrow{true} \text{See (4)} \\
&\quad \langle\langle 0+y, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle \\
&\xrightarrow{true} \text{See (5)} \\
&\quad \langle\langle 0+1, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle \\
&\xrightarrow{true} \text{See (6)} \\
&\quad \langle\langle 1, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle
\end{aligned}$$

$$\begin{array}{l} \textit{Rule } 1 \\ \textit{Rule } 2(a) \\ \textit{Rule } 3(c) \\ \textit{Rule } 3(b) \end{array} \qquad \dfrac{\dfrac{\dfrac{x \xrightarrow{x=0} 0}{x+y \xrightarrow{x=0} 0+y}}{\langle x+y, \{x \mapsto 0\}\rangle \xrightarrow{true} \langle 0+y, \{x \mapsto 0\}\rangle}}{\langle\langle x+y, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle \xrightarrow{true} \langle\langle 0+y, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle} \tag{4}$$

$$\begin{array}{l} \textit{Rule } 1 \\ \textit{Rule } 2(b) \\ \textit{Rule } 3(d) \\ \textit{Rule } 3(c) \end{array} \qquad \dfrac{\dfrac{\dfrac{y \xrightarrow{y=1} 1}{0+y \xrightarrow{y=1} 0+1}}{\langle 0+y, \{x \mapsto 0\}\rangle \xrightarrow{y=1} \langle 0+1, \{x \mapsto 0\}\rangle}}{\langle\langle 0+y, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle \xrightarrow{true} \langle\langle 0+1, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle} \tag{5}$$

$$\begin{array}{l} \textit{Rule } 2(c) \\ \textit{Rule } 3(b) \\ \textit{Rule } 3(b) \end{array} \qquad \dfrac{\dfrac{0+1 \xrightarrow{true} 1}{\langle 0+1, \{x \mapsto 0\}\rangle \xrightarrow{true} \langle 1, \{x \mapsto 0\}\rangle}}{\langle\langle 0+1, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle \xrightarrow{true} \langle\langle 1, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle} \tag{6}$$

$$\begin{array}{l} \textit{Rule } 3(a) \\ \textit{Rule } 3(b) \end{array} \qquad \dfrac{\langle 1, \{x \mapsto 0\}\rangle \xrightarrow{true} 1}{\langle\langle 1, \{x \mapsto 0\}\rangle, \{y \mapsto 1\}\rangle \xrightarrow{true} \langle 1, \{y \mapsto 1\}\rangle} \tag{7}$$

$$\begin{array}{l} \textit{Rule } 1 \\ \textit{Rule } 3(c) \\ \textit{Rule } 2(a) \end{array} \qquad \dfrac{\dfrac{x \xrightarrow{x=0} 0}{\langle x, \{x \mapsto 0\}\rangle \xrightarrow{true} \langle 0, \{x \mapsto 0\}\rangle}}{\langle x, \{x \mapsto 0\}\rangle + \langle x, \{x \mapsto 1\}\rangle \xrightarrow{true} \langle 0, \{x \mapsto 0\}\rangle + \langle x, \{x \mapsto 1\}\rangle} \tag{8}$$

Figure 2: Expression evaluation steps

$\xrightarrow{true}$ See (7)
$\langle 1, \{y \mapsto 1\}\rangle$
$\xrightarrow{true}$ Rule 3(a)
1

Because states are partial, variable names may be reused in separate (non-overlapping) parts of the expression, as demonstrated in the following evaluation.

$\langle x, \{x \mapsto 0\}\rangle + \langle x, \{x \mapsto 1\}\rangle$
$\xrightarrow{true}$ See (8)
$\langle 0, \{x \mapsto 0\}\rangle + \langle x, \{x \mapsto 1\}\rangle$
$\xrightarrow{true}$ Rule 3(a), Rule 2(a)

$0 + \langle x, \{x \mapsto 1\}\rangle$
$\xrightarrow{true}$ Similar reasoning to (8)
$0 + \langle 1, \{x \mapsto 1\}\rangle$
$\xrightarrow{true}$ Rule 3(a), Rule 2(a)
$0 + 1$
$\xrightarrow{true}$ Rule 2(c)
1

Note that in the first step (8) the test on the value of the left-most reference to x does not appear in the transition label at the outer level – it is resolved local to the left-most expression, and does not cause interference with the separate part of the expression which declares its own variable x.

Similarly, if the same identifier appears in nested scopes, the innermost state takes precedence, as shown in the following evaluation.

$\langle\langle x, \{x \mapsto 1\}\rangle, \{x \mapsto 0\}\rangle$
$\xrightarrow{true}$ Rule 1, Rule 3(c), Rule 3(b)
$\langle\langle 1, \{x \mapsto 1\}\rangle, \{x \mapsto 0\}\rangle$
$\xrightarrow{true}$ Rule 3(a), Rule 3(b)
$\langle 1, \{x \mapsto 0\}\rangle$
$\xrightarrow{true}$ Rule 3(a)
1

3. Command execution

In this section we describe the semantics of an imperative programming language with concurrency. Because we wish to allow interleaving execution of concurrent statements, the semantics of execution is small-step, as is expression evaluation.

3.1. Syntax

The abstract syntax of a command $c \in \mathit{Cmd}$ is described below, where $x \in \mathit{Ident}$, $e, b \in \mathit{Expr}$,

$$c ::= \mathbf{nil} \mid (x := e) \mid (c_1 \; ; \; c_2) \mid (c_1 \parallel c_2) \mid (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \mid (\mathbf{while}\ b\ \mathbf{do}\ c) \mid (\mathbf{state}\ \sigma \bullet c)$$

The special command $\mathbf{nil}$ indicates a terminated command. It can partake in no further action.

An assignment $x := e$ is the standard assignment command. For the moment we assume $x \in \mathit{Ident}$ and e is an expression as defined in the previous section, although in Sect. 5 we give a richer expression syntax for e, and in Sect. 7.3 consider other possibilities for the left-hand side.

Sequential composition "$c_1 \; ; \; c_2$" follows our usual understanding, and we assume an interleaving semantics for a parallel composition $c_1 \parallel c_2$. The conditional and while commands are standard.

The *local state* command $(\mathbf{state}\ \sigma \bullet c)$ declares variables in the domain of σ to be 'local' to c. This abstract command type corresponds to the concrete syntax of declaring a new variable and providing it with an initial value, that is its value in σ. Standard SOS differs as a variable is mapped to a location through an environment, and the global store maps that location to a value.

Local states may also be used to model constants, i.e., variables that do not appear on the left-hand side of an assignment during execution (or as by-reference parameters to procedures, Sect. 4). This constraint can be enforced statically through the concrete syntax, and hence we do not complicate the dynamic semantics by treating them differently.

3.2. Transitions

The semantics are defined with respect to a transition relation.

$$\longrightarrow : \mathit{Label} \to (\mathit{Cmd} \leftrightarrow \mathit{Cmd})$$

It is the least relation that satisfies the operational rules. The syntax of Labels (1) is extended to include assignments.

$$l ::= \mathit{true} \mid x = v \mid x := v$$

We do not syntactically distinguish transitions on expressions from transitions on commands, but we ensure the correct relation is always clear from the intended types. As an abbreviation we also omit labels of true from the transition arrow.

The new label type, $x := v$, requires that the context updates x to the value v. Hence, a transition $c \xrightarrow{x := v} c'$ states that command c transitions to c' and has the behaviour of setting x to v in some outer state.

3.3. Standard commands

The semantics of the language appears in Fig. 3. The command $\mathbf{nil}$ can take no transitions, and therefore there is no corresponding rule. An assignment (Rule 4) first evaluates the expression e (eventually) to a value, then exposes the assignment of that value in the label. The sequential composition rule (Rule 5) is standard: if the first operand can take some step then the composition also can, and when the first operand has terminated the second may execute. Rule 6 states that concurrent processes interleave their operations; a concurrent process terminates when both processes have terminated (Rule 6(c)).

Rule 7 states that a conditional first evaluates its guard (eventually) to a boolean value, and then chooses c_1 or c_2 depending on the result of the evaluation.

An iteration command is expanded into a conditional command (Rule 8), in which (a new copy of) the loop body c is executed if the condition holds before executing the loop again, and terminates otherwise.

Note that the rules are relatively uncluttered – there is no need to carry an environment or store around. We now look at the novel rules for states.

3.4. State-based rules

Rule 9 states that a local state is eliminated when its command terminates. This is similar to Rule 3(a).

Rule 10 states that if c can transition independently of the context, then so can $(\mathtt{state}\ \sigma \bullet c)$. Being context-independent, there is no constraint on σ, cf. Rule 3(b).

Rule 11(a) applies when the command c requires $x = v$ to be true to transition to c', and x is in the domain of σ. Syntax aside, this rule is effectively the same as Rule 3(c) – a transition is

$$Assume \quad x \in Ident \quad v \in Val \quad e, b \in Expr \quad c, c_i \in Cmd \quad l \in Label$$

Rule 4 (Assignment)

$$(a)\ \frac{e \xrightarrow{l} e'}{(x := e) \xrightarrow{l} (x := e')}$$

$$(b)\ \ (x := v) \xrightarrow{x = v} \mathbf{nil}$$

Rule 5 (Sequential composition)

$$(a)\ \frac{c_1 \xrightarrow{l} c_1'}{c_1 \ ; \ c_2 \xrightarrow{l} c_1' \ ; \ c_2}$$

$$(b)\ \ \mathbf{nil} \ ; \ c_2 \longrightarrow c_2$$

Rule 6 (Interleave)

$$(a)\ \frac{c_1 \xrightarrow{l} c_1'}{c_1 \parallel c_2 \xrightarrow{l} c_1' \parallel c_2} \qquad (b)\ \frac{c_2 \xrightarrow{l} c_2'}{c_1 \parallel c_2 \xrightarrow{l} c_1 \parallel c_2'} \qquad (c)\ \ (\mathbf{nil} \parallel \mathbf{nil}) \longrightarrow \mathbf{nil}$$

Rule 7 (Conditional)

$$(a)\ \frac{b \xrightarrow{l} b'}{(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \xrightarrow{l} (\mathbf{if}\ b'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)}$$

$$(b)\ \ (\mathbf{if}\ \mathit{true}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \longrightarrow c_1 \qquad (c)\ \ (\mathbf{if}\ \mathit{false}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) \longrightarrow c_2$$

Rule 8 (While)

$$(\mathbf{while}\ b\ \mathbf{do}\ c) \longrightarrow (\mathbf{if}\ b\ \mathbf{then}\ (c\ ;\ \mathbf{while}\ b\ \mathbf{do}\ c)\ \mathbf{else\ nil})$$

Figure 3: Command execution

possible only when v has the correct value for x in σ. The behaviour no longer depends on the context. Rule 11(b) applies when x is not in the domain of σ. Syntax aside, this rule is effectively the same as Rule 3(d). Since x is not in the domain of the state, the context-dependent behaviour remains the same.

The rules for assignments are the more interesting cases. Rule 12(a) is the rule where state changes occur. If $x := v$ is the context-dependent behaviour, and x is local to σ, then σ is updated appropriately. This is now context-independent. Following Plotkin's style we let $\sigma[x \mapsto v]$ represent the state which is equal to σ in every argument except x, which it maps to the value v.

For example,

$$(\mathbf{state}\ \{x \mapsto 1\} \bullet x := 0) \longrightarrow (\mathbf{state}\ \{x \mapsto 0\} \bullet \mathbf{nil})$$

In Rule 12(b), c is updating a variable that is not in the local state. There is therefore no change in σ, and the label remains the same, i.e.,

$$(\mathbf{state}\ \{x \mapsto 1\} \bullet y := 0) \xrightarrow{y = 0} (\mathbf{state}\ \{x \mapsto 1\} \bullet \mathbf{nil})$$

Rule 9 (Eliminate state)

$$(\mathbf{state}\ \sigma \bullet \mathbf{nil}) \longrightarrow \mathbf{nil}$$

Rule 10 (State – context independent)

$$\frac{c \longrightarrow c'}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma \bullet c')}$$

Rule 11 (State – guard)

(a)
$$\frac{c \xrightarrow{x=v} c' \quad x \in \mathit{dom}(\sigma) \quad \sigma(x) = v}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma \bullet c')}$$

(b)
$$\frac{c \xrightarrow{x=v} c' \quad x \notin \mathit{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet c) \xrightarrow{x=v} (\mathbf{state}\ \sigma \bullet c')}$$

Rule 12 (State – assignment)

(a)
$$\frac{c \xrightarrow{x \coloneqq v} c' \quad x \in \mathit{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet c) \longrightarrow (\mathbf{state}\ \sigma[x \mapsto v] \bullet c')}$$

(b)
$$\frac{c \xrightarrow{x \coloneqq v} c' \quad x \notin \mathit{dom}(\sigma)}{(\mathbf{state}\ \sigma \bullet c) \xrightarrow{x \coloneqq v} (\mathbf{state}\ \sigma \bullet c')}$$

Figure 4: Local state execution

3.5. Examples

In this section we demonstrate the semantics on some examples.

3.5.1. Sequential computation

Consider a simple program that swaps the value of two variables, x and y, before executing some command c. The swap makes use of a local variable tmp, initially 0, which is not available inside c.

$$\mathbf{state}\ \{x \mapsto 1, y \mapsto 2\} \bullet (\mathbf{state}\ \{\mathit{tmp} \mapsto 0\} \bullet \mathit{tmp} := x\ ;\ x := y\ ;\ y := \mathit{tmp})\ ;\ c$$

(We include c because otherwise the whole command would be equivalent to $\mathbf{nil}$.)

The expression, x, of the first assignment, $\mathit{tmp} := x$, is evaluated in context to 1. The evaluation is shown in detail in Fig. 5. The evaluation requires straightforward propagation of the label $x = 1$ outwards through the program until it finds the relevant state which contains the value for x. We omit the details of such trivial expression evaluations throughout the rest of the paper.

The assignment $\mathit{tmp} := 1$ then transitions to $\mathbf{nil}$, with the context-dependent behaviour being exactly $\mathit{tmp} := 1$. This is shown in detail in Fig. 5. As above, we omit the details of such trivial transitions.

The $\mathbf{nil}$, which has taken the place of the assignment to tmp, is eliminated through Rule 5(b), and then the expression y in $x := y$ is evaluated in context to 2.

$$\mathbf{state}\ \{x \mapsto 1, y \mapsto 2\} \bullet (\mathbf{state}\ \{\mathit{tmp} \mapsto 1\} \bullet x := 2\ ;\ y := \mathit{tmp})\ ;\ c$$

As above, the assignment transitions to $\mathbf{nil}$, with the context-dependent behaviour $x := 2$. This is passed through the inner state via Rule 12(b), and then results in a change in the outer state

Let σ^t_v abbreviate the state $\{tmp \mapsto v\}$ for all v, let $\sigma^{xy}_{v_1 v_2}$ abbreviate the state $\{x \mapsto v_1, y \mapsto v_2\}$ for all v_1, v_2, and let d abbreviate the command $(x := y \ ; \ y := tmp)$. Also abbreviate $\mathbf{state}$ to $\mathbf{st}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x \xrightarrow{x=1} 1}{tmp := x \xrightarrow{x=1} tmp := 1}}{tmp := x \ ; \ d \xrightarrow{x=1} tmp := 1 \ ; \ d}}{(\mathbf{st}\ \sigma^t_0 \bullet tmp := x \ ; \ d) \xrightarrow{x=1} (\mathbf{st}\ \sigma^t_0 \bullet tmp := 1 \ ; \ d)}}{(\mathbf{st}\ \sigma^t_0 \bullet tmp := x \ ; \ d) \ ; \ c \xrightarrow{x=1} (\mathbf{st}\ \sigma^t_0 \bullet tmp := 1 \ ; \ d) \ ; \ c}}{(\mathbf{st}\ \sigma^{xy}_{12} \bullet (\mathbf{st}\ \sigma^t_0 \bullet tmp := x \ ; \ d) \ ; \ c) \longrightarrow (\mathbf{st}\ \sigma^{xy}_{12} \bullet (\mathbf{st}\ \sigma^t_0 \bullet tmp := 1 \ ; \ d) \ ; \ c)}$$

The transition is justified, top-to-bottom, by Rule 1; Rule 4(a); Rule 5(a); Rule 11(b) (since $x \notin \text{dom}(\sigma_t)$); Rule 5(a); Rule 11(a).

$$\dfrac{\dfrac{\dfrac{\dfrac{tmp := 1 \xrightarrow{tmp \,:=\, 1} \mathbf{nil}}{tmp := 1 \ ; \ d \xrightarrow{tmp \,:=\, 1} \mathbf{nil} \ ; \ d}}{(\mathbf{st}\ \sigma^t_0 \bullet tmp := 1 \ ; \ d) \longrightarrow (\mathbf{st}\ \sigma^t_1 \bullet \mathbf{nil} \ ; \ d)}}{(\mathbf{st}\ \sigma^t_0 \bullet tmp := 1 \ ; \ d) \ ; \ c \longrightarrow (\mathbf{st}\ \sigma^t_1 \bullet \mathbf{nil} \ ; \ d) \ ; \ c}}{(\mathbf{st}\ \sigma^{xy}_{12} \bullet (\mathbf{st}\ \sigma^t_0 \bullet tmp := 1 \ ; \ d) \ ; \ c) \longrightarrow (\mathbf{st}\ \sigma^{xy}_{12} \bullet (\mathbf{st}\ \sigma^t_1 \bullet \mathbf{nil} \ ; \ d) \ ; \ c)}$$

The transition is justified, top-to-bottom, by Rule 4(b); Rule 5(a); Rule 12(a); Rule 5(a); Rule 10.

Figure 5: Transitions for swap

through Rule 12(a).

$$\mathbf{state}\ \{x \mapsto 2, y \mapsto 2\} \bullet (\mathbf{state}\ \{tmp \mapsto 1\} \bullet \mathbf{nil} \ ; \ y := tmp) \ ; \ c$$

After eliminating the $\mathbf{nil}$ and evaluating the expression tmp to 1, the remaining assignment results in an update to y in the outer state.

$$\mathbf{state}\ \{x \mapsto 2, y \mapsto 1\} \bullet (\mathbf{state}\ \{tmp \mapsto 1\} \bullet \mathbf{nil}) \ ; \ c$$

The inner state is no longer necessary, and is eliminated through Rule 9, resulting in the following command.

$$\mathbf{state}\ \{x \mapsto 2, y \mapsto 1\} \bullet c$$

3.5.2. Iteration

We present the execution of a simple sequential program below, which calculates $\sum_{i=1}^{N} i$ where N is the initial value of local variable a, and stores both intermediate results and the final value in some non-local variable b. In this case a is initially 2 and hence the final value of b is 3.

Rule 13 (Sequential assignment)

$$(a) \quad (x := v \; ; \; c) \xrightarrow{x := v}{}^{*} \; c$$

$$(b) \quad (\textbf{state}\; \{x \mapsto v_1\} \bullet x := v_2 \; ; \; c) \longrightarrow^{*} (\textbf{state}\; \{x \mapsto v_2\} \bullet c)$$

Proof. The proof of (a) is justified by Rule 4(b) and Rule 5(a), followed by Rule 5(b). The proof of (b) is justified similarly, using Rule 12(a). □

Rule 14 (Iterate while)

$$(a) \quad \frac{b \xrightarrow{cl}{}^{*} \; \mathit{true}}{(\textbf{while}\; b\; \textbf{do}\; c) \xrightarrow{cl}{}^{*} (c \; ; \textbf{while}\; b\; \textbf{do}\; c)} \qquad (b) \quad \frac{b \xrightarrow{cl}{}^{*} \; \mathit{false}}{(\textbf{while}\; b\; \textbf{do}\; c) \xrightarrow{cl}{}^{*} \; \textbf{nil}}$$

Proof. For (a):

$$\begin{array}{ll}
 & \textbf{while}\; b\; \textbf{do}\; c \\
\longrightarrow & \text{Rule 8} \\
 & \textbf{if}\; b\; \textbf{then}\; (c \; ; \textbf{while}\; b\; \textbf{do}\; c)\; \textbf{else nil} \\
\xrightarrow{cl}{}^{*} & \text{Assumption and Rule 7(a)} \\
 & \textbf{if}\; \mathit{true}\; \textbf{then}\; (c \; ; \textbf{while}\; b\; \textbf{do}\; c)\; \textbf{else nil} \\
\longrightarrow & \text{Rule 7(b)} \\
 & c \; ; \textbf{while}\; b\; \textbf{do}\; c
\end{array}$$

The proof of (b) follows from Rule 8, Rule 7(a) and Rule 7(c). □

Figure 6: Derived rules

$$\begin{array}{l}
\textbf{state}\; \{a \mapsto 2\} \bullet \\
\quad b := 0 \; ; \\
\quad (\textbf{while}\; a > 0\; \textbf{do} \\
\qquad b := a + b \; ; \\
\qquad a := a - 1)
\end{array} \tag{9}$$

To simplify the presentation, we make use of the derived rules in Fig. 6. The rules summarise a sequence of steps which include some simple manipulations, such as eliminating **nil** commands through Rule 5(b) and unfolding **while** commands. We use the notation $\xrightarrow{cl}{}^{*}$ to indicate a sequence of transitions with context-dependent behaviour given by the list of labels cl. If an element of cl is true we omit that element, and if all elements of cl are true we leave the label blank.

The execution of (9) is given below. We use the following abbreviation to save space.

$$WH \quad \widehat{=} \quad (\textbf{while } a > 0 \textbf{ do } b := a + b \ ; \ a := a - 1)$$

$(\textbf{state } \{a \mapsto 2\} \bullet b := 0 \ ; \ WH)$

$\xrightarrow{b\,:=\,0}{}^{*}$ Rule 13(a), Rule 12(b)
$(\textbf{state } \{a \mapsto 2\} \bullet WH)$

$\longrightarrow^{*}$ Expression evaluation, Rule 14(a), Rule 11
$(\textbf{state } \{a \mapsto 2\} \bullet b := a + b \ ; \ a := a - 1 \ ; \ WH)$

$\xrightarrow{b=0}{}^{*}$ Expression evaluation
$(\textbf{state } \{a \mapsto 2\} \bullet b := 2 \ ; \ a := a - 1 \ ; \ WH)$

$\xrightarrow{b\,:=\,2}{}^{*}$ Rule 13(a), Rule 12(b)
$(\textbf{state } \{a \mapsto 2\} \bullet a := a - 1 \ ; \ WH)$

$\longrightarrow^{*}$ Expression evaluation, Rule 13(b)
$(\textbf{state } \{a \mapsto 1\} \bullet WH)$

$\longrightarrow^{*}$ Expression evaluation, Rule 14(a), Rule 11
$(\textbf{state } \{a \mapsto 1\} \bullet b := a + b \ ; \ a := a - 1 \ ; \ WH)$

$\xrightarrow{b=2}{}^{*}$ Expression evaluation
$(\textbf{state } \{a \mapsto 2\} \bullet b := 3 \ ; \ a := a - 1 \ ; \ WH)$

$\xrightarrow{b\,:=\,3}{}^{*}$ Expression evaluation, Rule 13(a), Rule 12(b)
$(\textbf{state } \{a \mapsto 1\} \bullet a := a - 1 \ ; \ WH)$

$\longrightarrow^{*}$ Expression evaluation, Rule 13(b)
$(\textbf{state } \{a \mapsto 0\} \bullet WH)$

$\longrightarrow^{*}$ Expression evaluation, Rule 14(b), Rule 11
$(\textbf{state } \{a \mapsto 0\} \bullet \textbf{nil})$

$\longrightarrow$ Rule 9
$\textbf{nil}$

If we extract the labels from the execution sequence we have the context-dependent, or "observable", behaviour of the program.

$$b := 0 \ ; \ b = 0 \ ; \ b := 2 \ ; \ b = 2 \ ; \ b := 3$$

All other steps are local steps (a label of $true$), i.e., tests or updates of a, or unfolding steps.

Note that the above sequence can't be simplified, because it is context-dependent. For example, it is possible (although not desirable) to put this program in parallel with another program that also updates b, and hence may interleave steps that lead to other sequences (because b may be evaluated differently).

3.5.3. Concurrency

Now we give the execution of a concurrent program. The commands share a non-local variable y and each have their own local variable x.

$$(\textbf{state } \{x \mapsto 0\} \bullet y := x) \quad \| \quad (\textbf{state } \{x \mapsto 1\} \bullet y := x) \tag{10}$$

Assuming that there are no concurrent processes also modifying y, the final value of y will be either 0 or 1, depending on the order of execution.

Taking the first branch, from Rule 12 we have

$$\begin{array}{ll} & (\mathbf{state}\ \{x \mapsto 0\} \bullet y := x) \\ \longrightarrow & (\mathbf{state}\ \{x \mapsto 0\} \bullet y := 0) \\ \xrightarrow{y=0} & (\mathbf{state}\ \{x \mapsto 0\} \bullet \mathbf{nil}) \\ \longrightarrow & \mathbf{nil} \end{array}$$

Note that x does not appear in the labels, as it is local. A similar sequence holds for the second branch. Outside of the parallel composition the name spaces are distinct, and therefore the reused variable name does not effect the computation.

Rule 6(a) introduces nondeterminism in which of the branches will execute, since their steps may be interleaved. Hence, collecting the possible sequences of context-dependent behaviour of the command (10), we find two possibilities:

$$y := 0;\ y := 1 \qquad \text{and} \qquad y := 1;\ y := 0$$

We have included the parallel operator in our language, partly because concurrency is increasingly prevalent in modern programming languages, and partly to demonstrate that our encapsulated state commands handle multiple instances of the same variable name without requiring the complexity involved in allocating new locations.

4. Procedures

A practical language must provide procedures for abstraction purposes, and should allow recursion. We provide the semantics for two types of parameters: by-reference ($\mathbf{ref}$) and by-value ($\mathbf{val}$). Other procedure and parameter mechanisms are discussed later.

4.1. Syntax

We extend the language with procedure declarations and calls.

$$c ::= \ldots \mid p(\vec{y}, \vec{e}) \mid (\mathbf{procdecl}\ \rho \bullet c)$$

A procedure call is of the form $p(\vec{y}, \vec{e})$, where p is a procedure identifier, $\vec{y}$ is a list of variables which are the actual by-reference parameters, and $\vec{e}$ is a list of expressions which are the actual by-value parameters. Procedure identifiers are taken from the set *Ident*, but are assumed not to be used as identifiers for any other purpose.

A command $(\mathbf{procdecl}\ \rho \bullet c)$, where ρ is a *State*, states that c uses the procedures declared in ρ. Because ρ has the type *State*, there is semantically no difference to $(\mathbf{state}\ \rho \bullet c)$, and therefore we define

$$(\mathbf{procdecl}\ \rho \bullet c) = (\mathbf{state}\ \rho \bullet c)$$

However, we distinguish the two syntactically because we wish to limit their use: the domain of a procedure declaration ρ is limited to a subset of *Ident* reserved for procedure identifiers only, and the range of ρ are special elements of *Val* called *procedure denotations*, described below. Furthermore, we expect that a procedure declaration may never be changed, that is, it is constant.

As with states, procedure declarations can be nested, and subprocesses can define their own local version of procedures. This abstract syntax corresponds to concrete syntax of a new block containing c which declares of the procedures in dom(ρ) as local to c.

4.2. Procedure denotations

A procedure takes two lists of parameters, the first being the actual by-reference parameters and the second being the actual by-value parameters. As such, a procedure denotation is a function from two lists to a command, the body of the procedure. The body of the procedure is contained within two scoping constructs: the formal by-reference formal parameters are *dynamically renamed* to the corresponding actual parameters, and, as is usual, the actual by-value parameters are used to initialise the corresponding formal by-value parameters.

For example, consider the following concrete syntax that declares a procedure $square$ for updating a variable with the square of the value of its by-value parameter.

$$square(\mathbf{ref}\, z, \mathbf{val}\, x) \ \widehat{=}\ z := x^2$$

This is syntactic sugar representing the mapping of the procedure identifier $square$ to the following procedure denotation, $\mathsf{sq}_{\mathsf{den}}$.

$$\lambda\, U \bullet \lambda\, V \bullet (\mathbf{rn}\ \{z \mapsto U\} \bullet (\mathbf{state}\ \{x \mapsto V\} \bullet z := x^2)) \tag{11}$$

It is a function which accepts the metavariable U of type $Ident$, representing the actual by-reference parameter, and the metavariable V of type Val, representing the actual by-value parameter. The formal by-reference parameter z is *dynamically renamed* ($\mathbf{rn}$) to U, and the value V is used to initialise the formal by-value parameter x. The body of the procedure remains the same. The treatment of by-value parameters is standard, i.e., the formal parameter is treated as a variable local to the procedure body and is initialised with the corresponding actual parameter value. The renaming is novel; its semantics is explained in Sect. 4.3. This renaming avoids many of the problems associated with variable names, in particular clashes between local variable names and actual parameters.

A procedure declaration which defines $square$ as accessible by program c is written

$$\mathbf{procdecl}\ \{square \mapsto \mathsf{sq}_{\mathsf{den}}\} \bullet c$$

More generally, we allow procedure denotations to accept multiple by-reference and by-value parameters, which are passed in two lists, respectively.

$$ProcDen \ \widehat{=}\ \mathrm{seq}\, Ident \rightarrow\!\!\!\!\!\!+\ \mathrm{seq}\, Val \rightarrow\!\!\!\!\!\!+\ Cmd$$

Although technically lists are the parameters to a procedure denotation, when dealing with particular definitions we write (actual and formal) parameters as comma-separated lists of expressions. We require that the by-reference parameters precede the by-value, although this requirement can be relaxed in concrete syntax. When there are no by-value parameters or no by-reference parameters, we omit the corresponding empty sequence of parameters.

Therefore, the general form of a procedure declaration in pseudo-code:

$$p(\mathbf{ref}\, \vec{z}, \mathbf{val}\, \vec{x}) \ \widehat{=}\ B$$

is represented in our language by a mapping from procedure identifier p to

$$(\lambda\, \vec{U} \bullet \lambda\, \vec{V} \bullet\ \mathbf{rn}\ \vec{z} \mapsto \vec{U} \bullet (\mathbf{state}\ \vec{x} \mapsto \vec{V} \bullet B))$$

The syntax $\vec{a} \mapsto \vec{b}$ is the mapping formed by matching corresponding parameters in each list. The lists $\vec{U}$ and $\vec{V}$ refer to the actual by-reference and by-value parameters, respectively[3]. A local state mapping the formal by-value parameters $\vec{x}$ to $\vec{V}$ is created as a context for the body of the procedure B. In addition, any context-dependent behaviour of B which contains the formal by-reference parameters is renamed to contain the actual by-reference parameters. To be well-formed, the length of $\vec{z}$ and $\vec{U}$, and $\vec{x}$ and $\vec{V}$, must be the same, i.e., the number of actuals must match the number of formals.

4.3. Renaming

Before explaining the rules for executing a procedure call, we give the semantics of a renaming command. We extend the syntax of commands thus

$$c ::= \ldots \mid (\mathbf{rn}\ \Theta \bullet c)$$

where Θ is of type $Renaming$, that is, a partial function on identifiers $Ident \nrightarrow Ident$. The context-dependent behaviour of this command is the context-dependent behaviour of c, but with each variable $y \in \mathrm{dom}(\Theta)$ replaced with $\Theta(y)$. This corresponds with our intuition about procedure calls, which is that the effect (or constraints) on the formal parameters is mapped to an equivalent effect on the actuals.

By doing the renaming dynamically, we avoid issues to do with locations of actual/formal variables. In particular, this mechanism easily allows recursion, including mutual recursion, and concurrent processes calling the same procedure in parallel.

The label $l\Theta$ is just straightforward replacement of variables in the domain of Θ with their values in the range. It is defined formally below, assuming $x \in \mathrm{dom}(\Theta)$ and $y \notin \mathrm{dom}(\Theta)$.

$$\begin{array}{rcl} (true)\Theta & = & true \\ (x = v)\Theta & = & \Theta(x) = v \\ (y = v)\Theta & = & y = v \\ (x := v)\Theta & = & \Theta(x) := v \\ (y := v)\Theta & = & y := v \end{array} \tag{12}$$

For example:

$$\begin{array}{rcl} (z = v)\{z \mapsto a\} & = & (a = v) \\ (z := v)\{z \mapsto a\} & = & (a := v) \end{array}$$

The renaming has no effect on variables outside its domain.

The rules for renaming are presented in Fig. 7. Rule 15(a) states that the renaming is applied to any command (guard or assignment, including the left-hand side of an assignment). Rule 15(b) states that a renaming terminates when its command terminates.

We assume that renamings do not appear explicitly in the concrete syntax of a programming language, but are used only in procedure denotations. However, see Sect. 7.4 for other uses.

[3]The placeholder names $\vec{U}$ and $\vec{V}$ are metavariables which do not appear otherwise in the syntax of commands, and therefore we assume they can be selected so as to avoid clashes.

$$\textit{Assume} \quad \Theta \in \textit{Renaming} \quad \vec{y} \in \mathrm{seq}\,\textit{Ident} \quad \vec{e} \in \mathrm{seq}\,\textit{Expr} \quad \vec{v} \in \mathrm{seq}\,\textit{Val}$$

Rule 15 (Rename)

$$(a)\ \frac{c \xrightarrow{l} c'}{(\mathbf{rn}\ \Theta \bullet c) \xrightarrow{l\Theta} (\mathbf{rn}\ \Theta \bullet c')} \qquad (b)\ \ (\mathbf{rn}\ \Theta \bullet \mathbf{nil}) \longrightarrow \mathbf{nil}$$

Rule 16 (Procedure call)

$$(a)\ \frac{\vec{e} \xrightarrow{l} \vec{e}'}{p(\vec{y}, \vec{e}) \xrightarrow{l} p(\vec{y}, \vec{e}')} \qquad (b)\ \frac{p \xrightarrow{l} \mathsf{p_{den}}}{p(\vec{y}, \vec{v}) \xrightarrow{l} \mathsf{p_{den}}(\vec{y})(\vec{v})}$$

$$(c)\ \frac{p \xrightarrow{l} (\lambda\, \vec{U} \bullet \lambda\, \vec{V} \bullet (\mathbf{rn}\ \vec{z} \mapsto \vec{U} \bullet (\mathbf{state}\ \vec{x} \mapsto \vec{V} \bullet c)))}{p(\vec{y}, \vec{v}) \xrightarrow{l} (\mathbf{rn}\ \vec{z} \mapsto \vec{y} \bullet (\mathbf{state}\ \vec{x} \mapsto \vec{v} \bullet c))}$$

Figure 7: Rules for procedure call and renaming

4.4. Procedure call

Our general approach is that procedure bodies replace procedure calls at the point of the call (the *Copy Rule* from *ALGOL-60* [11]). We present the semantics in such a way that global variables appearing in procedure bodies are *dynamically bound* [12], that is, refer to the closest defining occurrence at run time. We discuss how to achieve *static binding* in procedure bodies in Sect. 7.2.

The rules for a procedure call are given in Fig. 7. Rule 16(a) states that a procedure call $p(\vec{y}, \vec{e})$ first evaluates its by-value actual parameters, $\vec{e}$. The evaluation of a sequence of expressions is left-to-right – its formal definition is straightforward and deferred until later (Rule 19). When the evaluation process is finished, then, assuming p is defined as $\mathsf{p_{den}}$ in context, the procedure call is replaced by the effect of applying the actual parameters to the procedure denotation. This is equivalent to evaluating p to $\mathsf{p_{den}}$, as with any other variable. To be well-formed, of course, p must evaluate to a procedure denotation, which may be checked statically. Taking the general form of a denotation and substituting it into Rule 16(b) gives Rule 16(c).

4.5. Examples

In this section we provide examples of procedure calls in our framework, including recursion. To simplify the presentation we use the following equivalences.

$$(\mathbf{state}\ \sigma \bullet c) \equiv c \tag{13}$$
$$(\mathbf{rn}\ \{x \mapsto x\} \bullet c) \equiv c \tag{14}$$

Equivalence (13) holds under the side conditions that c is not $\mathbf{nil}$ and does not contain procedure calls and that, for all $x \in \mathrm{dom}(\sigma)$, x does not appear free in c. The equivalence holds because any context-dependent behaviour l exhibited by c can never reference variables in dom(σ), and hence the state has no effect on l. Equivalence (14) holds under the side condition that c is not $\mathbf{nil}$. The equivalence holds because the renaming is an identity function, and hence has no effect on any context-dependent behaviour of c from (12).

The notion of equivalence we adopt is *weak bisimilarity*, which in our setting means that the context-dependent behaviour of the commands on each side of the equivalence, excluding transitions with the label $true$, are identical. We defer a proof of these equivalences, and discussion of bisimilarity, until Sect. 6.

4.5.1. Standard procedure call

Consider a program for calculating the square of 3 and storing it in a global variable a, using the procedure $square$ from (11),

$$\mathbf{procdecl}\ \rho \bullet square(a, 3) \tag{15}$$

where the state ρ is defined below.

$$\rho \mathrel{\widehat{=}} \{square \mapsto \mathsf{sq_{den}}\} \tag{16}$$

From Rule 16(b) and Rule 11(a) the procedure call is replaced by the application of the actual parameters to the procedure denotation for $square$, i.e., $\mathsf{sq_{den}}(a)(3)$, which gives the following command.

$$\begin{array}{l}\mathbf{procdecl}\ \rho \bullet \\ \quad \mathbf{rn}\ \{z \mapsto a\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2)\end{array}$$

The assignment expression is evaluated to 9 using the evaluation rules. This eliminates references to x and we omit the state from the presentation through (13). The step of the program immediately after the evaluation is shown below.

$$\frac{z := 9 \xrightarrow{z = 9} \mathbf{nil}}{\mathbf{rn}\ \{z \mapsto a\} \bullet z := 9 \xrightarrow{a = 9} \mathbf{rn}\ \{z \mapsto a\} \bullet \mathbf{nil}} \tag{17}$$

The transition is justified by Rule 4(b) and Rule 15(a). After renaming, the context-dependent behaviour is updating a to 9, as expected.

To summarise:

$$\begin{array}{ll}
 & square(a, 3) \\
\xrightarrow{sq} & \text{Rule 16(b)} \\
 & \mathsf{sq_{den}}(a)(3) \\
= & \text{(11)} \\
 & \mathbf{rn}\ \{z \mapsto a\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2) \\
\longrightarrow^* & \text{Expression evaluation, (13)} \\
 & \mathbf{rn}\ \{z \mapsto a\} \bullet z := 9 \\
\xrightarrow{a = 9} & \text{From the above calculation} \\
 & \mathbf{rn}\ \{z \mapsto a\} \bullet \mathbf{nil} \\
\longrightarrow & \text{Rule 15} \\
 & \mathbf{nil}
\end{array}$$

The sq label on the first step abbreviates $square = \mathsf{sq}_{\mathsf{den}}$. This would be hidden outside the scope of ρ in (15). Therefore, the effect of this execution is updating a to 9. Note that there was no need to allocate locations for x or z.

4.5.2. Resolving name clashes

In this section we demonstrate how name clashes that result from procedure calls are resolved in our framework. Firstly, consider the case where a call to $square$ occurs in a context which already contains the variable x, i.e., the a variable with the same name as the formal by-reference parameter. We implicitly assume the declaration of ρ as in (15).

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet \mathit{square}(a, x - 2))$$

The procedure call first evaluates its by-value parameter, $x - 2$, to 3, in the current context in which x is 5.

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet \mathit{square}(a, 3))$$

The call itself is then expanded as follows.

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet \mathbf{rn}\ \{z \mapsto a\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2))$$

There is a new inner state which maps x to the value 3. However, any computation within the call that refers to x will be limited to updating only the innermost x, since Rule 12(a) hides the effect on x in the label. An alternative approach, as mentioned in the ALGOL-60 report [11], is to resolve name clashes between actual parameters and local procedure variables by renaming the latter. To formally specify this approach requires tedious bookkeeping of used variable names; in our setting clashes are resolved by keeping the name spaces separate through renamings.

Another potential name clash occurs when x is passed as the by-reference parameter.

$$(\mathbf{state}\ \{x \mapsto 5\} \bullet \mathit{square}(x, 3))$$

After expanding the call, we have

$$\mathbf{state}\ \{x \mapsto 5\} \bullet \mathbf{rn}\ \{z \mapsto x\} \bullet (\mathbf{state}\ \{x \mapsto 3\} \bullet z := x^2)$$

The expression x^2 is evaluated with respect to the innermost x, that is, to 9. The context-dependent behaviour of the assignment becomes $z := 9$, which is unaffected by the inner state, and is renamed to $x := 9$ by the renaming. This updates the outermost x to 9, as expected.

If x appears as a non-local variable in the body of a procedure, and is also passed as an actual by-reference parameter to that procedure, then aliasing occurs. However, the renaming will have no effect on the non-local occurrences of x, and the procedure body will execute as expected. For instance, the following procedure increases the value of its by-reference parameter by the value of a non-local variable a.

$$p(\mathbf{ref}\ z) \mathrel{\widehat{=}} z := a + z$$

A call $p(a)$ expands as follows.

$$\mathbf{rn}\ \{z \mapsto a\} \bullet z := a + z$$

The evaluation of the assignment first looks for the value of a in context, and then the value of z, which is renamed back to a. The assignment itself therefore has the effect of doubling the value of a (assuming no other concurrent process modifies a).

As a final case, aliasing can occur if x is passed as the actual parameter to two by-reference parameters. This will result in a renaming $\Theta = \{z_1 \mapsto x, z_2 \mapsto x\}$, which has a similar effect to the case above.

4.5.3. Recursion

Consider the following program for finding the summation from 1 to some number a, and storing the (successive) results in b. It is a recursive version of (9).

$$\begin{aligned} &\mathit{sum}(\mathbf{ref}\ b, \mathbf{val}\ a) \mathrel{\widehat=} \\ &\quad \mathbf{if}\ a = 0\ \mathbf{then}\ b := 0 \\ &\quad \mathbf{else}\ (\mathit{sum}(b, a-1)\ ;\ b := a + b) \end{aligned} \tag{18}$$

Call the corresponding procedure denotation $\mathsf{sum}_{\mathsf{den}}$. We use this to calculate the sum to 2 and store the result in x.

$$\mathbf{procdecl}\ \{\mathit{sum} \mapsto \mathsf{sum}_{\mathsf{den}}\} \bullet \mathit{sum}(x, 2) \tag{19}$$

Applying Rule 16(b) the procedure call expands to the following.

$$\begin{aligned} &\mathbf{rn}\ \{b \mapsto x\} \bullet \mathbf{state}\ \{a \mapsto 2\} \bullet \\ &\quad \mathbf{if}\ a = 0\ \mathbf{then}\ b := 0 \\ &\quad \mathbf{else}\ (\mathit{sum}(b, a-1)\ ;\ b := a + b) \end{aligned}$$

By expression evaluation and Rule 7(a) and Rule 7(c) we select the second branch of the conditional.

$$\begin{aligned} &\mathbf{rn}\ \{b \mapsto x\} \bullet \mathbf{state}\ \{a \mapsto 2\} \bullet \\ &\quad (\mathit{sum}(b, a-1)\ ;\ b := a + b) \end{aligned} \tag{20}$$

We expand the recursive call in the same manner as the original call, as (20) is still in the context as in (19). Note we get nested renamings, as well as nested states.

$$\begin{aligned} &\mathbf{rn}\ \{b \mapsto x\} \bullet \mathbf{state}\ \{a \mapsto 2\} \bullet \\ &\quad (\mathbf{rn}\ \{b \mapsto b\} \bullet \mathbf{state}\ \{a \mapsto 1\} \bullet \\ &\qquad \mathbf{if}\ a = 0\ \mathbf{then}\ b := 0 \\ &\qquad \mathbf{else}\ (\mathit{sum}(b, a-1)\ ;\ b := a + b))\ ; \\ &\quad b := a + b \end{aligned}$$

The inner renaming of b to b has no effect (this is because the recursive calls all refer to the same by-reference variable). For presentation purposes we omit the renaming by (14). By Rule 7 we select the second branch of the conditional again.

$$\begin{aligned} &\mathbf{rn}\ \{b \mapsto x\} \bullet \mathbf{state}\ \{a \mapsto 2\} \bullet \\ &\quad (\mathbf{state}\ \{a \mapsto 1\} \bullet \\ &\qquad (\mathit{sum}(b, a-1)\ ;\ b := a + b))\ ; \\ &\quad b := a + b \end{aligned}$$

Note there are nested names a. The inner a has the value 1 (the innermost recursive call), while the outer a has the value 2.

We expand the call to sum again, with 0 as the actual value for a, which means by Rule 7(b) we select the first branch of the conditional.

```
$\mathbf{rn}\ \{b \mapsto x\} \bullet \mathbf{state}\ \{a \mapsto 2\} \bullet$
    $(\mathbf{state}\ \{a \mapsto 1\} \bullet$
        $(\mathbf{state}\ \{a \mapsto 0\} \bullet$
            $b := 0)\ ;$
        $b := a + b)\ ;$
    $b := a + b$
```

This program now takes a series of straightforward steps which involve expression evaluation, removing $\mathbf{nil}$s, and assignments to b. Within the scope of the renaming $b \mapsto x$, the context-dependent behaviour is the following sequence:

$$b := 0\ ;\ b = 0\ ;\ b := 1\ ;\ b = 1\ ;\ b := 3$$

Outside the renaming, that is, in the context of the original call, the context-dependent behaviour is renamed to

$$x := 0\ ;\ x = 0\ ;\ x := 1\ ;\ x = 1\ ;\ x := 3$$

What is notable about this recursive unfolding is that no further mechanisms are needed beyond that of handling normal procedure calls. Because we expand procedure calls in-place, recursive calls have access to the same context as the original call. Furthermore our renaming and local state declarations neatly encapsulate the effect of procedure calls, and prevent naming conflicts.

In the case of a non-terminating recursion, our rules will result in an infinite sequence of transitions.

We make a final point that mutual recursion, which can occur between two procedures appearing within the same state, is also handled using the same basic laws.

4.6. Other types of parameters

4.6.1. Call by name

Call-by-name can be viewed as a generalisation of call by reference, where instead of variables, unevaluated expressions are passed as actual parameters. Under this interpretation, renamings and labels must also be generalised to handle expressions in place of variables. Therefore, operations on the formal by-name parameters are renamed so that they become operations on the actual by-name parameters (expressions). When the behaviour is expression evaluation, the renaming results in straightforward behaviour. If the actual parameters are assigned to, the language must be rich enough to handle expressions on the left-hand side of assignments. This is discussed in Sect. 7.3.

4.6.2. Call by value-result

A call to procedure p with an actual by-value-result parameter a is designed so that the value of a is used to initialise the corresponding formal parameter, say z, and at the end of executing p the final value of z is copied back to a. The variable a will be otherwise unchanged during the execution of p (unless a occurs free in the body of p, or a is also passed as a by-reference parameter).

The value-result mechanism can be handled through two formal parameters, z_v and z_r, which are by-value and by-reference, respectively. The only use of z_r is an assignment $z_r := z_v$ at the end of the procedure call. If there is more than one value-result parameter, the final assignments occur in the order they appear in the parameter list. Appropriate syntactic translation in a concrete syntax can hide the above syntactic details from the programmer.

4.6.3. Function calls

We assume that a *function* is a procedure which returns a value, using the special command $\mathbf{return}$. Functions can simplify some definitions, removing the need for reference parameters. For instance, we would like to write the function denotation of $square$, (11), as

$$\lambda\ V \bullet (\mathbf{state}\ \{x \mapsto\ V\} \bullet \mathbf{return}\, x^2)$$

and rewrite (15) as

$$\mathbf{procdecl}\ \rho \bullet\ a := square(3)$$

One approach to allowing functions is to assume that a function is a shorthand for a procedure with an extra by-reference parameter, say x, with $\mathbf{return}\ e$ commands replaced by $x := e$. A function call $z := p(\vec{y}, \vec{v})$ is treated semantically as $p(\vec{y} \frown \langle z \rangle, \vec{v})$, where $\langle z \rangle$ is a singleton sequence containing z and $\frown$ is concatenation of sequences. This is again a simple translational issue for the concrete syntax.

However, this does not extend easily to the case where a function call appears as part of a complex expression in an assignment, or as a parameter to another procedure call, for instance, $y{:}= square(3) + 1$. In Sect. 5 we consider other mechanisms for handling functions, and functions appearing within expressions.

5. Complex expressions

In the main section of the paper we have assumed a basic expression language. However programming languages often allow function calls to appear in expressions, either in assignments or as parameters to other function or procedure calls. We explore this option in this section, and show how we can relatively easily allow small-step function evaluation within expressions.

5.1. Syntax

We extend the syntax of expressions below, where op is some operator, $\vec{e}$ is a list of expressions, $\vec{y}$ is a list of variables, and f is a function identifier.

$$e ::= v \mid x \mid op(\vec{e}) \mid f(\vec{y}, \vec{e}) \mid \mathsf{fneval}(c)$$

An expression may be a value, a variable, some operator op applied to a list of expressions, a function call, or a function body, written $\mathsf{fneval}(c)$, where c is a command.

The $op(\vec{e})$ syntax generalises the basic expressions we gave earlier, which allowed only binary addition. In abstract syntax, binary infix addition $e_1 + e_2$ may appear as $plus(\langle e_1, e_2 \rangle)$. For each operator, op, we assume an associated evaluator operator, opf, which gives the semantics of op on lists of values. For addition, $plusf(\langle v_1, v_2 \rangle) = v_1 + v_2$.

A function call is handled like a procedure call (including by-reference parameters), with the assumption that the body of the procedure terminates with a $\mathbf{return}$ command. The semantics of $\mathbf{return}$ commands is described below. If the body of the function call is instantiated to c, the function call expression becomes $\mathsf{fneval}(c)$. The evaluation strategy for this expression type is to execute c as normal, until a $\mathbf{return}$ command is executed, at which point the expression will be replaced by the returned value.

5.2. Return commands

A command $\mathbf{return}\ e$ first evaluates the expression e (Rule 17(a)), then exposes the calculated value in the transition label before terminating (Rule 17(b)).

Rule 17 (Return)

$$(a)\ \frac{e \xrightarrow{l} e'}{\mathbf{return}\ e \xrightarrow{l} \mathbf{return}\ e'} \qquad (b)\ \mathbf{return}\ v \xrightarrow{\mathbf{return}\ v} \mathbf{nil}$$

We extend the type of *Label*s to include $\mathbf{return}$ expressions.

$$l ::= \ldots \mid \mathbf{return}\ v$$

The return label is uninteresting to any construct except a fneval expression. This means that all of the rules in Fig. 3 and Fig. 7 still hold for $l \in \mathit{Label}$ using the extended definition. The only rule we must add is for a local state.

Rule 18 (State – return)

$$\frac{c \xrightarrow{\mathbf{return}\ v} c'}{(\mathbf{state}\ \sigma \bullet c) \xrightarrow{\mathbf{return}\ v} (\mathbf{state}\ \sigma \bullet c')}$$

We also define a renaming to have no effect on a $\mathbf{return}$ label, that is, we extend (12) such that $(\mathbf{return}\ v)\Theta = (\mathbf{return}\ v)$.

5.3. Semantics

The expression evaluation rules are given in Fig. 8. Note that with the addition of function bodies in expression syntax, the labels in expression evaluation transitions are richer than in Fig. 1, that is, they include assignments and $\mathbf{return}$ labels.

Recall that a value does not evaluate to anything; it is used as the base case for evaluating expressions. Therefore there is no corresponding rule. A variable x evaluates to value v, provided $x = v$, as in Rule 1.

A list of expressions is evaluated from left-to-right (Rule 19(a) and Rule 19(b)). An operation op on parameters $\vec{e}$ evaluates $\vec{e}$ until it is a ground list of values (Rule 20(a)). The meaning of op, opf, is used to calculate the resulting value (Rule 20(b)). Rule 2 may be derived from Rules 19 and 20. Operators are distinct from functions – operators are expression-level objects we assume are primitive to our language, while functions (below) are command-level objects and must be defined explicitly within a program.

Rule 19 (Evaluate expression sequence)

$$(a)\,\frac{e \xrightarrow{l} e'}{(\langle e\rangle \frown \vec{e}) \xrightarrow{l} (\langle e'\rangle \frown \vec{e})} \qquad (b)\,\frac{\vec{e} \xrightarrow{l} \vec{e}'}{(\langle v\rangle \frown \vec{e}) \xrightarrow{l} (\langle v\rangle \frown \vec{e}')}$$

Rule 20 (Evaluate general operator)

$$(a)\,\frac{\vec{e} \xrightarrow{l} \vec{e}'}{op(\vec{e}) \xrightarrow{l} op(\vec{e}')} \qquad (b)\,\frac{opf(\vec{v}) = v}{op(\vec{v}) \longrightarrow v}$$

Rule 21 (Evaluate function call)

$$(a)\,\frac{\vec{e} \xrightarrow{l} \vec{e}'}{f(\vec{y}, \vec{e}) \xrightarrow{l} f(\vec{y}, \vec{e}')} \qquad (b)\,\frac{f \xrightarrow{l} \mathsf{f_{den}}}{f(\vec{y}, \vec{v}) \xrightarrow{l} \mathsf{fneval}(\mathsf{f_{den}}(\vec{y})(\vec{v}))}$$

Rule 22 (Evaluate fneval)

$$(a)\,\frac{c \xrightarrow{l} c'}{\mathsf{fneval}(c) \xrightarrow{l} \mathsf{fneval}(c')} \quad l \neq \mathbf{return}\, v \qquad (b)\,\frac{c \xrightarrow{\mathbf{return}\, v} c'}{\mathsf{fneval}(c) \longrightarrow v}$$

Figure 8: Small-step complex expression evaluation

A function call on f first evaluates its parameters (Rule 21(a)), and when ground, applies the definition of f, $\mathsf{f_{den}}$, to the parameters, and wraps the resulting command as an fneval expression (Rule 21(b)).

An expression $\mathsf{fneval}(c)$ is evaluated through executing c (Rule 22(a)), unless c returns a value, as required by the side condition $l \neq \mathbf{return}\, v$. When c returns a value v, the fneval expression evaluates to v (Rule 22(b)).

5.4. Example

We rewrite the sum procedure from (18) as a recursive function. Let $\mathsf{sumf_{den}}$ be the following function definition of sum, which no longer requires a by-reference parameter.

$$\begin{aligned}(\lambda\, V \bullet\, &\mathbf{state}\ \{a \mapsto V\} \bullet \\ &\quad \mathbf{if}\ a = 0\ \mathbf{then\ return}\, 0 \\ &\quad \mathbf{else\ return}\, sum(a-1) + a)\end{aligned}$$

First consider executing the following command.

$$\mathbf{procdecl}\ \{sum \mapsto \mathsf{sumf_{den}}\} \bullet x := sum(0)$$

The assignment evaluates as follows, from Rule 21.

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 0\} \bullet\ \mathbf{if}\ a = 0\ \mathbf{then\ return}\, 0\ \mathbf{else\ return}\, sum(a-1) + a)$$

From expression evaluation and Rule 7(b), the first branch of the conditional is selected.

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 0\} \bullet \mathbf{return}\, 0)$$

We eliminate the now unnecessary state containing a following (13).

$$\cfrac{\cfrac{\mathbf{return}\, 0 \xrightarrow{\mathbf{return}\, 0} \mathbf{nil}}{\mathsf{fneval}(\mathbf{return}\, 0) \longrightarrow 0}}{x := \mathsf{fneval}(\mathbf{return}\, 0) \longrightarrow x := 0} \qquad \begin{array}{l} \textit{Rule } 17(b) \\ \textit{Rule } 22(b) \\ \textit{Rule } 4(a) \end{array}$$

Now consider executing the following program.

$$\mathbf{procdecl}\ \{sum \mapsto \mathsf{sumf}_{\mathsf{den}}\} \bullet x := sum(1)$$

Following similar reasoning as above, the assignment simplifies to

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 1\} \bullet \mathbf{return}\, sum(0) + a)$$

Now the evaluation of the expression $sum(0)$ also continues as above, resulting in a nested fneval expression (eliminating again an unnecessary declaration of a).

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 1\} \bullet (\mathbf{return}\, \mathsf{fneval}(\mathbf{return}\, 0) + a))$$

The inner fneval evaluates to 0, as described above.

$$x := \mathsf{fneval}(\mathbf{state}\ \{a \mapsto 1\} \bullet (\mathbf{return}\, 0 + a))$$

The $\mathbf{return}$ command evaluates to 1, and the assignment overall evaluates to $x := 1$.

A similar transformation can be used to straightforwardly execute $x := sum(2)$, which eventually has the context-dependent behaviour of $x := 3$. Note that this is in contrast to earlier calculations of the sum which resulted in successive updates to x.

6. Bisimilarity

For the purposes of this paper, we define equivalence of two commands through *weak bisimilarity* [13]. Intuitively, this states that two commands are equivalent if their context behaviour is the same, ignoring transitions with the label $true$. This is because transitions with a label of $true$ are deemed *unobservable*, corresponding to our intuition that such steps are independent of the context.

First we introduce some extra notation:

- Given a label l, $\widehat{l}$ is an empty or singleton sequence of labels, defined below to remove labels of $true$.

$$\widehat{l} = \begin{cases} \langle\,\rangle & \text{if } l = true \\ \langle l \rangle & \text{otherwise} \end{cases}$$

- Given a sequence of labels ls of length n, $c_0 \overset{ls}{\Longrightarrow} c_m$ iff there exists $c_1, c_2, \ldots c_{m-1}$ such that

  $$c_0 \longrightarrow^* c_1 \xrightarrow{ls(0)} c_2 \longrightarrow^* c_3 \xrightarrow{ls(1)} \ldots \xrightarrow{ls(n)} c_{m-1} \longrightarrow^* c_m$$

  That is, a finite number (≥ 0) of transitions with label $true$ may be inserted between the transitions in ls. Note that $c \longrightarrow^* c'$ may represent an empty sequence of $true$ transitions, in which case $c = c'$.

Definition 21 (Weak bisimilarity) *A binary relation $R \subseteq Cmd \times Cmd$ is a weak bisimulation if, for all $(c, d) \in R$, all $c', d' \in Cmd$, and all $l \in Label$, the following two conditions hold.*

$$c \xrightarrow{l} c' \Rightarrow (\exists\, d' \bullet d \overset{\widehat{l}}{\Longrightarrow} d' \wedge (c', d') \in R) \qquad (22)$$

$$d \xrightarrow{l} d' \Rightarrow (\exists\, c' \bullet c \overset{\widehat{l}}{\Longrightarrow} c' \wedge (c', d') \in R) \qquad (23)$$

Two commands c and d are *weakly bisimilar*, written $c \equiv d$, iff there exists a weak bisimulation R such that $(c, d) \in R$.

As an example we prove equivalence (14).

Theorem 24. *For any $cmd \in Cmd$ and $x \in Ident$,*

$$(\mathbf{rn}\ \{x \mapsto x\} \bullet cmd) \equiv cmd$$

Proof. Let Θ_x abbreviate $\{x \mapsto x\}$. We prove that the relation R defined below is a weak bisimulation.

$$R = \{((\mathbf{rn}\ \Theta_x \bullet cmd), cmd) \mid cmd \in Cmd\} \cup \{(\mathbf{nil}, \mathbf{nil})\}$$

We first note

$$cmd \xrightarrow{l} cmd' \Leftrightarrow (\mathbf{rn}\ \Theta_x \bullet cmd) \xrightarrow{l} (\mathbf{rn}\ \Theta_x \bullet cmd') \qquad (25)$$

The $\Rightarrow$ direction is obvious from Rule 15(a); the $\Leftarrow$ direction holds because the relation $\longrightarrow$ is the least relation which satisfies the operational rules, and Rule 15(a) is the only rule which applies to a renaming command on both sides of a transition.

To prove (23) we choose $(\mathbf{rn}\ \Theta_x \bullet cmd')$ as the witness for c'.

$$\begin{array}{l} cmd \xrightarrow{l} cmd' \Rightarrow \\ \qquad (\mathbf{rn}\ \Theta_x \bullet cmd) \xrightarrow{l} (\mathbf{rn}\ \Theta_x \bullet cmd') \wedge ((\mathbf{rn}\ \Theta_x \bullet cmd'), cmd') \in R \end{array}$$

This follows from (25) and the definition of R.

To prove (22) we partition into cases where $cmd = \mathbf{nil}$. Firstly, when $cmd \neq \mathbf{nil}$, we choose cmd' as the witness for d'.

$$\begin{array}{l} (\mathbf{rn}\ \Theta_x \bullet cmd) \xrightarrow{l} (\mathbf{rn}\ \Theta_x \bullet cmd') \Rightarrow \\ \qquad cmd \xrightarrow{l} cmd \wedge ((\mathbf{rn}\ \Theta_x \bullet cmd'), cmd') \in R \end{array}$$

This follows from (25) and the definition of R.

When $cmd = \mathbf{nil}$ we have $true$ as the only possible choice for l (Rule 15(b)), and choose $\mathbf{nil}$ as the witness for d'.

$$(\mathbf{rn}\ \Theta_x \bullet \mathbf{nil}) \xrightarrow{true} \mathbf{nil} \Rightarrow (\mathbf{nil} \overset{\widehat{true}}{\Longrightarrow} \mathbf{nil} \wedge (\mathbf{nil}, \mathbf{nil}) \in R)$$

This holds by choosing the empty sequence of transitions for $\overset{\widehat{true}}{\Longrightarrow}$, and from the definition of R. $\square$

We may also define *strong* bisimilarity following Milner [13], although other definitions of equivalence, and techniques for proving such equivalences, are outside the scope of this paper.

7. Other aspects

In this section we address some other aspects of programming and the style of rules we have adopted. To assist we first outline Plotkin's use of locations. We then discuss three programming aspects which, in addition to by-reference parameters, Plotkin identified as leading to the introduction of locations: static binding, array types, and reference types [2, Sect. 3.4]. In all cases we are able to provide the desired semantics at a higher level of abstraction, without going beyond standard variable/value mappings. We conclude the section by sketching how higher-order programming may be defined in our setting.

7.1. Locations

In the semantics we have thus far presented, we map variables directly to their values. In Plotkin's notes, variables are mapped to *locations* (the *environment*), and a separate mapping relates locations to values (the *store*). Environments appear throughout the program, while the store is a meta-level construct appearing in all transition rules. During execution of a program, a new variable declaration x is replaced by an environment mapping x to some new location l, and the store is updated so that l takes the initial value of x. The added level of indirection allows the same variable name to refer to different values, depending on the most recently declared environment.

To understand how our approach works, in particular when considering static binding, the key insight is that the allocation of a new location is essentially allocating a new variable name. Plotkin's approach works because each identifier (variable) is renamed to some new identifier (location) which is guaranteed to be unique within that execution of the program. This renaming adds considerable complexity to the dynamic semantics in the rules.

7.2. Static binding

The effect of a procedure on its free variables, that is, variables that are neither declared locally nor in the parameter list, is determined by the *binding* strategy the language employs. There are two main strategies, *dynamic* and *static*. With *dynamic binding*, procedure and variable names refer to the most recently declared version at run-time. This semantics is easy to describe, and is the semantics originally used in Lisp [12]. In the rules we have presented, variable and procedure references appearing as global within procedure bodies will follow this semantics, that is, we have provided a dynamic binding semantics.

However, most programming languages use static binding, which means that procedure and variable references refer to the closest statically enclosing definition within the code. This style of name binding is more intuitive when statically reading the code.

Static binding in Plotkin's notes is enforced by *elaborating* a procedure definition p: a copy of the environment at the point of the declaration, restricted to the free identifiers appearing in the body of p, is inserted into the procedure definition. The elaborated definition is called a *closure*. Hence the environment at the point of declaration is used when executing the body of the procedure.

We base our approach to enforcing static binding on the following observation:

Proposition 1. *If a program, P, contains no declarations which cause a name clash with any previously declared identifier, then the behaviour of P is identical whether P is executed with a dynamic or static binding strategy.*

To informally justify the observation, consider the program below, which contains a parameterless procedure declaration p.

$$\begin{array}{l} \textbf{state}\ \{a \mapsto 0\} \bullet \\ \quad \textbf{procdecl}\ \{p \mapsto (a := 10)\} \\ \quad\quad \textbf{state}\ \{a \mapsto 5\} \bullet\ p() \end{array}$$

The question is to which instance of a does the assignment in the definition of p refer. In a statically bound language, it is the outermost a, while in a dynamically bound language it is the innermost. However, if we rename the inner a to b, then there can be no choice, the procedure clearly refers to the (outermost, only) a.[4]

Any program in which there are no nested scopes of variables which share the same name contains no ambiguity about which variable is being referred to, as each variable is unique. Hence, both strategies will bind the variable to the same identifier.

Note that a recursive procedure can lead to overlapping variable names, even if variable names are otherwise unique. This occurs when the recursive procedure body declares a local variable, as shown by local variable a in Sect. 5.4. Since such variables are not free, this does not pose a problem for resolving references.

Given Proposition 1, our approach to static binding is to eliminate all name clashes (including procedure names) before execution, through a straightforward pre-processing step. This step is essentially the same as allocating a unique location for each variable name, except that we may perform the step prior to execution, and we may selectively rename only those variables that require it, i.e., those that are using a name that has already been declared. There is no need to rename the x variables in (10), for instance. The naming may be performed in any way that results in the absence of name clashes; one option is to rename in an hierarchical manner, i.e., we may rename the innermost "a" in the above program to "$p.a$", indicating it is declared in the scope of p.

In summary, the advantage of our approach to static binding is that the restriction applies only in the abstract syntax. The concrete syntax of a real programming language with static binding may allow redeclaration of variables and procedures, as long as they are renamed appropriately before using the semantics.

7.3. Arrays and records

In this section we consider the extensions necessary to allow array and record types, and in particular, passing array indexing and field access expressions as reference parameters. In an

[4]We make the usual assumption that procedure bodies cannot refer to variables that have not been declared at the time of the procedure declaration.

implementation setting, individual entries in an array or record can be accessed directly through their mapping to locations. In our framework we provide a semantics for interpreting updates to array indexes and record field accesses as overriding the value of a function at an index.

7.3.1. Basics and syntax

We refer arrays and records as special elements of Val called *compound values*, which are functions from elements of Val to Val (we use the term compound value to distinguish them from command-level functions and function denotations). Array values are compound values whose domain is a contiguous subrange of the natural numbers starting from 0, and record values are compound values whose domain is some set of constants.

The syntax of expressions is extended to allow array indexing and record field access.

$$e ::= \ldots \mid e_1[e_2] \mid e.n$$

Here e_2 is an expression assumed to be integer-valued, and n is a field name[5]. This syntax allows arrays of arrays, arrays of records, etc. For instance, an array A of records with field name n can be accessed through expressions such as $A[0].n$. The evaluation of such expressions are given by Rules 23 and 24 in Fig. 9.

7.3.2. L-values

We identify a subset of expressions, $\mathit{LValExpr}$, which are indexes into a variable through array indexing and field access. Their syntax is given by the following simple production for $\mathit{lve} \in \mathit{LValExpr}$.

$$\mathit{lve} ::= x \mid \mathit{lve}[e] \mid \mathit{lve}.n$$

Hence, $A[x].n$ is an $\mathit{LValExpr}$, but $(A+B).n$ is not. We now extend the syntax of commands to allow $\mathit{LValExpr}$s to appear on the left-hand side of assignments, and in the actual by-reference parameters of procedure calls (previously only variables could be found in those places).

$$c ::= \ldots \mid \mathit{lve} := e \mid \ldots \mid p(\vec{\mathit{lve}}, \vec{e}) \mid \ldots$$

Our approach is to treat, for example, an update of the first element of a one-dimensional array of numbers, $A[0] := 3$, as an update of A following the pattern $A := A[0 \mapsto 3]$. One approach to handling this would be to treat the former assignment as a syntactic sugar for the latter, however, this does not extend easily to allowing array index expressions as by-reference parameters.

Instead we introduce a new syntactic class LVal, which correspond to $\mathit{LValExpr}$s where all but the main variable have been evaluated. For convenience, rather than defining them recursively as with $\mathit{LValExpr}$s, we denote them by the pairing of a variable with a list of values, the indexes into the structure of the variable.

$$\mathit{LVal} \mathrel{\widehat{=}} \mathit{Ident} \times \text{seq } \mathit{Val}$$

[5]We are assuming that field names may not be constructed from general expressions, although this is not a restriction that simplifies the dynamic semantics particularly. However, the restriction does simplify the static semantics of expressions, in particular type checking.

$$Assume \quad i, n, vF \in Val \quad lve \in LValExpr \quad lv \in LVal$$

Rule 23 (Array index evaluation)

$$(a)\ \frac{e_1 \xrightarrow{l} e_1'}{e_1[e_2] \xrightarrow{l} e_1'[e_2]} \qquad (b)\ \frac{e_2 \xrightarrow{l} e_2'}{vF[e_2] \xrightarrow{l} vF[e_2']} \qquad (c)\ vF[i] \longrightarrow vF(i)$$

Rule 24 (Record field evaluation)

$$(a)\ \frac{e \xrightarrow{l} e'}{e.n \xrightarrow{l} e'.n} \qquad (b)\ vF.n \longrightarrow vF(n)$$

Rule 25 (L-Value evaluation)

$$(a)\ \frac{lve \xrightarrow{l}_{\mathsf{lval}} lve'}{lve[e] \xrightarrow{l}_{\mathsf{lval}} lve'[e]} \quad (b)\ \frac{e \xrightarrow{l} e'}{lv[e] \xrightarrow{l}_{\mathsf{lval}} lv[e']} \quad (c)\ \frac{lve \xrightarrow{l}_{\mathsf{lval}} lve'}{lve.n \xrightarrow{l}_{\mathsf{lval}} lve'.n}$$

$$(d)\ (x, ls)[i] \longrightarrow_{\mathsf{lval}} (x, ls \frown \langle i \rangle) \qquad (f)\ x \longrightarrow_{\mathsf{lval}} (x, \langle\rangle)$$
$$(e)\ (x, ls).n \longrightarrow_{\mathsf{lval}} (x, ls \frown \langle n \rangle)$$

Figure 9: Rules for arrays

Only $LValExpr$s may be evaluated to an $LVal$. This evaluation is given by the relation $\longrightarrow_{\mathsf{lval}}$, which is the least relation that satisfies Rule 25. For example, the expression $A[0]$ is evaluated to the $LVal$ $(A, \langle 0 \rangle)$, and the expression $A[0].n$ is evaluated to the $LVal$ $(A, \langle 0, n \rangle)$. A variable is evaluated to an $LVal$ with an empty list of indexes (Rule 25(f)) and the index into an array is evaluated as an ordinary expression (Rule 25(b)); otherwise Rule 25 is straightforward. Lists of $LVal$s may be evaluated analogously to Rule 19 for lists of expressions.

7.3.3. Labels and states

We also generalise labels so that $LVal$s may appear in place of variables.

$$l ::= true \mid lv = v \mid lv := v$$

This necessitates corresponding generalisations of the rules for local state, as given in Rules 27 and 28 in Fig. 10. Rule 27(a) differs from Rule 11(a) in that the evaluation of an $LVal$ requires extra work.

$$\begin{array}{lcl} eval(vF, \langle\rangle) & = & vF \\ eval(vF, \langle i \rangle \frown ls) & = & eval(vF(i), ls) \end{array}$$

The rule collapses to Rule 11(a) when the sequences of indexes is empty, that is, when the $LVal$ corresponds to an unindexed variable. Rule 11(b) corresponds to Rule 27(b).

Similarly, Rule 28 is a generalisations of Rule 12. This new rule makes use of the function nv, defined below such that $nv(xv, ls, v)$ is the new value obtained by updating (the compound value) vF, at the point given by index list ls, to the value v.

$$\begin{array}{rcl} nv(vF, \langle\rangle, v) & = & v \\ nv(vF, \langle i \rangle \frown ls, v) & = & vF[i \mapsto nv(vF(i), ls, v)] \end{array}$$

Rule 28(a) collapses to Rule 12(a) when the sequences of indexes is empty.

As an example, assume vF is a one-dimensional array, and it is being updated in its first element to value 0. The new value is constructed below:

$$\begin{array}{rl} & nv(vF, \langle 0 \rangle, v) \\ = & vF[0 \mapsto nv(vF(0), \langle\rangle, v)] \\ = & vF[0 \mapsto v] \end{array}$$

The derived rule for the update of an element in a one-dimensional array is as follows.

Rule 26 (Vector update)

$$\frac{c \xrightarrow{(A, \langle i \rangle) := v} c' \quad A \in dom(\sigma) \quad A' = \sigma(A)[i \mapsto v]}{(\textbf{state}\ \sigma \bullet c) \longrightarrow (\textbf{state}\ \sigma[A \mapsto A'] \bullet c')}$$

7.3.4. Commands

The rule for allowing assignments to accept $LVal$s on the left-hand side (Rule 29) subsumes Rule 4, adding clause (a) to evaluate the $LValExpr$ first.

The rule for allowing procedures to accept $LVal$s in place of variables in the by-reference actual parameter list (Rule 30), subsumes Rule 16, adding clause (a) to evaluate to $LVal$ parameters first.

The instantiation of the procedure or function occurs in the usual way (Rule 30(b)), except that now the procedure denotations include renamings from $Ident$ to $LVal$. Therefore, context-dependent behaviour involving the formal by-reference parameters (which must always be variables) such as $x = v$ and $x := v$, will be renamed to $(A, \langle i \rangle) = v$ and $(A, \langle i \rangle) := v$.

As an example of using $LVal$s as actual by-reference parameters, consider the following code, using (11).

$$\begin{array}{l} \textbf{state}\ \{A \mapsto \{0 \mapsto 3, 1 \mapsto 0\}\} \bullet \\ \textbf{procdecl}\ \{square \mapsto \mathsf{sq}_{\mathsf{den}}\} \bullet square(A[1], A[0]) \end{array}$$

The result of the call should be updating $A(1)$ to 9.

The expression $A[0]$ is evaluated to 3 from Rule 23(b) and (c). Recalling that the range of renamings is now $LVal$, the expansion of the procedure call in context gives:

$$\textbf{rn}\ \{z \mapsto (A, \langle 1 \rangle)\} \bullet (\textbf{state}\ \{x \mapsto 3\} \bullet z := x^2)$$

As shown in (17), the context-dependent behaviour of this command becomes

$$(A, \langle 1 \rangle) := 9$$

$$Assume \quad \vec{lve} \in \text{seq}\, LValExpr \quad ls \in \text{seq}\, Val \quad \vec{lv} \in \text{seq}\, LVal$$

Rule 27 (State – test)

$$(a)\ \frac{c \xrightarrow{(x,ls)=v} c' \quad x \in dom(\sigma) \quad eval(\sigma(x), ls) = v}{(\textbf{state}\ \sigma \bullet c) \longrightarrow (\textbf{state}\ \sigma \bullet c')}$$

$$(b)\ \frac{c \overset{(x,ls)=v}{\longrightarrow} c' \quad x \notin dom(\sigma)}{(\textbf{state}\ \sigma \bullet c) \xrightarrow{(x,ls)=v} (\textbf{state}\ \sigma \bullet c')}$$

Rule 28 (State – assignment)

$$(a)\ \frac{c \xrightarrow{(x,ls) := v} c' \quad x \in dom(\sigma)}{(\textbf{state}\ \sigma \bullet c) \rightarrow (\textbf{state}\ \sigma[x \mapsto nv(\sigma(x), ls, v)] \bullet c')}$$

$$(b)\ \frac{c \overset{(x,ls) := v}{\longrightarrow} c' \quad x \notin dom(\sigma)}{(\textbf{state}\ \sigma \bullet c) \overset{(x,ls) := v}{\longrightarrow} (\textbf{state}\ \sigma \bullet c')}$$

Rule 29 (Assignment)

$$(a)\ \frac{lve \xrightarrow{l}_{\mathsf{lval}} lve'}{lve := e \xrightarrow{l} lve' := e}$$

$$(b)\ \frac{e \xrightarrow{l} e'}{lv := e \xrightarrow{l} lv := e'}$$

$$(c)\quad lv := v \xrightarrow{lv := v} \textbf{nil}$$

Rule 30 (Procedure call)

$$(a)\ \frac{\vec{lve} \xrightarrow{l}_{\mathsf{lval}} \vec{lve}'}{p(\vec{lve}, \vec{e}) \xrightarrow{l} p(\vec{lve}', \vec{e})}$$

$$(b)\ \frac{\vec{e} \xrightarrow{l} \vec{e}'}{p(\vec{lv}, \vec{e}) \xrightarrow{l} p(\vec{lv}, \vec{e}')}$$

$$(c)\ \frac{p \xrightarrow{l} \mathsf{p_{den}}}{p(\vec{lv}, \vec{v}) \xrightarrow{l} \mathsf{p_{den}}(\vec{lv})(\vec{v})}$$

Figure 10: Rules for *LVal*s in local state

Through Rule 26 this has the desired effect in the enclosing state, i.e.,

$$\begin{array}{rl} & nv(\{0 \mapsto 3, 1 \mapsto 0\}, \langle 1 \rangle, 9) \\ = & (\{0 \mapsto 3, 1 \mapsto 0\})[1 \mapsto nv(0, \langle\rangle, 9)] \\ = & (\{0 \mapsto 3, 1 \mapsto 0\})[1 \mapsto 9] \\ = & (\{0 \mapsto 3, 1 \mapsto 9\}) \end{array}$$

7.4. Reference types

Pointers, accessed through a global heap, may be described in our system by declaring the heap as a global variable, i.e., in the outermost enclosing state. Allocating and deallocating are then operations on the heap variable. However, if the pointers are ubiquitous through the language, as is the case with C, having the heap at the meta level of the rules, as in Plotkin-style semantics, may be more convenient. Given the variable scoping we employ and the lack of interference through a global store, it would be interesting to investigate whether the operational semantics can be used to nicely describe separation logic for pointer-based programs [14, 15, 16].

A major problem for formal analysis of pointer-based programs is *aliasing*, where different variable names refer to the same entity. This occurs in implementation languages when two variables map to the same location in memory, as happens when a procedure call matches formal and actual by-reference parameters. This is handled in our semantics by dynamic renaming, and it may be the case that renamings can also define an explicit aliasing command in our semantics (the `x == y` command of Plotkin). Using a renaming command the scope of the aliasing is restricted by the syntax, rather than exposed globally through the store, and it is possible that formal analysis in the presence of aliasing may be simpler in our framework; however, investigation of that topic is beyond the scope of this paper.

7.5. Higher-order programming

The essence of higher-order programming is to allow parameters and variables to take the value of procedure denotations. Then procedure calls may be made using a variable instead of the procedure name, and procedures can be passed by-value (and by-reference) to procedure bodies. For example, consider the following declaration of higher-order function denotation $\mathsf{map}_{\mathsf{den}}$, which returns a new list which is the result of applying a function to all the elements of the input list.

$$\begin{array}{l} (\lambda\, F, L \bullet \textbf{state}\ \{f \mapsto F, l \mapsto L\} \bullet \\ \quad \textbf{if}\ l = \langle\rangle\ \textbf{then return}\ \langle\rangle \\ \quad \textbf{else return}\ \langle f(\mathit{head}(l)) \rangle \frown \mathit{map}(f, \mathit{tail}(l))) \end{array}$$

We assume the formal parameter l is a sequence, and use $\mathit{head}(l)$ and $\mathit{tail}(l)$ for partitioning a non-empty list l into its first element and the remainder of the list, respectively. List concatenation is represented by the symbol "$\frown$".

The following program copies the sequence $\langle 1, 4, 9 \rangle$ into variable x.

$$\begin{array}{l} \textbf{procdecl}\ \{\mathit{map} \mapsto \mathsf{map}_{\mathsf{den}}, \mathit{square} \mapsto \mathsf{sq}_{\mathsf{den}}\} \bullet \\ \textbf{state}\ \{\mathit{fn} \mapsto 0, x \mapsto 0\} \bullet \\ \quad \mathit{fn} := \mathit{square}\ ; \\ \quad x := \mathit{map}(\mathit{fn}, \langle 1, 2, 3 \rangle) \end{array}$$

The mechanics required to execute this program do not require any new laws. The first assignment $fn := square$ evaluates $square$ to its definition ($\mathsf{sq_{den}}$ from (11)) and this becomes the value for fn in context. Then F is instantiated with $\mathsf{sq_{den}}$ in the by-value parameters for map, and hence f is initialised to $\mathsf{sq_{den}}$. In evaluating the non-trivial $\mathbf{return}$ expression in map, the value $\mathsf{sq_{den}}$ is extracted for f in the function application $f(head(l))$ via Rule 21(b). This process is repeated in the recursive call to map.

The only difference to our earlier presentation is that we allow procedure denotations to appear in $\mathbf{state}$ commands, as well as $\mathbf{procdecl}$ commands.

8. Related work

Plotkin's *syntax-directed* operational semantics was a great improvement over earlier lower-level stack machine descriptions. Surveys on current trends in operational semantics research are given by Mousavi, Reniers and Groote [17] and Aceto, Fokkink and Verhoef [6]. The historical development of operational semantics is discussed by Plotkin [18] and by Jones [19]. In this section we first compare our framework with that of Plotkin [2], and then other related work.

8.1. Plotkin-style SOS

We have separated the state-based manipulations, i.e., testing and updating the value of variables, from the basic commands (tests and assignments) which cause them. This means the rules for the structural operators, such as sequential composition and parallel composition, do not need to explicitly mention the state. Instead, the state is treated as a construct of the language, corresponding to variable declarations and procedure definitions. Interestingly, including state and procedures (data) as part of the program seems to contradict a literal interpretation of the classic equation given by Plotkin, SYSTEM = PROGRAM + DATA.

If we ignore language constructs which require the use of environments, and therefore treat Plotkin's store as a total mapping from variables directly to values, our style may be related to his by declaring the entire state of the program as local to the program at the highest level. That is, for some program c, with initial state given by σ, an execution of

$$(\mathbf{state}\ \sigma \bullet c)$$

using our rules should theoretically result in the same sequence of commands given by execution of

$$\langle c, \sigma \rangle$$

using Plotkin's rules (assuming no name clashes in identifiers, by-reference procedure calls, etc). Since σ is total on variable names, the transition labels in our execution sequence will all be $true$.

The rules themselves may be unified, under the simplifying assumptions above, through deriving transitions in our setting in which the (total) state is explicitly passed through the structure of the program. For instance, consider the following rule for sequential composition, which may be derived from the rules in Figs. 3 and 4.

$$\frac{(\mathbf{state}\ \sigma \bullet c_1) \longrightarrow (\mathbf{state}\ \sigma' \bullet c_1')}{(\mathbf{state}\ \sigma \bullet c_1\ ;\ c_2) \longrightarrow (\mathbf{state}\ \sigma' \bullet c_1'\ ;\ c_2)}$$

Hence in our style the basic commands are nondeterministic and their effect is propagated outwards to the enclosing context, while in Plotkin-style rules the context is gathered starting from the "outside" and then used to determine the effect of the basic operation. For instance, we must describe the effect of an assignment directly within a local (in this case, global) state declaration.

$$(\mathbf{state}\ \sigma \bullet x := v) \longrightarrow (\mathbf{state}\ \sigma[x \mapsto v] \bullet \mathbf{nil})$$

This rule can be derived from Rule 4(b) and 12(a) assuming $x \in \mathrm{dom}(\sigma)$.

To make the similarity with the Plotkin-style approach more apparent, we rewrite the above two rules using the more familiar syntax $\langle c, \sigma \rangle$ instead of the $\mathbf{state}$ keyword[6].

$$\frac{\langle c_1, \sigma \rangle \longrightarrow \langle c_1', \sigma' \rangle}{\langle c_1 \ ; \ c_2, \sigma \rangle \longrightarrow \langle c_1' \ ; \ c_2, \sigma' \rangle} \qquad \langle x := v, \sigma \rangle \longrightarrow \langle \mathbf{nil}, \sigma[x \mapsto v] \rangle \tag{26}$$

These rules still differ from Plotkin's because he considers *final configurations*, which makes redundant the $\mathbf{nil}$ in rules such as (26).

Our rules contain a significant amount of nondeterminism, which is not normally expected when evaluating the value of an expression, but the nondeterminism is resolved neatly in the operational semantics style through the intercession of outer contexts. The attractiveness of operational semantics in handling nondeterminism is noted by Jones [19] with respect to concurrency (and in comparison with denotational semantics).

In pragmatic terms, an advantage of the style we present is that we do not need to specify a rule for every $State/Cmd$ combination (although we must give a rule for every $State/Label$ combination). The reasoning about states and labels appears in Rules 11 and 12, while the meaning of the commands can be given succinctly in their own rules, e.g., Rules 4, 7 and 8. This results in a separation of concerns, or *modularity*, as defined by Mosses [20]. The advantage of modularity becomes apparent as more constructs are added. By the end of Plotkin's notes, there is a significant amount of baggage carried around in each rule, which we are able to separate out. For example, consider a simple command "$[x = v]$", the informal meaning of which is to block (in a concurrent setting) until variable x has the value v in context. It can be formally specified in our framework by the following rule.

$$[x = v] \xrightarrow{x=v} \mathbf{nil}$$

This is the only rule we need to add, and it is relatively concise and intuitive. In Plotkin's full framework, where environments are required, the corresponding rule would include state, environment, and evaluation information.

$$\frac{\rho(x) = l \quad \sigma(l) = v}{\rho \vdash \langle [x = v], \sigma \rangle \longrightarrow \sigma}$$

Finally, the major advantage of our approach is that we do not need to introduce locations to handle by-reference parameters to procedures, or reuse of variable names. This is because our novel command $(\mathbf{rn}\ \Theta \bullet c)$ can operate directly on the context dependent behaviour of a procedure call, and therefore avoid many issues to do with variables names, as discussed in Sect. 4.5.2. Our treatment is more abstract (one can think of locations as being one way to implement the renaming we describe), and overall resulting in a more intuitive framework.

[6]As an example of how the semantics of a local state may treated in a standard SOS rule base, see Reynolds' treatment [4, Chapter 6], attributed to Eugene Fink.

8.2. Other related work

The majority of operational descriptions of imperative programming languages do not use labelled transitions. One exception is the work of Degano and Priami [21], who use enhanced labels in their transitions, but not with respect to state-based operations. Closer to our work is the Modular Structural Operational Semantics (MSOS) of Mosses [20]. In Mosses' framework, the labels on transitions contain the state and environment information, and can be extended to contain other information about program execution. Due to syntactic conventions, only the context information required to execute a particular construct needs to be extracted from the labels. This results in rules which are more concise than Plotkin's, and comparable to our own. The labels on our transitions, in comparison, specify properties that the pre (and post) states must satisfy, rather than the states (stores and environments) themselves. The MSOS papers [20, 22, 23] do not include semantics for procedures and parameters, and hence it is not clear whether the MSOS style can remove the need for the extra level of complexity introduced by locations.

The utility of MSOS is particularly demonstrated for programs with abrupt termination, where the occurrence of an error at run time (such as division by 0) is exposed in the label. Then at an outer context the error can result in termination of the entire program, or be "caught" by some enclosing exception handler. This approach dovetails neatly with our use of labels for propagating context-dependent behaviour. Potentially, our use of labels can be extended to use the category theory that Mosses presents, thereby generalising our rules to a richer label type.

Our semantics for procedure call is based on the *copy rule* of Algol-60, where the procedure body replaces the call. An alternative approach is to execute the procedure body at the point of declaration, as is the approach taken by Jones in [5]. However, this approach, while simplifying the definition of static binding, creates a significant number of problems to do with reuse of variable names if a procedure p is called concurrently by two processes, or is recursive.

The use of labelled transitions in operational semantics is widespread in process calculi (e.g., [24, 9, 25, 6]). The labels in such languages contain the "observable behaviour" of processes, which are typically process interactions. In this paper we have applied this approach to a state-based imperative language, where state tests and assignments are the obvious candidates to appear in labels. We have also taken the language further towards the process algebra world by allowing partial states to "hide" local computation, in an analogous manner to the way synchronisation events may be hidden in process calculi. In previous work [10] we used our approach for introducing state tests and assignments into CSP [24]. That paper includes a more general state modification construct, which can nondeterministically update several variables at a time. It also evaluates expressions in a big-step style. The labels were extended to be a pair which included both state manipulation information and synchronisation events. This allows operations on the state to be synchronised using the existing CSP operators.

9. Conclusions

We have presented an alternative style for expressing operational semantics of imperative languages. In this style commands at the basic level are nondeterministic, and it is left to the outer context(s) to choose the correct transition. In contrast, the traditional SOS style collects the context starting from the outermost scope until the effect of the command at the basic level is fully determined. That style, however, results in the need for a ubiquitous store of the values of variables, which appears in every transition rule, and the need for a level of indirection through locations to handle repeated variables declarations.

Our style appears to result in simpler and more compact rules, and removes the need for locations. The resulting semantics is more abstract than that of Plotkin's. The simplicity is particularly evident in the description of procedures and parameters, including both by-value and by-reference. The simplicity of the use of labels comes through the dynamic renaming operator, where only the effect on nonlocal variables need be renamed, while operations on by-value parameters and locally declared variables are hidden. This avoids many of the tedious issues to do with formally defining the semantics, where name clashes become apparent. Casual investigation by the authors indicates that a similar construct is not possible in Plotkin-style semantics.

Although we argue that our rules are simpler and more abstract, for the basic commands they are perhaps less accessible to non-experts than those of Plotkin, since one does not normally associate nondeterminism with, for instance, assignment commands.

Acknowledgements. This research was supported by Australian Research Council (ARC) Discovery Grant DP0987452, *Combining Time Bands and Teleo-Reactive Programs for Advanced Dependable Real-Time Systems*. We would like to thank Kirsten Winter and Brijesh Dongol for feedback on earlier drafts of this paper.

References

[1] G. D. Plotkin, A structural approach to operational semantics., Tech. rep., Computer Science Department, Aarhus University (1981).

[2] G. D. Plotkin, A structural approach to operational semantics., J. Log. Algebr. Program. 60-61 (2004) 17–139.

[3] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, MIT Press, 1993.

[4] J. C. Reynolds, Theories of Programming Languages, Cambridge University Press, 1998.

[5] C. B. Jones, Understanding programming language concepts via operational semantics, in: C. George, Z. Liu, J. Woodcock (Eds.), Domain Modeling and the Duration Calculus, International Training School, Advanced Lectures, Vol. 4710 of Lecture Notes in Computer Science, Springer, 2007, pp. 177–235.

[6] L. Aceto, W. Fokkink, C. Verhoef, Structural operational semantics, in: J. Bergstra, A. Ponse, S. Smolka (Eds.), Handbook of Process Algebra, Elsevier Science, 2001, pp. 197–292.

[7] A. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1998.

[8] S. Schneider, Concurrent and Real-time Systems: The CSP Approach, Wiley, 2000.

[9] R. Milner, A Calculus of Communicating Systems, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[10] R. Colvin, I. J. Hayes, CSP with hierarchical state, in: M. Leuschel, H. Wehrheim (Eds.), Integrated Formal Methods (IFM 2009), Vol. 5423 of Lecture Notes in Comp. Sci., Springer, 2009, pp. 118–135.

[11] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, P. Naur, Revised report on the algorithm language ALGOL 60, Commun. ACM 6 (1) (1963) 1–17.

[12] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, Commun. ACM 3 (4) (1960) 184–195. doi:http://doi.acm.org/10.1145/367177.367199.

[13] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[14] S. S. Ishtiaq, P. W. O'Hearn, BI as an assertion language for mutable data structures, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM Press, 2001, pp. 14–26.

[15] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: IEEE Symposium on Logic in Computer Science (LICS), IEEE Computer Society, 2002, pp. 55–74.

[16] J. C. Reynolds, An overview of separation logic, in: B. Meyer, J. Woodcock (Eds.), Verified Software: Theories, Tools, Experiments (VSTTE), Vol. 4171 of Lecture Notes in Computer Science, Springer, 2005, pp. 460–469.

[17] M. R. Mousavi, M. A. Reniers, J. F. Groote, SOS formats and meta-theory: 20 years after, Theoretical Computer Science 373 (3) (2007) 238 – 272. doi:DOI: 10.1016/j.tcs.2006.12.019.

[18] G. D. Plotkin, The origins of structural operational semantics, J. Log. Algebr. Program. 60-61 (2004) 3–15.

[19] C. B. Jones, Operational semantics: Concepts and their expression, Inf. Process. Lett. 88 (1-2) (2003) 27–32.

[20] P. D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60-61 (2004) 195–228.

[21] P. Degano, C. Priami, Enhanced operational semantics: a tool for describing and analyzing concurrent systems, ACM Comput. Surv. 33 (2) (2001) 135–176. doi:http://doi.acm.org/10.1145/384192.384194.

[22] P. D. Mosses, Pragmatics of modular SOS, in: H. Kirchner, C. Ringeissen (Eds.), Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Proceedings, Vol. 2422 of Lecture Notes in Computer Science, Springer, 2002, pp. 21–40.

[23] P. D. Mosses, Exploiting labels in structural operational semantics, Fundam. Inform. 60 (1-4) (2004) 17–31.

[24] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, Inc., 1985.

[25] J. A. Bergstra, J. W. Klop, J. V. Tucker, Process algebra with asynchronous communication mechanisms, in: S. D. Brookes, A. W. Roscoe, G. Winskel (Eds.), Seminar on Concurrency, Vol. 197 of Lecture Notes in Computer Science, Springer, 1985, pp. 76–95.