University of Amsterdam
Programming Research Group

Predictable and Reliable Program Code: Virtual Machine-based Projection Semantics (submitted for inclusion in the Handbook of Network and Systems Administration)

J.A. Bergstra
I. Bethke

Report PRG0603 September 2006

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

I. Bethke

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7583
e-mail: inge@science.uva.nl

Programming Research Group Electronic Report Series

Predictable and Reliable Program Code: Virtual Machine-based Projection Semantics (submitted for inclusion in the Handbook of Network and Systems Administration)

J.A. Bergstra [a,b,1] I. Bethke [a,2,*]

[a] *University of Amsterdam, Faculty of Science, Programming Research Group*

[b] *Utrecht University, Department of Philosophy, Applied Logic Group*

1 Introduction

Network and systems administration seems to be a subject not based on theory, or at least not primarily based on theory. Mark Burgess [22–25] provides aspects of theory and so do Alva Couch and Yizhan Sun in [29], but the gap between these papers and conventional foundational theory of computing is quite large. Theoretical computer science has its empty areas as well, but we hold that many critical issues in network and systems administration cannot be reliably addressed without the application of semantic theory, be it tailor made theories.

The line between system administration and other aspects of computing is never clear. System administrators are often amongst the first users to test new technologies, and are involved in building new applications in a production environment. When things go wrong, the system administrator is often one of the first people to be involved in problem finding.

Many problems in the security and reliability of mission-critical software can result from misleading program semantics. The possibility for mistakes in coding to weaken a system or result in unexpected behaviour was one of the motivations for the introduction of the Ada language. Here we present a simple

* Corresponding author. Address: Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

[1] E-mail: janb@{phil.uu.nl, science.uva.nl}

[2] E-mail: inge@science.uva.nl

theory developing a reliable syntax whose execution semantics are unambiguous. Of all the languages used by system administrators and operators in deploying services (e.g. PHP, Perl, TCL, Scheme etc.) few can be said to have such a clear connection between syntax and behaviour. The system world would do well to foster this kind of predictability in future technologies.

Since the introduction of time sharing operating systems, command languages or scripts have been used to manage the running of programs. Job Control Language was introduced to manage program execution for batch control. In the 1980s, the DOS environment on the IBM PC mimicked the idea of scripts with its simple "batch files", used to start applications from the command line, but these were never as sophisticated as those coming from timesharing systems.

The Unix Bourne shell, and later C shell, were a turning point for system scripting. They allowed—and continue to allow—a user to combine independent programs using 'pipes' or communication channels which behave like a Virtual Private Network interface between applications. Later, the inefficiencies and irregularities of syntax led to the development of operating system independent languages such as Awk, then Perl and Tcl which have only partially replaced the shell.

Today there is a need for a new generation of languages for system management, in a variety of contexts. One example is configuration management. Configration management is currently an active area—see the chapter by Alva Couch in this book. A prototype language that is widely used for here is Cfengine [20], which takes a declarative view that has been likened to Prolog [21]. Cfengine uses an irregular syntax but aims to give clear semantics based on fixed point behaviour [24].

As languages become high level, the details of their semantics become important in order to understand their behaviour. Today, scripting languages like shell, Perl, PHP and Tcl are very low level and leave the higher level semantics entirely to imperative, declarative, functional and object oriented programmers. Python and Ruby are bridging this gap.

In this chapter we wish to consider this subject—the semantics of scripting languages—as part of the foundations of system and network administration and provide a theoretical account to elementary higher level object oriented scripting languages using the field of semantic frameworks. Our aim is to highlight the role of layers of language abstraction and their effect on semantics of language statements. We show how these languages can be built up from low level primitives through virtualization layers to high level constructs, so that we might build new languages that are "correct by construction". This is a large field of research, so we have modest aims for our chapter, but we

hope that this will inspire more work of an applied nature to feed the current debates and controversies over semantics.

We start with providing a survey of semantic theories of programming and propose the viewpoint that all of these theories have the intrinsic problem of asking their readers to think in terms of mathematical domains that may fail to be self evident for the practical minds working in network and systems administration. Then we provide an alternative semantic theory that we have developed under the name 'projection semantics'. Projection semantics is a way to state that compilers determine the meaning of programs. A projection is a theoretical account of a possibly slow but conceptually convincing compiler, or rather of the transformation that the compiler is supposed to produce. The classical assumption that such a projection should be validated against a denotational or mathematical semantic theory is dropped! Semantic intuitions are supposed to reside at the level of the projections, i.e. conceptual abstractions of what compilers do or might do.

A second aspect of the appoach that we advocate is that it is execution architecture based. With this we mean that in giving conceptual models for programs and their meaning it is essential that an operating context, or execution architecture is taken as the point of departure. Here we will use the execution architecture of [14] which proposes that a program executes in a context where it interacts with both local (auxiliary) and external services. The external services represent the world to which the running program ultimately directs its activity—i.e. whether its operation is supposed to achieve intended effects—while the services local to the execution architecture represent transformations on auxiliary and volatile data which are thrown away as soon as the execution of the program halts and which cannot be inspected by any other system component than the running program. Because this viewpoint is reasonably close to a possible description of a virtual machine our projection semantics is called *virtual machine based*. Perhaps a more informative name is *execution architecture based* projection semantics. Programming with program notations that permit being modelled by these techniques might be called *execution architecture based programming*. Below we will reconstruct a small fragment of object oriented scripting as execution architecture based programming, by providing a projection from a high level—though very simple—syntax to a low level one for which an execution architecture is known in detail.

Many matters can be clarified on the basis of projection semantics for an execution architecture based reconstruction of programs and program notations. A further development in the direction of concurrent programming languages, for instance the large variety of thread scheduling mechanisms also called *interleaving strategies*, will capture major user intuitions while being almost irresponsibly naive in comparison to existing dominant concurrency theories.

The instruction sequence language ISLA that is used below finds its semantics in so-called *thread algebras*. Thread algebras are conceptually simplified—and for that same reason technically more complex and less stable—process algebras fully dedicated to the semantic issue to be solved. Therefore our projection semantics may be classified as a special case of process algebra based semantics. Process algebra based semantics of program notations has a history of more than 20 years: see e.g. Milner [49], Vaandrager [63], Cleaveland et al. [26], Andrews [3], Bergstra, Middelburg and Usenko [12] and Mauw and Reniers [46].

2 Semantics of programming languages

The study of programs has a profound history in the setting of formal languages and grammars. Originating from theoretical studies of automata and parsing, the path from program texts to program execution has been subject to many investigations and has to some extent been found useful in various practical applications such as compiler-writing systems, type-checkers, code-generators and documentation generators. However, in contrast to the popularity of formal syntax, formal semantics—the field concerned with the rigorous mathematical study of the meaning of programming languages and their models of computation—has been less exploited in practical applications concerning the design and implementation of programming languages. Currently, only few systems generating any kind of implementation from semantic descriptions are available [41]. In particular, semantic results are rarely incorporated in practical systems that help language designers to implement and test a language under development, or to assist programmers in answering their questions about the meaning of some language feature not properly documented in a language's reference manual. Among the reasons for this state of affairs, one stands out. The use of even a properly-implemented and well-maintained system is limited to the input available for it. For significant practical uses of semantic descriptions, the input should be a complete, fully *formal* semantic description of the observable behaviour of a program in contrast to the kind of descriptions usually found in textbooks and manuals, where crucial details are often omitted and informal conventions are introduced in the interest of conciseness. Yet despite the theoretical efforts in establishing the foundations of various frameworks, good pragmatic features needed for the efficient use of semantic descriptions are often lacking. Below we give a recap of the semantic frameworks of current interest. For an extensive survey along with a list of uses that have been made of them, and speculations on the hindrances for greater use, the reader is referred to [52].

Before we give a brief summary of the semantic frameworks of current interest, let us distinguish between *static* and *dynamic* semantics. Static se-

mantics—which appears to be used more often in practical applications than dynamic semantics—concerns the checking for well-formedness of a program without actually executing it. It thus corresponds to its compile-time behaviour and is independent of any input that is provided to the program at run-time. Well-formedness is usually decidable, so static semantics may be treated as a kind of (context-sensitive) syntax specified by attribute grammars. For compiled programs, their well-formedness has already been checked, and their dynamic semantics corresponds to their run-time behaviour. For languages implemented by interpreters, programs usually run without a foregoing overall well-formedness check. Any required checks happen at run-time and can thus be considered part of the dynamic semantics. In the sequel we shall focus entirely on dynamic semantics and do not worry about what semantics is given to programs containing ill-typed expressions, assuming that such programs are filtered out by a preceding static semantics.

The dynamic semantics of a language is given by a mathematical model which represents the possible computations described by the language. The three main classes of approach can be classified as *operational*, *denotational*, or *axiomatic*.

Starting from the early 1960's [47], operational semantics models the computations of programs as a—perhaps infinite—sequence of computational steps between states. Although states in computations generally depend on the syntactical appearance of the executed program, syntactically-distinct programs can have the same sets of states. In order to obtain a reasonable notion of semantic equivalence in operational semantics, some equivalence relation that ignores the syntactic dependence of states has to be introduced; popular choices are bisimulation equivalences [64]. Various ways of modelling computations have been developed yielding different frameworks for operational semantics of programming languages. The most prominent ones are:

(1) *Structural operational semantics (SOS)*—the pioneering work on operational semantics—was proposed by Plotkin in [1]. Here computations are modelled as sequences of—possibly labelled—transitions between states—a mixture of syntax and computed values—involving syntax, computed values, and auxiliary entities. SOS has been widely used to describe process calculi used for specification, but also for the design and description of the programming languages Facile and Ada.
(2) *Natural semantics* was developed by Kahn in the mid-1980's [44]. Terminating computations are modelled as evaluation relations between syntax and computed values, possibly involving auxiliary entities; nonterminating computations are generally ignored. This semantics was used during the design and official definition of Standard ML.
(3) *Reduction semantics* is a framework proposed by Felleisen and Friedman [32] towards the end of the 1980's. This type of operational semantics

models computations as sequences of term rewriting steps where the intermediate terms consist again of computed values and auxiliary entities. Reduction semantics was used during the design of Concurrent ML.

(4) *Abstract state machines (ASM)*, previously called "evolving algebras" [3] [40], was developed by Gurevich in the late 1980's. In contrast to the preceding approaches, states are mathematical structures where data come as abstract objects which are equipped with basic operations—partial functions— and relations. The notion of computation is given by a transition system which determines which functions in a given state have to be updated. ASM was adopted by ISO for the use in the Prolog standard. More recently in the new millenium, ASM has been used to provide specifications of Java and the JVM [61], and of the JCVM [57].

Denotational semantics models each part of a program as its denotation—typically a higher-order function between complete partial orders [39]— representing its observable behaviour. The semantics of loops and recursion usually involves the explicit use of fixed-point operators or a specification as equations whose least solution is to be found. Apart from the original Scott-Strachey style of denotational semantics, we consider here also *monadic semantics* and *predicate transformers*.

(5) *Scott-Strachey semantics* is the original style of denotational semantics developed at the end of the 1960's [56]. Here domains of denotations and auxiliary entities are defined by domain equations, i.e. equations involving complete partial orders with continuous functions. Many standard techniques for representing programming concepts as pure mathematical functions have been established in this framework. For instance, sequencing may be represented by composition of functions, or by the use of continuations. The denotational description of nondeterminism, concurrency, and interleaving has led to the development of so-called power domains [60]. Because of Strachey's fundamental contributions, researchers were able to give semantic definitions to programming languages like Ada, Algol60, Algol68, PL/1, Lisp, Pascal, Scheme and Snobol [16,48,6,51,53,62].

(6) *Monadic semantics* is based on category-theoretic concepts and was developed by Moggi [50] at the end of the 1980's. Denotations in monadic semantics are elements of so-called monads, i.e. type constructors that capture various notions of sequential computation. Monad transformers construct monads incrementally and add new aspects of a computation to a given monad. Monadic semantics appears to be currently lacking an example of its use in connection with a practical programming language.

(7) *Predicate transformer semantics* was proposed by Dijkstra [30] in the

[3] The idea of evolving algebras has probably independently been developed and applied during the design of the wide-spectrum language COLD. For more details see [43,34,35].

mid-1970's. Here the semantics of a programming language is defined by assigning to each command in the language a corresponding predicate transformer returning the weakest pre-condition which ensures termination of the command under a certain post-condition. Predicate transformers are required to have properties corresponding to the continuity of functions on Scott-domains. Unlike other semantic formalisms, predicate transformer semantics was not designed as an investigation into the foundations of computations. Rather, it was intended—and still is—to provide programmers with a methodology to develop their programs as "correct by construction" in a "calculational style".

Axiomatic semantics is an approach based on mathematical. Its purpose is to provide a set of logical axioms and rules in order to reason about the correctness of computer programs with the rigour of mathematical logic. As usual with axiomatic specifications, there may be insufficiently many properties been given—in which case there may be more than one model—or the set of properties may be inconsistent—in which case there are no models at all. As with predicate transformers, predicates are used as formulae. We consider *Hoare logic*.

(8) *Hoare logic* (also known as *Floyd-Hoare logic*) was developed by Hoare [42] in the late 1960's. Hoare's logic has axioms and inference rules for all the constructs of a simple imperative programming language. In addition to the rules for the simple language in Hoare's original paper, rules for other language constructs—such as concurrency, procedures, jumps, and pointers—have been developed since then by Hoare and many other researchers. Hoare logic was used during the design of Pascal, and the resulting description as the basis for program verification.

A large number of semantic frameworks have been provided during the past three decades. We have classified them mainly as operational, denotational, and axiomatic in a deliberately brief and superficial style, focussing on the most important differences between the frameworks, and ignoring many details. Compared to the amount of effort that has been devoted to the development of these semantic frameworks and their uses listed above, the list of significant practical uses may be considered as relatively small. Hindrances to greater use are user-unfriendliness and the fact that most frameworks do not scale up smoothly from tidy illustrative to full-scale practical languages.

3 Projection semantics

A major difficulty with the approaches to programming language semantics that stem from theory is that the audience is asked to think in terms of cer-

tain more or less sophisticated semantic domains, typically function spaces or transition systems of some particular kind. This is always felt as an obstacle and textbooks on programming never seem to pay any attention to such mathematical background.

Projection semantics (PS) was developed in the setting of *program algebra* in the years 1998 and 1999 [9,10] and is different in that it explains the semantics of a program in a 'higher' language in terms of a translation, here called *projection*, in a known program notation. A program is supposed to have the same meaning as its projection, by definition. The semantics of the projection, i.e., the resulting program after projection, is therefore given in terms of programming language semantics as well, the only progress made being that it is in a notation which one claims to be understood.

As an example we consider the basic program notation based on a parameter set of basic instructions $\{a, b, c, \ldots\}$ representing a command issued by a program during execution to its context. That context may involve classical ingredients such as a printing device but it may also involve the calculation of a value contained in a variable which is often taken simply as something inside a program. The execution of a basic instruction takes place during program execution. It is done by the machine architecture that executes the program and it will have two effects: a possible state change in the execution context and a Boolean value which is produced as a result of processing the instruction—viewed as a request to its environment—and which is subsequently returned to the program under execution where it may be used to decide which is the next instruction to be carried out. In addition to the parameter set, the basic program notation contains test instructions—each basic instruction can be turned into positive or negative test instruction by prefixing it with - or + —and absolute jump instructions $\#\#k$ with $k \in \mathbb{N}$.

A program is then a finite stream of instructions. Execution starts at the first instruction and ends when the last instruction is executed or if a jump is made to a nonexisting instruction. In particular, $\#\#0$ represents program termination because the instruction count is taken to start with 1. The working of test instructions is as follows: if the i-th instruction is $+a$ its execution begins with the execution of a. After completion of that action a reply value is at hand. If the reply was `true` the run of the program proceeds with instruction number $i + 1$—if that exists otherwise execution terminates. Alternatively, if the reply was `false` the execution proceeds with instruction number $i + 2$—again assuming its existence, leading to termination in the other case as well. Thus a negative reply enacts that an instruction is skipped. Execution of a negative test instruction $-a$ proceeds in a similar way yielding a skip if a returns `true`. Finally, the execution of a jump instruction $\#\#k$ makes the execution continue with the k-th instruction if that instruction exists. Otherwise the program execution terminates. If $\#\#k$ itself is the k-th instruction the execution

is caught in a never ending loop.

An example of a basic program is $a; +b; \#\#0; c; c; \#\#2$ which may be read as: first do a and then b, and then repeat $c; c; b$ as long as the last reply for b was negative, and terminate after the first positive reply to an execution of b is observed.

The behavioural semantics of basic programs is defined in [8,10,11]. This semantics is based on the basic *thread algebra* TA. TA has a constant S for termination, a constant D for inaction or divergent behaviour, and a composition mechanism named *post conditional composition*: the expression $P \trianglelefteq a \trianglerighteq Q$ represents the execution of action a, and if `true` was returned, behaviour continues as P, and otherwise as Q. A shorthand for $P \trianglelefteq a \trianglerighteq P$ is $a \circ P$, and this resembles the usual action prefix in process algebra. Furthermore TA is equipped with a family of approximation operators which can be used to handle infinite behaviour. Now any basic program X gives rise to a (possibly infinite) TA process. For example, the process P corresponding to the above mentioned basic program $a; +b; \#\#0; c; c; \#\#2$ can be defined by the two equations $P = a \circ Q$, $Q = S \trianglelefteq b \trianglerighteq (c \circ c \circ Q)$.

Based on this extremely simple program notation more advanced program features can be developed within projection semantics such as conditional statements, while loops, recursion, object classes, method calls etc. In taking care of a cumulative and bottom up introduction of such complex features while providing appropriate projections into the lower levels of language development, an *intermediate* languages can be developed that keeps all definitions rigorous, ensures a clear meaning of higher program constructs and serves as an intermediate step for programs written in a higher-order language.

4 Virtual machines and intermediate languages

Virtual machines (VMs) are a popular target for language implementors. They reside as application programs on top of an operating system and export abstract machines that can run programs written in a particular *intermediate language* (IL). As an intermediate step, programs written in a higher-order language (HL) are first translated into a more suitable form before object or machine code is generated for a particular target machine. Masking differences in the hardware and software layers below the virtual machine, any program written in the high-level language and compiled for these virtual machines will run on them. In an ideal situation the frontend of a virtual machine will be entirely independent of the target hardware, while the backend will be sensibly independent of the particular language in which source programs are written. In this way the task of writing compilers for n languages on m machines is

factored into $n + m$ part-compilers rather than $n \times m$ compilers.

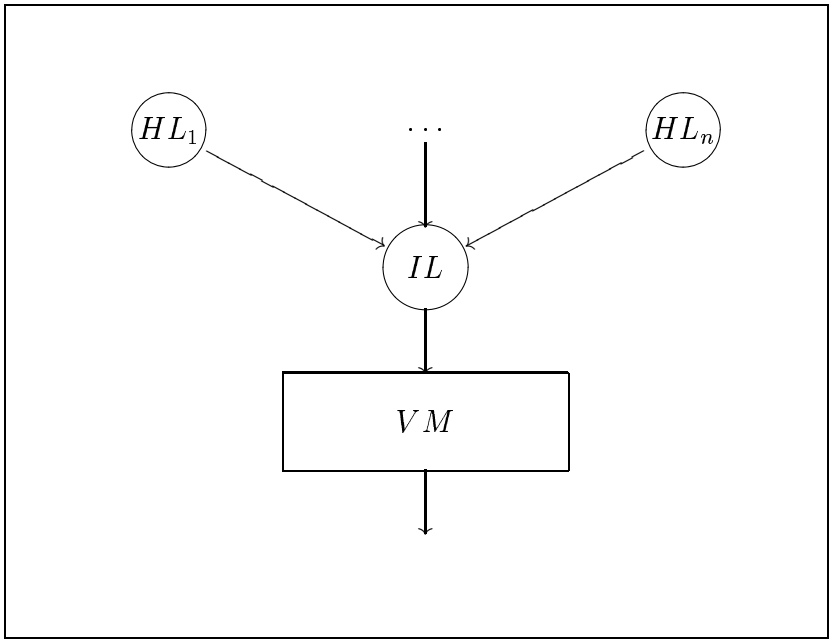

Fig. 1. Outline of a virtual machine design

One of the first virtual machines was the *p-Code machine* invented in the late 1970's as an intermediate form for the ETH Pascal compilers [2] but becoming pervasive as the machine code for the UCSD Pascal system. What had been noted was that a program encoded for an abstract machine may be used in two ways: a compiler backend may compile the code down to the machine language of the actual target machine, or an interpreter may be written which emulates the abstract machine on the target. This interpretative approach surrenders a significant factor of speed, but has the advantage that programs are much more dense in the abstract machine encoding. As a consequence of this technology high-level languages became available for the first time on microcomputers. As an additional benefit, the task of porting a language system to a new machine reduced to the relatively simple task of creating a new interpreter on the new machine.

In the late 1990's Sun Microsystems released their Java [37] language system. This system is also based on an abstract machine - the *Java Virtual Machine (JVM)* [45]. Nowadays, JVMs are available for almost all computing platforms. In mid-2000 Microsoft revealed a new technology based on a wider use of the world wide web for service delivery. This technology became known as the *.NET* system and was designed with the objective of supporting multiple languages. At the heart of this initiative stands the *Common Language Infrastructure (CLI)* virtual machine executing the *Common Intermediate Language (CIL)*.

A long-running question in the design of virtual machines has been whether register or stack architectures can be implemented more efficiently with an interpreter. Many designers favour stack architectures since the location of the operands is implicit in the stack location, prominent examples being the stack-based JVM and CLI. In contrast, the operands of register machine instructions must be specified explicitly. A more recent example of this sort of virtual machine is Parrot intended to run dynamic languages. In the sequel we shall only consider stack-based VMs. For a translation from stack-based to register-based code and efficiency comparisons see e.g. [38,58].

5 The base language

The base language is the PS answer to the conceptual question: 'what is a program?'. The tricky aspect is that at the same time as the base language is claimed to provide this answer PS predicts that this answer may be in need of refactoring at some future stage. PS runs on the unproven assumption that whatever formal definition of a program is given, there will always be a setting in which this definition needs to be compromised or amended in order to get a comprehensible theory. But at the same time a PS answer to any problem needs full precision definitions. There is no room for an open ended concept of program like 'a text meant to tell a machine what to do', or—in the absence of precise definitions of 'unit' and of 'deployment'—'a definition of a software component as a unit of deployment'. Likewise, a machine language may not be defined as a program notation meant for machine execution—in the absence of a clear model of the concept of machine execution—and a scripting language is not definable as a language for writing scripts—unless, again the concept of a script is rigorously defined beforehand.

As a start for syntax development *instruction stream language with absolute jumps (ISLA)* (cf. Section 3) may be taken which features among the sequence of the program notations that have been designed on the basis of program algebra under the name PGLD [10]. The basis of the definition of ISLA is formed by the notion of a *basic action*. Basic actions may be used as instructions in ISLA. A basic instruction a represents a command issued by the program during execution to its context. That context may involve classical ingredients such as a printing device but it may also involve the operation of a value contained in a variable which is often taken simply as something inside a program. The execution of a basic action a takes place during program execution. It is done by the machine architecture that executes the program and it will have two effects: a possible state change in the execution context and a Boolean value which is produced as a result of processing the action—viewed as a request to its environment—and which is subsequently returned to the program under execution where it may be used to decide which is the next instruction

to be carried out.

For each basic action a the following are ISLA instructions: a, $+a$ and $-a$. These three versions of the basic actions are called: *void basic action* (a), *positive test instruction* ($+a$), and *negative test instruction* ($-a$). A void basic action represents the command to execute the basic action while ignoring the Boolean return value that it produces. Upon completion of the execution of the basic action by the execution environment of the program processing will proceed with the next instruction. The test actions represent the two ways that the Boolean reply can be used to influence the subsequent execution of a program. The only other instructions in ISLA are *absolute* jumps $\#\#k$ with k a natural number.

An ISLA program is a finite stream of ISLA instructions. Execution by default starts at the first (left most) instruction. It ends if the last instruction is executed or if a jump is made to a nonexisting instruction. In particular, $\#\#0$—which will be abbreviated by !—represents program termination because the instruction count is taken to start with 1. The working of test instructions is as follows: if the i-th instruction is $+a$ its execution begins with the execution of a. After completion of that action a reply value is at hand. If the reply was `true` the run of the program proceeds with instruction number $i+1$ if that exists, otherwise execution terminates. Alternatively, if the reply was `false` the execution proceeds with instruction number $i+2$ again assuming its existence, leading to termination in the other case as well. Thus a negative reply enacts that an instruction is skipped. In the case of a negative test instruction $-a$ execution proceeds with instruction number $i+2$ at a positive reply and with instruction number $i+1$ at reply `false`. The execution of a jump instruction $\#\#k$ makes the execution continue with the k-th instruction if that exists. Otherwise the program execution terminates. If $\#\#k$ is itself the k-th instruction the execution is caught in a never ending loop.

Example 1 Typical ISLA programs are

- $a; +b; !; c; \#\#2$
  which may be read as first do a and then b and then repeat $c; b$ as long as the last reply for b was negative, and terminate after the first positive reply to an execution for b is observed.
- $-a; \#\#6; b; c; !; e; f$
  which may be read as first do a, and if a returns `true` then do b and c, else do e and f, and then terminate.

The definition of ISLA in PS has the following intention. At the initial stage of PS development it is reasonable to view ISLA as a definition of the concept of a program. Moreover, any text P is a program provided there is at hand

a projection ϕ which maps the text to an LP NA instruction sequence $\phi(P)$ which, by definition, represents the meaning of P as a program. The position regarding ISLA is therefore not the untenable assertion that every program is identical to an ISLA instruction sequence but the much more flexible assertion that for some entity P to qualify as a program it must be known how it represents an ISLA instruction sequence. By means of the application of the projection that ISLA program is found and the meaning of the entity P as a program is determined.

An alternative to ISLA is the program notation ISLR which admits basic actions, positive and negative test actions and the termination instruction !. The jumps are different, however. Only forward jumps ($\#k$) and backward jumps ($\backslash\#k$) (with k a natural number) are admitted. Thus ISLR is ISL with relative jumps. Termination also occurs if a jump is performed outside the range of instructions. [4]

Example 2 A typical ISLR program is $+a; \#3; b; !; c; \backslash\#4$ which may be read as first do a and, if the reply to a was negative, continue with b and terminate; otherwise enter an infinite repetition of c.

ISLA and ISLR are intertranslatable by the mappings $\psi : \text{ISLA} \to \text{ISLR}$ and $\phi : \text{ISLR} \to \text{ISLA}$ which transform jumps depending on the place of their occurrence and leave the other instructions unmodified. The mappings are defined by

- $\psi(u_1; \ldots; u_n) = \psi_1(u_1); \ldots; \psi_n(u_n)$ where

$$\psi_i(\#\#k) = \begin{cases} \#k - i & \text{if } k \geq i, \\ \backslash\#i - k & \text{otherwise.} \end{cases}$$

- $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$ where $\phi_i(\#k) = \#\#k + i$ and

$$\phi_i(\backslash\#k) = \begin{cases} \#\#i - k & \text{if } k < i, \\ ! & \text{otherwise.} \end{cases}$$

[4] The program notation ISLR was originally defined in the setting of program algebra in [10] under the name PGLC.

Example 3 The translations of the programs in Example 1 and 2 are

$$\begin{aligned}
\psi(a;+b;!;c;\#\#2) &= \psi_1(a);\psi_2(+b);\psi_3(!);\psi_4(c);\psi_5(\#\#2)\\
&= a;+b;!;c;\backslash\#3,\\
\psi(-a;\#\#6;b;c;!;e;f) &= \psi_1(-a);\psi_2(\#\#6);\psi_3(b);\psi_4(c);\psi_5(!);\psi_6(e);\psi_7(f)\\
&= -a;\#4;b;c;!;e;f,\\
\phi(+a;\#3;b;!;c;\backslash\#4) &= \phi_1(+a);\phi_2(\#3);\phi_3(b);\phi_4(!);\phi_5(c);\phi_6(\backslash\#4)\\
&= +a;\#\#5;b;!;c;\#\#2.
\end{aligned}$$

A virtue of ISLR is that it supports *relocatable* programming. Programs may be concatenated without disturbing the meaning of jumps. Placing a program behind another program may be viewed as a relocation which places each instruction at an incremented position. This property can be best exploited if programs are used that always use ! for termination. In some cases it is useful to use only a forward jump to the first missing instruction for termination. For programs of that form concatenation corresponds to sequential composition.

Another alternative to ISLA as a base language would be ISLAR, the instruction stream notation with absolute *and* relative jumps. Throughout this paper, however, we shall take ISLA as base language, and projections will be (chained) mappings to ISLA programs. We shall write A $\subseteq$ B for a program notation B extending A and use the operator ϕ for projections. In most cases a projection transforms some program notation back to a simpler one. Sometimes the choice of the right order of such steps is important. A formalism for specifying a strategy for chaining projections will not be included below, however. It is assumed that a reader will be easily able to determine the right order of the various projections. A full precision story may need an annotation of all projection functions with domains and codomains. But as it turns out these matters are often clear from the context and merely needed for formalization rather than for rigorous explanation. Therefore that kind of bookkeeping has been omitted, with the implicit understanding that it can always be introduced if additional clarification is necessary.

6 Extensions of ISLA with labels and goto's, conditional constructs and while loops

6.1 Projective syntax for labels and goto's

Labels have been introduced in the program algebra setting in [10] in the program notation PGLDg. Here, however, labels will have the form $[s]$, with s a non-empty alphanumerical string, i.e., a sequence over the alphabet $\{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9\}$; $\#\#[s]$ and $\#\#[s][t]$ are the corresponding *single* or *chained goto* instructions. The intended meaning is that a label is executed as a skip. A single goto stands for a jump to the leftmost instruction containing the corresponding label if that exists and a termination instruction otherwise, and a chained goto of the form $\#\#[s][t]$ stands for a jump to the leftmost instruction containing the label $[s]$ followed by a jump to the leftmost instruction thereafter containing the label $[t]$ assuming hat they both exist and termination otherwise. The label occurrence that serves as the destination of a single or chained goto is called the *target* occurrence of the goto instruction.

In order to provide a projection for A:GL (program notation A with goto's and labels) the introduction of annotated goto's is useful. $\#\#[s]m$ ($\#\#[s][t]m$) represents the goto instruction $\#\#[s]$ ($\#\#[s][t]$) in a program where the target label is at position m. $\#\#[s]0$ ($\#\#[s][t]0$) represents the case that no target label exists. The projection from A:GLa (A with annotated goto's and labels) to A and from A:GL to A:GLa are obvious.

Projection 4 Let ISLA $\subseteq$ A.

(1) ϕ : A:GLa $\to$ A is defined by $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$ where
   (a) $\phi_i([s]) = \#\#i + 1$,
   (b) $\phi_i(\#\#[s]m) = \phi_i(\#\#[s][t]m) = \#\#m$.
   The other instructions remain unmodified.

(2) ϕ : A:GL $\to$ A:GLa is defined by $\phi(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$ where $\phi(\#\#[s]) = \#\#[s]m$ and $\phi(\#\#[s][t]) = \#\#[s][t]m$ with m the instruction number of the target label if it exists and 0 otherwise. The other instructions remain unmodified.

6.2 Second level instructions

When transforming a program it may be necessary to insert one or more instructions. Unfortunately that renders the counting of jumps useless. Of course jump counters may be updated simultaneously thus compensating for the introduction of additional instructions. Doing so has proved to lead to

unreadable descriptions of projection functions, however, and the device of two level instruction counting will be proposed as a more readable alternative.

Instructions will be split in *first level* instructions and *second level* instructions. The idea is that expanding projections—projections that replace instructions by non-unit instructions—are given in such way that each instruction sequence replacing a single instruction begins with a first level instruction while all subsequent instructions are taken at the second level.

Second level instructions are instructions prefixed with a $\sim$. First level instructions do not have a prefix; they are made second level by prefixing them with a $\sim$. First and second level absolute jumps $\#\#k$ and $\sim \#\#k$ will represent a jump to the k-th first level instruction if that exist (and termination otherwise) simply ignoring the second level instructions. The extension of a ISLA based program notation A with second level instructions is denoted with A:SL. For any A $\supseteq$ ISLA, the projection ϕ : A:SL $\to$ A removes second level instruction markers and updates the jumps.

Projection 5 Let ISLA $\subseteq$ A.
ϕ : A:SL $\to$ A is defined by $\phi(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$ where

$$\phi(\#\#k) = \begin{cases} ! & \text{if } k > max, \\ \#\#k + l & \text{otherwise.} \end{cases}$$

Here max is the number of first level instructions occurring in $u_1; \ldots; u_n$ and l is the number of second level instructions preceding the k-th first level instruction. The other first level instructions remain unmodified. Moreover, $\phi(\sim \#\#k) = \phi(\#\#k)$ and $\phi(\sim u) = u$ for all other first level instructions u.

Example 6 $\phi(\sim a; \sim +b; \sim \#\#2; \#\#1) = a; +b; !; \#\#4$

The only use of A:SL is as a target notation for projections from an ISLA based program notation. There will be a number of examples of this in the sequel.

Notation 7 For notational convenience we introduce the following abbreviations. We write

- $a\#\#jmp$ for $a; \sim \#\#jmp$, $+a\#\#jmp$ for $+a; \sim \#\#jmp$, and $-a\#\#jmp$ for $-a; \sim \#\#jmp$, and likewise
- $\sim a\#\#jmp$ for $\sim a; \sim \#\#jmp$, $\sim +a\#\#jmp$ for $\sim +a; \sim \#\#jmp$ and $\sim -a\#\#k$ for $\sim -a; \sim \#\#jmp$

with $jmp \in \mathbb{N} \cup \{[s], [s][t] \mid s, t \in \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9\}^+\}$.

6.3 Projective syntax for the conditional construct

Conditional constructs can be added to ISLA by using four new forms of instruction: for each basic action a, $+a\{$ and $-a\{$ are *conditional header* instructions, further $\}\{$ is the *separator* instruction and $\}$ is the *end of construct* instruction. The idea is that in $+a\{;X;\}\{;Y;\}$ after performing a, at a positive reply X is performed and at a negative reply Y is performed. The program notation combining an ISLA based program notation A and these conditional instructions is denoted with A:C. A projection for the conditional construct instructions is found using second level instructions. Due to the use of second level instructions the projection can be given by replacing instructions without the need to update jump counters elsewhere in the program. The projection to second level instructions will be such that e.g. $\phi(b;+a\{;c;-d;\#\#0;\}\{;+e;\#\#4;\};f) =$

$$b;-a\#\#7;c;-d;\#\#0;\#\#10;+e;\#\#4;\#\#10;f.$$

The general pattern of this projection is as follows: the conditional header instructions are mapped onto a sequence of two instructions, one performing the test and the second instruction containing the second level jump to the position following the projected separator instruction. The separator instruction is mapped to a jump to the position following the projected end of construct instruction, and the end of construct instruction is mapped to a skip, i.e., to a jump to the position thereafter. For this projection to work out some parsing of the program is needed in order to find out which separator instruction matches with which conditional construct header instruction and which end of construct instruction.

The annotated brace instructions needed for the projection of the conditional construct instructions are as follows: $+a\{k,\ -a\{k,\ \}\{k,\ k\}$ with k a natural number. This number indicates the position of instructions that contain the corresponding opening, separating or closing braces, or 0 if such an instruction cannot be found in the program. A:Ca is the extension of a program notation A with annotated versions of the special conditional statements. An annotated version of the program given above is

$$b;+a\{6;c;-d;\#\#0;2\}\{9;+e;\#\#4;6\};f.$$

The projection from A:Ca to A:SL is now again obvious.

Projection 8 Let ISLA $\subseteq$ A.

ϕ : A:Ca $\rightarrow$ A:SL is defined by $\phi(u_1;\ldots;u_n) = \phi_1(u_1);\ldots;\phi_n(u_n)$ where

$$\begin{aligned}\phi_i(+a\{k) &= -a\#\#k+1\\ \phi_i(-a\{k) &= +a\#\#k+1\\ \phi_i(\}\{k) &= \#\#k+1\\ \phi_i(k\}) &= \#\#i+1\end{aligned}$$

for $k > 0$ and,

$$\begin{aligned}\phi_i(+a\{0) &= -a\#\#0\\ \phi_i(-a\{0) &= +a\#\#0\\ \phi_i(\}\{0) &= \#\#0\\ \phi_i(0\}) &= \#\#0\end{aligned}$$

The other instructions remain unmodified.

The projection of A:C to A:Ca involves a global inspection of the entire A:C program. There is room for confusion, for instance, in the case

$$P = a;\};+b\{;c;\}\{;d;\}\{;e;\{.$$

In spite of the fact that the program P is not a plausible outcome of programming in ISLA:C, an annotated version of can be established as

$$a;0\};+b\{5;c;\}\{0;d;\}\{0;e;\{0,$$

which contains all semantic information needed for a projection.

Projection 9 Let ISLA $\subseteq$ A.
ϕ : A:C $\rightarrow$ A:Ca is defined by $\phi(u_1;\ldots;u_n) = \phi(u_1);\ldots;\phi(u_n)$ where

(1) $\phi(+a\{) = +a\{k$ and $\phi(-a\{) = -a\{k$
with k the instruction number of the corresponding separator instruction if it exists and 0 otherwise,
(2) $\phi(\}\{) = \}\{k$
with k and the instruction number of the corresponding end of construct instruction if it exists and 0 otherwise,
(3) $\phi(\}) = k\}$
with k the instruction number of the corresponding separator instruction if it exist and 0 otherwise.

The other instructions remain unmodified.

6.4 Projective syntax for while loops

The following three instructions support the incorporation of while loops in programs projectible to ISLA: $+a\{*, -a\{*$ (*while loop header* instructions) and $*\}$ (*while loop end* instruction). This gives the program notation A:W. As in the case of conditional constructs, while constructs will be projected using annotated versions.

A:Wa allows the annotated while loop instructions $+a\{*k$, $-a\{*k$ and $k*\}$. The projection of annotated while loop instructions is again obvious.

Projection 10 Let ISLA $\subseteq$ A.
ϕ : A:Wa $\rightarrow$ A:SL is defined by $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$ where

$$\begin{aligned} \phi_i(+a\{*k) &= -a\#\#k + 1 \\ \phi_i(-a\{*k) &= +a\#\#k + 1 \\ \phi_i(k*\}) &= \#\#k \end{aligned}$$

for $k > 0$ and,

$$\begin{aligned} \phi_i(+a\{*0) &= -a\#\#0 \\ \phi_i(-a\{*0) &= +a\#\#0 \\ \phi_i(0*\}) &= \#\#0 \end{aligned}$$

The other instructions remain unmodified.

The introduction of annotated while braces by means of an annotating projection follows the same lines as in the case of conditional statement instructions.

Projection 11 Let ISLA $\subseteq$ A.
ϕ : A:W $\rightarrow$ A:Wa is defined by $\phi(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$ where

(1) $\phi(+a\{*) = +a\{*k$ and $\phi(-a\{*) = -a\{*k$
with k the instruction number of the corresponding while loop end instruction if it exists and 0 otherwise, and
(2) $\phi(*\}) = k*\}$
with k the instruction number of the corresponding while loop header instruction if it exist and 0 otherwise.

The other instructions remain unmodified.

Notice that a projection of the extension A:C of an ISLA based program notation A obtained by a simultaneous introduction of conditional and while

constructs can be given by a concatenation of the projections defined above by starting either with conditional or while instructions. Thus

$$\text{A:CW} = \text{(A:C):W} \to \text{(A:C):Wa} \to \text{(A:C):SL} \to \text{A:C} \to \cdots \to \text{A}.$$

We end this section with an example containing conditional and while construct instructions.

Example 12 Let $P = +a\{; b; +c\{*; d; *\}; \}\{; e; f; \}$. P can be read as do a and continue with e and f and terminate if a returns `false`; otherwise do b and repeat $c; d$ as long as the last reply for c was positive, and terminate after the first negative reply to an execution for c. We may view P as a program in (ISLA:C):W. The annotated version of P given by the above projections is $+a\{; b; +c\{*5; d; 3*\}; \}\{; e; f; \}$ and projection to (ISLA:C):SL yields

$$+a\{; b; -c\#\#6; d; \#\#3; \}\{; e; f; \}.$$

Projecting to ISLA:C we obtain

$$+a\{; b; -c; \#\#7; d; \#\#3; \}\{; e; f; \}.$$

We now proceed with the projection of the conditional constructs. The annotated version is

$$+a\{7; b; -c; \#\#7; d; \#\#3; \}\{10; e; f; 7\}$$

and projection to ISLA:SL yields

$$-a\#\#8; b; -c; \#\#7; d; \#\#3; \#\#11; e; f; \#\#11.$$

Finally, projecting to ISLA we obtain

$$-a; \#\#9; b; -c; \#\#8; d; \#\#4; !; e; f; !.$$

The projections discussed in this section are summarized in Figure 2.

7 Extension with constructs involving recursion

In some cases a program needs to make use of a data structure for its computation. This data structure is active during the computation only and helps the control mechanisms of the program. Such a data structure is called a *service* or *state machine* [13]. A formalization of this matter involves a refinement of the syntax for basic actions. Any service action used in a program is now supposed to consist of two parts, respectively the *focus* and the *method*, glued together

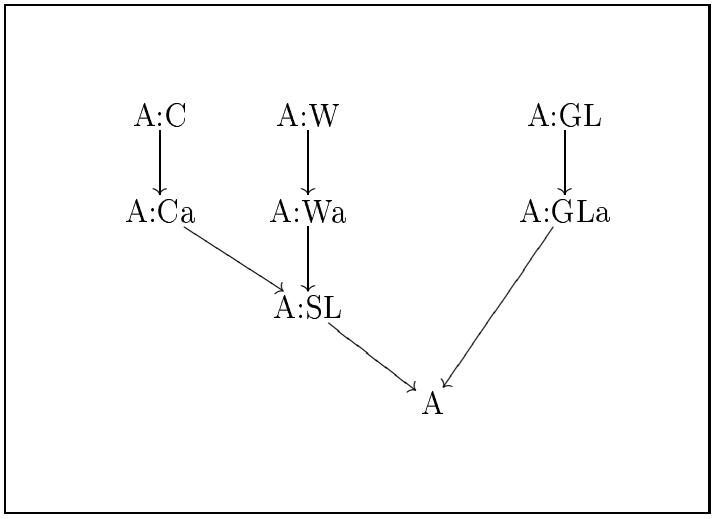

Fig. 2. Basic extensions of an ISLA based program notation A

with a period that is not permitted to occur in either one of them. For the service only the method matters because the focus is used to identify to which service (or other system component) an atomic instruction is directed. We denote the extension of a program in notation A by the instructions of service SER by A/SER. A service action a used in the extension will be written as $ser.a$. An example is given below.

NNV is a very simple data structure holding a single natural number—initially 0—which can be set and returned. The only basic actions performed by NNV are

(1) $set(n)$. The set action always succeeds setting the value to n.
(2) $eq(n)$ [5]. This action fails if the value is unequal to n; otherwise it returns $true$.

With a natural number value a *case statement*

$$nnv.case(m)\{;X_0;\}\{;X_1;\}\{;\cdots;\}\{;X_{m-1};\}\{;X_m;\}$$

becomes available providing a multi-way branching based on a natural number. The idea here is that if the value i of NNV lies in the range 0 to m, then X_i is performed and execution continues after the last closing brace, otherwise this construct is skipped. The program notation combining an ISLA based program notation A with NNV based case statements is denoted with A:CS$_{nnv}$. Its projection can be given by the use of annotations and second level instructions.

The annotated brace instructions needed for the projection are $nnv.case(m)\{k$ and $i\}\{k$ with natural numbers k, i where k indicates the instruction number

[5] Since basic actions are supposed to return Boolean values, we have replaced the return operation by an equality indicator function.

of the corresponding separator or end of construct instruction and i the value to be tested. k is set to 0 in the case of ill formed program syntax. A:CS$_{nnv}$a denotes the extension with annotated case statements.

Projection 13 Let ISLA $\subseteq$ A.
ϕ : A:CS$_{nnv}$a $\rightarrow$ A/NNV:SL is defined by $\phi(u_1;\ldots;u_n) = \phi_1(u_1);\ldots;\phi_n(u_n)$ where

$$\begin{aligned}\phi_i(nnv.case(m)\{k) &= -nnv.eq(0)\#\#k+1\\ \phi_i(i\}\{k) &= -nnv.eq(i)\#\#k+1\\ \phi_i(\}) &= \#\#i+1\end{aligned}$$

for $k > 0$ and,

$$\begin{aligned}\phi_i(nnv.case(m)\{0) &= \#\#0\\ \phi_i(i\}\{0) &= \#\#0\end{aligned}$$

The other instructions remain unmodified.

The projection of A:CS$_{nnv}$ to A:CS$_{nnv}$a involves again a complete inspection of the A:CS program.

Projection 14 Let ISLA $\subseteq$ A.
ϕ : A:CS$_{nnv}$ $\rightarrow$ A:CS$_{nnv}$a is defined by $\phi(u_1;\ldots;u_n) = \phi(u_1);\ldots;\phi(u_n)$ where

(1) $\phi(nnv.case(m)\{) = nnv.case(m)\{k$
with k the instruction number of the corresponding separator or end of construct instruction if it exists and 0 otherwise,
(2) $\phi(\}\{) = i\}\{k$
with i the test value and k and the instruction number of the corresponding separator or end of construct instruction if it exists and 0 otherwise.

The other instructions remain unmodified.

A more advanced data structure is the *stack*. The key application of a stack is to obtain projections for program constructs involving recursion.

The basic actions performed by a stack S are

(1) *pop*. This instruction fails if the stack is empty. Otherwise it succeeds and removes the top from the stack while giving a positive reply.
(2) *push*(n). The push action always succeeds placing the natural number n on top of the stack and producing a positive reply.
(3) *topeq*(n). This action returns `true` if the top of the stack equals n; otherwise it returns `false`.

Similar to the case statement for natural number values one can now introduce case statements

$$s.case(m)\{;X_0;\}\{;X_1;\}\{;\cdots;\}\{;X_{m-1};\}\{;X_m;\}$$

based on the topmost element of the stack. Annotation and projection are defined in the same fashion. The program notation combining an ISLA based program notation A with stack based case statements is denoted with A:CS$_s$; its annotated version is denoted A:CS$_s a$.

Projection 15 Let ISLA $\subseteq$ A.
ϕ : A:CS$_s$a $\to$ A/S:SL is defined by $\phi(u_1;\ldots;u_n) = \phi_1(u_1);\ldots;\phi_n(u_n)$ where

$$\begin{aligned}\phi_i(s.case(m)\{k) &= -s.topeq(0)\#\#k+1\\ \phi_i(i\}\{k) &= -s.topeq(i)\#\#k+1\\ \phi_i(\}) &= \#\#i+1\end{aligned}$$

for $k > 0$ and,

$$\begin{aligned}\phi_i(s.case(m)\{0) &= \#\#0\\ \phi_i(i\}\{0) &= \#\#0\end{aligned}$$

The other instructions remain unmodified.

Projection 16 Let ISLA $\subseteq$ A.
ϕ : A:CS$_s$ $\to$ A:CS$_s$a is defined by $\phi(u_1;\ldots;u_n) = \phi(u_1);\ldots;\phi(u_n)$ where

(1) $\phi(s.case(m)\{) = s.case(m)\{k$
with k the instruction number of the corresponding separator or end of construct instruction if it exists and 0 otherwise,
(2) $\phi(\}\{) = i\}\{k$
with i the test value and k and the instruction number of the corresponding separator or end of construct instruction if it exists and 0 otherwise.

The other instructions remain unmodified.

In preparation of the incorporation of recursion, *dynamic jumps* $\#\#s.pop$—where the counter depends on the top of the current stack—will be introduced. The program notation that combines an extension A of ISLA with dynamic jumps is denoted with A:DJ. The projection to A/S:CS$_s$:SL takes into account that in a program $u_1;\ldots;\#\#s.pop;\ldots;u_n$ at most n different values of the jump counter are needed.

Projection 17 Let ISLA $\subseteq$ A.
ϕ : A:DJ $\to$ A/S:CS$_s$:SL is defined by $\phi(u_1;\ldots;u_n) = \phi(u_1);\ldots;\phi(u_n)$ where

$\phi(\#\#s.pop) =$

$$s.case(n)\{;\sim s.pop\#\#0;\sim\}\{;\cdots;\sim\}\{;\sim s.pop\#\#n;\sim\}$$

All other instructions remain unmodified.

Having dynamic jumps available, we can now introduce recursion. The instruction pair $R\#\#k$, $\#\#R$ represents recursion in the following way: if a *returning jump instruction* $R\#\#k$ occurring at position i is executed a jump to instruction k is made, moreover the instruction counter $i+1$ is placed on the stack. Whenever a *return instruction* $\#\#R$ is executed a jump is performed to the top of the stack which is simultaneously popped.

A:R is the program notation A augmented with returning jump instructions of the form $R\#\#k$ and return instructions $\#\#R$. A projection to A/S:SL:DJ can be given in the following way.

Projection 18 Let ISLA $\subseteq$ A.
ϕ : A:R $\to$ A/S:SL:DJ is defined by $\phi(u_1;\ldots;u_n) = \phi_1(u_1);\ldots;\phi_n(u_n)$ where

$$\phi_i(R\#\#k) = s.push(i+1)\#\#k$$
$$\phi_i(\#\#R) = \#\#s.pop$$

All other instructions remain unmodified.

The part of the computation taking place between a returning jump and its corresponding return instruction may be called a subcomputation. Recursion using returning jumps and return instructions can be made more expressive by means of parameters, i.e., values that serve as inputs to a subcomputation. They are put in place before executing a returning jump, and the program part executed as a subcomputation 'knows' where to find these parameters. Various strategies can be imagined for arranging the transfer of parameters during returning jump and return instructions. Here we have chosen for an automatic parameter transfer to and from a stack to local spots.

A *stack with value pool* is a data structure that maintains a stack and a value pool. The stack stores values of primitive types and instruction counters which do not count as values. The value pool consists of finitely many pairs with the first element being the *spot* relative to the value pool and the second element its content. There are two distinguished spots: *this* and *that*. *this* contains the instances for object oriented instance method calls and *that* the results of subcomputations. In addition to the usual stack actions a stack with value pool can perform the following actions which transfer values between stack and pool:

(4) $store(s)$. The precondition of this basic action is that the stack is non-empty; otherwise it fails. If the stack is non-empty its top is removed and made to be the content of spot s. The previous content of s, if any, is lost in the process.
(5) $load(s)$. This basic action takes the content of the spot s, which must have been introduced at an earlier stage, and pushes that on the stack. A spot is defined if it has been assigned a value by means of a *store* instruction at some stage. If the spot is still undefined the action fails. This is a copy instruction because the value is still the content of s as well after successful execution.

We now assume that the m parameters of a recursion—and in the case of an object oriented instance method call also an instance parameter that plays the role of the target instance—are initially stored consecutively on a stack. From here they are placed in the value pool at spots $arg1, \ldots, argm$—and possibly *this*—but only after having copied the contents of these spots to a second auxiliary stack in order to allow recovery of the original contents. Just after the returning jump and just before the return instruction the top of the auxiliary stack contains the parameters in decreasing order. The projection of the return instruction will then involve the recovery of the several spots before the returning jump from the data that were placed on the auxiliary stack.

In the sequel we shall write $Rm\#\#k$ and $IRm\#\#k$ $(m, k \in \mathbb{N})$ for returning jump instructions with m parameters and a possible instance. Given a program notation A the extension by parametrized returning jumps and return instructions is denoted A:Rp. The projection uses two stacks sharing a single value pool called svp and svp_{aux}, respectively. In order to recover the copied spots it is necessary to replace returning jumps by instruction sequences containing 2 first level instructions. As a consequence jump counters have to be updated.

Projection 19 Let ISLA $\subseteq$ A.
ϕ : A:RP $\to$ A/SVP:SL:DJ is defined by $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$

where

$$\begin{aligned}\phi_i(Rm\#\#k) = {} & svp_{aux}.load(arg1);\\ & \sim svp_{aux}.load(arg2);\cdots;\sim svp_{aux}.load(argm);\\ & \sim svp.store(arg1);\cdots;\sim svp.store(argm);\\ & \sim svp.push(i+1)\#\#k;\\ & svp_{aux}.store(argm);\\ & \sim svp_{aux}.store(argm-1);\cdots;\sim svp_{aux}.store(arg1)\end{aligned}$$

$$\begin{aligned}\phi_i(IRm\#\#k) = {} & svp_{aux}.load(arg1);\\ & \sim svp_{aux}.load(arg2);\cdots;\sim svp_{aux}.load(argm);\\ & \sim svp_{aux}.load(this);\\ & \sim svp.store(this);\sim svp.store(arg1);\cdots;\sim svp.store(argm);\\ & \sim svp.push(i+1)\#\#k;\\ & svp_{aux}.store(this);\\ & \sim svp_{aux}.store(argm);\cdots;\sim svp_{aux}.store(arg1)\end{aligned}$$

$$\phi_i(\#\#R) = \#\#svp.pop$$

$$\phi_i(\#\#l) = \#\#l + r$$

where r is the number of (instance) returning jumps preceding u_l. All other instructions remain unmodified.

The two stacks used in the projection above van be combined into a single stack with value pool by considering a stack svp that performs in addition to the usual stack and transfer actions also the action

(6) $down(n)$. This action places the top just below the n-th position from above after removing the top. Thus after $down(n)$ the top has migrated to the $n+1$-th position from above. If that is impossible the action fails, a negative reply is given and no change is made to the stack.

Allowing for $down$ actions, one can replace in Projection 19 instructions of the form $svp_{aux}.load(x)$ by $svp.load(x); down(m)$ if no instance parameter is involved and by $svp.load(x); down(m+1)$ otherwise; instructions of the form

$svp_{aux}.store(x)$ have to be stripped off their subscript.

It is important to notice that for the projection of a specific program only a finite number of $down(n)$ instructions is used. This implies that the stack manipulations do not reach the expressive power of a full stackwalk, but rather may be simulated (per program) from a stack and a finite memory service, or more easily obtained using a bounded value buffer.

The projections of recursion instructions are summarized in Figure 3. The final projection to A/S can be obtained by chaining the appropriate projections discussed in the preceding sections.

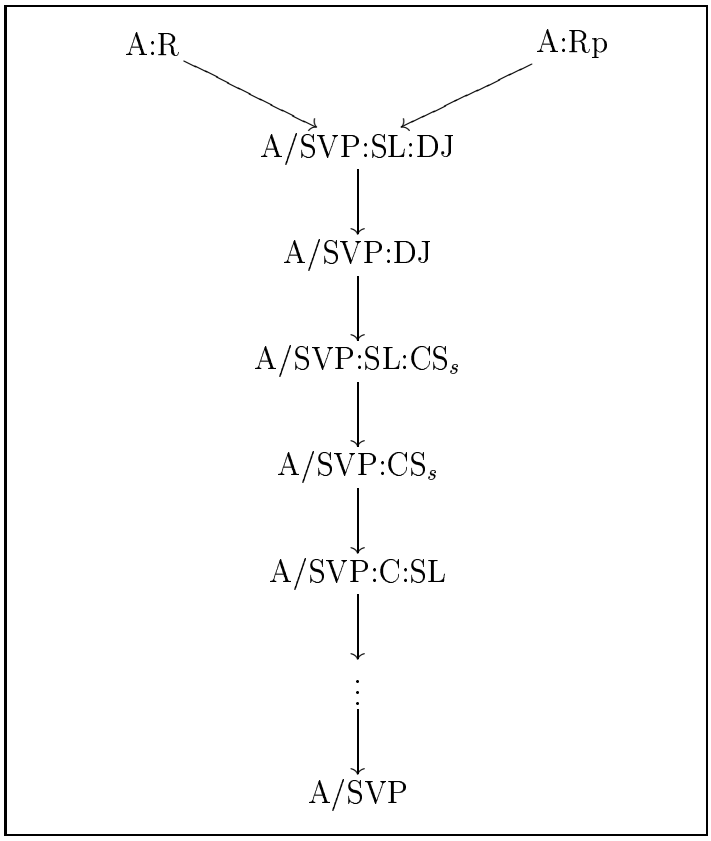

Fig. 3. Projections of recursion and parametrized recursion with automatic parameter transfer using a stack with value pool

8 Intermediate level programs

The design of instructions can be viewed as proceeding in three phases. Until this point only so-called low level instructions have been designed. Low level instructions have either fixed projections or their projection trivially depends on the program context. Program context dependence occurs for in-

stance with a closing brace instruction for which the projection may depend on the index of its corresponding opening brace instruction. The second design phase introduces an intermediate language ILN for high-level object oriented programming constructs which makes use of an *operation interface*.

An operation interface OI is a data structure that maintains in addition to a stack and a value pool a heap which memorizes instance data and the class of objects. The stack stores values of primitive types, instruction counters and references to objects.

Assuming also the existence of a family of operators, the basic actions performed by OI are in addition to the stack and value pool actions:

(7) *compute*(f, k) (with f the name of an operator with arity k). It is assumed that k values $v_1, ..., v_k$ are stored consecutively on the stack. If there are too few values on top of the stack the action fails and the state is unchanged. If there are sufficiently many values the reply is positive and the value $f(v_1, ..., v_k)$ is placed on the stack after removing all k arguments from it.

(8) *create*. This basic action allocates a new reference and puts it on top of the stack. If a new reference is not available this action fails.

(9) *getfield*(F). The precondition of this action is that the top of the stack is a reference denoting an instance with field F; otherwise it fails. If the precondition is met, the content of field F of the instance denoted by the top is pushed from the heap on the stack after having removed the top.

(10) *setfield*(F). The precondition of this action is that the top of the stack is a reference followed by a value or a reference; otherwise it fails. If the precondition is met, the reference and the value are popped from the stack and field F of the instance denoted by the popped reference is set to the popped value.

(11) *instanceof*(C). The precondition of this action is that the top of the stack is a reference; otherwise it fails. If the precondition is met, this action returns `true` if the class of the instance denoted by the reference is C, otherwise it returns `false`. Afterwards the reference is taken from the stack.

(12) *setclass*(C). Again, the precondition is that the top of the stack is a reference; otherwise the action fails. If the precondition is met, the class of the instance denoted by the reference is set to C. Afterwards the reference is popped.

In order to make this data structure less abstract, we provide a more detailed and concrete description.

Let us denote with *BinSeq* the finite sequences of 0's and 1's. Three subsets of *BinSeq* are then at the basis of the concrete operation interface:

- *Loc* is a finite or infinite set of so-called *locations*—in practical cases it must be finite, in some formal modeling cases an infinite set of locations may be more amenable.
- *Ic* is the finite or infinite collection of instruction counters. Here the most plausible choice is that *Ic* contains binary forms of all natural numbers.
- *Pval* is the collection of primitive values. This is again a finite set in most practical cases, just like *Loc*.

On the stack one finds

- references: taking the form rw with $w \in Loc$, e.g. $r00110$; the set of rw's with $w \in Loc$ is denoted *Ref* below,
- instruction counters: taking the form iw with $w \in Ic$, and
- primitive values: taking the form w with $w \in Pval$.

In any state of computation the spots have either no content—the initial state for all spots—or, if a spot has a content, it is either a reference rw or a primitive value.

Several collections of names are used: *ClassNames*, *FieldNames* and *SpotNames*. Any convention will do and these sets may overlap without generating confusion. A state of the concrete operation interface consists now of the following sets:

- *Obj* is a subset of *Loc*. *Obj* contains those locations that contain an object. Initially a computation will start with *Obj* empty.
- *Classification* is a subset of $Obj \times ClassNames$. We may regard *Classification* in fact as a partial function from *Obj* to *ClassNames*.
- *Fields* is a subset of $Obj \times FieldNames \times (Ref \cup Pval)$. Again we assume that *Fields* is a partial function from $Obj \times FieldNames$ to $Ref \cup Pval$.
- *Spots* is a subset of $SpotNames \times (Ref \cup Pval)$ and hence a partial function from *SpotNames* to $Ref \cup Pval$.

It is important to make a distinction between vital objects and garbage objects. Vital objects are those that can be reached from spots containing a reference and references positioned anywhere on the stack via zero or more selections of fields containing a reference. Non-vital objects are called garbage objects. After each method execution garbage objects are removed and so are their outgoing fields and classifications.

Now the operations of the heap can be interpreted as follows:

- *create*. If $Obj = Loc$ then return `false`; otherwise find $w \in Loc - Obj$, place rw on the top of the stack, put w in *Obj*, and then return `true`.
- *getfield*(f). If the top of the stack is not a reference or if the top of the stack contains rw, say, but $Fields(rw, f)$ is not defined return `false`; otherwise

remove the top of the stack, place $Fields(rw, f)$ on the stack and return `true`.

- *setfield*(f). If the top of the stack is a reference rw, say, and if it lies on top of a value v in $Ref \cup Pval$, pop the two uppermost items from the stack, define $Fields(rw, f) = v$ and return `true`; otherwise return `false`.
- *instanceof*(C). If the top of the stack is not a reference return `false`; if the top of the stack contains rw, say, pop the stack, return `true` if $Classification(rw) = C$, and return `false` otherwise.
- *setclass*(C). If the top of the stack is not a reference return `false`; if the top of the stack contains rw, say, define $Classification(rw) = C$, pop the stack and return `true`.

In this concrete operation interface a reference takes the form rw and spots may also be considered references. In other models references may work differently so this description of a concrete operation interface cannot be taken as an analysis of what constitutes a reference in general.

In preparation of the projection of ILN we introduce a *dynamic chained goto* instruction $\#\#[instanceof][M]$ where the first goto depends on the class of the instance *this*. The projection from A:DCG (A with dynamic chained goto's) to A/OI:C:GL:SL can be given in the following way.

Projection 20 Let ISLA $\subseteq$ A.
ϕ : A:DCG $\rightarrow$ A/OI:GL:SL is defined by $\phi(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$ where

$$\begin{aligned}\phi(\#\#[instanceof][M]) = {} & oi.load(this); \sim +instanceof(C_1)\#\#[C_1][M]; \\ & \sim oi.load(this); \sim +instanceof(C_2)\#\#[C_2][M]; \\ & \vdots \\ & \sim oi.load(this); \sim +instanceof(C_k)\#\#[C_k][M]\end{aligned}$$

where $C1, \ldots, Ck$ are the classes introduced by *setclass* instructions in u_1; $\ldots$; u_n. All other instructions remain unmodified.

We now proceed with the second design phase. A typical ILN program consists of a sequence of labeled class parts. E.g., an ILN program with 4 classes looks as follows:

$$[C1]; CB1; \ldots; [C4]; CB4.$$

The subprograms $CB1, \ldots, CB4$ are so-called class bodies, which consist of zero or more labeled method body parts with end markers

$$[M1]; MB1; end; \ldots; [Mk]; MBk; end.$$

In addition to the usual labels, ILN has the following instruction set.

new(C) This instruction pushes a new reference to an instance of class C onto the stack.

stack push The instruction $E \Rightarrow$ represents pushing an entity with name E onto the stack. E may be a single non-empty alphanumerical string or of the form $E.F$, i.e., a string consisting of two alphanumerical parts glued together with a period.

top from stack The instruction $\Rightarrow E$ takes the top from the stack and places that entity on the place denoted with E. Again, E may be a single non-empty alphanumerical string or of the form $E.F$.

method end marker The method end marker *end* serves as the end of a method.

function call A function call $fc(f, n)$ represents the call of the function f from a given operator family with arity n. The algorithmic content of an operator call is not given by the program that contains it.

class method call A class method call instruction has the form $mc(C, M, n)$. Here M is a method, C a class and n is the number of arguments. We tacitly assume that the names occurring in the stack push and pop instructions of M are amongst $arg1, \ldots, argn$.

instance method call An instance method call instruction has the form $mc(M, n)$. Here M is a method and n is the number of arguments. We tacitly assume that the names occurring in the stack push and pop instructions of M are amongst $this, arg1, \ldots, argn$.

if header An if-then-else construction has the form $if(E); X; else; Y; endif$ separated by *else* and completed by an *endif* instruction. If the value of E is *true* then the instruction stream X is performed; otherwise Y is executed.

while header A while construction has the form $while(E); X; endwhile$ and performs the instruction stream X as long as the value of E is *true*.

separator and end of construct *else* and the end markers *endif* and *endwhile* serve as separator and end markers for the conditional construct and the while loop.

ILN instructions can be projected to ISLA/OI:DCG:GL:Rp:CW.

Projection 21 ϕ : ILN $\rightarrow$ ISLA/OI:DCG:GL:Rp:CW is defined by $\phi(u_1; \ldots; u_n) = \phi_1(u_1); \ldots; \phi_n(u_n)$ where

$$
\begin{aligned}
\phi_i(new(C)) &= oi.create; oi.setclass(C) \\
\phi_i(E \Rightarrow) &= oi.load(E) \\
\phi_i(\Rightarrow E) &= oi.store(E) \\
\phi_i(E.F \Rightarrow) &= oi.load(E); oi.getfield(F) \\
\phi_i(\Rightarrow E.F) &= oi.load(E); oi.setfield(F) \\
\phi_i(end) &= \#\#R
\end{aligned}
$$

$$
\begin{aligned}
\phi_i(fc(f,m)) &= oi.compute(f,m)\\
\phi_i(mc(C,M,m)) &= Rm\#\#3k_1+2k_2+k_3+i+2;\#\#3k_1+2k_2+k_3+i+3;\\
&\quad \#\#[C][M]\\
\phi_i(mc(M,m)) &= IRm\#\#3k_1+2k_2+k_3+i+2;\#\#3k_1+2k_2+k_3+i+3;\\
&\quad \#\#[instanceof][M]\\
\phi_i(if\,(E)) &= oi.load(E);oi.compute(==true,1);+oi.topeq(1)\{;oi.pop\\
\phi_i(while(E)) &= oi.load(E);oi.compute(==true,1);+oi.topeq(1)\{*;oi.pop\\
\phi_i(else) &= \}\{;oi.pop\\
\phi_i(endif) &= \}\\
\phi_i(endwhile(E)) &= oi.load(E);oi.compute(==true,1);*\};oi.pop
\end{aligned}
$$

Here k_1 is the number of if and while headers and endwhile instructions, k_2 the number of method calls and k_3 the number of creation, field pushing and popping, and else instructions occurring in $u_1;\ldots;u_{i-1}$. Moreover, $==true$ is the name of the operator that returns 1 if the argument has value *true* and 0 otherwise. In a setting with relative jumps one can replace the complex absolute jumps by relative jumps of length 2. Labels remain unmodified.

A simple ILN program P and its projection are given below. P consists of a single class part labeled $[C]$ with method parts $[CC]$, $[get]$ and $[M]$. $[CC]$ is a class constructor initializing the F field of any object of this class with a random Boolean value, $[get]$ is an instance method returning the value of the F field of the calling object, and $[M]$ is a static method creating two objects of this class—X and Y—by calling the constructor for X but setting the F field of Y to the complement of the F field of X by using the constant operators *true* and *false*.

$$
\begin{aligned}
&[C];\\
&\quad [CC];fc(random,0);\Rightarrow this.F;end;\\
&\quad [get];arg1.F\Rightarrow;\Rightarrow that;end;\\
&\quad [M];new\ C;\Rightarrow X;X\Rightarrow;mc(CC,0);\\
&\qquad new\ C;\Rightarrow Y;\\
&\qquad X\Rightarrow;mc(C,get,1);\\
&\qquad if\,(that);fc(false,0);\Rightarrow Y.F;else;fc(true,0);\Rightarrow Y.F;endif
\end{aligned}
$$

Projection into ISLA/OI:DCG:GL:Rp:CW yields

$[C];$

$[CC]; oi.compute(random, 0); oi.load(this); oi.setfield(F); \#\#R;$

$[get]; oi.load(arg1); oi.getfield(F); oi.store(that); \#\#R;$

$[M]; oi.create; oi.setclass(C); oi.store(X); oi.load(X); IR0\#\#19; \#\#20;$

$\#\#[instanceof][CC];$

$oi.create; oi.setclass(C); oi.store(Y);$

$oi.load(X); R1\#\#26; \#\#27; \#\#[C][get];$

$oi.load(that); oi.compute(== true, 1);$

$+s.topeq(1)\{; oi.pop; fc(false, 0); \Rightarrow Y.F; \}\{; oi.pop; fc(true, 0); \Rightarrow Y.F; \}$

9 A high-level program notation HLN

In the third phase we deviate from the projective syntax paradigm and introduce instructions at a level close to the high-level program notations used by human programmers and computer software users.

In this section a toy high-level program notation HLN is developed. Its meaning is given by a compiler projection into ILN. Like ILN programs HLN programs are supposed to have a certain fixed structure. In this structure a program is a series of class parts as depicted in Figure 4. Class parts consist of a

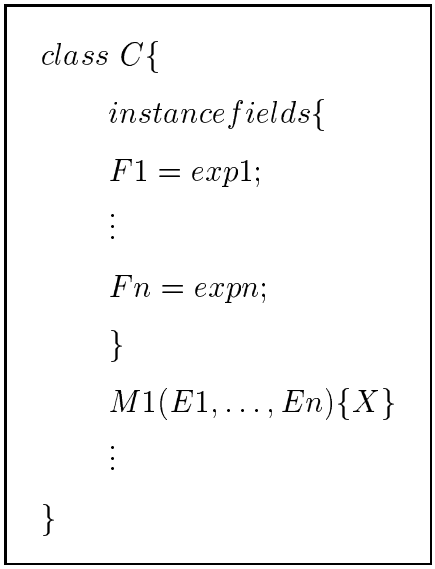

Fig. 4. The structure of class parts in HLN

number of segments. These segments can be of two kinds: instance field parts

and method declarations. The compiler projection will transform the list of instance field parts in a class to a class constructor for that class with label $[CC]$. Method declarations—which have the form $M(E1, \ldots, En)\{X\}$ and are named different from CC—will be projected to class method parts.

HLN has the following expressions.

$$exp := E \mid E.F \mid f(exp1, \ldots, expn) \mid I.M(exp1, \ldots, expn) \mid C.M(exp1, \ldots, expn)$$

with f an n-ary operator from a given operator family, I a class instance calling an n-ary instance method and $C.M(exp1, \ldots, expn)$ an n-ary class method call. Assignments and statements are of the form

$$\begin{aligned}
asg &:= E = exp \mid E.F = exp \mid E = new\ C \mid E.F = new\ C \\
stm &:= asg \mid return\ exp \mid I.M(exp1, \ldots, expn) \mid C.M(exp1, \ldots, expn) \mid \\
&\quad\ if\ (exp)\ blk\ else\ blk \mid while\ (exp)\ blk \\
blk &:= \{stm1; \ldots; stmn\}
\end{aligned}$$

The projections to ILN can be given by

exp :

$$\begin{aligned}
\phi(E) &= E \Rightarrow \\
\phi(E.F) &= E.F \Rightarrow \\
\phi(f(exp1, \ldots, expn)) &= \phi(exp1); \ldots; \phi(expn); fc(f, n) \\
\phi(I.M(exp1, \ldots, expn)) &= \phi(exp1); \ldots; \phi(expn); I \Rightarrow; mc(M, n); that \Rightarrow \\
\phi(C.M(exp1, \ldots, expn)) &= \phi(exp1); \ldots; \phi(expn); mc(C, M, n); that \Rightarrow
\end{aligned}$$

asg :

$$\begin{aligned}
\phi(E = exp) &= \phi(exp); \Rightarrow E \\
\phi(E.F = exp) &= \phi(exp); \Rightarrow E.F \\
\phi(E = new\ C) &= new\ C; mc(CC, 0); \Rightarrow E \\
\phi(E.F = new\ C) &= new\ C; mc(CC, 0); \Rightarrow E.F
\end{aligned}$$

stm :

$$\begin{aligned}
\phi(return\ exp) &= \phi(exp); \Rightarrow that \\
\phi(I.M(exp1, \ldots, expn)) &= \phi(exp1); \ldots; \phi(expn); I \Rightarrow; mc(M, n) \\
\phi(C.M(exp1, \ldots, expn)) &= \phi(exp1); \ldots; \phi(expn); mc(C, M, n) \\
\phi(if\ (exp)\ blk\ else\ blk) &= if\ (\phi(exp)); \phi(blk); else; \phi(blk); endif \\
\phi(while\ (exp)\ blk) &= while\ (\phi(exp)); \phi(blk); endwhile(\phi(exp))
\end{aligned}$$

blk :

$$\phi(\{stm1; \ldots; stmn\}) = \phi(stm1); \ldots; \phi(stmn)$$

The meaning of class parts can finally be given by

$$\begin{aligned}
\phi(class\ C\{X\}) &= [C]; \phi(\{X\}) \\
\phi(instancefields\{F1 = exp1; \ldots; Fn = expn\}) &= [CC]; \\
&\quad \phi(this.F1 = exp1); \\
&\quad \ldots \\
&\quad \phi(this.Fn = expn); \\
&\quad end
\end{aligned}$$

$$\begin{aligned}
\phi(M(E1, \ldots, En)\{X\}) &= [M]; \\
&\quad E1 \Rightarrow; \ldots; En \Rightarrow; \\
&\quad \phi(E1 = arg1); \\
&\quad \ldots \\
&\quad \phi(En = argn); \\
&\quad \phi(\{X\}); \\
&\quad \Rightarrow En; \ldots; \Rightarrow E1; \\
&\quad end
\end{aligned}$$

Note that the projection of HLN to ILN is dynamically correct. That is, the elements on the stack meet the requirements of the current instructions: whenever an instruction counter is expected, the top of the stack will contain one, and whenever n values are expected, the n uppermost elements will be values.

We end this section with a typical HLN program and its projection into ILN.

Let P be the the following program

$$
\begin{array}{l}
class\ A\{ \\
\quad instancefields\{F = 0;\} \\
\quad get()\{return\ this.F;\} \\
\quad set(E)\{this.F = E;\} \\
\quad inc(E)\{this.set(this.get() + E);\} \\
\quad \} \\
\\
class\ B\{ \\
\quad get(E)\{return\ E.F;\} \\
\quad test()\{ \\
\qquad X = new\ A; \\
\qquad Y = new\ A; \\
\qquad X.set(1); \\
\qquad X.inc(1); \\
\qquad Y.F = B.get(X); \\
\quad \}
\end{array}
$$

Assuming that the constants $0, 1$ and addition $+$ are predefined operators, P can be projected into ILN by

$$
\begin{array}{l}
[A]; \\
\quad [CC]; fc(0,0); \Rightarrow this.F; end; \\
\quad [get]; this.F \Rightarrow; \Rightarrow that; end; \\
\quad [set]; E \Rightarrow; arg1 \Rightarrow; \Rightarrow E; E \Rightarrow; \Rightarrow this.F; \Rightarrow E; end; \\
\quad [inc]; E \Rightarrow; arg1 \Rightarrow; \Rightarrow E; this \Rightarrow; mc(get,0); that \Rightarrow; E \Rightarrow; fc(+,2); \\
\qquad this \Rightarrow; mc(set,1); end;
\end{array}
$$

$[B];$

$[get]; E \Rightarrow; arg1 \Rightarrow; \Rightarrow E; E.F \Rightarrow; \Rightarrow that; \Rightarrow E; end;$

$[test];$

$new\ A; mc(CC, 0); \Rightarrow X;$

$new\ A; mc(CC, 0); \Rightarrow Y;$

$fc(1, 0); X \Rightarrow; mc(set, 1);$

$fc(1, 0); X \Rightarrow; mc(inc, 1);$

$X \Rightarrow; mc(B, get, 1); that \Rightarrow; \Rightarrow Y.F;$

end

10 Concluding remarks

At this point a simple program notation for object oriented programming has been provided admitting a projection semantics. Many more features exist and one might say that the project of syntax design has only been touched upon. Still the claim is made that the above considerations provide a basis for the design of reliable syntax for much more involved program notations. Projection semantics provides a scientifically well-founded and rigorous yet simple approach to the semantics of programming languages. Future work will have to clarify that this framework is industrially viable for the high-level design and analysis of complex systems, and for natural refinements of models to executable and reliable code.

Acknowledgements

This paper benefited greatly from the insightful comments made by Mark Burgess and Kees Middelburg.

References

[1] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In [15], 197–292 (2001).

[2] U. Ammann. Code generation for a Pascal compiler. *Software - Practice and Experience*, 7, 391–423 (1977).

[3] J.H. Andrews. Process-algebraic foundations of aspect-oriented programming. In [69], 187–209 (2001).

[4] Jos C.M. Baeten (ed). *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press (1990).

[5] Jos C.M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger (eds). *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003*, Lecture Notes in Comput. Sci. 2719, Springer (2003).

[6] H. Bekić, D. Bjørner, W. Henhapl, C.B. Jones and P. Lucas *A Formal Definition of a PL/1 Subset.* IBM Laboratory Vienna, TR25.139 (1974).

[7] J.A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In [5], 1–21 (2003).

[8] J.A. Bergstra and I. Bethke. Polarized process algebra with reactive composition. *Theoretical Computer Science*, 343(3), 285–304 (2005).

[9] J.A. Bergstra and M.E. Loots. Program algebra for component code. *Formal Aspects of Computing*, 12(1), 1–17 (2000).

[10] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2), 125–156 (2002).

[11] J.A. Bergstra and C.A. Middelburg. A thread algebra with multi-level strategic interleaving. In [27], 35–48 (2005).

[12] J.A. Bergstra, C.A. Middelburg, and Y.S. Usenko. Discrete time process algebra and the semantics of SDL. In [15], 1209–1268 (2001).

[13] J.A. Bergstra and A. Ponse. Combining programs and services. *Journal of Logic and Algebraic Programming*, 51(2), 175–192 (2002).

[14] J.A. Bergstra and A. Ponse. Execution architectures for program algebra. *Journal of Applied Logic*, to appear (2006).

[15] J.A. Bergstra, A. Ponse, and S.A. Smolka (eds). *Handbook of Process Algebra*, North-Holland (2001).

[16] D. Bjørner and O. Oest (eds). *Towards a Formal Description of Ada*, Lecture Notes in Comput. Sci. 98, Springer (1980).

[17] D. Bjørner, M. Broy, and A.V. Zamulin (eds). *Perspectives of System Informatics. 4th International Andrei Ershov Memorial Conference, PSI 2001*, Novosibirsk (2201).

[18] E. Börger (ed). *Specification and Validation Methods*, Oxford University Press (1995).

[19] F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing (eds). *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, Lecture Notes in Comput. Sci. 247, Springer (1987).

[20] M. Burgess A site configuration engine. *Computing Systems* 8(2), 309 – 337 (1995).

[21] M. Burgess It's elementary, Dear Watson: Applying logic programming to convergent system management processes. In [28], 123 – 138 (1999).

[22] M. Burgess. *Principles of Network and System Administration*, J. Wiley and Sons (2000).

[23] M. Burgess. On the theory of system administration. *Science of Computer Programming* 49(1-3), 1 – 46 (2003).

[24] M. Burgess. Configurable immunity for evolving human-computer systems. *Science of Computer Programming* 51(3), 197 – 213 (2004).

[25] M. Burgess. *Analytical Network and System Administration - Managing Human-Computer Systems*. J. Wiley & Sons (2004).

[26] R. Cleaveland, X. Du, and S.A. Smolka. GCCS: a graphical coordination language for system specification. In [55], 284–298 (2000).

[27] S.B. Cooper, B. Löwe, and L. Torenvliet (eds). *New Computational Paradigms: First Conference on Computability in Europe, CiE 2005, Amsterdam, The Netherlands, June 8-12, 2005*. Springer-Verlag, LNCS 3526 (2005).

[28] A. Couch and M. Gilfix. *Proceedings of the 13th USENIX conference on System administration*, Seattle, Washington. USENIX Association, Berkely, CA, USA (1999).

[29] A. Couch and Y. Sun. On observable reproducibility in network configuration management. *Science of Computer Programming*, 53(2), 215–253 (2004).

[30] E.W. Dijkstra. Guarded commands, non-determinacy, and formal derivations of programs. *Commun. ACM*, 18, 453–457 (1975).

[31] M.A. Ertl (ed). Advances in interpreters, virtual machines and emulators. *Science of Computer Programming*, 57(3), 251–380 (2005).

[32] M. Felleisen and D.P. Friedman. Control operators, the SECD machine, and the λ-calculus. In [67], 193–217 (1987).

[33] J. Fox (ed). *Proceedings of the Symposium on Computers and Automata*, Vol.21 of Microwave Research Institute Symposia Series, Polytechnic Institute of Brooklyn Press (1971).

[34] L.M.G. Feijs and H.B.M. Jonkers. *Formal Specification and Design*, Cambridge University Press, Cambridge Tracts in Theoretical Computer Science 35 (1992).

[35] L.M.G. Feijs, H.B.M. Jonkers and C.A. Middelburg. *Notations for Software Design*, Springer Verlag, FACIT Series (1994).

[36] D. Gabbay and F. Guenthner (eds). *Handbook of Philosophical Logic*, volume II, Reidel Publishing Company, Dordrecht (1984).

[37] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, Addison-Wesley, Reading MA (1997).

[38] D. Gregg, A. Beatty, K. Casey, B. Davis, and A. Nisbet. The case for virtual register machines. In [31], 319–338 (2005).

[39] C.A Gunter and D.S. Scott. Semantic domains. In [65], 633–674 (1990).

[40] Y. Gurevich. Evolving algebras 1993: Lipari guide. In [18], 9–36 (1995).

[41] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach, *ACM SIGPLAN Notices*, 35(3), 39–48 (2000).

[42] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12, 576–580 (1969).

[43] H.B.M. Jonkers. An Introduction to COLD-K. In [68], 139–205 (1989).

[44] G. Kahn. Natural semantics. In [19], 22–39 (1987).

[45] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, Addison-Wesley, Reading MA (1997).

[46] S. Mauw and M.A. Reniers. A process algebra for Interworkings. In [15], 1269–1328 (2001).

[47] J. McCarthy. Towards a mathematical science of computation. In [54], 21–28 (1962).

[48] R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[49] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Comput. Sci. 92, Springer (1980).

[50] E. Moggi. Notions of computation and monads. *Information and Control*, 93, 55–92 (1991).

[51] P.D. Mosses. Mathematical semantics of Algol60. Technical Report Technical Monograph PRG-12, Programming Research Group, University of Oxford (1974).

[52] P.D. Mosses. The varieties of programming language semantics and their uses. In [17], 165–190 (2001).

[53] S. Muchnick and U. Pleban. A semantic comparison of LISP en SCHEME. In *Proceedings of the 1980 ACM Symposium on LISP and Functional Programming*, 56–64 (1980).

[54] C.M. Popplewell (ed). *Information Processing 1962, Proceedings of the IFIP Congress 62*, North-Holland (1962).

[55] A. Porto and G.-C. Roman (eds). *Proceedings of the Fourth International Conference on Coordination Models and Languages (COORDINATION 2000)*, Lecture Notes in Comput. Sci. 1906, Springer (2000).

[56] D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In [33], 19–46 (1971).

[57] I.A. Severoni. Operational semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58, 3–25 (2004).

[58] Y. Shi, D. Gregg, A. Beatty, and M.A. Ertl. Virtual machine showdown: stack versus registers. In [66], 153–163 (2005).

[59] P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak (eds). *Advances in Grid Computing - EGC 2005*, LNCS 3470, Springer-Verlag (2005).

[60] M.B. Smyth. Power domains. *Journal of Computer and System Sciences* , 16(1), 23–36 (1978).

[61] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine. Definition, Verification, Validation.* Springer (2001).

[62] R.D. Tennent. A denotational description of the programming language Pascal. Technical Report 77-47, Dept. of Computing and Inforamtion Sciences, Queen's University, Kingston, Ontario, Canada (1977).

[63] F.W. Vaandrager. Process algebra semantics of POOL. In [4], 173–236 (1990).

[64] R.J. van Glabbeek. The linear time—branching time spectrum I. The semantics of concrete, sequential processes. In [15], 3–100 (2001).

[65] J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin (eds). *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, Elsevier Science Publishers, Amsterdam; and MIT Press (1990).

[66] *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments.* Chicago, IL, USA. ACM Press, USA (2005).

[67] M. Wirsing (ed). *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference*, Gl. Avernaes, 1986, North-Holland (1987).

[68] M. Wirsing and J.A. Bergstra (eds). *Algebraic Methods: Theory, Tools and Applications*, Lecture Notes in Comput. Sci. 394, Springer (1989).

[69] A. Yonezawa and S. Matsuoka (eds.) *3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION 2001)*, Lecture Notes in Comput. Sci. 2192, Springer (2001).

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0602] J.A. Bergstra and A. Ponse, *Program Algebra with Repeat Instruction*, Programming Research Group - University of Amsterdam, 2006.

[PRG0601] J.A. Bergstra and A. Ponse, *Interface Groups for Analytic Execution Architectures*, Programming Research Group - University of Amsterdam, 2006.

[PRG0505] B. Diertens, *Software (Re-)Engineering with PSF*, Programming Research Group - University of Amsterdam, 2005.

[PRG0504] P.H. Rodenburg, *Piecewise Initial Algebra Semantics*, Programming Research Group - University of Amsterdam, 2005.

[PRG0503] T.D. Vu, *Metric Denotational Semantics for BPPA*, Programming Research Group - University of Amsterdam, 2005.

[PRG0502] J.A. Bergstra, I. Bethke, and A. Ponse, *Decision Problems for Pushdown Threads*, Programming Research Group - University of Amsterdam, 2005.

[PRG0501] J.A. Bergstra and A. Ponse, *A Bypass of Cohen's Impossibility Result*, Programming Research Group - University of Amsterdam, 2005.

[PRG0405] J.A. Bergstra and I. Bethke, *An Upper Bound for the Equational Specification of Finite State Services*, Programming Research Group - University of Amsterdam, 2004.

[PRG0404] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Strategic Interleaving*, Programming Research Group - University of Amsterdam, 2004.

[PRG0403] B. Diertens, *A Compiler-projection from PGLEc.MSPio to Parrot*, Programming Research Group - University of Amsterdam, 2004.

[PRG0402] J.A. Bergstra and I. Bethke, *Linear Projective Program Syntax*, Programming Research Group - University of Amsterdam, 2004.

[PRG0401] B. Diertens, *Molecular Scripting Primitives*, Programming Research Group - University of Amsterdam, 2004.

[PRG0302] B. Diertens, *A Toolset for PGA*, Programming Research Group - University of Amsterdam, 2003.

[PRG0301] J.A. Bergstra and P. Walters, *Projection Semantics for Multi-File Programs*, Programming Research Group - University of Amsterdam, 2003.

[PRG0201] I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences*, Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

www.science.uva.nl/research/prog/