TLP **2** (4 & 5): 425–460, July & September 2002. © 2002 Cambridge University Press
DOI: 10.1017/S1471068402001448 Printed in the United Kingdom

A refinement calculus for logic programs

IAN HAYES, ROBERT COLVIN, DAVID HEMER, PAUL STROOPER
School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

RAY NICKSON
School of Mathematical and Computing Sciences,
Victoria University of Wellington, New Zealand

Abstract

Existing refinement calculi provide frameworks for the stepwise development of imperative programs from specifications. This paper presents a refinement calculus for deriving logic programs. The calculus contains a wide-spectrum logic programming language, including executable constructs such as sequential conjunction, disjunction, and existential quantification, as well as specification constructs such as general predicates, assumptions and universal quantification. A declarative semantics is defined for this wide-spectrum language based on *executions*. Executions are partial functions from states to states, where a state is represented as a set of bindings. The semantics is used to define the meaning of programs and specifications, including parameters and recursion. To complete the calculus, a notion of correctness-preserving refinement over programs in the wide-spectrum language is defined and refinement laws for developing programs are introduced. The refinement calculus is illustrated using example derivations and prototype tool support is discussed.

KEYWORDS: refinement calculus, logic programming

1 Introduction

Our goal is to provide a method for the systematic development of logic programs from specifications. We follow a refinement calculus approach (Back, 1980; Morgan & Robinson, 1987; Morris, 1987; Morgan, 1994), which provides a framework for the stepwise development of imperative programs from specifications. It makes use of a wide-spectrum language that includes both specification and programming language constructs. This allows a specification to be refined, step by step, to program code within a single language. The refinement steps are performed using refinement laws that have been proven correct within the framework. Once they are proven correct, they can be safely applied in any refinement, although in some cases their application involves proof obligations that must be discharged. The *programs* produced during the intermediate steps of the refinement process may contain specification constructs as components, and hence may not be *code* suitable for execution. The refinement is completed when the program contains only executable constructs.

We define a refinement calculus for logic programming. The calculus contains a wide-spectrum logic programming language, including executable conjunction, disjunction, and existential quantification, as well as specification constructs such as general predicates, assumptions and universal quantification, which are not in general executable. General predicates allow the effect of a program to be specified via properties expressed as logical formulae. Assumptions represent information about the context in which a program fragment will execute. An implementation is obliged to produce the specified result only if its assumptions are satisfied by the context. The language also supports parametrised procedures and recursion.

We define a declarative semantics for the wide-spectrum language in terms of *executions*, which are partial functions from initial to final states. A state in turn is represented as a set of bindings, where each binding is a mapping from variables to values. As is traditionally the case with logic programs, we consider only executions where the set of bindings in the final state are a subset of the bindings in the initial state. To complete the calculus, we define a notion of correctness-preserving refinement over programs in the wide-spectrum language and a set of refinement laws. The declarative nature of the semantics means that we do not distinguish a program that produces an answer once from a program that produces the same answer multiple times, nor do we consider the order in which answers are produced. Our semantics makes use of a least fixed point semantics that equates non-termination with the least defined program, **abort**.

Section 2 of this paper presents related work. Section 3 summarizes the wide-spectrum logic programming language. Section 4 gives the basic definitions necessary for our formal semantics. Section 5 presents the semantics of the base language in terms of executions. Section 6 defines our notion of refinement, and section 7 gives the machinery for dealing with procedures and parameters. Section 8 discusses the semantics of recursion, and section 9 discusses refinement laws and presents a small example. Section 10 presents an extended example, which is the refinement of a program for the N-queens problem. Section 11 discusses `Marvin`, a prototype tool that supports the refinement calculus.

Our semantics is described using the mathematical notation of the Z specification language (Spivey, 1992), except that we write sequences within square brackets to be consistent with logic programming notation. No familiarity with Z is assumed. A number of properties of the semantics are presented here without proof; the proofs of these properties are presented in Hayes *et al.* (2000).

2 Related work

Traditionally, the refinement calculus has been used to develop imperative programs from specifications (Back, 1980; Morgan & Robinson, 1987; Morris, 1987; Morgan, 1994). The increase in expressive power of logic programming languages, when compared with imperative languages, leads to a reduced conceptual gap between a problem and its solution, which means that fewer development steps are required during refinement. An additional advantage of logic programming languages over procedural languages is their simpler, cleaner semantics, which leads to simpler proofs of the refinement steps. Finally, the higher expressive level of logic programming languages means that the individual refinement steps typically achieve more.

There have been previous proposals for developing a refinement calculus for declarative languages. A refinement calculus for functional programming languages is discussed by Ward (1994). Kok (1990) has applied the refinement calculus to logic programming languages, but his approach is quite different to ours. Rather than defining a wide-spectrum logic programming language, Kok embeds logic programs into a more traditional refinement calculus framework and notation.

There have been several proposals for the constructive development of logic programs, for example in Jacquet (1993). Much of this work has focused on program transformations or equivalence transformations from a first-order logic specification (Clark, 1978; Hogger, 1981). Read & Kazmierczak (1991) propose a stepwise development of modular logic programs from first-order specifications, based on three refinement steps that are much coarser than the refinement steps proposed in this paper. This leaves most of the work to be done in discharging the proof obligations for the refinement steps, for which they provide little guidance. Another approach to constructing logic programs is through *schemata* (Marakakis, 1997). A logic program is designed through the application of common algorithmic structures. The designer chooses which program structure is most suitable to a task based on the data types in question. As such, the focus of this method is to aid the design of large programs. The refinement steps and corresponding verification proofs are therefore much larger. Deductive logic program synthesis (Deville & Lau, 1994) is probably the most similar to the refinement calculus approach. In deductive synthesis, a specification is successively transformed using synthesis laws proven in an underlying framework (typically first-order logic).

Two key aspects that make the refinement calculus approach different from these other proposals are the smaller refinement steps and the use of assumptions. Refinement steps are performed by applying individual refinement laws that have been proven correct within the framework; these laws have a much smaller granularity than the refinement steps in the other approaches. Applying these refinement laws may introduce proof obligations that must be discharged, however discharging these proof obligations is usually straightforward, and can often be handled automatically with suitable tool support. The use of assumptions allows refinements to be proved correct with respect to the context in which they appear. To our knowledge, such *refinements in context* are novel in the logic programming community.

Deville (1990) introduces a systematic program development method for Prolog that incorporates assumptions and types similar to ours. The main difference is that Deville's approach to program development is mostly informal, whereas our approach is fully formal. A second distinction is that Deville's approach concentrates on the development of individual procedures. By using a wide-spectrum language, our approach blurs the distinction between a logic description and a logic program. For example, general predicates may appear anywhere within a program, and the refinement rules allow them to be transformed within that context. Similarly, programming language constructs may be used and transformed at any point.

The motivation for the work by Hoare (2000) is to come up with unifying theories for programming, which is quite different from the motivation for our work. However, the logic programming constructs he considers and the semantics he uses are both very similar to those we use.

$\langle P \rangle$	-	specification
$\{A\}$	-	assumption
$(c_1 \vee c_2)$	-	disjunction
$(c_1 \wedge c_2)$	-	parallel conjunction
(c_1, c_2)	-	sequential conjunction
$(\exists V \bullet c)$	-	existential quantification
$(\forall V \bullet c)$	-	universal quantification
$pc(t)$	-	procedure call

Fig. 1. Summary of wide-spectrum language commands.

3 Wide-spectrum language

This section presents the wide-spectrum logic programming language (Hayes *et al.*, 1997), which combines both logic programming language and specification language constructs. It allows constructs that may not be executable, similar to Back's (1980) inclusion of specification constructs in Dijkstra's imperative language. This has the benefit of allowing gradual refinement without the need for notational changes during the refinement process.

The constructs in the language are specifications (also called general predicates), assumptions, propositional operators, quantifiers, and procedure calls. A summary of the language is shown in figure 1.

Specifications: a specification $\langle P \rangle$, where P is a predicate, represents a set of instantiations of the free variables of the program that satisfy P. For example, the command $\langle X = 5 \vee X = 6 \rangle$ represents the set of instantiations $\{5, 6\}$ for X, and the command $\langle V = \mathit{fact}(U) \rangle$ specifies the set of pairs of V and U such that V equals the factorial of U. The specification **fail** is defined by:

$$\textbf{fail} == \langle \mathit{false} \rangle$$

(We use the notation "==" to indicate a definition.) The program **fail** always computes an empty answer set, like Prolog's `fail`. The null command, **skip**, is defined by

$$\textbf{skip} == \langle \mathit{true} \rangle$$

Assumptions: an assumption $\{A\}$, where A is a predicate, expresses a requirement on the context for a program fragment. For example, we may augment our specification of the factorial example by an assumption that U has been instantiated to be an element of the natural numbers before the factorial is evaluated: $\{U \in \mathbb{N}\}, \langle V = \mathit{fact}(U) \rangle$. If assumptions about the context are formally expressed, implementations may take advantage of the assumptions, but need not establish (or indeed check) them. If these assumptions do not hold, the program fragment may abort. Aborting includes program behaviour such as nontermination and abnormal termination due to exceptions like division by zero, as well as termination with arbitrary results. We define the (worst possible) program **abort** by

$$\textbf{abort} == \{\mathit{false}\}$$

Note that **abort** is quite different from the program **fail**, which never aborts, but has an empty solution set.

Propositional operators: there are two forms of conjunction: a sequential form (c_1, c_2) where command c_1 is evaluated before c_2; and a parallel version $(c_1 \wedge c_2)$ where c_1 and c_2 are evaluated independently and the intersection of their respective results is formed. The disjunction of two programs $(c_1 \vee c_2)$ computes the union of the results of the two programs. We overload the symbols "$\wedge$" and "$\vee$" as both operators on predicates and operators on commands. Because, for example, the meanings of $\langle P \wedge Q \rangle$ and $\langle P \rangle \wedge \langle Q \rangle$ are identical, this does not usually cause confusion.

The following three programs illustrate the behaviour of sequential and parallel conjunction, and show the difference between abortion and failure:

$$P1 == \{X \neq 0\}, \langle Y = 1/X \rangle$$
$$P2 == \langle X \neq 0 \rangle, \langle Y = 1/X \rangle$$
$$P3 == \langle X \neq 0 \rangle \wedge \langle Y = 1/X \rangle$$

If each of the three programs is executed from a state where $X = 0$, $P1$ will abort, while $P2$ will fail, producing an empty answer set. The behaviour of $P3$ is also equivalent to abort, because the predicate $Y = 1/0$ is not defined.

Quantifiers: the existential quantifier $(\exists V \bullet c)$ generalises disjunction, computing the union of the results of command c for all possible values of V. For example, the following may be part of the development of a factorial procedure:

$$\exists U1, V1 \bullet \langle U1 = U - 1 \wedge V1 = fact(U1) \wedge V = V1 * U \rangle$$

Similarly, the universal quantifier $(\forall V \bullet c)$ computes the intersection of the results of command c for all possible values of V.

Parametrised commands and procedures: we separate the definition of parametrised commands from procedures. A parametrised command has the form

$$V :\!- c$$

where V is a variable, and c is a wide-spectrum command. This notation is similar to the lambda calculus notation, $(\lambda V \bullet c)$, and allows anonymous procedures to be used.

A procedure definition has the form

$$id \mathrel{\widehat=} pc$$

where id is an identifier representing the name of the procedure, and pc is a parametrised command as defined above. For example,

$$id \mathrel{\widehat=} V :\!- c$$

defines a procedure called id with a formal parameter V and body c. A parametrised command or a procedure identifier that is defined as a parametrised command may be applied to an actual parameter term. For example, a call on the procedure id is of the form $id(t)$, where t is a term: the actual parameter.

Our semantics only deals with procedures with a single parameter. However, no generality is lost because multiple parameters may be encoded using compound terms. For example, a procedure to calculate factorials may be specified by

$$factorial \mathrel{\widehat{=}} (U, V) \mathrel{:\!-} \{U \in \mathbb{N}\}, \langle V = fact(U)\rangle$$

which is an abbreviation for the following definition with a single parameter W:

$$factorial \mathrel{\widehat{=}} W \mathrel{:\!-} (\exists\, U, V \bullet \langle W = (U, V)\rangle, \{U \in \mathbb{N}\}, \langle V = fact(U)\rangle)$$

Commands: we define *Cmd* to be the set of commands in our language, built up from the constructs shown in figure 1.

4 Semantic domains

We begin our formal treatment of the semantics by defining the domains over which our semantics of programs are given.

4.1 Variables, values and functors

We have fixed domains of *variables* (*Var*), *values* (*Val*), and *functors* (*Fun*). Elements of *Var* represent the variables that can occur in programs. This includes variables for which a program's answers give values, variables that are bound by universal and existential quantifiers, and variables used as formal parameters. Values are the objects in the universe of discourse, denoted by ground terms. Functors represent the function symbols in our language that are used to construct compound terms. The arity of a functor is given by arity, which is a total function from functors to natural numbers (notationally, $\mathsf{arity} : Fun \to \mathbb{N}$). Atoms are functors of arity zero.

If we restrict our interpretation of ground terms to the Herbrand interpretation, as is typical in logic programming, *Val* is structured into atoms and compound terms. But we allow more structure than this; *Val* may be structured according to any algebra, taking into account whatever kind of term equality is appropriate for the application under consideration. For the purposes of this paper, all we assume is that there is a function apply, which models the application of a function to a sequence of values of length equal to the arity of the function, resulting in a value. We present the definition of apply as a Z axiomatic definition; the signature of apply is given above the line, and the definition in the form of a predicate is given below the line. In this case the signature states that apply is a (total) function that takes a functor, and returns a *partial* function ($\rightarrow\!\!\!\!\!+$) mapping sequences of values to values.

$$\begin{array}{|l}
\mathsf{apply} : Fun \to (\mathrm{seq}\, Val \rightarrow\!\!\!\!\!+ Val) \\
\hline
\forall f : Fun \bullet \mathrm{dom}(\mathsf{apply}\ f) \subseteq \{s : \mathrm{seq}\, Val \mid \#s = \mathsf{arity}\ f\}
\end{array}$$

With this more general interpretation, given a functor f of arity n, "apply f" may be undefined for some sequences of values of length n. We write *def* E (or *def* P) for the predicate that is true precisely when the expression E (or predicate P) is well-defined: that is, when all function applications occurring within it are well-defined.

For the Herbrand interpretation, or indeed for any other interpretation in which all functions are total, $def\ E = def\ P = true$ for all expressions E and predicates P.

4.2 Bindings, states and predicates

A *binding* is a total function, mapping every variable to a value.

$$Bnd == Var \rightarrow Val$$

Each binding corresponds to a single ground answer to a Prolog-like query. The mechanism for representing "unbound" variables is described below.

A *state* is a set of bindings.

$$State == \mathbb{P}\, Bnd$$

A state corresponds to our usual notion of a predicate with some free variables, which is true or false once provided with a binding for those variables, i.e., for a binding in the state. Given a predicate P, we write $\overline{P}$ to denote the set of bindings satisfying P. For example:

$$\begin{array}{rcl} \overline{false} & = & \varnothing \\ \overline{true} & = & Bnd \\ \overline{X = 3} & = & \{b : Bnd \mid b\ X = 3\} \\ \overline{X < Y} & = & \{b : Bnd \mid b\ X < b\ Y\} \end{array}$$

A completely unbound variable is represented by a possibly infinite state that has one binding to each element of Val. For example, if we suppose that Var contains just the variables $\{X, Y, Z\}$, Val is the set $\{0, 1\}$, and f has arity one and is total, the set of solutions to the equation $Y = f(X)$ is represented by the following set of bindings.

$$\begin{array}{l} \{\{X \mapsto 0, Y \mapsto \mathsf{apply}\ f\ [0], Z \mapsto 0\}, \\ \ \{X \mapsto 0, Y \mapsto \mathsf{apply}\ f\ [0], Z \mapsto 1\}, \\ \ \{X \mapsto 1, Y \mapsto \mathsf{apply}\ f\ [1], Z \mapsto 0\}, \\ \ \{X \mapsto 1, Y \mapsto \mathsf{apply}\ f\ [1], Z \mapsto 1\}\} \end{array}$$

The expression $X \mapsto x$ constructs a pair of its operands; it is equivalent to (X, x).

We do not concern ourselves in this semantics with finite or infinite representation of infinite states. Our model allows a successful execution that terminates in an infinite state. Some infinite states may correspond to Prolog answer sets that can be finitely enumerated, and others may not.

4.3 Terms

A *Term* is a variable, or a functor with a (possibly empty) finite sequence of terms (the arguments). We model a term via the following data type constructor.

$$Term ::= varT\langle\langle Var\rangle\rangle \mid funT\langle\langle Fun \times \mathrm{seq}\ Term\rangle\rangle$$

For any variable V, $varT(V)$ is a term, and if f is a functor and ts is a sequence of terms, then $funT(f, ts)$ is a term.

A term may have a value when evaluated relative to some binding, or it may be undefined if the term involves the incorrect application of a function. We define a partial function eval that evaluates a term relative to a binding.

$$\begin{array}{|l}
\mathsf{eval} : \mathit{Term} \rightarrow (\mathit{Bnd} \nrightarrow \mathit{Val}) \\
\hline
\mathsf{eval}(varT(V)) = (\lambda b : \mathit{Bnd} \bullet b\ V) \\
\mathsf{eval}(funT(f, ts)) = \{b : \mathit{Bnd};\ vs : \mathrm{seq}\,\mathit{Val} \mid \\
\qquad (\forall t : \mathrm{ran}\, ts \bullet b \in \mathrm{dom}(\mathsf{eval}\ t)) \wedge \\
\qquad vs = map(\lambda t : \mathit{Term} \bullet \mathsf{eval}\ t\ b)(ts) \wedge \\
\qquad vs \in \mathrm{dom}(\mathsf{apply}\ f) \\
\qquad \bullet\, b \mapsto \mathsf{apply}\ f\ vs\}
\end{array}$$

The notation $\{x : T \mid P \bullet E\}$ describes the set of values of the expression E, for each x of type T for which P holds (when P is *true* we may omit it, i.e. $\{x : T \bullet E\}$). Thus, $\{x : T \mid P \bullet x \mapsto E\}$ is the function (set of pairs) whose domain is the set of $x : T$ satisfying predicate P, and which maps each such x to the corresponding value of expression E. In the definition of eval, to evaluate a compound term $funT(f, ts)$ with respect to a binding b, all of the terms in the sequence ts must be able to be evaluated with respect to b, the resultant sequence of values vs is defined by mapping a function that evaluates a single term with respect to b over the sequence ts, and vs must be in the domain of apply f. For a term t, defined t is the set of states for which t is defined for every binding in the state.

$$\begin{array}{|l}
\mathsf{defined} : \mathit{Term} \rightarrow \mathbb{P}\,\mathit{State} \\
\hline
\mathsf{defined}\ t = \{s : \mathit{State} \mid s \subseteq \mathrm{dom}(\mathsf{eval}\ t)\}
\end{array}$$

For a variable V, term t, and state s, assign $V\ t\ s$ is the same as state s, except that in each binding within s the value of V is replaced by the value of t in that binding.

$$\begin{array}{|l}
\mathsf{assign} : \mathit{Var} \rightarrow \mathit{Term} \rightarrow \mathit{State} \nrightarrow \mathit{State} \\
\hline
\mathsf{assign}\ V\ t = (\lambda s : \mathsf{defined}\ t \bullet \{b : s \bullet b \oplus \{V \mapsto \mathsf{eval}\ t\ b\}\})
\end{array}$$

In the definition, $f \oplus g$ stands for function f overridden by function g. The expression $f \oplus g$ is the same as function f, but with everything in the domain of g mapped to the same objects that g maps them to (outside of the domain of g, $f \oplus g$ behaves like f).

For some term t, free t is the set of free variables in t:

$$\begin{array}{|l}
\mathsf{free} : \mathit{Term} \rightarrow \mathbb{P}\,\mathit{Var} \\
\hline
\mathsf{free}(varT(V)) = \{V\} \\
\mathsf{free}(funT(f, ts)) = \bigcup\{t : \mathrm{ran}\, ts \bullet \mathsf{free}\, t\}
\end{array}$$

Similarly, for a command c, free c defines the set of free variables in c. Since we do not formally define the predicates allowed in specifications and assumptions here, we do not define $\mathsf{free}\, c$ formally.

5 Program execution

5.1 Executions

We define the semantics of our language in terms of *executions*, which are mappings from initial states to final states. The mapping is partial because the program is only well-defined for those initial states that guarantee satisfaction of all the program's assumptions. Executions satisfy three healthiness properties below. These properties restrict executions to model pure logic programs.

1. If a command is guaranteed to terminate from an initial state $\overline{P}$ whose bindings all satisfy some predicate P, it must also guarantee to terminate from all those initial states $\overline{P'}$, where $P' \Rrightarrow P$. We thus require that any subset s' of a set s that is in the domain of an execution e, is also in the domain of e.

   $$\forall\, s : \mathrm{dom}\, e \bullet (\forall\, s' : \mathbb{P}\, s \bullet s' \in \mathrm{dom}\, e)$$

   In addition, if a command is guaranteed to terminate from initial state $\overline{P}$ and it is also guaranteed to terminate from initial state $\overline{Q}$, it must terminate from an initial state $\overline{P \vee Q}$. Thus, if all sets in a set of states ss are in the domain of e, then their union is also in the domain of e.

   $$\forall\, ss : \mathbb{P}(\mathrm{dom}\, e) \bullet \bigcup ss \in \mathrm{dom}\, e$$

   As we show (Hayes *et al.*, 2000), these together are equivalent to the fact that the domain of e is the powerset of the set of all bindings, b, such that the singleton set $\{b\}$ is in the domain of e.

   $$\mathrm{dom}\, e = \mathbb{P}\{b : \mathit{Bnd} \mid \{b\} \in \mathrm{dom}\, e\}$$

2. Because of the constraining nature of logic programs (command execution cannot decrease "groundedness") for any state s in the domain of an execution e, the set of bindings in the output state $e(s)$ must be a subset of s.

   $$\forall\, s : \mathrm{dom}\, e \bullet e(s) \subseteq s$$

   Consider a state with variables X and Y which can take values in the set $\{0, 1\}$ and the execution e that represents the program $\langle X = Y \rangle$. Intuitively the set of initial states for a specification $\langle P \rangle$ consists of those states for which P is defined. In this example, the domain of e consists of all subsets of

   $$\begin{aligned} S \;\; = \;\; & \{\{X \mapsto 0, Y \mapsto 0\}, \{X \mapsto 0, Y \mapsto 1\}, \\ & \{X \mapsto 1, Y \mapsto 0\}, \{X \mapsto 1, Y \mapsto 1\}\} \end{aligned}$$

   Given an initial state, the effect of executing $\langle X = Y \rangle$ is to filter out those bindings b for which $X = Y$ is not true. In particular $e(S) = \{\{X \mapsto 0, Y \mapsto 0\}, \{X \mapsto 1, Y \mapsto 1\}\} \subseteq S$.

3. For a set of bindings s, the output set of bindings consists of all those bindings b, such that the singleton set $\{b\}$ is preserved by e.

$$\forall s : \operatorname{dom} e \bullet e(s) = \{b : s \mid e(\{b\}) = \{b\}\}$$

For each of the singleton states in the example state S above,

$$\begin{aligned} e(\{\{X \mapsto 0, Y \mapsto 0\}\}) &= \{\{X \mapsto 0, Y \mapsto 0\}\} \\ e(\{\{X \mapsto 0, Y \mapsto 1\}\}) &= \{\} \\ e(\{\{X \mapsto 1, Y \mapsto 0\}\}) &= \{\} \\ e(\{\{X \mapsto 1, Y \mapsto 1\}\}) &= \{\{X \mapsto 1, Y \mapsto 1\}\} \end{aligned}$$

Thus $e(S)$ consists of the bindings which are individually preserved by e.

We thus define:

$$\begin{aligned} &\mathit{Exec} == \{e : \mathit{State} \nrightarrow \mathit{State} \mid \\ &\qquad \operatorname{dom} e = \mathbb{P}\{b : \mathit{Bnd} \mid \{b\} \in \operatorname{dom} e\} \wedge && (1) \\ &\qquad (\forall s : \operatorname{dom} e \bullet e(s) \subseteq s) \wedge && (2) \\ &\qquad (\forall s : \operatorname{dom} e \bullet e(s) = \{b : s \mid e(\{b\}) = \{b\}\})\} && (3) \end{aligned}$$

Note that Property (1) implies that $\varnothing \in \operatorname{dom} e$ for all executions e. Also, from Property (2), $e(\{b\})$ is either $\{b\}$ or $\varnothing$. In Hayes *et al.* (2000), Property (3) is shown to be equivalent to either of the following two properties.

$$\forall s : \operatorname{dom} e \bullet e(s) = \bigcup\{b : s \bullet e(\{b\})\} \qquad (4)$$
$$\forall s : \operatorname{dom} e \bullet \forall s' : \mathbb{P} s \bullet e(s') = e(s) \cap s' \qquad (5)$$

Property (4) shows that an execution e can be determined by considering the effect of the execution on each singleton binding, and then forming the union of the results. Property (5) shows that the result of executing a command in a subset s' of some state s is consistent with executing the command in state s and restricting the results to those in s'. This property is similar to the property quoted by Hoare (2000) and attributed to He Jifeng, as one that characterises a pure logic program. For example, Prolog's `var` does not satisfy the property.

5.2 Semantic function for commands

We define the semantics of the commands in our language via a function that takes a command and returns the corresponding execution.

$$\mathsf{exec} : \mathit{Cmd} \rightarrow \mathit{Exec}$$

The semantics of the basic commands (excluding procedure calls, which are treated in section 7.5) is shown in figure 2. In the remainder of this section, we explain the definitions. In Hayes *et al.* (2000), we show that all executions constructed using the definitions satisfy the healthiness properties of executions.

In section 7, where we discuss procedures and parameters, we extend the definition of exec with an *environment*, which maps procedure identifiers to their corresponding

$$\begin{aligned}
&\mathsf{exec}(\langle P\rangle) = (\lambda\, s : \mathbb{P}(\overline{def\ \ P}) \bullet s \cap \overline{P})\\
&\mathsf{exec}(\{A\}) = (\lambda\, s : \mathbb{P}(\overline{def\ \ A \wedge A}) \bullet s)\\
&\mathsf{exec}(\mathbf{fail}) = \mathsf{exec}(\langle \mathit{false}\rangle) = (\lambda\, s : \mathit{State} \bullet \varnothing)\\
&\mathsf{exec}(\mathbf{abort}) = \mathsf{exec}(\{\mathit{false}\}) = \{\varnothing \mapsto \varnothing\}\\
&\mathsf{exec}(\mathbf{skip}) = \mathsf{exec}(\langle \mathit{true}\rangle) = (\lambda\, s : \mathit{State} \bullet s)\\
&\mathsf{exec}(c_1 \vee c_2) = \mathsf{exec}\, c_1 \dot{\cup}\, \mathsf{exec}\, c_2\\
&\mathsf{exec}(c_1 \wedge c_2) = \mathsf{exec}\, c_1 \dot{\cap}\, \mathsf{exec}\, c_2\\
&\mathsf{exec}(c_1\, ,\, c_2) = \mathsf{exec}\, c_1 \,;\, \mathsf{exec}\, c_2\\
&\mathsf{exec}(\exists\, V \bullet c) = \mathsf{exists}\, V\, (\mathsf{exec}\, c)\\
&\mathsf{exec}(\forall\, V \bullet c) = \mathsf{forall}\, V\, (\mathsf{exec}\, c)
\end{aligned}$$

Fig. 2. Execution semantics of basic commands

(parametrised) executions. For simplicity, we first present the semantics of the basic commands ignoring the environment.

5.3 Specifications and assumptions

A specification $\langle P\rangle$ is defined for all states s such that P is defined for all bindings in s; the result of executing specification $\langle P\rangle$ consists of those bindings in s that satisfy P.

An assumption $\{A\}$ is defined for all states s such that A is defined and A holds for all bindings in s; the result of executing assumption $\{A\}$ has no effect (the set of bindings remains unchanged).

The definition for the special-case specification **fail** is the constant function that returns the empty state, no matter what the initial state is. Hence for any command c, including **abort**,

$$\mathbf{fail}\, ,\, c = \mathbf{fail}$$

because **fail** maps any state to the empty state.

The definition for the special-case assumption **abort** is the function mapping the empty state to the empty state. Hence for any command c,

$$\mathbf{abort}\, ,\, c = \mathbf{abort}$$

because the domain of **abort** contains only the empty state, which it maps to the empty state. Note that the empty state is preserved by any command, i.e. for any command c, $(\mathsf{exec}\ \ c)(\varnothing) = \varnothing$.

The definition of the special-case specification **skip** is the identity function on states. Hence for any command c,

$$c\, ,\, \mathbf{skip} = c = \mathbf{skip}\, ,\, c.$$

5.4 Propositional operators

Disjunction and parallel conjunction are defined as pointwise union and intersection of the corresponding executions.

$$\_ \Cap \_ : Exec \times Exec \rightarrow Exec$$
$$\_ \Cup \_ : Exec \times Exec \rightarrow Exec$$
$$(e_1 \Cap e_2) = (\lambda s : \operatorname{dom} e_1 \cap \operatorname{dom} e_2 \bullet (e_1\ s) \cap (e_2\ s))$$
$$(e_1 \Cup e_2) = (\lambda s : \operatorname{dom} e_1 \cap \operatorname{dom} e_2 \bullet (e_1\ s) \cup (e_2\ s))$$

For a conjunction $(c_1 \wedge c_2)$, if a state s is mapped to s' by $\mathsf{exec}\, c_1$ and s is mapped to s'' by $\mathsf{exec}\, c_2$, then $\mathsf{exec}(c_1 \wedge c_2)$ maps s to $s' \cap s''$. Disjunction is similar, but gives the union of the resulting states instead of intersection. The command **abort** is a zero of both disjunction and parallel conjunction, **skip** is the unit of parallel conjunction, and **fail** is the unit of disjunction.

Sequential conjunction (c_1 , c_2) is defined as function composition of the corresponding executions. The domain of the function $(\lambda x : T \mid P \bullet E)$ is those values of x in T for which P holds, and the range is the corresponding set of values for E.

$$\_ \fatsemi \_ : Exec \times Exec \rightarrow Exec$$
$$(e_1 \fatsemi e_2) = (\lambda s : \operatorname{dom} e_1 \mid e_1(s) \in \operatorname{dom} e_2 \bullet e_2(e_1(s)))$$

If $\mathsf{exec}\, c_1$ maps state s to s' and $\mathsf{exec}\, c_2$ maps s' to s'', then $\mathsf{exec}(c_1 , c_2)$ maps s to s''. If either s is not in the domain of $\mathsf{exec}\, c_1$ or s' is not in the domain of $\mathsf{exec}\, c_2$, then s is not in the domain of $\mathsf{exec}(c_1 , c_2)$. The command **skip** is the unit of sequential conjunction, and both **abort** and **fail** are left zeroes.

5.5 Quantifiers

For a variable V and a state s, we define the state "$\mathsf{unbind}\ V\ s$" as one whose bindings match those of s in every place except V, which is completely unconstrained.

$$\mathsf{unbind} : Var \rightarrow State \rightarrow State$$
$$\mathsf{unbind}\ V\ s = \{b : s;\ x : Val \bullet b \oplus \{V \mapsto x\}\}$$

Execution of an existentially quantified command $(\exists V \bullet c)$ from an initial state s is defined if executing c is defined in the state s', which is the same as s except that V is unbound. The resultant state after executing c consists of all those bindings b in s such that there is a value, x, for V such that execution of c retains the binding $b \oplus \{V \mapsto x\}$. We thus make the following definition of the existential quantifier for executions.

$$\mathsf{exists} : Var \rightarrow Exec \rightarrow Exec$$
$$\mathsf{exists}\ V\ e = (\lambda s : State \mid \mathsf{unbind}\ V\ s \in \operatorname{dom} e$$
$$\bullet \{b : s \mid (\exists x : Val \bullet e(\{b \oplus \{V \mapsto x\}\}) \neq \varnothing)\})$$

Universal quantification behaves in a similar fashion, except that for $(\mathsf{forall}\ V\ e)$ to retain a binding b, execution of e must retain $b \oplus \{V \mapsto x\}$ for all values x.

$$\begin{array}{|l}
\mathsf{forall} : Var \rightarrow Exec \rightarrow Exec \\
\hline
\mathsf{forall}\ V\ e = (\lambda s : State \mid \mathsf{unbind}\ V\ s \in \mathrm{dom}\, e \\
\qquad \bullet \{b : s \mid (\forall x : Val \bullet e(\{b \oplus \{V \mapsto x\}\}) \neq \varnothing)\})
\end{array}$$

6 Refinement

An execution e_1 is refined by an execution e_2 if and only if e_2 is defined wherever e_1 is and they agree on their outputs whenever both are defined. This is the usual "definedness" order on partial functions, as used, for example, by Manna (1974): it is simply defined by the subset relation of functions viewed as sets of pairs. We define refinement, $\sqsubseteq$, as a relation ($\leftrightarrow$) between *Exec*s.

$$\begin{array}{|l}
\_ \sqsubseteq \_ : Exec \leftrightarrow Exec \\
\hline
e_1 \sqsubseteq e_2 \Leftrightarrow e_1 \subseteq e_2
\end{array}$$

For the commands *Cmd* in our language, we define refinement in terms of refinement of the corresponding executions.

$$\begin{array}{|l}
\_ \sqsubseteq \_ : Cmd \leftrightarrow Cmd \\
\hline
c_1 \sqsubseteq c_2 \Leftrightarrow \mathsf{exec}\ c_1 \sqsubseteq \mathsf{exec}\ c_2
\end{array}$$

Finally, refinement equivalence ($\sqsubseteq\!\!\!\sqsupseteq$) is defined for *Cmd* and *Exec* as refinement in both directions.

$$\begin{array}{|l}
\_ \sqsubseteq\!\!\!\sqsupseteq \_ : Cmd \leftrightarrow Cmd \\
\_ \sqsubseteq\!\!\!\sqsupseteq \_ : Exec \leftrightarrow Exec \\
\hline
c_1 \sqsubseteq\!\!\!\sqsupseteq c_2 \Leftrightarrow c_1 \sqsubseteq c_2 \wedge c_2 \sqsubseteq c_1 \\
e_1 \sqsubseteq\!\!\!\sqsupseteq e_2 \Leftrightarrow e_1 \sqsubseteq e_2 \wedge e_2 \sqsubseteq e_1
\end{array}$$

Refinement is a preorder — a reflexive and transitive relation — because subset is a preorder on sets.

6.1 Lattice properties

The refinement relation forms a meet semi-lattice over *Exec*.

- "$\sqsubseteq$" is a partial order because "$\subseteq$" is a partial order on sets;
- meets exist: $e_1 \sqcap e_2 = e_1 \cap e_2$;
- there is a unique bottom element, corresponding to the command **abort** (recall that exec **abort** $= \{\varnothing \mapsto \varnothing\}$).

Note that joins do not exist in general, because $e_1 \cup e_2$ may not be a function; and there is no top element. For $e_1 \sqcup e_2$ to be defined, we require that $e_1 \cup e_2$ is a function and not simply a relation (i.e. e_1 returns the same state as e_2 for the states where both are defined). Even under that condition, the result is not simply $e_1 \cup e_2$, because that would violate condition (1) of executions. Instead, we define $e_1 \sqcup e_2$ by adding to $e_1 \cup e_2$ mappings for all states that consist of bindings for which either e_1 or e_2 is defined.

$$\begin{array}{|l}
\_ \sqcap \_ : Exec \times Exec \to Exec \\
\_ \sqcup \_ : Exec \times Exec \nrightarrow Exec \\
\hline
e_1 \sqcap e_2 = e_1 \cap e_2 \\
(e_1, e_2) \in \mathrm{dom}(\_ \sqcup \_) \Leftrightarrow (\forall s : \mathrm{dom}\, e_1 \cap \mathrm{dom}\, e_2 \bullet e_1(s) = e_2(s)) \\
(e_1, e_2) \in \mathrm{dom}(\_ \sqcup \_) \Rightarrow \\
\qquad e_1 \sqcup e_2 = (\lambda s : \mathbb{P}(\bigcup(\mathrm{dom}\, e_1 \cup \mathrm{dom}\, e_2)) \bullet \{b : s \mid (e_1 \cup e_2)(\{b\}) \neq \varnothing\})
\end{array}$$

We show (Hayes *et al*., 2000) that $\sqcap$ and $\sqcup$ (when the latter is well-defined) preserve the healthiness properties for executions.

If $e_1 \sqsubseteq e_2$ then their join $e_1 \sqcup e_2$ is defined (and is e_2). As a result, we can define joins for chains. We first define a chain of executions:

$$Chain == \{ec : \mathbb{N} \to Exec \mid (\forall i : \mathbb{N} \bullet ec(i) \sqsubseteq ec(i+1))\}$$

Note that we define every chain as an infinite sequence; a finite chain is simply modelled as an infinite chain in which the last element is repeated infinitely often. If ec is a chain, then $ec(i) \sqcup ec(i+1)$ exists for all i and is equal to $ec(i+1)$. We therefore define the join of a chain as follows.

$$\begin{array}{|l}
\bigsqcup : Chain \to Exec \\
\hline
\bigsqcup ec = \bigcup(\mathrm{ran}\, ec)
\end{array}$$

7 Procedures and parameters

To simplify the semantics, we treat procedures, parameters and recursion as separate, though related, concerns in a manner similar to Morgan (1988).

7.1 Parametrised commands

To deal with parameters, we introduce the notion of a *parametrised command*. Given a variable V and a command c, the expression $V :\!- c$ denotes the parametrised command ($PCmd$) that, when provided a term argument t, behaves like $c[t/V]$, i.e. the command c with every occurrence of V replaced by t. For $V :\!- c$ to be well-formed, we require that c has no free variables other than V, i.e., $\mathsf{free}(c) \subseteq \{V\}$; any other variables in c must be explicitly quantified.

The semantics of parametrised commands is given by parametrised executions, which are functions mapping actual parameter terms to executions.

$$PExec == Term \to Exec$$

7.2 Environments

To handle procedure definitions and recursion, we introduce a given set of *procedure identifiers* ($PIdent$) and an *environment*, which maps procedure identifiers to their corresponding procedure executions.

$$Env == PIdent \rightarrowtail PExec$$

Hence we change the definition of exec to add an environment parameter.

$$\mathsf{exec} : Env \rightarrow Cmd \rightarrow Exec$$

The definitions we have given in section 5 do not depend directly upon the environment. The only change required is to add the environment parameter to the calls on exec for subcomponents, e.g. for an environment ρ:

$$\mathsf{exec}(\rho)(c_1 \wedge c_2) = (\mathsf{exec}\ \ \rho\ c_2) \Cap (\mathsf{exec}\ \ \rho\ c_2)$$

7.3 Parametrised executions

For an actual parameter term t, the execution of a parametrised command V **:-** c is a function that is defined for all states s for which evaluation of t is defined for all bindings in s, and for which c is defined when V is bound to the value of t for each binding in s. We determine the result of executing the parametrised command by determining its result for each binding b in s. A binding b from a state s will be retained by the execution of the parametrised command applied to term t if the execution of c from a state consisting of $\{b\}$, but with the formal parameter V replaced with the value of term t in binding b, retains that binding. Recall that the result of executing a command on a singleton set is either that singleton set or the empty set.

$$\mathsf{pexec} : Env \rightarrow PCmd \rightarrow PExec$$

$$\begin{aligned}&\mathsf{pexec}(\rho)(V \mathrel{:-} c) = (\lambda t : Term \bullet\\ &\quad (\lambda s : \mathsf{defined}\ \ t \mid \mathsf{assign}\ \ V\ \ t\ \ s \in \mathrm{dom}(\mathsf{exec}\ \ \rho\ c) \bullet\\ &\qquad \{b : s \mid (\mathsf{exec}\ \ \rho\ c)(\mathsf{assign}\ \ V\ \ t\ \{b\}) \neq \varnothing\}))\end{aligned}$$

This definition only applies to non-recursive parametrised commands. The definition of pexec for recursion is presented in section 8.

7.4 Refinement

We define refinement between parametrised executions p_1 and p_2 by requiring refinement for every possible value of the parameter. We also define refinement equivalence between parametrised executions in the obvious way.

$$\begin{aligned}&\_ \sqsubseteq \_ : PExec \leftrightarrow PExec\\ &\_ \sqsubseteq\!\!\sqsupseteq \_ : PExec \leftrightarrow PExec\end{aligned}$$

$$\begin{aligned}&(p_1 \sqsubseteq p_2) \Leftrightarrow (\forall t : Term \bullet (p_1\ \ t) \sqsubseteq (p_2\ \ t))\\ &(p_1 \sqsubseteq\!\!\sqsupseteq p_2) \Leftrightarrow (\forall t : Term \bullet (p_1\ \ t) \sqsubseteq\!\!\sqsupseteq (p_2\ \ t))\end{aligned}$$

Refinement between parametrised commands pc_1 and pc_2 is defined, as expected, in terms of refinement between the corresponding parametrised executions. Because the meaning of a parametrised command depends on the environment, this becomes a parameter to the refinement relation; this parameter is written as a subscript ρ.

$$\begin{array}{|l}
\_ \sqsubseteq_\rho \_ : Env \rightarrow (PCmd \leftrightarrow PCmd) \\
\_ \sqsubseteq\!\!\sqsupseteq_\rho \_ : Env \rightarrow (PCmd \leftrightarrow PCmd) \\
\hline
(pc_1 \sqsubseteq_\rho pc_2) \Leftrightarrow (\mathsf{pexec}\ \ \rho\ pc_1 \sqsubseteq \mathsf{pexec}\ \ \rho\ pc_2) \\
(pc_1 \sqsubseteq\!\!\sqsupseteq_\rho pc_2) \Leftrightarrow (\mathsf{pexec}\ \ \rho\ pc_1 \sqsubseteq\!\!\sqsupseteq \mathsf{pexec}\ \ \rho\ pc_2)
\end{array}$$

When the environment ρ is clear from the context it may be elided.

7.5 Procedure call

A parametrised command may be directly applied to a term; the result is a command, the semantics of which is defined as follows:

$$\mathsf{exec}\ \ \rho\ ((v :\!- c)(t)) = \mathsf{pexec}\ \ \rho\ (v :\!- c)\ t$$

The syntax of a procedure call is $id(t)$, where t is a term and id is a procedure identifier. The parametrised command which is the definition of id in the environment is applied to t. If id is not defined in the environment, the result of a call on id is **abort**.

$$\mathsf{exec}\ \ \rho\ id(t) = \mathbf{if}\ id \in \operatorname{dom} \rho\ \mathbf{then}\ (\rho\ id\ t)\ \mathbf{else}\ (\mathsf{exec}\ \ \rho\ \mathbf{abort})$$

7.6 Lattice properties

The lattice properties of *Exec* can be lifted to *PExec*.

$$\begin{array}{|l}
\_ \dot{\sqcap} \_ : PExec \times PExec \rightarrow PExec \\
\_ \dot{\sqcup} \_ : PExec \times PExec \nrightarrow PExec \\
\hline
p_1 \dot{\sqcap} p_2 = (\lambda t : Term \bullet p_1\ t \sqcap p_2\ t) \\
(p_1, p_2) \in \operatorname{dom}(\_ \dot{\sqcup} \_) \Leftrightarrow (\forall t : Term \bullet (p_1\ t, p_2\ t) \in \operatorname{dom}(\_ \sqcup \_)) \\
(p_1, p_2) \in \operatorname{dom}(\_ \dot{\sqcup} \_) \Rightarrow p_1 \dot{\sqcup} p_2 = (\lambda t : Term \bullet p_1\ t \sqcup p_2\ t)
\end{array}$$

$$PChain == \{p : \mathbb{N} \rightarrow PExec \mid (\forall i : \mathbb{N} \bullet p(i) \sqsubseteq p(i+1))\}$$

$$\begin{array}{|l}
\dot{\bigsqcup} : PChain \rightarrow PExec \\
\hline
\dot{\bigsqcup}\ p = (\lambda t : Term \bullet \bigsqcup(\lambda i : \mathbb{N} \bullet p\ i\ t))
\end{array}$$

8 Recursion

If id is an identifier and pc is a parametrised command, possibly containing instances of id, the recursion block $\mathbf{re}\, id \bullet pc\ \mathbf{er}$ is also a parametrised command. Intuitively, a call on the parametric recursion block $\mathbf{re}\, id \bullet V :\!- \ldots id(t) \ldots \mathbf{er}$ is similar to a call on the Prolog procedure defined by `id(V) :- ... id(t) ....`

8.1 Semantics of recursion blocks

A recursion block embeds one or more recursive calls on the block inside a context. Thus, a context is a function from one parametrised command (the recursive call) to another (the entire body of the recursion block):

$$Ctx == PExec \rightarrow PExec$$

Intuitively, to represent $\mathbf{re}\, id \bullet V$ **:-** $\ldots id(t) \ldots \mathbf{er}$, we define the context $\mathscr{C}$ such that $\mathscr{C}(p) = V$ **:-** $\ldots p(t) \ldots$. In this example, $\mathscr{C}$ embeds a call on p in the context given by id, and also provides p with the parameter t. Formally, we define a function that extracts a context from a recursion block.

$$\begin{array}{|l} \mathsf{context} : Env \rightarrow PCmd \nrightarrow Ctx \\ \hline \mathsf{context}(\rho)(\mathbf{re}\, id \bullet pc\ \mathbf{er}) = (\lambda\, p : PExec \bullet \mathsf{pexec}(\rho \oplus \{id \mapsto p\})(pc)) \end{array}$$

This function is partial because it is only defined for *PCmd*s that are recursion blocks.

To define the semantics of recursion blocks we use a fixed point construction, which is defined for all monotonic contexts – see the Knaster–Tarski Theorem (Nelson, 1989). We first define the set of monotonic contexts.

$$MCtx == \{\mathscr{C} : Ctx \mid (\forall\, p, p' : PExec \bullet (p \sqsubseteq p') \Rightarrow (\mathscr{C}(p) \sqsubseteq \mathscr{C}(p')))\}$$

This monotonicity property holds for every context $\mathscr{C}$ that can be constructed in our language (Hayes *et al.*, 2000).

Now the meaning of a recursion block is the least fixed point of the context $\mathscr{C}$ provided by the recursion block (written fix $\mathscr{C}$), where:

$$\begin{array}{|l} \mathsf{fix} : MCtx \rightarrow PExec \\ \hline (\forall\, \mathscr{C} : MCtx \bullet \mathsf{fix}\ \mathscr{C} = \mathscr{C}(\mathsf{fix}\ \mathscr{C})) \\ (\forall\, \mathscr{C} : MCtx;\ p : PExec \bullet (\mathscr{C}(p) = p) \Rightarrow (\mathsf{fix}\ \mathscr{C} \sqsubseteq p)) \end{array}$$

Hence the meaning of a recursion block is the least fixed point of the context corresponding to the recursion block.

$$\mathsf{pexec}(\rho)(\mathbf{re}\, id \bullet pc\ \mathbf{er}) = \mathsf{fix}(\mathsf{context}(\rho)(\mathbf{re}\, id \bullet pc\ \mathbf{er}))$$

8.2 Constructing the fixed point

To simplify this and the next section, we use the syntax of parametrised commands to stand for their *PExec*s and we assume a fixed environment ρ, which is augmented with a single recursive definition.

The least defined command is **abort**. The least defined parametrised command is a parametrised command with **abort** as its body:

$$\mathbf{abort}_1 == V \text{ :- } \mathbf{abort}$$

For a recursion based on a monotonic context $\mathscr{C}$, we construct the chain of programs:

$$\begin{array}{|l} pc : PChain \\ \hline pc = (\lambda\, i : \mathbb{N} \bullet \mathscr{C}^i(\mathbf{abort}_1)) \end{array}$$

That is, we have

$$pc(0) = \mathscr{C}^0(\mathbf{abort}_1) = \mathbf{abort}_1$$
$$pc(i+1) = \mathscr{C}^{i+1}(\mathbf{abort}_1) = \mathscr{C}(pc(i))$$

The sequence pc forms a chain ordered by $\sqsubseteq$ and has a join ($\bigsqcup pc$). By the Limit Theorem (Nelson, 1989), $\bigsqcup pc = \mathsf{fix}\ \mathscr{C}$, as long as $\mathscr{C}$ is a chain-continuous function. For a chain pc for which the join $\bigsqcup pc$ exists, a function $\mathscr{C}$ is *chain-continuous* provided $\mathscr{C}(\bigsqcup pc) = \bigsqcup(\lambda\, i : \mathbb{N} \bullet \mathscr{C}(pc(i)))$. All the contexts $\mathscr{C}$ that can be constructed in our language are chain-continuous (Hayes *et al.*, 2000).

For example, the following equivalences hold in our semantics.

$$\begin{array}{rcl}
\mathbf{re}\, p \bullet X \text{ :- } p(X)\, \mathbf{er} & \sqsubseteq\!\!\!\sqsupseteq & \mathbf{abort}_1 \\
\mathbf{re}\, p \bullet X \text{ :- } \langle X = 1\rangle, p(X)\, \mathbf{er} & \sqsubseteq\!\!\!\sqsupseteq & X \text{ :- } \langle X = 1\rangle, \mathbf{abort} \\
\mathbf{re}\, p \bullet X \text{ :- } \langle X = 1\rangle \wedge p(X)\, \mathbf{er} & \sqsubseteq\!\!\!\sqsupseteq & \mathbf{abort}_1 \\
\mathbf{re}\, p \bullet X \text{ :- } \langle X = 1\rangle \vee p(X)\, \mathbf{er} & \sqsubseteq\!\!\!\sqsupseteq & \mathbf{abort}_1
\end{array}$$

The first is the trival non-terminating recursion. The second fails if X is bound to a value other than one, otherwise it is a non-terminating recursion. The last two both use parallel operators that evaluate both arguments; hence they both degenerate to a non-terminating recursion.

8.3 Mutual recursion

In this paper we do not explicitly handle the definition of a set of mutually recursive procedures. Such a set can always be encoded as a single procedure and hence given a semantics via this encoding. For example, a set of mutually recursive procedures

$$\begin{array}{l} p_1 \mathrel{\widehat=} V_1 \text{ :- } C_1 \\ \vdots \\ p_n \mathrel{\widehat=} V_n \text{ :- } C_n \end{array}$$

may be encoded as a single procedure

$$p \mathrel{\widehat=} (I, V_1, \ldots, V_n) \text{ :- } \langle I = 1\rangle, C_1 \vee \ldots \vee \langle I = n\rangle, C_n$$

where the parameter I (a fresh name) encodes which of the original procedures is being called and the parameter names $V_1, \ldots, V_n$ are assumed to be distinct. A call of the form $p_1(t)$ is then encoded as $p(1, t, \_, \ldots, \_)$.

9 Refinement laws

This section presents a number of refinement laws for the step-wise refinement of logic programs. Each of these laws has been proven with respect to the semantics, using the execution properties listed in section 5.1. Appendix A presents a summary of refinement laws used in this paper.

9.1 Algebraic laws

The commands in the wide-spectrum language obey a number of algebraic properties. Parallel conjunction and disjunction are commutative, but sequential conjunction is not. Parallel conjunction and disjunction as well as sequential conjunction are associative. Parallel conjunction can be refined to sequential conjunction using the rule pand-to-sand because in the sequential form c_2 can assume the context established by c_1, but in the parallel form it cannot:

$$c_1 \wedge c_2 \sqsubseteq c_1 \, , c_2$$

To prove such a law we show

$$\begin{aligned} & \mathsf{exec}(\rho)(c_1 \wedge c_2) \sqsubseteq \mathsf{exec}(\rho)(c_1 \, , c_2) \\ \equiv\ & \mathsf{exec}\ \ \rho\ c_1 \Cap \mathsf{exec}\ \ \rho\ c_2 \subseteq \mathsf{exec}\ \ \rho\ c_1 \fatsemi \mathsf{exec}\ \ \rho\ c_2 \end{aligned}$$

which can be shown via the definitions of $\Cap$ and $\fatsemi$, and the properties of executions.

Parallel conjunction and disjunction distribute over each other. Sequential conjunction distributes over parallel conjunction and disjunction in the following ways:

$$\begin{array}{rcl} c_1 \, , (c_2 \wedge c_3) & \sqsubseteq\!\!\sqsupseteq & (c_1 \, , c_2) \wedge (c_1 \, , c_3) \\ c_1 \, , (c_2 \vee c_3) & \sqsubseteq\!\!\sqsupseteq & (c_1 \, , c_2) \vee (c_1 \, , c_3) \\ c_1 \wedge (c_2 \, , c_3) & \sqsubseteq & (c_1 \wedge c_2) \, , c_3 \\ (c_1 \vee c_2) \, , c_3 & \sqsubseteq\!\!\sqsupseteq & (c_1 \, , c_3) \vee (c_2 \, , c_3) \end{array}$$

The existential and universal quantifiers distribute over disjunction and (parallel) conjunction respectively.

$$\begin{array}{rcl} (\exists\, V \bullet c_1 \vee c_2) & \sqsubseteq\!\!\sqsupseteq & (\exists\, V \bullet c_1) \vee (\exists\, V \bullet c_2) \\ (\forall\, V \bullet c_1 \wedge c_2) & \sqsubseteq\!\!\sqsupseteq & (\forall\, V \bullet c_1) \wedge (\forall\, V \bullet c_2) \end{array}$$

9.2 Monotonicity laws

Each of the language constructs is monotonic with respect to the refinement relation. Monotonicity guarantees that the result of replacing a component of a program by its refinement is itself a refinement of the original program.

For parallel and sequential conjunction, and disjunction, there are laws stating that the command can be refined by refining either of the arguments of the top-level operator, e.g. the rule por-mono:

$$\frac{c_1 \sqsubseteq c_2;\ c_3 \sqsubseteq c_4}{c_1 \vee c_3 \sqsubseteq c_2 \vee c_4}$$

For the existential and universal quantifiers we give monotonicity laws stating that the command can be refined by refining the quantified sub-command, e.g. the rule exists-mono for existential quantifiers:

$$\frac{c_1 \sqsubseteq c_2}{\exists\, V \bullet c_1 \sqsubseteq \exists\, V \bullet c_2}$$

9.3 Predicate lifting

The predicate lifting laws state that predicate operators inside of specifications can be lifted to their corresponding command operator. For example, the rule lift-pand states that predicate conjunction within a specification can be lifted to parallel conjunction at the command level:

$$\langle P \wedge Q \rangle \sqsubseteq\!\sqsupseteq \langle P \rangle \wedge \langle Q \rangle$$

Similar rules are given that lift predicate disjunction to command disjunction, and predicate quantifiers within specifications to the corresponding command quantifiers.

9.4 Specification and assumption laws

Programs can be refined by weakening, removing and introducing assumptions. For example, the law weaken-assumpt states that an assumption can be refined by weakening the predicate:

$$\frac{A \Rrightarrow B}{\{A\} \sqsubseteq \{B\}}$$

where for predicates A and B, A entails B ($A \Rrightarrow B$) iff for all bindings b, A is true at b implies B is true at b, i.e., $A \Rrightarrow B \Leftrightarrow \overline{A} \subseteq \overline{B}$.

A program can be refined by replacing a specification by an equivalent one; the law equivalent-spec states that if P and Q are equivalent, then the program $\langle P \rangle$ is equivalent to the program $\langle Q \rangle$, i.e.

$$\frac{P \equiv Q}{\langle P \rangle \sqsubseteq\!\sqsupseteq \langle Q \rangle}$$

where predicates P and Q are said to be equivalent ($P \equiv Q$) iff for all bindings b, P is true at b iff Q is true at b, i.e. $P \equiv Q \Leftrightarrow \overline{P} = \overline{Q}$.

9.5 Context

We have already seen that commands are refined in the context of a program environment ρ. A second kind of context is an "assumption" context (based on the precondition context used in the program window inference presented in Nickson & Hayes (1997)). For example, consider the command "$\{A\}, S$"; in refining S and its subcommands, we may assume that the predicate A holds. Rather than propagating the assumption throughout S and its subcommands, we add the assumption A to the context of the refinement.

To allow refinement in the context of assumptions, we need to introduce the notion of pointwise refinement, $\dot{\sqsubseteq}$, defined as:

$$
\begin{array}{|l}
\_ \sqsubseteq_\rho \_, \_ \sqsubseteq\!\sqsupseteq_\rho \_ : Env \to Cmd \times Cmd \to (Bnd \to boolean) \\
\hline
c_1 \sqsubseteq_\rho c_2 = (\lambda b : Bnd \bullet \{b\} \in \mathrm{dom}(\mathsf{exec}\ \rho\ c_1) \Rightarrow \\
\qquad (\{b\} \in \mathrm{dom}(\mathsf{exec}\ \rho\ c_2) \wedge \mathsf{exec}\ \rho\ c_1\ \{b\} = \mathsf{exec}\ \rho\ c_2\ \{b\}) \\
c_1 \sqsubseteq\!\sqsupseteq_\rho c_2 = c_1 \sqsubseteq_\rho c_2 \wedge c_2 \sqsubseteq_\rho c_1
\end{array}
$$

As with command refinement we shall omit the environment when it is clear from the context. Note that when there are no assumptions (or equivalently the assumption is true), pointwise refinement is equivalent to command refinement, i.e.

$$c_1 \sqsubseteq c_2 \Leftrightarrow (\mathrm{true} \Rrightarrow (c_1 \sqsubseteq c_2))$$

The laws weaken-assumpt and equivalent-spec can be restated to include the current context, Γ.

$$\frac{\Gamma \Rrightarrow (A \Rightarrow B)}{\Gamma \Rrightarrow (\{A\} \sqsubseteq \{B\})} \qquad \frac{\Gamma \Rrightarrow (P \Leftrightarrow Q)}{\Gamma \Rrightarrow (\langle P\rangle \sqsubseteq\!\sqsupseteq \langle Q\rangle)}$$

When refining c_1 in the context $\{A\}, c_1$ we may assume A holds by extending the current context; this is encapsulated in the rule assumpt-in-context:

$$\frac{\Gamma \wedge A \Rrightarrow (c_1 \sqsubseteq c_2)}{\Gamma \Rrightarrow \{A\}, c_1 \sqsubseteq \{A\}, c_2}$$

Similarly for specifications we have the rule spec-in-context:

$$\frac{\Gamma \wedge P \Rrightarrow (c_1 \sqsubseteq c_2)}{\Gamma \Rrightarrow \langle P\rangle, c_1 \sqsubseteq \langle P\rangle, c_2}$$

If a command c_1 refines to c_2 in a context Γ, then it refines to c_2 in any stronger context. In particular, given a refinement law of the form $c_1 \sqsubseteq c_2$, then $c_1 \sqsubseteq c_2$ in any context, i.e.

$$\frac{c_1 \sqsubseteq c_2}{\Gamma \Rrightarrow (c_1 \sqsubseteq c_2)}$$

This means that any of the (command) refinement laws that do not have proof obligations, such as the algebraic and predicate laws given earlier, can be applied in any context. We also observe that each of our language constructs are monotonic with respect to the pointwise refinement relation in any context. For example, for disjunction the following law holds.

$$\frac{\Gamma \Rrightarrow (c_1 \sqsubseteq c_2);\ \Gamma \Rrightarrow (c_3 \sqsubseteq c_4)}{\Gamma \Rrightarrow (c_1 \vee c_3 \sqsubseteq c_2 \vee c_4)}$$

Notice this law is more general than por-mono, which is an instance of the above law with the assumption context being true.

9.6 Recursion introduction law

We desire a refinement law that allows us to refine a non-recursive parametrised command into a recursive block. That is, for pc a parametrised command and $\mathscr{C}(id)$ a command that may depend on the (fresh) procedure identifier id and the variable V, we want to derive a law with the following as the conclusion.

$$pc \sqsubseteq_\rho \mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er}$$

We will derive such a law using well-founded induction, the general form of which is, for some predicate ϕ and well-founded relation $\prec$,

$$\frac{(\forall Y \bullet Y \prec V \Rightarrow \phi(Y)) \Rrightarrow \phi(V)}{\forall X \bullet \phi(X)}$$

We take $\phi(X)$ to be $pc(X) \sqsubseteq_\rho (\mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er})(X)$. Given this, and the definition of refinement for parametrised commands, the predicate $\forall X \bullet \phi(X)$ on the bottom line is equivalent to our initially stated refinement,

$$pc \sqsubseteq_\rho \mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er}$$

Now we instantiate the top line of the well-founded induction law using the same ϕ.

$$\begin{aligned}&(\forall Y : Term \bullet Y \prec V \Rightarrow pc(Y) \sqsubseteq_\rho (\mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er})(Y)) \Rrightarrow \\ &\quad pc(V) \sqsubseteq_\rho (\mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er})(V)\end{aligned}$$

As id was a fresh name, we define $\rho' = \rho \cup \{id \mapsto \mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er}\}$. The top line can now be written as

$$\begin{aligned}&(\forall Y : Term \bullet Y \prec V \Rightarrow pc(Y) \sqsubseteq_{\rho'} id(Y)) \Rrightarrow \\ &\quad pc(V) \sqsubseteq_{\rho'} (\mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er})(V)\end{aligned}$$

Now we simplify the expression on the right-hand side of the entailment so as to make the whole expression useful as a proof obligation for the recursion introduction law.

$$\begin{aligned}&pc(V) \sqsubseteq_{\rho'} (\mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er})(V)\\ \equiv\ &\text{unfold}\\ &pc(V) \sqsubseteq_{\rho'} (V :\!- \mathscr{C}(\mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er}))(V)\\ \equiv\ &\text{parameter application; definition of } id \text{ in } \rho'\\ &pc(V) \sqsubseteq_{\rho'} \mathscr{C}(id)\end{aligned}$$

Noting that $A \Rightarrow c \sqsubseteq_\rho c' \equiv \{A\}, c \sqsubseteq_\rho c'$, the process produces the following recursion introduction law, recursion-intro.

$$\frac{(\forall Y : Term \bullet \{Y \prec V\}, pc(Y) \sqsubseteq_{\rho'} id(Y)) \Rrightarrow pc(V) \sqsubseteq_{\rho'} \mathscr{C}(id)}{pc \sqsubseteq_\rho \mathbf{re}\, id \bullet V :\!- \mathscr{C}(id)\, \mathbf{er}}$$

9.7 Example

We derive a recursive implementation of a factorial function from the definition of factorial, which is:

$$\begin{aligned}&\mathit{fact}(0) = 1\\ &\mathit{fact}(n+1) = \mathit{fact}(n) \times (n+1), \quad \text{for} \;\; n \geqslant 0\end{aligned}$$

Starting with an initial abstract specification S, we wish to refine this to a concrete program C, such that C can easily be translated (automatically) into an executable program. We adopt a step-wise refinement approach, where we introduce a number of intermediate programs I_j (where $1 \leqslant j \leqslant n$ for some natural n), representing refinements of the previous program, i.e. $S \sqsubseteq I_1 \sqsubseteq \ldots \sqsubseteq I_n \sqsubseteq C$. This is achieved by making use of the transitive nature of the refinement relations.

Monotonicity rules are used to transform a subcommand of the program. The subcommand to be refined is indicated by the markers $\llcorner$ and $\lrcorner$. For binary commands we use the more specific monotonicity rules used to focus on the left or right-hand side of a command. Often a single step in the example actually corresponds to the application of multiple nested monotonicity rules.

Our factorial program has parameters U and V. It can assume $U \in \mathbb{N}$ and must establish $V = \mathit{fact}(U)$.

$$(U, V) \mathrel{:\!-} \{U \in \mathbb{N}\}, \langle V = \mathit{fact}(U)\rangle$$

We make use of recursion-intro, which allows one to assume

$$\forall\, U1, V1 \bullet \{U1 \prec U\}, \{U1 \in \mathbb{N}\}, \langle V1 = \mathit{fact}(U1)\rangle \mathrel{\dot\sqsubseteq_{\rho'}} f(U1, V1)$$

where f is a fresh procedure identifier and $\rho' = \rho \cup \{f \mapsto \mathbf{re}\, f \bullet (U, V) \mathrel{:\!-} \mathscr{C}(f)\, \mathbf{er}\}$ and refine

$$\{U \in \mathbb{N}\}, \langle V = \mathit{fact}(U)\rangle$$

in this context. Focusing on the specification $\langle V = \mathit{fact}(U)\rangle$, we can assume the assumptions prior to the specification; in order that this assumption (and assumptions introduced to the context later) can be used we now use the pointwise refinement relation with the extended environment ρ'.

Assumption 1:
  $\forall\, U1, V1 \bullet \{U1 < U\}, \{U1 \in \mathbb{N}\}, \langle V1 = \mathit{fact}(U1)\rangle \mathrel{\dot\sqsubseteq_{\rho'}} f(U1, V1)$
Assumption 2: $U \in \mathbb{N}$
$\bullet\ \langle V = \mathit{fact}(U)\rangle$
$\dot\sqsubseteq$ case analysis, Assumption 2
  $(\langle U = 0\rangle, \llcorner\langle V = \mathit{fact}(U)\rangle\lrcorner) \vee (\langle U > 0\rangle, \langle V = \mathit{fact}(U)\rangle)$
$\dot\sqsubseteq$ spec-in-context
  Assumption 3: $U = 0$
  $\bullet\ \langle V = \mathit{fact}(U)\rangle$
  $\dot\sqsubseteq$ equivalent-spec $(U = 0 \Rightarrow (V = \mathit{fact}(U) \Leftrightarrow V = 1))$
    $\langle V = 1\rangle$
  $(\langle U = 0\rangle, \ulcorner\langle V = 1\rangle\urcorner) \vee (\langle U > 0\rangle, \llcorner\langle V = \mathit{fact}(U)\rangle\lrcorner)$

Focusing on the second occurrence of $\langle V = fact(U)\rangle$, we can again make a contextual assumption, this time from the specification $\langle U > 0\rangle$:

$$
\begin{array}{ll}
& \text{Assumption 4: } U > 0 \\
& \bullet\ \langle V = fact(U)\rangle \\
\sqsubseteq & \textsf{equivalent-spec}\text{, factorial def., assumptions} \\
& \langle \exists\, U1, V1 \bullet U1 = U - 1 \wedge V1 = fact(U1) \wedge V = V1 * U\rangle \\
\sqsubseteq & \textsf{lift-exists, lift-pand, pand-to-sand}\text{ (x2)} \\
& \exists\, U1, V1 \bullet \langle U1 = U - 1\rangle, \langle V1 = fact(U1)\rangle, \langle V = V1 * U\rangle \\
\sqsubseteq & \textsf{assumpt-after-spec}\text{ (x2)} \\
& \exists\, U1, V1 \bullet \langle U1 = U - 1\rangle, \\
& \qquad\qquad \llcorner\{U1 < U\}, \{U1 \in \mathbb{N}\}, \langle V1 = fact(U1)\rangle\lrcorner, \\
& \qquad\qquad \langle V = V1 * U\rangle
\end{array}
$$

Next we use Assumption 1 to introduce a recursive call, making use of the assumption $U1 = U - 1$ to discharge the variant on the well-founded relation.

$$
\begin{array}{ll}
& \bullet\ \{U1 < U\}, \{U1 \in \mathbb{N}\}, \langle V1 = fact(U1)\rangle \\
\sqsubseteq & \text{Assumption 1} \\
& f(U1, V1)
\end{array}
$$

Putting this all together and removing the assumptions, using remove-assumpt, we get

$$
\begin{array}{ll}
& \{U \in \mathbb{N}\}, \langle V = fact(U)\rangle \\
\sqsubseteq_{\rho'} & (\langle U = 0\rangle, \langle V = 1\rangle) \vee \\
& (\langle U > 0\rangle, (\exists\, U1, V1 \bullet \langle U1 = U - 1\rangle, f(U1, V1), \langle V = V1 * U\rangle))
\end{array}
$$

Closing the introduction of recursion, we have proved:

$$
\begin{array}{ll}
& (U, V) \mathrel{:\!-} \{U \in \mathbb{N}\}, \langle V = fact(U)\rangle \\
\sqsubseteq_{\rho} & \mathbf{re}\, f \bullet (U, V) \mathrel{:\!-} \\
& \quad (\langle U = 0\rangle, \langle V = 1\rangle) \vee \\
& \quad (\langle U > 0\rangle, (\exists\, U1, V1 \bullet \langle U1 = U - 1\rangle, f(U1, V1), \langle V = V1 * U\rangle)) \\
& \mathbf{er}
\end{array}
$$

To translate this (informally) into Prolog, we:

- turn the recursion block into a recursive procedure;
- implement the arithmetic specifications using `is`;
- express the disjunction using separate clauses;
- make the existential quantification implicit.

The result is:

```
f(U,V) :- U=0, V=1.
f(U,V) :- U>0, U1 is U-1, f(U1,V1), V is V1*U.
```

10 Example: N-queens

In this section we present a non-trivial logic program refinement to demonstrate some refinement techniques that have been developed. The case study chosen is the N-queens problem. It is a generalisation of a chess problem, where eight queens are to be placed on a chess board so that no two queens may attack each other (they do not share a row, column, or diagonal). The generalisation is to have N queens on an $N \times N$ board.

10.1 Specification of N-queens

We first describe and motivate our specification of the N-queens problem. Given a natural number N, a solution to the N-queens problem can be represented by a list S of length N that contains numbers between 1 and N. The row number of the queen in column i is given by $S(i)$, i.e. a queen at location (4,5) is indicated by $S(4) = 5$. This representation implicitly guarantees that there is only a single queen in each column.

We define a predicate $psoln(S, N)$, which checks that in the list S there are no row clashes – no two row numbers are the same – and no diagonal clashes – the absolute value of the difference in the column numbers does not equal the absolute difference in the row numbers – as well as requiring the elements of S are valid row numbers in the range 1 to N.

$$\begin{aligned} &psoln(S,N) == list(S) \wedge (\forall i : 1..\#S \bullet S(i) \in 1..N \wedge \\ &\qquad (\forall j : i+1..\#S \bullet S(i) \neq S(j) \wedge |i-j| \neq |S(i) - S(j)|)) \end{aligned}$$

Our specification of N-queens constrains the length of a solution S to be N.

$$nqueens \mathrel{\widehat{=}} (N, S) \mathrel{:\!-} \{N \in \mathbb{N}\}, \langle \#S = N \wedge psoln(S, N)\rangle$$

10.2 Refinement of N-queens

We note the following derived property of $psoln$.

$$psoln([H \mid T], N) \Leftrightarrow \left(\begin{array}{l} psoln(T,N) \wedge H \in 1..N \wedge \\ notrow(H,T) \wedge notdiag(H,T) \end{array} \right) \tag{6}$$

where, for H a natural number and T a list of natural numbers,

$$notrow(H,T) == (\forall i : 1..\#T \bullet H \neq T(i)) \tag{7}$$

$$notdiag(H,T) == (\forall i : 1..\#T \bullet i \neq |H - T(i)|) \tag{8}$$

We implement the N-queens solution using an accumulator-style program, by introducing a partial solution P as a parameter. P starts as an empty list, and is extended on each recursive call until a full solution of length N is built. The accumulator version of N-queens includes $psoln(P, N)$ as an assumption, and must establish that the full solution S has a suffix of P and satisfies $psoln(S, N)$.

$$\begin{aligned} &nqacc \mathrel{\widehat{=}} (N, P, S) \mathrel{:\!-} \{N \in \mathbb{N}\}, \{psoln(P, N)\}, \\ &\qquad \langle P \text{ suffix } S \wedge \#S = N \wedge psoln(S, N)\rangle \end{aligned}$$

We implement *nqueens* by calling *nqacc* with the empty list, i.e.

$$nqueens(N, S) \sqsubseteq nqacc(N, [\,], S)$$

The equivalence of these programs can be seen to hold by observing:

$$psoln([\,], N) \land [\,]\ \text{suffix}\ S$$

Our task is now to refine *nqacc*. As with the refinement of the factorial program in section 9.7, we use the recursion-intro law. We may assume the following inductive hypothesis during the refinement.

$$\begin{array}{l}(\forall N, P1, S \bullet \\ \quad \{P1 \prec P\}, \{N \in \mathbb{N}\}, \{psoln(P1, N)\}, \\ \quad \langle P1\ \text{suffix}\ S \land \#S = N \land psoln(S, N)\rangle \\ \quad \sqsubseteq nq(N, P1, S))\end{array} \tag{9}$$

where $P1 \prec P$ is the well-founded relation which holds when $N \geqslant \#P1 > \#P$. (Note that $P1$ is longer than P, but the well-foundedness is maintained by the upper bound of N.)

Now we begin the refinement of the body of *nqacc*. We apply the case-analysis law, to split the specification into the cases $\#P = N$ and $\#P < N$, because $psoln(P, N)$ implies $\#P \leqslant N$.

$$\begin{array}{l}\{N \in \mathbb{N}\}, \{psoln(P, N)\}, \\ (\langle \#P = N\rangle, \langle P\ \text{suffix}\ S \land \#S = N \land psoln(S, N)\rangle \\ \lor \\ \langle \#P < N\rangle, \langle P\ \text{suffix}\ S \land \#S = N \land psoln(S, N)\rangle)\end{array}$$

The first disjunct is the base case:

$$\langle \#P = N\rangle, \langle P\ \text{suffix}\ S \land \#S = N \land psoln(S, N)\rangle$$

From the context we know $psoln(P, N)$. Hence the above is equivalent to

$$\langle \#P = N\rangle, \langle S = P\rangle$$

Next we focus on the branch that requires a recursive call.

$$\langle \#P < N\rangle, \langle P\ \text{suffix}\ S \land \#S = N \land psoln(S, N)\rangle$$

Since we are accumulating the solution, we prepend an element X to the front of P. We note that

$$\#P < \#S \Rightarrow (P\ \text{suffix}\ S \Leftrightarrow (\exists X \bullet [X \mid P]\ \text{suffix}\ S))$$

Using the law spec-in-context we refine to

$$\langle \#P < N\rangle, \langle (\exists X \bullet [X \mid P]\ \text{suffix}\ S) \land \#S = N \land psoln(S, N)\rangle$$

The predicate $psoln(S, N)$ has the property that any suffix P of S also satisfies $psoln(P, N)$. Hence the above is equivalent to the following.

$$\begin{array}{l}\langle \#P < N\rangle, \\ \langle (\exists X \bullet psoln([X \mid P], N) \land [X \mid P]\ \text{suffix}\ S) \land \#S = N \land psoln(S, N)\rangle\end{array}$$

We extend the scope of X to the end of the program, then use lift-exists to take it to the program level. We also convert the conjunction involving $psoln([X \mid P], N)$ to a sequential conjunction, with $psoln$ on the left-hand side (using lift-pand and pand-to-sand).

$$\begin{array}{l} \langle \#P < N \rangle, \\ (\exists X \bullet \langle psoln([X \mid P], N) \rangle, \langle [X \mid P] \text{ suffix } S \wedge \#S = N \wedge psoln(S, N) \rangle) \end{array}$$

We now establish the assumptions needed to match the inductive hypothesis. Since $N \geqslant \#[X \mid P] > \#P$ we may introduce $[X \mid P] \prec P$. Since $N \in \mathbb{N}$ is in context we may introduce it as an assumption. We also introduce $\{psoln([X \mid P], N)\}$ using law establish-assumpt.

$$\begin{array}{l} \langle \#P < N \rangle, \\ (\exists X \bullet \langle psoln([X \mid P], N) \rangle, \\ \quad \{[X \mid P] \prec P\}, \{N \in \mathbb{N}\}, \{psoln([X \mid P], N)\}, \\ \quad \langle [X \mid P] \text{ suffix } S \wedge \#S = N \wedge psoln(S, N) \rangle) \end{array}$$

We may now use the inductive hypothesis (9) to introduce a recursive call on $nq(N, [X \mid P], S)$.

$$\langle \#P < N \rangle, (\exists X \bullet \langle psoln([X \mid P], N) \rangle, nq(N, [X \mid P], S))$$

By observing the equivalence (6), and noting that the context includes $psoln(P, N)$, we may simplify $\langle psoln([X \mid P], N) \rangle$ to $\langle X \in 1..N \wedge notrow(X, P) \wedge notdiag(X, P) \rangle$ using spec-in-context. We also lift the conjunctions to the program level using lift-pand. Now closing the introduction of recursion, we get

$$\begin{array}{l} \textbf{re}\ nq \bullet (N, P, S)\text{:-} \\ \quad \{N \in \mathbb{N}\}, \{psoln(P, N)\}, \\ \quad (\langle \#P = N \rangle, \langle P = S \rangle \\ \quad \vee \\ \quad \langle \#P < N \rangle, \\ \quad (\exists X \bullet (\langle X \in 1..N \rangle \wedge \langle notrow(X, P) \rangle \wedge \langle notdiag(X, P) \rangle), \\ \qquad nq(N, [X \mid P], S))) \\ \textbf{er} \end{array}$$

This is the overall structure of the program. However we must still implement $\langle X \in 1..N \rangle$, $\langle notrow(X, P) \rangle$ and $\langle notdiag(X, P) \rangle$.

10.3 Subproblems

The implementation of $\langle X \in 1..N \rangle$ via a recursive procedure is straightforward and is omitted here.

To refine the other two specifications we observe that they each require that a property $\mathscr{P}$ is established for every element in a list (cf. (7) and (8)). However universal quantification is not executable. To remove the quantification we recursively establish the property $\mathscr{P}$ for each element in the list. This is encapsulated in the following law.

property-over-list
$\langle list(L)\rangle \wedge \langle \forall i : 1..\#L \bullet \mathscr{P}(V, L(i))\rangle \sqsubseteq proc(V, L)$
where
$proc \mathrel{\widehat{=}} \mathbf{re}\; p \bullet (V, L)$:-
$\quad\langle L = [\,]\rangle \vee (\exists H, T \bullet \langle L = [H \mid T]\rangle, \langle \mathscr{P}(V, H)\rangle, p(V, T))\;\mathbf{er}$

We use this law to derive an implementation of $\langle notrow(X, P)\rangle$.

$\langle list(P)\rangle \wedge \langle \forall i : 1..\#P \bullet X \neq P(i)\rangle \sqsubseteq norowclash(X, P)$
where
$norowclash \mathrel{\widehat{=}} \mathbf{re}\; norowc \bullet (X, P)$:-
$\quad\langle P = [\,]\rangle \vee (\exists H, T \bullet \langle P = [H \mid T]\rangle, \langle X \neq H\rangle, norowc(X, T))\;\mathbf{er}$

In a similar way we derive an implementation of $\langle notdiag(X, P)\rangle$, though in this case we use a more general law property-over-list-indexed.

property-over-list-indexed
$\langle list(L)\rangle \wedge \langle \forall i : 1..\#L \bullet \mathscr{P}(i, V, L(i))\rangle \sqsubseteq proc(V, L, 1)$
where
$proc \mathrel{\widehat{=}} \mathbf{re}\; p \bullet (V, L, J)$:-
$\quad\langle L = [\,]\rangle \vee (\exists H, T \bullet \langle L = [H \mid T]\rangle, \langle \mathscr{P}(J, V, H)\rangle, p(V, T, J+1))\;\mathbf{er}$

The implementation of $\langle notdiag(X, P)\rangle$ follows.

$\langle list(P)\rangle \wedge \langle \forall i : 1..\#P \bullet i \neq |X - P(i)|\rangle \sqsubseteq nodiagAcc(X, P, 1)$
where
$nodiagAcc \mathrel{\widehat{=}} \mathbf{re}\; nod \bullet (X, P, J)$:-
$\quad(\langle P = [\,]\rangle \vee$
$\quad(\exists H, T \bullet \langle P = [H \mid T]\rangle, \langle J \neq |X - H|\rangle, nod(X, T, J+1)))\;\mathbf{er}$

Whereas earlier laws are primitives corresponding to language constructs, the laws property-over-list and property-over-list-indexed are examples of higher-level refinement laws that encapsulate a design pattern for a particular data structure, in this case lists.

10.4 Prolog implementation

The refinement is completed by converting any parallel conjunctions to sequential conjunctions using the law pand-to-sand. From this program the following Prolog code can be generated by performing the translations noted in section 9.7.

```
nqueens(N,S) :- nqacc(N,[],S).

nqacc(N,P,S) :- length(P,N), S = P.
nqacc(N,P,S) :-
    length(P,M), M < N,
    memrng(X,N), norowclash(X,P), nodiagAcc(X,P,1),
    nqacc(N,[X|P], S).

memrng(X,N) :- N > 0, X = N.
memrng(X,N) :- N > 0, Nm1 is N-1, memrng(X,Nm1).

norowclash(_,[]).
norowclash(X, [H|T]) :- X =\= H, norowclash(X,T).

nodiagAcc(_,[],_).
nodiagAcc(X, [H|T],J):-
    XmH is X - H, abs(XmH,AbsXmH),
    J1 is J+1, AbsXmH =\= J,
    nodiagAcc(X, T, J1).
```

11 Tool support

To support the refinement calculus, a prototype tool, `Marvin` (Hemer *et al.*, 2001), has been developed based on the Isabelle theorem prover (Paulson, 1994). `Marvin` includes an embedding of the wide-spectrum language (and its associated semantics) in Isabelle/HOL. All of the refinement laws listed in Appendix A have been proven using the tool, and the factorial example presented in section 9 has been performed.

Isabelle is a generic interactive theorem prover, supporting a variety of different object logics, including first-order logic, higher-order logic and Zermelo Frankel set theory. Isabelle/HOL provides the `datatype` facility, used to declare recursive data structures. Isabelle/HOL also provides the `primrec` construct, allowing the user to define primitive recursive operators on recursive datatypes.

11.1 Tool architecture

Figure 3 shows the structure of the theories defined in `Marvin`.

`Marvin_lemmas` extends the existing Isabelle/HOL theories with additional lemmas required by the tool. The theories `Term` and `Pred` model terms and predicates respectively, using a datatype to declare their syntax. The theory `Command` models *Cmd*, representing the commands in the wide-spectrum language; commands are defined using the datatype facility. `State` models program state as a set of bindings. `Pointwise` models a pointwise union and pointwise intersection operator, corresponding to generalisations of $\uplus$ and $\Cap$. The theory `RefRels` models the polymorphic refinement relations "refines to" ($\sqsubseteq$) and "refinement equivalence" ($\sqsubseteq\!\!\sqsupseteq$). `Execs` models the type *Exec* and the function exec. `RefCmd` models refinement of commands, from which a number of useful refinement laws are derived. `Recurse` models the recursive features of the language. `RefAssume` extends the notion of command refinement to include assumption context.

The theories `Command`, `Execs` and `RefCmd` are described briefly below.

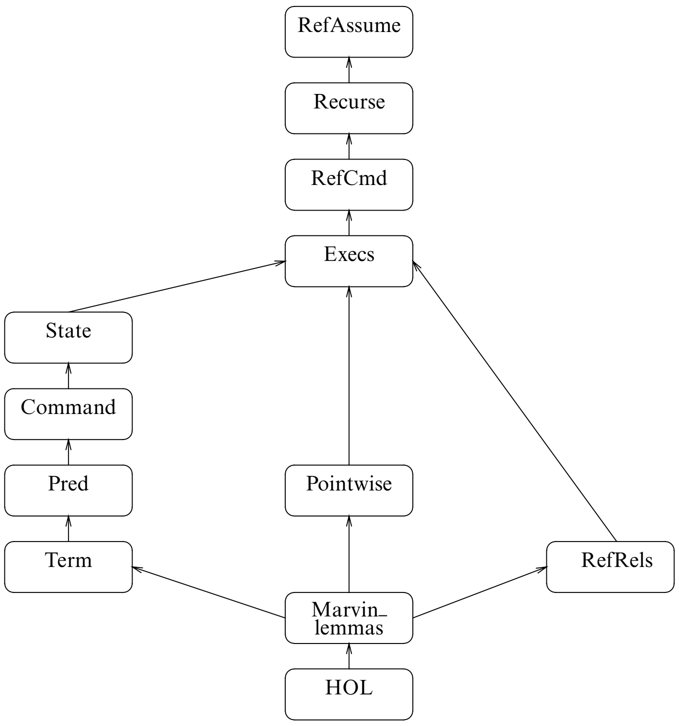

Fig. 3. Structure of the RefLP theory.

11.2 Commands

The theory `Command` introduces the type `Cmd`, corresponding to *Cmd* defined in section 3. Commands are defined using Isabelle/HOL's datatype facility, and each command is described via a type constructor and the type of its arguments.

```
datatype Cmd   ==   ⟨ pred ⟩
                |   { pred }
                |   Cmd ∧ Cmd
                |   Cmd ∨ Cmd
                |   Cmd , Cmd
                |   ∃ Var • Cmd
                |   ∀ Var • Cmd
                |   fail
                |   abort
                |   skip
                |   PIdent(rterm)
```

11.3 Executions

The theory `Execs` introduces the execution type *Exec* and the function exec, mapping a command to an execution. Firstly the type `StateMap` is defined, representing the set of partial mappings between initial and final states.

```
StateMap == State ⇸ State
```

Partial mappings are defined in Isabelle/HOL in terms of total functions. The labels `Some` and `None` are used to distinguish defined and undefined values. More precisely, for a partial mapping f and element x, if x is in the domain of f then there exists a y such that `f(x) = Some y`, otherwise `f(x) = None`. The label `the` can be used to strip away the label `Some`.

We define the type `Exec`, as a subtype of `StateMap`, using Isabelle's `typedef` facility. The set of executions corresponds to those partial functions satisfying the three properties of executions (1), (2) and (3) defined in section 5.1:

```
Exec == {e: StateMap | dom(e) = ℙ {b. {b} ∈ dom(e)}
          ∧ ∀ s ∈ dom(e). the e(s) ⊆ s
          ∧ ∀ s ∈ dom(e). e(s) = Some {b. b ∈ s & (e({b}) = Some {b})}}
```

The function `exec` is defined using primitive recursion. The definition in Isabelle follows that given in figure 2, except, for simplicity, definedness has been dropped from the definition of `exec` for specifications and assumptions. We also include the procedure call semantics (defined in section 7.5).

```
"exec p ⟨ x ⟩ = (λ s . Some (s ∩ bnds(x)))"
"exec p { x } = (λ s . if s ⊆ bnds(x) then Some s else None)"
"exec p (c1 ∧ c2) = ((exec p c1) ⋒ (exec p c2))"
"exec p (c1 ∨ c2) = ((exec p c1) ⋓ (exec p c2))"
"exec p (c1 , c2) = ((exec p c1) ⨾ (exec p c2))"
"exec p (∃ v • c) = exists v (exec p c)"
"exec p (∀ v • c) = forall v (exec p c)"
"exec p fail = (λ s. Some ∅)"
"exec p abort = (λ s. if s = ∅ then Some ∅ else None)"
"exec p skip = (λ s. Some s)"
"exec p fid(t) =
     (if fid: dom(p)
     then Rep_Exec((the (p fid))(t))
     else (λ s. if s = ∅ then Some ∅ else None))"
```

When processing a `primrec` declaration, Isabelle performs a number of checks. These checks include ensuring that there is exactly one reduction rule for each constructor, and that the variables on the right-hand side of each definition are captured by any variables given on the left. In this way `primrec` declarations are a safe and conservative way of defining functions on recursively defined data structures.

Each of the reduction rules in the `primrec` declaration are added to the default simplification set, and are applied automatically when a simplification tactic is called. Thus, in general, the user does not need to refer to the reduction rules explicitly.

11.4 Refinement rules

The theory `RefCmd` introduces the refinement relations $\sqsubseteq$ and $\sqsupseteq\sqsubseteq$ for commands, defined in the context of an environment ρ. For example, the rule exists-mono is represented in `Marvin` as:

```
env ρ ⊩ S ⊑ T ==> env ρ ⊩ ∃ v S ⊑ ∃ v T
```

Refinement laws proven within Marvin include algebraic properties, predicate lifting laws, monotonicity laws, specification and assumption laws, and laws which extend and use the assumption context. In particular, the laws listed in Appendix A have all been proven within Marvin. The refinement laws have been used to refine a number of programs in Marvin.

12 Conclusion

We have presented a refinement calculus for logic programming. The calculus contains a wide-spectrum logic programming language, including both specification and executable constructs. We presented a declarative semantics for this wide-spectrum language based on executions, which are partial functions from states to states. Part of these semantics is a formal notion of refinement over programs in the wide-spectrum language. To illustrate the semantics we presented refinement laws proved in our semantics, and included the refinement of a non-trivial program. We also discussed a tool that supports the refinement calculus, based on the Isabelle theorem prover.

In earlier work (Hayes *et al.*, 1997), we defined the meaning of a command c in the wide-spectrum language by a pair of predicates: $ok.c$ is a predicate defining the initial condition under which execution of the command is well-defined (essentially representing the assumptions the command makes about its context), while $ef.c$ is the effect of the command, provided that its assumptions are satisfied. The two semantics are closely related in that, for every command c,

$$\begin{aligned}\overline{ok.c} &= \bigcup \mathrm{dom}(\mathsf{exec}\, c)\\ \overline{ok.c \wedge ef.c} &= (\mathsf{exec}\, c)(\overline{ok.c})\end{aligned}$$

However, the earlier paper lacked a rigorous treatment of procedures, parameters, and recursion, and the semantics we use here is chosen to facilitate the presentation of those concepts.

In Colvin *et al.* (1997) we demonstrate how we represent types in the refinement calculus via specifications and assumptions, and include a wider discussion on contextual refinement. In Colvin *et al.* (1998) we introduce *data refinement*, where we change the representation of the types of a procedure. This allows a specification type to be replaced by an implementation type, or a procedure to be implemented by a more efficient representation. Data refinement is performed on a procedure-by-procedure basis. We have three kinds of data refinement, distinguished by interface issues. This work is extended in Colvin *et al.* (2001), where we consider data refinement on groups of procedures. We introduce a notion of *module* into the wide spectrum language, and by restricting the set of programs that may use a module, we can develop efficient representations of the original types. In Colvin *et al.* (2002) we present a prototype tool for generating Mercury (Somogyi *et al.*, 1995) code from programs in our wide-spectrum language. We define what it means for a wide-spectrum program to be executable, and derive type and mode declarations by analysing the structure of the program.

Acknowledgements

This work reported in this paper is supported by Australian Research Council grant number A49937007: *Refinement Calculus for Logic Programming.*

A Refinement laws

A.1 Algebraic laws

pand-commute	$c_1 \wedge c_2 \sqsubseteq\sqsupseteq c_2 \wedge c_1$
pand-assoc	$(c_1 \wedge c_2) \wedge c_3 \sqsubseteq\sqsupseteq c_1 \wedge (c_2 \wedge c_3)$
pand-idempotent	$c \wedge c \sqsubseteq\sqsupseteq c$
por-commute	$c_1 \vee c_2 \sqsubseteq\sqsupseteq c_2 \vee c_1$
por-assoc	$(c_1 \vee c_2) \vee c_3 \sqsubseteq\sqsupseteq c_1 \vee (c_2 \vee c_3)$
por-idempotent	$c \vee c \sqsubseteq\sqsupseteq c$
sand-assoc	$(c_1 , c_2) , c_3 \sqsubseteq\sqsupseteq c_1 , (c_2 , c_3)$
sand-idempotent	$(c_1 , c_1) \sqsubseteq\sqsupseteq c_1$
pand-distrib-por	$c_1 \wedge (c_2 \vee c_3) \sqsubseteq\sqsupseteq (c_1 \wedge c_2) \vee (c_1 \wedge c_3)$
por-distrib-pand	$c_1 \vee (c_2 \wedge c_3) \sqsubseteq\sqsupseteq (c_1 \vee c_2) \wedge (c_1 \vee c_3)$
left-sand-distrib-por	$c_1, (c_2 \vee c_3) \sqsubseteq\sqsupseteq (c_1, c_2) \vee (c_1, c_3)$
left-sand-distrib-pand	$c_1, (c_2 \wedge c_3) \sqsubseteq\sqsupseteq (c_1, c_2) \wedge (c_1, c_3)$
pand-distrib-sand	$c_1 \wedge (c_2 , c_3) \sqsubseteq (c_1 \wedge c_2) , c_3$
right-sand-distrib-por	$(c_1 \vee c_2) , c_3 \sqsubseteq\sqsupseteq (c_1 , c_3) \vee (c_2 , c_3)$
forall-distrib	$(\forall X \bullet (c_1 \wedge c_2)) \sqsubseteq\sqsupseteq (\forall X \bullet c_1) \wedge (\forall X \bullet c_2)$
exists-distrib	$(\exists X \bullet (c_1 \vee c_2)) \sqsubseteq\sqsupseteq (\exists X \bullet c_1) \vee (\exists X \bullet c_2)$
pand-to-sand	$c_1 \wedge c_2 \sqsubseteq c_1 , c_2$
extend-scope-exists-over-pand	$\dfrac{X \text{ nfi } c_2}{(\exists X \bullet c_1) \wedge c_2 \sqsubseteq\sqsupseteq (\exists X \bullet c_1 \wedge c_2)}$

A.2 Refinement relation laws

refsto-reflex	$c \sqsubseteq c$
refeq-reflex	$c \sqsubseteq\sqsupseteq c$
refsto-trans	$\dfrac{c_1 \sqsubseteq c_2;\; c_2 \sqsubseteq c_3}{c_1 \sqsubseteq c_3}$
refeq-trans	$\dfrac{c_1 \sqsubseteq\sqsupseteq c_2;\; c_2 \sqsubseteq\sqsupseteq c_3}{c_1 \sqsubseteq\sqsupseteq c_3}$
refeq-symm	$\dfrac{c_1 \sqsubseteq\sqsupseteq c_2}{c_2 \sqsubseteq\sqsupseteq c_1}$
refsto-anti-symm	$\dfrac{c_1 \sqsubseteq c_2;\; c_2 \sqsubseteq c_1}{c_1 \sqsubseteq\sqsupseteq c_2}$
refeq-stronger-than-refsto	$\dfrac{c_1 \sqsubseteq\sqsupseteq c_2}{c_1 \sqsubseteq c_2}$

A.3 Monotonicity laws

pand-mono	$\dfrac{c_1 \sqsubseteq c_2;\; c_3 \sqsubseteq c_4}{c_1 \wedge c_3 \sqsubseteq c_2 \wedge c_4}$
por-mono	$\dfrac{c_1 \sqsubseteq c_2;\; c_3 \sqsubseteq c_4}{c_1 \vee c_3 \sqsubseteq c_2 \vee c_4}$
sand-mono	$\dfrac{c_1 \sqsubseteq c_2;\; c_3 \sqsubseteq c_4}{c_1 \,,\, c_3 \sqsubseteq c_2 \,,\, c_4}$
exists-mono	$\dfrac{c_1 \sqsubseteq c_2}{(\exists X \bullet c_1) \sqsubseteq (\exists X \bullet c_2)}$
forall-mono	$\dfrac{c_1 \sqsubseteq c_2}{(\forall X \bullet c_1) \sqsubseteq (\forall X \bullet c_2)}$

A.4 Predicate lifting

lift-pand	$\langle P \rangle \wedge \langle Q \rangle \mathrel{\underline{\square}} \langle P \wedge Q \rangle$
lift-por	$\langle P \rangle \vee \langle Q \rangle \mathrel{\underline{\square}} \langle P \vee Q \rangle$
lift-exists	$\exists X \bullet \langle P \rangle \mathrel{\underline{\square}} \langle \exists X \bullet P \rangle$
lift-forall	$\forall X \bullet \langle P \rangle \mathrel{\underline{\square}} \langle \forall X \bullet P \rangle$

A.5 Specification and assumption laws

weaken-assumpt	$\dfrac{P \Rrightarrow Q}{\{P\} \sqsubseteq \{Q\}}$
remove-assumpt	$\{A\} \,,\, c \sqsubseteq c$
combine-assumpt	$\{A\} \,,\, \{B\} \mathrel{\underline{\square}} \{A \wedge B\}$
establish-assumpt	$\langle P \rangle \mathrel{\underline{\square}} \langle P \rangle \,,\, \{P\}$
equivalent-spec	$\dfrac{P \equiv Q}{\langle P \rangle \mathrel{\underline{\square}} \langle Q \rangle}$
assumpt-after-spec	$\dfrac{P \Rrightarrow A}{\langle P \rangle \mathrel{\underline{\square}} \langle P \rangle \,,\, \{A\}}$

A.6 Context

introduce-assumpt	$\dfrac{\Gamma \Rrightarrow A}{\Gamma \Rrightarrow (c \mathrel{\underline{\boxdot}} \{A\}, c)}$
introduce-spec	$\dfrac{\Gamma \Rrightarrow B}{\Gamma \Rrightarrow (c \mathrel{\underline{\boxdot}} \langle B \rangle \wedge c)}$

assumpt-in-context
$$\frac{\Gamma \wedge A \Rrightarrow (c_1 \mathrel{\dot{\sqsubseteq}} c_2)}{\Gamma \Rrightarrow (\{A\}\,, c_1 \mathrel{\dot{\sqsubseteq}} \{A\}\,, c_2)}$$

spec-in-context
$$\frac{\Gamma \wedge A \Rrightarrow (c_1 \mathrel{\dot{\sqsubseteq}} c_2)}{\Gamma \Rrightarrow (\langle A\rangle\,, c_1 \mathrel{\dot{\sqsubseteq}} \langle A\rangle\,, c_2)}$$

equivalent-under-assumpt
$$\frac{A \Rrightarrow (P \Leftrightarrow Q)}{\{A\}\,, \langle P\rangle \mathrel{\sqsubseteq\!\sqsupseteq} \{A\}\,, \langle Q\rangle}$$

use-parallel-spec
$$\frac{I \Rrightarrow (P \Leftrightarrow Q)}{\langle I\rangle \wedge \langle P\rangle \mathrel{\sqsubseteq\!\sqsupseteq} \langle I\rangle \wedge \langle Q\rangle}$$

case-analysis
$$\frac{\Gamma \Rrightarrow P \vee Q}{\Gamma \Rrightarrow (c \mathrel{\dot{\sqsubseteq}} (\langle P\rangle\,, c) \vee (\langle Q\rangle\,, c))}$$

References

Back, R.-J. (1980) *Correctness preserving program refinements: Proof theory and applications.* Tract 131, Mathematisch Centrum, Amsterdam.

Clark, K. (1978) *The synthesis and verification of logic programs.* Research report, Imperial College.

Colvin, R., Hayes, I. J. and Strooper, P. (1998) Data refining logic programs. In: Grundy, J., Schwenke, M. & Vickers, T. (eds), *International Refinement Workshop and Formal Methods Pacific (NFMW 1998)*, pp. 100–116. Springer-Verlag.

Colvin, R., Hayes, I. J. and Strooper, P. (2000) Refining logic programs using types. In: Edwards, J. (ed), *Australasian Computer Science Conference (ACSC 2000)*, pp. 43–50. IEEE Computer Society.

Colvin, R., Hayes, I. J. and Strooper, P. (2001) A technique for modular logic program refinement. In: Lau, K.-K. (ed.), *Logic Based Program Synthesis and Transformation LOPSTR 2000, selected papers.* LNCS, vol. 2402, pp. 38–56. Springer-Verlag.

Colvin, R., Hayes, I. J., Hemer, D. and Strooper, P. (2002) Translating refined logic programs to Mercury. In: Oudshoorn, M. (ed.), *Australasian Computer Science Conference (ACSC 2002)*, pp. 33–40. Australian Computer Society.

Deville, Y. (1990) *Logic Programming: Systematic program development.* Addison-Wesley.

Deville, Y. and Lau, K.-K. (1994) Logic program synthesis. *J. Logic Program.* **19/20**, 321–350. (Special Issue: Ten Years of Logic Programming.)

Hayes, I., Nickson, R. and Strooper, P. (1997) Refining specifications to logic programs. In: Gallagher, J. (ed.), *Logic Program Synthesis and Transformation. Proceedings 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 1996*, LNCS 1207, pp. 1–19. Springer-Verlag.

Hayes, I., Nickson, R., Strooper, P. and Colvin, R. (2000) *A declarative semantics for logic program refinement*. Technical Report 00-30, Software Verification Research Centre, University of Queensland.

Hemer, D., Hayes, I. and Strooper, P. (2001) Refinement calculus for logic programming in Isabelle/HOL. In: Boulton, R. and Jackson, P. (eds.), *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001*, LNCS 2152, pp. 249–264. Springer-Verlag.

Hoare, C. A. R. (2000) Unifying theories for logic programming. In: Lloyd, J. (ed.), *Australian Workshop on Computational Logic*, pp. 31–56.

Hogger, C. J. (1981) Derivation of logic programs. *J. ACM*, **28**(2), 372–392.

Jacquet, J.-M. (ed). (1993) *Constructing Logic Programs.* Wiley Professional Computing.

Kok, J. N. (1990) *On logic programming and the refinement calculus: Semantics-based program transformations.* Technical Report RUU-CS-90-39, Department of Computer Science, Utrecht University.

Manna, Z. (1974) *Mathematical Theory of Computation.* McGraw-Hill.

Marakakis, E. I. (1997) *Logic program development based on typed moded schemata and data types.* PhD thesis, Department of Computer Science, University of Bristol.

Morgan, C. C. (1988) Procedures, parameters and abstraction: Separate concerns. *Sci. Comput. Program.* **11**, 17–27.

Morgan, C. C. (1994) *Programming from Specifications* (2nd ed). Prentice Hall.

Morgan, C. C. and Robinson, K. A. (1987) Specification statements and refinement. *IBM J. Res. & Dev.* **31**(5).

Morris, J. M. (1987) A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, **9**(3), 287–306.

Nelson, G. (1989) A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, **11**(4), 517–561.

Nickson, R. and Hayes, I. (1997) Supporting contexts in program refinement. Science of Computer Programming, **29**, 279–302.

Paulson, L. C. (1994) *Isabelle – a generic theorem prover.* LNCS 828. Springer-Verlag.

Read, M. G. and Kazmierczak, E. A. (1991) Formal program development in Modular Prolog: A case study. In: Clement, T. P. and Lau, K.-K. (eds.), *Proc. of LOPSTR'91.* Workshops in Computing, pp. 69–93. Springer Verlag.

Somogyi, Z., Henderson, F. J. and Conway, T. C. (1995) Mercury, an efficient purely declarative logic programming language. In: Kotagiri, R. (ed.), *Proceedings 18th Australasian Computer Science Conference*, pp. 499–512. Glenelg, South Australia.

Spivey, J. M. (1992) *The Z Notation: A reference manual* (2nd ed). Prentice Hall International.

Ward, N. (1994) *A refinement calculus for nondeterministic expressions.* PhD thesis, Department of Computer Science, University of Queensland.