A LOGIC FOR CORRECT PROGRAM DEVELOPMENT

Joseph L. Bates

TR 81-455
March 1981

Department of Computer Science
Cornell University
Ithaca, New York 14853

This research was supported in part by grants from the National Science Foundation.

This report is a revised version of a dissertation submitted in August 1979 to the Department of Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

A Logic for Correct Program Development*

Joseph L. Bates

TR 81-455
March 1981

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This research was supported in part by grants from the National Science Foundation.

This report is a revised version of a dissertation submitted in August 1979 to the Department of Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

Existing verification technology, though theoretically adequate, is not directly applicable to the construction of large software systems. This thesis explores the view that reasoning about code is not the proper paradigm for correct program development. Instead, specifications should be the objects of study and a logic should be formulated for constructively proving that specifications have acceptable implementations; from these proofs code may be extracted. Thus, constructive existence proofs become the programmer's main concern, while executable text is seen as a valuable by-product of correct reasoning which cannot be produced from incorrect reasoning.

The thesis captures this view of program development in a logic for the formal refinement of specifications. Specifications are written in an imperative notation of generalized assignment; they allow calculations in integer arithmetic and finite set theory. Classical reasoning techniques are shown inconsistent in this domain (where propositions may claim the constructive existence of programs), hence an alternative logic is developed based on intuitionistic reasoning. Proofs in this constructive refinement logic are trees, stylistically similar to those of Gerhard Gentzen's sequent calculus. It is argued that while linear proofs are appropriate when reasoning is developed and presented on paper, hierarchical proofs are appropriate when reasoning is developed and presented with machine aid. A mechanism is described that extracts correct code from valid proofs; its existence assures the

consistency of the logic. Finally, several code optimization techniques are examined and applied to code extracted from sample proofs.

The thesis concludes with a discussion of the expounded view of correct program development, suggestions for a program development system based on this view, and a look at the numerous research problems remaining in this area.

Table of Contents

Chapter 1

Introduction

1.1 The software problem

With the advent of the silicon integrated circuit in the early 1960's, the fundamental electronics technology that underlies modern computers began doubling in effectiveness every year [Siewiorek et. al. 1978, Bhandarkar and Juliussen 1978]. Prior to this technological achievement, machines were limited, society's expectations modest, and potential applications of computers were adequately realized. However, as exponentially improving hardware encouraged increasingly complex applications, the burden of producing correct, efficient, ever larger systems came to rest on the programmer. Too often the programmer failed miserably.

As hardware advances continued into the 1970's, the growing gap between the expectation and reality of software development prompted computer scientists to examine the programming task in depth. Dijkstra wrote the frequently mimicked "GOTO Statement Considered Harmful" letter [Dijkstra 1968a] advancing the idea that the goto statement led to programs with "intellectually unmanageable" control flow. The term "software engineering" was coined to convey the notion that programs are structured objects to be constructed according to sound engineering principles [Randell and Naur 1968]. Programming language semantics took on a new significance as Hoare's axiomatic method provided an effective tool for analyzing both the complexity of languages and the properties of individual programs [Hoare 1969]. These ideas grew into several broad areas of research aimed at solving the software problem.

Languages:

Programming language designers realized that languages must aid, not burden, the programmer [Hoare 1973, Wirth 1974]. Languages were developed to support the major intellectual techniques for handling complexity -- abstraction, case analysis, induction, etc. -- without themselves becoming too complex to understand. Study of such areas as data abstraction, control abstraction, scoping, and modularity led to proposals for various supporting language constructs.

Methodology:

After considerable debate, the "code then debug" philosophy gave way to the idea that programs could and should be constructed to function properly the first time executed [Mills 1975]. Top down design from well specified requirements became an accepted development technique. Programmers recognized that testing alone could only mildly increase their confidence that a program was properly constructed; assured correctness had to be built in, it couldn't be tested in.

Verification:

The notion of program correctness was formalized in various ways and techniques were developed to prove programs correct.

(Correctness means performing according to specifications. Assuring the aptness of specifications is an important, but distinct, problem.) These ideas affected programming languages and methodology; they led to the suggestion that programs and their proofs be developed simultaneously. Systems were built to support program verification with varying degrees of automated aid.

As of the late 1970's, many of these advances in programming languages and methodology have had an impact on the construction of real world software. Top down development and structured programming have been adopted by a wide variety of software producers. Articles proposing or analyzing other techniques appear constantly in the data processing industry's trade journals. Myriad introductory programming texts describe top down development in PL/I subsets or PASCAL. "Structured programming" preprocessors for FORTRAN, COBOL, and other old languages seem as common as the compilers themselves.

In contrast, though the idea has been in the technical literature for over a decade, formal verification of programs has not had a direct impact on real world software production. That is, people almost never carry out complete formal verifications of their programs. For interesting algorithms, existing formal proof techniques and their informal counterparts are adequate. These short programs, often published, are subject to the detailed scrutiny of many individuals. Real programs, however, are often so dull that only the original programmer ever reads them. Existing techniques, though theoretically adequate, are exceedingly tedious in this domain, even if mechanical aid is supplied. Thus, one finds that enforced formal verification with current technology is unacceptable in practice; yet, because real world programs are too large to be fully tested and are produced by programmers with a persistent tendency to err, formal reasoning of some kind (checked by machine) is exactly what is needed to guarantee the correctness of large programs.

1.2 Toward a solution

The work presented herein is based on the belief that current verification technology is inadequate largely because it fails to directly support effective methods of program development. Most verification techniques are analytic tools, applicable to arbitrary code segments divorced from their development; similarly, most verification systems analyze arbitrary programs and yield a set of lemmas (verification conditions) that must be proven. In my experience, however, careful, accurate programmers having low error rates develop correct programs with informal reasoning -- they seldom verify completed programs with formal reasoning. Though formal techniques for analyzing code can help guide informal development .e.g., Dijkstra's use of predicate transformers [Dijkstra 1976], verification systems based on these techniques are code readers -- they can only insure correctness by analysis performed after code is written. In contrast, this thesis claims the need for a habitable formal logic and associated support system for correct program construction.

A logic for program construction should formalize a particular development methodology, not just the semantics of a conventional programming language. The text produced by a programmer should contain the complete record of an idealized (i.e., no backup) development of his program, including all reasoning needed to demonstrate its correctness. Programmers are generally aware of this reasoning during development of correct code. Providing a formal notation for recording it permits program and proof synthesis to be integrated into one unified process, with correctness of the evolving program insured at all stages. This approach formalizes the methodology advocated by many authors of structured programming texts [Wirth 1973, Conway and Gries 1975, Dijkstra 1976].

Establishing program correctness is viewed differently from this vantage. Verification as a distinct activity no longer exists since incorrect programs cannot be produced. Executable code is obtained only by means of correct formal development. This change in viewpoint is of great psychological importance because it captures the idea of programming as a methodical process of discrete, manageable decisions in which the concepts of bugs, quick and dirty programs, etc., never arise. When constructing programs to meet given specifications, I believe limiting one's thought in this way is not just appropriate, but essential.

A program development logic requires programmers to manipulate substantially more text than is usual. Also, formal reasoning has an advantage over rigorous reasoning only when it is mechanically checked for validity. Thus, an interactive system that assists and watches the programmer is an indispensable part of this approach. The minimal version of such a system must provide clerical help in manipulating program developments and should check the validity of reasoning used therein. In addition, the system should take over the work of proving the mass of trivial propositions found in developments. The constantly increasing ratio of software to hardware costs has made any such assistance economically attractive, even at the expense of substantial dedicated computing power.

The view of program development advocated here clearly contradicts the well known argument that "programs should not be written at terminals". The reasoning behind that claim applies to conventional (edit-run-debug) systems, because they effectively support only the "modify until all the test cases work" methodology. My proposal is for a system supporting a more tenable methodology, requiring and helping the programmer to reason about all possible executions. I envision construction proceeding via a man-machine dialogue, perhaps a variation of that suggested by Floyd [Floyd 1971]. The dialogue would emphasize correct reasoning in a style suited to expressing complete idealized developments.

1.3 Organization of the thesis

The body of this thesis is the presentation and analysis of PRL, a logic for the top down development of programs. PRL, an acronym for Program Refinement Logic, is a formalism that provides for the specification and correct refinement of programs operating on integers, booleans, and well typed, homogeneous, finite sets. The logic is the result of my attempt at the "habitable formal logic ... for program construction" called for in the previous section.

Chapters 3 and 4 contain the major technical results of my work. Chapter 3 describes PRL in depth; it includes examples dispersed throughout the text. Chapter 4 presents a detailed account of a method for extracting (i.e., compiling) programs from proofs. Inherent in this account is a demonstration of the logic's consistency.

Though I have tried to limit the bulk of the thesis, chapters 3 and 4 have steadfastly resisted my efforts. In chapter 2, the "casual" reader will find the general ideas of refinement logics, motivated and developed from previous work on program correctness. Chapter 5 first summarizes the three fundamental ideas that underlie PRL, then enumerates a variety of areas subject to future study. Thus, chapters 2 and 5 provide an overview of refinement logics in the context of past and suggested research.

Chapter 2

Correct Program Development by Formal Refinement

2.1 Some existing approaches to correctness

I shall now discuss a variety of methods aimed at helping people produce correct imperative programs. These methods range from the early work on formally defining correctness to automatic systems for synthesizing correct code from specifications. They roughly fall into three catagories: development methodologies, formal logics, and computer based programming support systems.

The fundamental idea in establishing correctness is that executable programs may be interpreted as defining classes of computations. One can analyze a given program to determine its meaning, that is, the class of computations it performs. A program is said to be correct exactly when its determined meaning is consistent with a separate specification of its intended meaning.

Early workers on program verification [Naur 1966, Floyd 1967] developed a method of assigning predicates, called inductive assertions, to various points in the control flow of a program so that every time execution reaches any particular point the corresponding assertion is true. By choosing assertions appropriately and examining all execution paths between each pair, the program may be proved correct, in that the *output* assertion will hold at termination whenever the *input* assertion holds initially. The pair of input/output assertions forms a specification of the program. For example, consider the following asserted program segment to sum the first N positive integers.

```
{N≥1}
S ← 1;
K ← 1;
{1≤K≤N ∧ S=sum of 1 up to K}
while K≠N do
    {1≤K<N ∧ S=sum of 1 up to K}
    K ← K+1;
    S ← S+K;
    {1≤K≤N ∧ S=sum of 1 up to K}
od
{S=sum of 1 up to N}
```

The assertions are boolean expressions of the variables of the program. They are written in curly braces and serve to describe the state of the variables when control passes by the assertion. The specification for the program segment is that if N≥1 initially then S will be the sum of 1 up to N when the segment completes.

Hoare extended the idea of adding assertions to programs into a logical system that provides both a formal definition of the semantics of several language constructs and a method of reasoning about individual programs [Hoare 1969]. This *axiomatic* technique defines each language construct by describing the relationship that must hold between assertions placed before and after any instance of the construct. Thus, the meaning of an assignment statement $x \leftarrow e$ is defined by $\{P_{x\leftarrow e}\}\ x\leftarrow e\ \{P\}$, where $P_{x\leftarrow e}$ is the assertion P with all free occurrences of x replaced by e. (Roughly, an occurrence of x is free in P if it represents the value of the variable x global to P. Thus, in $x=5 \vee (\exists x)(f(x)=5)$ the first occurrence of x is free but the remaining two are not, they are bound occurrences of x.) For example, $\{x+2=3\}\ x\leftarrow x+2\ \{x=3\}$ and $\{1=2\}\ x\leftarrow 1\ \{x=2\}$ are both valid since the truth of the _precondition_ (to the left) before executing the assignment insures the truth of the _postcondition_ (to the right) afterward. In a correct asserted program, if a precondition is equivalent to _false_ (as in the latter example) then control will never reach the succeeding statement.

The axioms that characterize simple statements are augmented by inference rules that describe composite constructs, thus:

selection
$$\frac{\{P \wedge B\}\ S_1\ \{Q\},\quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{Q\}}$$

iteration
$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{P \wedge \neg B\}}$$

(P is called an _invariant_ because it is true before and after every iteration of the loop.)

consequence
$$\frac{P \Rightarrow P',\quad Q' \Rightarrow Q,\quad \{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}}$$

where the horizontal line indicates that the conclusion below follows from the hypotheses above. Since Hoare's original work, much attention has been directed at axiomatizing a wide variety of additional language constructs [Clint and Hoare 1972, Hoare 1972a, Hoare 1972b, Clint 1973, Hoare and Wirth 1973, Owicki and Gries 1973, Shaw et. al. 1976].

A program may be formally shown correct by providing assertions throughout that are related by the axioms and inference rules of the logic. The following shows a fully asserted program to search a sorted array. It is correct according to the above rules.

let sorted≡(∀i,j)(1≤i≤j≤N⇒A[i]≤A[j])

and in(i,j)≡(∃k)(i≤k≤j ∧ A[k]=X)

the total specification for the program is

    {N≥1 ∧ sorted}

    •
    •
    •

    {in(1,N)⇒A[K]=X ∧ ¬in(1,N)⇒K=0}

```
{N≥1 ∧ sorted}
{1≤1≤N≤N ∧ sorted ∧ in(1,N)⇒in(1,N)}
L←1;
{1≤L≤N≤N ∧ sorted ∧ in(1,N)⇒in(L,N)}
H←N;
{1≤L≤H≤N ∧ sorted ∧ in(1,N)⇒in(L,H)}
while L<H do
    {1≤L≤H≤N ∧ sorted ∧ in(1,N)⇒in(L,H) ∧ L<H}
    {1≤L≤FLOOR((L+H)/2)<H≤N ∧ sorted ∧ in(1,N)⇒
        ( [A[FLOOR((L+H)/2)]<X∧in(FLOOR((L+H)/2)+1,H)]
        ∨ [A[FLOOR((L+H)/2)]≥X∧in(L,FLOOR((L+H)/2))] )}
    K←FLOOR((L+H)/2);
    {1≤L≤K<H≤N ∧ sorted ∧ in(1,N)⇒
        ([A[K]<X∧in(K+1,H)] ∨ [A[K]≥X∧in(L,K)])}
    if A[K]<X then
        {1≤L≤K<H≤N ∧ sorted ∧ in(1,N)⇒
            ([A[K]<X∧in(K+1,H)] ∨ [A[K]≥X∧in(L,K)]) ∧ A[K]<X}
        {1≤K+1≤H≤N ∧ sorted ∧ in(1,N)⇒in(K+1,H)}
```

```
        L←K+1;
        {1≤L≤H≤N ∧ sorted ∧ in(1,N)⇒in(L,H)}
    else
        {1≤L≤K<H≤N ∧ sorted ∧ in(1,N)⇒
            ([A[K]<X∧in(K+1,H)] ∨ [A[K]≥X∧in(L,K)]) ∧ ¬A[K]<X}
        {1≤L≤K≤N ∧ sorted ∧ in(1,N)⇒in(L,K)}
        H←K;
        {1≤L≤H≤N ∧ sorted ∧ in(1,N)⇒in(L,H)}
    fi
    {1≤L≤H≤N ∧ sorted ∧ in(1,N)⇒in(L,H)}
od
{1≤L≤H≤N ∧ sorted ∧ in(1,N)⇒in(L,H) ∧ ¬L<H}
{1≤L≤N ∧ in(1,N)⇒A[L]=X}
if A[L]=X then
    {1≤L≤N ∧ in(1,N)⇒A[L]=X ∧ A[L]=X}
    {in(1,N)⇒A[L]=X ∧ ¬in(1,N)⇒L=0}
    K←L;
    {in(1,N)⇒A[K]=X ∧ ¬in(1,N)⇒K=0}
else
    {1≤L≤N ∧ in(1,N)⇒A[L]=X ∧ ¬A[L]=X}
    {in(1,N)⇒A[0]=X ∧ ¬in(1,N)⇒0=0}
    K←0;
    {in(1,N)⇒A[K]=X ∧ ¬in(1,N)⇒K=0}
fi
{in(1,N)⇒A[K]=X ∧ ¬in(1,N)⇒K=0}
```

Certainly the above text is lacking in readability, but human engineering concerns aside, there are three technical ways in which the asserted text shown fails to capture one's intuitive understanding of the program's correctness. First, the program need not terminate -- the output assertion will hold if control reaches it, but it has not been shown that this will necessarily occur. The difficulty lies in the iteration rule; it guarantees only _partial_ correctness. To show _total_ correctness, i.e., that the program will indeed terminate with the output assertion true, a hypothesis must be added to the iteration rule placing some further constraints on the loop body. The usual technique is to require each execution of the loop body to decrease the value of some integer expression that must remain strictly positive for continued iteration. Thus, a total correctness iteration rule might be written

$$\underline{\text{iteration}} \quad \frac{P\wedge B\wedge t=t' \ \{S\}\ P\wedge t<t', \quad t\leq 0 \Rightarrow \neg B}{P\ \{\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\}\ P\wedge\neg B}$$

where t is an integer expression of the program variables and t' is a new name not occurring elsewhere in the proof. In the search program loop one could choose $t\equiv H-L$ since $H-L\leq 0\Rightarrow\neg L<H$ and the body will bring H and L closer at every iteration. Several recent formalisms, recognizing this limitation of Hoare's original approach, are based on total correctness [Dijkstra 1976, Constable and O'Donnell 1978].

The second failure of the asserted program method is that program segments cannot conveniently be restricted to changing only selected variables. Thus, the specification

$\{N\geq 1 \wedge \text{sorted}\}$
.
.
$\{in(1,N)\Rightarrow A[K]=X \wedge \neg in(1,N)\Rightarrow K=0\}$

is satisfied by a program that sets all N elements of A to one and K and X to zero. The usual solution here is to introduce logical constants that cannot be referenced by the program. These may be used to record the initial values of any program variables that should not change. The complete specification for the search example is then

$\{N\geq 1 \wedge \text{sorted} \wedge X=X' \wedge N=N' \wedge (\forall i)(1\leq i\leq N\Rightarrow A[i]=A'[i])\}$
.
.
$\{X=X' \wedge N=N' \wedge (\forall i)(1\leq i\leq N\Rightarrow A[i]=A'[i]) \wedge$
$in(1,N)\Rightarrow A[K]=X \wedge \neg in(1,N)\Rightarrow K=0\}$

where X', N', and A' are logical constants. A difficulty with this solution is that every assertion in the program must include clauses stating the invariance of selected variables. With top down development each program contains independent segments that have their own specifications; the proofs of these segments are then permeated with assertions about the invariance of global variables that are not referenced and should not even be considered within the segments.

The final problem with asserted programs is that uses of the rule of consequence are not fully justified in the asserted text. In the search example, each occurrence of two adjacent assertions corresponds to a logical implication that must hold for the proof to be valid. These implications, called verification conditions (VCs), are assumed to be proven in text external to the asserted program using the theory of the underlying logic (e.g., integer arithmetic). Since these proofs are necessary to demonstrate the correctness of the program, and often contain reasoning essential to understanding it, I believe they should be an integral part of the program text. As will be shown, PRL overcomes these technical problems of Hoare's logic.

Floyd's early work on the inductive assertion method has been used as the basis for many verification systems [King 1969, Good 1970, Deutsch 1973, Bledsoe and Bruell 1974, Waldinger and Levitt 1974, Igarashi et.al 1973, von Henke and Luckham 1975, Good et.al. 1975]. These systems typically require only that the user provide invariants to cut every loop; they automatically generate the necessary verification conditions [King 1976] which are sent to a theorem prover that attempts to complete the verification, sometimes with interactive user guidance. The fundamental approach to correctness here is "after the fact" verification in which completed programs are submitted for VC generation and proof. Thus, these systems are completely independent of the methods used to construct input programs.

Many researchers interested in correctness find existing verification systems lacking. Their objections divide them into two distinct groups, however. One group finds that the information users must supply to allow a posteriori verification is too great a burden; their goal is to increase the power of mechanical verifiers as much as possible so that all internal assertions can be automatically generated and proven from the specifications and code [Elspas 1974, German and Wegbreit 1975, Katz and Manna 1976]. The other group rejects a posteriori verification as a means of achieving correctness; rather, they argue that programs should be constructed along with their proofs in such a way that correctness is assured at all stages of development

[Dijkstra 1968b, Mills 1975, Dijkstra 1976, Gries 1976]. Study of this constructive approach to correctness, generally considered programming methodology, has yielded informal techniques as well as psychological attitudes that enable programmers to greatly increase their productivity [Mills 1975]. This thesis is based in part on the belief that automated aids for insuring correct software should support these techniques for correct program development rather than after the fact verification.

Dijkstra's calculus of guarded commands is a useful mathematical tool that aids in the formal development of correct programs [Dijkstra 1975a]. In his recent book each program is developed with an informal discussion that makes heavy use of formal techniques, but the correct programs that result are pure code with none of their correctness proofs contained in the program text [Dijkstra 1976]. I see this as a significant fault; since proof and program should be developed as a unit, so should they be presented.

In contrast, PL/CV is a formal logic whose well-formed formulas are correct program/proofs [Constable and O'Donnell 1978, Constable and Johnson 1979]. These combined objects are a variation of asserted programs in which all the reasoning needed to establish total correctness is combined with the program text in a relatively natural, readable way. The formality of the logic allows reasoning to be checked mechanically, thus justifying a high degree of confidence in the correctness of programs.

PL/CV is a significant step in the solution of the software crisis, yet it too is only a step. Its major failing is the lack of a methodology for producing program/proofs in stages, as the constructive approach calls for. PRL is the result of an attempt to remedy this failing by formalizing a logic of program development, not just programs, where each well formed formula describes a mechanically verifiable idealized top down development of a program. Such a logic should be a sound foundation from which to build automated interactive systems that aid in the construction of programs.

The interactive development of correct programs is an attractive idea that has been discussed before. Floyd presents an imaginary dialogue between man and machine that leads to a search program similar to the example above [Floyd 1971]. Though quite brief, his discussion provides much worth considering and clearly demonstrates the potential benefits of the system he envisions. Moriconi has developed a system for incremental development that aids the user in determining the effects of changes to specifications, code, and proofs when using the embedded VC generation based verifier [Moriconi 1977]. A limited facility is provided for partially refined programs and top down development, but Moriconi is mostly concerned with an intelligent natural language interface and reducing verification effort by keeping intact any work still valid after given changes are made. A recent paper from MIT describes the theory and initial construction of a programmer's apprentice that aids experts in the development of LISP programs [Rich and Shrobe 1978]. The proposed system "knows" enough about the programming task and specific programs to watch development, notice inconsistencies, suggest corrections, write programs by instantiating abstract schema, and answer questions. Though still under development, the LISP apprentice appears to be an important step toward tbe realization of Floyd's ideas.

A somewhat different approach to correct program development is taken in work on correctness preserving transformations [Gerhart 1975a, Gerhart 1975b, Buratall and Darlington 1975, Balzer et.al. 1976, Wegbreit 1976, Schwartz 1977]. Here much of the complexity of real programs is thought to result from the need for efficient execution. Thus, a methodology is proposed whereby the program is first written in as simple a form as possible, so that correctness is easily shown, and then correctness preserving optimizing transformations are applied to produce an acceptably efficient equivalent. Gerhart further proposes that many common program constructs, e.g., search loops, could be abstracted, proven correct, and listed in a handbook of valid schema. Large segments of the initial (simple) program might then be formed by combining appropriate instances of these established schema. A catalog of program transformations has been developed to support this methodology [Standish et.al. 1976].

In its pure form, the program transformation approach is limited by the ability of programmers to express short, clear solutions to real programming tasks. It seems to me that the approach would benefit from the introduction of a specification language of substantially greater power than the programming language. Creating a fully recorded transformation from initial specification to final code is similar to constructing a PRL program development. There are two main differences. First, PRL restricts the class of transformation rules to refinements of individual specifications, guaranteeing that each development is a simple hierarchy of transformations. Second, PRL specifications may be statements of pure logic, rather than only program specifications. These logical statements carry constructive (executable) meaning, thus, correctness and execution are highly intertwined concepts (see section 2.4).

The final area of correctness research that will be considered is [illegible] programming. The goal is to automate the programmer, and possibly the analyst, out of existence. There are two overall phases in the use of an automatic programming system, though they may overlap in time: problem acquisition and program synthesis. Some researchers, studying the simpler task, describe techniques for synthesis that start from some formal specification of the problem [Manna and Waldinger 1977, Long 1977, Manna and Waldinger 1978]. A few more ambitious researchers are attempting the construction of synthesis systems capable of acquiring the problem specifications directly from conversation in natural language with relatively naive users, much as would a programmer/analyst [Balzer et.al. 1974, Green 1976]. Though Dijkstra considers efforts at producing such systems "a theme that recurs with the same regularity as influenza epidemics" and the likely result of using them "design specifications produced as a kind of Pavlovian slobber" [Dijkstra 1975b], it seems probable that systems of this type would be extremely useful in bringing the power of computers to the public. In any case, many of the problems studied in automatic programming relate to similar concerns in conventional programming (e.g., specification languages, program synthesis methods, insuring correctness) and the work being done is of interest, even if successful automated systems are far off.

This completes my lengthy but necessarily incomplete review of existing work on the problem of producing correct imperative programs (see [London 1972, Elspas et.al. 1972, London 1975, Greenspan and Horning 1977, Barnard 1977] for further references). The last decade has seen a tremendous volume of work done in this area with the attendant publication of many hundreds of papers. I have tried to present enough background to enable the reader to understand and evaluate this thesis in the light of current knowledge; I will provide my own comments and analysis in the following pages.

2.2 The formal refinement approach to correctness

I have had the valuable opportunity to observe several programmers whose programs almost always work properly the first time executed. The most important aspect of their rare ability seems to be an attitude of precision that might be phrased "programming is a task in which I must carry out a series of discrete decisions and actions, each of which is simply acceptable or not; therefore, I can and will take care never to introduce a line of code whose correctness I am unsure of". Given this attitude, the appropriate time for machine aid in avoiding coding errors is not during program execution, nor is it during a post-development reading or verification. The appropriate time for machine aid is during program development.

In order to determine mechanically whether the construction of a program is proceeding along an acceptable path, one needs a formalism for expressing both the stages of partial development and the reasoning that justifies their correctness. The method of top down programming, in which problems are repeatedly broken down into a composition of smaller ones, has been proposed by many and found beneficial for constructing complex systems [Wirth 1971, Mills 1971, Liskov 1972, Baker 1972a, Baker 1972b]. I have chosen to explore a logic of formal top down development in which the well-formed formulas are correct partial refinements of programs from their specifications. The remaining three sections of this chapter informally describes the fundamental features of PRL in preparation for the formal development of chapter 3.

Consider the following top down description of the search program that was presented in section 2.1.

```
/* If X occurs in A[1:N] then set K so that A[K]=X
   otherwise set K to zero.  Assume N≥1 and A[1:N]
   is sorted in increasing order. */

 var L,H: integer;

 L←1;
 H←N;

 /* With the invariant (1≤L≤H≤N, A[1:N] is
    sorted in increasing order, and A[L:H] contains
    X if A[1:N] does), decrease H-L to zero. */

  while L<H do

      /* Split A[L:H] into nonempty parts
         and adjust L, H to bound a
         part of A that must contain X
         if A[L:H] itself does. */

        K←FLOOR((L+H)/2);
        if A[K]<X
            then L←K+1
            else H←K
        fi

  od

 if A[L]=X
     then K←L
```

```
        else K←0
    fi
```

Each English comment describes the function of the program text indented below it. In general, one may delete the refinements for any set of comments and obtain a correct, though incomplete, program. This provides a natural view of the text as a tree of specifications with the initial specification at the root and executable code at the leaves. The program may be read and understood level by level. Its correctness can be insured by checking the validity of each refinement step without regard for either surrounding text or lower levels of refinement. These two properties of refinement validity are most important: the freedom to ignore deeper refinements allows correctness to be checked continually during development, thus (in theory) permitting use of the attitude of precision; the freedom to ignore context insures that correctness is a local property of limited complexity, thus guaranteeing the practical feasibility of developing only correct programs. A formalization of top down description should maintain these essential properties.

The first step in creating a logic of program refinement is to capture the tree structure of developments. In PRL, each level of a development has the form

```
specification by
     .
     . refinement
     .
```

where the refinement specifies a set of subproblems and describes how solutions to these subproblems may be composed to form a solution to the original problem. All specifications are expressed in the same notation. A certain class, subject to direct implementation by the "compiler", may be written at the leaves of a completed development.

A specification is said to be valid when it defines a terminating computation. The easily implemented specifications that need no further refinement are the axioms of PRL. The rules of inference (allowing validity to be extended to other specifications) are formulated as a set of refinement rules that describe legal reasoning steps in formal developments. A successful development begins with a problem specification, proceeding by repeated application of refinement rules until all remaining unrefined specifications are axioms. Although backup and modification may occur during its construction, the final text describes a complete, idealized, top down development of a program meeting the original specification. Equivalently, the final text is a tree structured proof of the original specification's validity.

The informal example developed above leaves out much of the reasoning needed to assure correctness. In the formal system, however, each refinement should include the complete argument by which solutions of the subproblems may be combined to form a solution of the whole. Hence, refinements will include subproblems that are statements from the domain of pure logic. These will be proven in the same refinement style by which programs specifications are implemented. This style is a variation of Gentzen's natural deduction systems [Kleene 1952, Prawitz 1965, Szabo 1969] in which the structure of proofs is made explicit. I will use the term proposition to denote both program specifications and logical statements; the refinements of both are proofs. A program development is the constructive proof of its specification.

Since correctness should be a local property of the text, it is important in the development of a large proof that the validity of refinements not depend on dispersed information. This means that propositions must be stated in full without any hidden assumptions or dependency on surrounding text. Nevertheless, they must also be concise and readable, so PRL provides for a uniform user-defined global environment of definitions and theorems that can be explicitly referenced. Definitions allow the specification language to be extended to higher levels of abstraction; theorems serve to capture and name generally useful facts so that they can be used later without repeated proof. For instance, since a specification is proved by a program development, a theorem that claims a specification associates a name with a program development (i.e., a segment of code). Such a theorem is simply the definition of a procedure. The uniformity of the global environment allows non-local references to be interpreted without concern for their context.

The development of PRL has been confined to the study of imperative program segments whose only noticeable effects when executed are to modify the values of global variables. Though I believe the idea of formal refinement is applicable to other constructs, such as expressions, data types, and CLU iterators [Liskov et.al. 1977], these extensions are not herein discussed. Facilities for input/output are also omitted, though they may be modeled by introducing explicit input and output variables of appropriate types.

The specification for an imperative program segment must state the ay in which variables global to the segment are changed by its xecution and the _precondition_ under which the program segment may be xecuted to achieve the desired effect. I shall discuss the pecification language in depth in the next section; for now it is ufficient to assume these two components. Thus,

N>0 _set_ X to the minimum of A[1:N]

pecifies that the refinement only need apply if N is positive and then t should change only the variable X (to the minimum array element).

The following shows how the search example might look when xpressed (still informally and incompletely) according to the above iscussion.

```
def sorted ≡ (∀i,j st 1≤i≤j≤N)(A[i]≤A[j])
def in(i,j) ≡ (∃k)(i≤k≤j ∧ A[k]=X)

N≥1∧sorted set K st if in(1,L) then A[K]=X else K=0 fi by
    var L,H;
    N≥1∧sorted set L,H st 1≤L≤H≤N ∧ in(1,N)⇒in(L,H) by
        set L,H to 1,1;
    with invariant 1≤L≤H≤N ∧ sorted ∧ (in(1,N)⇒in(L,H))
     as H-L↓ while L<H do
     1≤L≤H≤N∧sorted set K,L,H to K',L',H' st
            L≤L'≤H'≤H ∧ H'-L'<H-L ∧ (in(L,H)⇒in(L',H')) by
        L≤H prove L≤FLOOR((L+H)/2)<H;
        set K to FLOOR((L+H)/2);
        1≤L≤H≤N∧sorted∧L≤K<H set L,H to L',H' st
              L≤L'≤H'≤H ∧ H'-L'<H-L ∧ (in(L,H)⇒in(L',H')) by
           1≤L≤H≤N∧sorted∧L≤K<H prove
                 if A[K]<X then in(L,H)⇒in(K+1,H)
                        else in(L,H)⇒in(L,K) fi
           if A[K]<X then set L to K+1
                 else set H to K fi
    1≤L=H≤N∧sorted∧(in(1,N)⇒in(L,H)) prove in(1,L)⇒A[L]=X
    if A[L]=X then set K to L else set K to 0 fi
```

The text of a development in PRL is not a program in the usual sense; rather, it is a proof presenting all the reasoning needed to establish that some program exists which meets a given specification. All such existence proofs are required to be constructive so that a compiler for the logic can extract a suitable program without undue difficulty. Aside from this restriction, however, there is considerable freedom in the choice of proof system employed. This work explores only a rather simple system in which the construction of programs from proofs is easily accomplished. However, I can imagine an interactive facility supporting semi-automated refinement in which a proof is any discussion or strategy that leads the system to an acceptable implementation of guaranteed correctness [Gordon et.al. 1977a, Gordon et.al. 1977b]. The final recorded text may be either the complete discussion (including dead ends) or only the reasoning actually needed to construct the implementation. I have chosen to study the latter variation, where texts are both idealized developments and static proofs.

This view of an entire development as a proof helps keep the programmer's attention on the logical soundness of his reasoning, not on the extracted program and its execution. It also assures that the development process is not programming followed by verification, but a unified activity of proof in a single logical system. In contrast, asserted program logics allow the development of code before much consideration is given to correctness. The very existence of executable code as an entity independent of any reasoning encourages the view that verification is a diversion from the true task of producing code. In reality, careful reasoning is the essence of programming. Avoiding a notation in which pure executable code can be expressed forces the shift, that I believe essential, away from the paradigm of verification by program analysis.

The natural method for producing PRL proofs is, of course, top down development. In such a development each refinement step represents a decomposition that the programmer feels is most appropriate and understandable. The pure logical propositions that must be proven arise only from local considerations of refinement correctness. Since they follow the top down development, rather than resulting from analysis of lower level code, they are expected and understood by the programmer. Thus, PRL avoids the problems of logics for program analysis in which programmers must often prove verification conditions involving assertions that have been "pushed through" large segments of program text. For example, in contrast to Dijkstra's weakest precondition method of code analysis [Dijkstra 1976], the top down derivation of specifications yields "appropriate" preconditions, i.e., logical formulas whose terminology and content are appropriate to their refinement level.

An apparent disadvantage of the PRL notation is that the hierarchical nature of proofs is not easily accommodated on paper. Reading and writing of tree structured proofs requires display of a node and its sons alone, without the descendants of the sons getting in the way. However, in an age of inexpensive and powerful computing hardware, there is reason to believe that the inherent limitations of paper are no longer relevant to programming logic design. Rather, advantage should be taken of the flexible modes of text structuring made possible by dedicated personal computers [Kay and Goldberg 1977, Nelson 1974]. One can envision the "smart editor" component of an interactive program development system conveniently allowing one to traverse hierarchical proofs, examining and creating them level by level. The presentation of PRL proofs in this exposition is hindered by the linear nature of the paper, but the reader should imagine that proofs are being manipulated on a display screen sufficiently large to present any single refinement step in full.

I would like to make one final comment on the refinement style of reasoning. Even before it was so apparent to me that PRL should be considered a logic for reasoning about specifications, I had decided to use refinement as the fundamental technique for structuring developments. When I needed to introduce a method for reasoning about logical formulas, I chose Gentzen's natural deduction method and transformed its inference rules into equivalent refinement rules. Subsequently, I have learned that in his later work Gentzen performed this very transformation himself. Though he still thought natural deduction was proper for presenting completed proofs, he developed the sequent calculus as a vehicle for studying decision procedures, i.e., methods for finding proofs [Kleene 1952, Szabo 1969]. If one considers the programmer as embodying a general decision procedure, Gentzen's arguments favoring the sequent calculus over natural deduction systems apply to the domain of program/proof development. With the advent of the technology discussed above, it is my belief that the refinement style is appropriate not only to proof development, but to proof presentation as well. This notion, one step beyond Gentzen's ideas, must await a PRL implementation for verification. If it is a valid view, it is applicable to more than the theory of program development; the refinement style applies to any proof system that might benefit from machine support.

2.3 The specification of imperative program segments

The question of correctness for a well specified program may be divided into two distinct concerns: first, that the program meets its specification and, second, that the specification adequately captures one's intuitive understanding of the program's intended behavior. The former is assured in PRL, since any executable code is always extracted from a correct proof ot the specification. The latter, however, cannot be guaranteed. Nonetheless, the validity of "correct" programs is limited by the aptness of their specifications. Thus, the design of a good specification language requires careful attention to the problem of obtaining appropriate specifications; it is insufficient that tbe language be technically complete (i.e., powerful enough to express any computable operation).

There are several ways to determine the aptness of a specification. One's intuitive understanding of the desired program behavior is often most precise for a set of specific inputs. In this case, the specification may be tested at selected points for conformity with the expected results. The known limitations of testing (that only a very small part of the full meaning of the specification is likely to be examined) apply here, of course. An alternative is to express the desired behavior in another form and prove the two specifications equivalent. However, this not only requires substantial additional effort, but reminds one of the philosophy of post-development program verification ("we can't get it right the first time so let's fix it up later"). Clearly, the point in producing a specification is to capture fully and precisely the ideas one has in mind, with the hope that after careful formulation the original specification is so concise and understandable that its correctness is evident from a simple reading. If this hope is to be realized, the specification language must permit the easiest and most natural (formal) expression of one's informal ideas.

The execution of an extracted program segment results in the change of a selected set of global variables. A PRL specification describes the way that these variables are accessed and modified. Thus, since a specification defines a mapping from initial to final state, it may be viewed as a very high level program. However, it need not be (efficiently) executable; indeed, a goal of the specification language is to separate completely concerns of correctness from those of performance, so that the complexity of specifications results from correctness requirements alone [Pratt 1977].

At each point of execution the _state_ of a PRL computation is quite simple: it is a mapping from program variables to values of the appropriate types. The meaning of a particular specification is a (nondeterministic) mapping, called a _transition_, from initial to final states. The comments used to specify transitions in the top down description of programs tend to be high level imperative statements, such as "set X so that ..." or "find the least Y satisfying ...". These imperatives frequently allow the specification of transitions in a more natural fashion than is possible with precondition/postcondition notation. For example, "increment X" would be written

$$\{X=X'\} \ldots \{X=X'+1\}$$

in the latter notation with the necessary introduction of a logical constant X' to record the old value of X. As was noted in section 2.1, the complete specification of transitions using assertions requires the introduction of logical constants to guarantee the invariance of variables that should not be changed. In an attempt to avoid the cumbersome task of carrying these assertions throughout proofs, I have chosen to base the PRL specification language on an imperative notation of generalized assignment. (As will be seen, this attempt has not entirely succeeded. Logical constants still enter into proofs, though less frequently. Some further approaches toward their elimination are discussed in chapter 5.)

The assignment symbol "$\leftarrow$" has the property that the name on the left represents the new value of a variable while the same name on the right represents the old value. Taking advantage of this interpretation, "increment X" is specified by writing "$X \leftarrow X+1$", requiring that any valid refinement increment X, but change no other variable. In general, a transition modifies the values of several variables; the corresponding transition specification has a tuple of names to the left of the arrow and a construct on the right that yields values for each variable. Thus, "$X,Y \leftarrow Y,X$" interchanges the values of X and Y.

In addition to conventional assignments, which define deterministic (functional) transitions, the specification language must permit the expression of nondeterministic transitions (those that do not have uniquely specified final states). Typically, this is accomplished by characterizing the final state with a predicate. One might apply this technique here by allowing transition specifications of the form

$$X \leftarrow X' \text{ such that } P(X,X')$$

where X' is a new name that denotes the final value of X. My interest in specification languages has led me to study a generalization of this form that retains the imperative flavor of conventional assignments. This specification notation is largely independent of other aspects of refinement logics; I view it as an orthogonal excursion into methods of program specification.

Nondeterministic transitions are specified by placing to the right of the arrow a construct that produces a set of (tuples of) values. Such a generalized assignment characterizes any program segment that places one of the generated (tuples of) values in the variables. Since set generating constructs of this sort are used in several contexts throughout PRL, I shall digress briefly to discuss their form and meaning.

A generator is a construct that evaluates to an unordered, nonrepeating group of values. For example, a set valued expression may be formed by placing a generator inside of the set brackets {...}. All the values produced by a generator are of the same type; the type may be determined easily by examination of the proof text. Generators include, among others, the forms:

expression	denoting a single value.
expression **to** expression	denoting a closed range of values.
$\in$ set-expression	denoting the elements of a set.
type	denoting all the values of a type.

Some contexts require a generated group of values with a locally defined bound variable ranging over elements of the group. The construct serving this function is called an [illegible]. It is used, for instance, in quantified boolean formulas such as $(\forall x$:1 **to** 50)P(x), where the bound variable x (ranging over the first fifty natural numbers) is accessible from P(x). Though generators simply yield sets of values, iterators may define several variables that are simultaneously bound to distinct values. For example, $(\exists x{:}\in S, y{:}\in T)P(x,y)$ contains an iterator in which the pair of bound variables x,y range over the cross product of sets S and T. Thus, in general, an iterator denotes a tuple of bound variable names ranging over an unordered, nonrepeating group of tuples of values. They include, among others, the forms:

var : generator	denoting a variable ranging over all values of a generator.
iterator **st** boolexp	denoting an iterator filtered to range only over values satisfying a boolean expression.
iterator , iterator	denoting the cross product of iterators.

One may use an iterator with one bound variable in place of a generator. A set of small prime numbers could be formed in this way by writing {p: 1 to 100 st prime(p)}.

Generators and iterators are presented formally in section 3.2. My development of them is based on ideas present in VERS2 [Earley 1973, Earley 1974a] and SETL [Schwartz 1975]. Their precise meaning depends on the context in which they are used, but the above informal description should suffice for the remainder of this chapter. I shall now return to their use in the specification of transitions.

The general form of a transition specification is

$$var_1,\ldots,var_n \leftarrow \textbf{any}\ iterator$$

where the iterator defines n bound variables and $var_1,\ldots,var_n$ are all distinct names. PRL is a language of pure values; it provides no structured variables. In conjunction with restrictions on procedure invocations, this assures that distinct names are never aliases for the same variable. Any valid refinement of the above transition specification must set $var_1,\ldots,var_n$ to some n-tuple that the iterator would produce (if evaluated), but must not modify any other variable. The reader may consider the following specifications and their informal interpretations.

$x \leftarrow$ **any** $b{:}\epsilon S$

Set x to an arbitrary element of S.

$x,y,z \leftarrow$ **any** (a:**int**, b:**int**, c:**int**) **st** $a,b,c>0 \wedge a^n+b^n=c^n$

Find positive values for x,y,z that satisfy $x^n+y^n=z^n$ (without changing n).

closure $\leftarrow$ **any** S:**set**(edge) **st** nodes(S)$\subset$nodes(G) $\wedge$ ($\forall$e:edge)
(e$\in$S$\Leftrightarrow$($\exists$p:path)(first(p)=tail(e)$\wedge$last(p)=head(e)))

Set closure to the transitive closure of graph G.

A transition specification, interpreted as a proposition, claims the existence of some program that changes variables in the specified way. Two aspects of such a claim are not presented in the transition specification, however. They must be provided by extending the form of propositions.

First, given a specification, a program might exist only under suitable restrictions on the initial values of variables. In this case, the proposition is written as an implication mediated by the symbol **pr** (prove), as in

S$\neq\emptyset$ **pr** x $\leftarrow$ **any** b:$\in$S.

In fact, for syntactic uniformity, the refineable propositions of PRL always take the form

hypothesis,...,hypothesis **pr** conclusion

where the hypotheses are boolean expressions and the conclusion is either another boolean expression or a transition specification.

Second, any program implementing a given specification may access only those variables declared in surrounding text. These variable names, together with their types, are collected into an <u>environment</u> for the proposition. The well-formedness of the proposition is determined by analyzing its hypotheses and conclusion in its environment. At the start of a refinement, the environment is small. It contains only those names needed to state the original problem. As refinement proceeds, new variables may be declared and added to the environment of lower level propositions. Alternatively, variable names may be dropped from environments to eliminate clutter or assure modularity of the proof. Each PRL refinement rule describes the environments of its subgoals in terms of the environment of its goal. Thus, environments carry global information downward into proofs with modifications at each refinement step determined by the choice of refinement rule.

These three components, hypotheaes, conclusion, and environment, constitute each refineable proposition. They are formally developed in chapter 3. However, the above background will suffice for an introduction to the PRL refinement rules.

2.4 <u>Refinement rules and the semantic content of proofs</u>

I shall now turn from the specification language to methods for reasoning about specifications. These methods, the refinement rules, enable one to extend the class of valid formulaa from the axioms by successively developing propositions of increasing complexity. The rules deacribed in this thesis formalize reasoning in the domains of first order predicate calculus, integer arithmetic, set theory, and imperative programs. They are briefly discussed below, as motivation for their formal introduction in chapter 3.

Following Gentzen's ideas on natural deduction systems [Kleene 1952, Constable and O'Donnell 1978, Prawitz 1965, Szabo 1969], I have divided the refinement rules into two classes according to their treatment of the goal proposition: the synthesis rules derive subgoals from the conclusion, the analysis rules derive subgoals from hypotheses. This division is somewhat fuzzy, indeed all the rules are of equal semantic standing and of similar form, but it aids one's memory and provides convenient terminology.

Consider the synthesis rule for a conclusion that is a conjunction ($\wedge$synthesis):

$$\begin{array}{ll} \wedge s \qquad & S \text{ pr } A \wedge B \text{ by} \\ & \quad S \text{ pr } A \\ & \quad S \text{ pr } B \end{array}$$

where A,B are boolean expressions and S is a set of hypotheses. This rule is one of a class for proving structured logical conclusions. In this case, the proof of A$\wedge$B from S breaks into two independent proofs, that of A from S and that of B from S. If these two propositions can be refined, then the original conclusion, A$\wedge$B, follows from the hypothesis set S.

A similar rule, $\Rightarrow$synthesis, is as follows.

$$\begin{array}{ll} \Rightarrow s \qquad & S \text{ pr } A \Rightarrow B \text{ by} \\ & \quad S, A \text{ pr } B \end{array}$$

This allows the implication's antecedent to be brought over into the hypothesis set. If B follows from S and A, then A$\Rightarrow$B follows from S.

Besides producing composite logical formulas, the synthesis rules refine transitions specifications into imperative control structures such as looping and sequencing. The following example is a valid instance of the composition rule, presented with other transition refinement rules in section 3.7.

$x \neq 0$ **pr** $x,y \leftarrow 1/x^2, 1/x$ **by**

$x \neq 0$ **pr** $y \leftarrow 1/x$

pr $x \leftarrow y*y$

Corresponding to ^synthesis there is an ^analysis rule for refining a proposition whose hypothesis set contains a conjunction:

^a $S, A \wedge B$ **pr** F **by**

S, A, B **pr** F

where A,B,S are as above and F is a transition specification or a logical formula. This rule allows one to replace the hypothesis $A \wedge B$ by the two distinct hypotheses A and B. With the remaining analysis rules, it allows the proof of propositions containing structured hypotheses. Two further examples follow.

va $S, A \vee B$ **pr** F **by**

S, A **pr** F

S, B **pr** F

<u>simplified $\exists$a</u>

S,($\exists$x:int)P(x) pr F by

S,P(c) pr F

where c is a new name ot type integer

The first rule, or analysis, permits the conclusion that F follows from A$\vee$B if it follows from A and also follows from B. The second rule, exists analysis, permits the conclusion that F follows from ($\exists$x)P(x) if it follows from P(c) for some fixed c (of which nothing else is known).

Executable programs must be automatically extractable from proofs of propositions whose conclusions are transition specifications. That is, proots of the existence of programs must be constructive. This is insured by associating with each refinement rule an inverse operation that combines the solutions of the subgoals into a solution of the goal. Starting with the solutions of axioms, which are provided by the "compiler", the extraction process proceeds up the proof tree inductively until an implementation is produced for the proposition at the root. Though this method is intuitively clear, I shall demonstrate that treating transition specifications as propositions introduces some subtlety into the logic and the extraction process. In particular, the classical semantics for boolean expressions must be abandoned in favor of a constructive interpretation.

Consider an instance of $\wedge$a in which the conclusion is a transition specification, T.

S, $A \wedge B$ **pr** T **by**

S, A, B **pr** T

The goal describes a program that computes T whenever S and $A \wedge B$ are true. A solution to the subgoal provides just such a program, since the truth of $A \wedge B$ guarantees that of A and that of B, so extraction from this rule is trivial.

Extraction from $\vee$a, however, presents problems. Choose a specific Turing machine, M, and consider an instance of $\vee$a in which S is empty, A is "M halts", B is "M diverges", and F specifies a program which yields "1" if M halts and "0" otherwise. A, B, and F are all easily formalized in any sufficiently powerful logic (including PRL). Thus, one has the following instance of $\vee$a.

M halts $\vee$ M diverges **pr** x ← 1 if M halts, 0 otherwise **by**

M halts **pr** x ← 1 if M halts, 0 otherwise

M diverges **pr** x ← 1 if M halts, 0 otherwise

Clearly, the first subgoal is satisfied by the simple program x←1, while the second subgoal is satisfied by x←0. Further, the hypothesis of the main goal is usally considered equivalent to **true**. Thus, the program extracted from the above proof should set x to 1 if M halts and to 0 if M diverges. Since this argument can be built for any given M, the assumption that programs can be extracted from $\vee$a leads to a solution of the halting problem. Therefore, programs cannot be extracted from the above form of $\vee$a.

The preceding argument shows that classical logic is inconsistent when applied to formulas having constructive meaning. In particular, ∨a may be incorrect when the conclusion is a statement of the form "we can find a program such that ...". However, transition specifications are concluaions of precisely this form. Thus, if specifications and logical formulaa are to be treated uniformly in conclusiona, PRL must adopt a non-classical approach to reasoning.

One possible approach is to replace tbe conventional ∨a rule by tbe rule

S,A∨B **pr** F **by**
  S,A **pr** F
  S,B **pr** F
  S,A∨B **pr** Q⇒A ∧ ¬Q⇒B

where Q is an obviously testable boolean expression (one that does not involve unbounded quantifiers, large sets, etc.). Though logically sound, this approach may require repetition at each use of A∨B of the reasoning that initially established its truth. Further, one would probably apply this rule only to the refinement of transitions and use the original, leas restrictive, ∨analysis rule for refinement of logical propositions, counter to the desire for a unified set of analysis rulea.

The alternative I have taken is based on the relatively well atudied area of constructive (intuitionistic) logic. In this approach, one maintains most of the classical reasoning rules by aasigning a non-classical interpretation to logical connectivea, and thus to logical formulas. For example, the interpretation of A∨B is that not only is one of A or B true, but we can determine which one is true. Thus, a proof of $A \vee B$ must supply a way of determining which alternative is true. Including $A \vee B$ as a hypothesis allows one to make use of the supplied decision method. With this interpretation, $\vee$a is a valid reasoning technique, because the decision method supplied with $A \vee B$ provides a way of choosing the subgoal that gives the desired result.

I cannot motivate and develop, in this dissertation, all of the constructive logic relevant to PRL. Instead, I will simply present an interpretation of connectives that is appropriate. Actually, PRL is founded on only the most basic ideas of constructive mathematics. I understand these ideas in terms of Computer Science concepts (e.g., compilation), so I will present them in these terms. The reader desiring a deeper understanding of intuitionism is referred to the bibliography [Kleene 1952, Prawitz 1965, Dummett 1977].

The following table informally describes the constructive meanings of the common logical connectives. These meanings might be used for informal constructive reasoning, e.g., reasoning constructively about PRL. Sections 3.4 and 4.2 introduce a formal interpretation for PRL formulas, mimicking the interpretation presented here.

Let C(F), for any formula F, be read as "F is constructively true". Then,

$C(A \wedge B)$ means $C(A)$ and $C(B)$.

$C(A \vee B)$ means either $C(A)$ or $C(B)$ and a method (called tbe selector) is available for deciding whicb.

$C(A \Rightarrow B)$ means that if $C(A)$ then $C(B)$.

$C(\neg A)$ means from $C(A)$ a contradiction may be constructively proven.

$C((\exists X)P(X))$ means a value, b, may be determined such that $C(P(b))$.

$C((\forall x)P(x))$ means that for any value, b, $C(P(b))$.

For example, $(\forall x)(\exists y)(y>x)$ is constructively interpreted as "for any x, one can find a value y which is larger than x". This statement would be satisfied by the program "set y to x+1".

The constructive interpretation of formulas forces the abandonment of certain classical truths. In particular, $A \vee \neg A$ is no longer valid for any A, e.g., if A is "M halts", because no mechanism may be available for deciding which alternative is true. Loss of this axiom forces loss of the method of proof by contradiction. Showing that $\neg A$ leads to a contradiction establishes $\neg\neg A$ (by the definition of $\neg$), but does not assure A, since both A and $\neg A$ could fail to be constructively true. Abandoning these "truths" is entirely appropriate when one is trying to constructively demonstrate the existence of programs that meet specifications. A programmer is generally not aided by a proof that the non-existence of the program he is about to write implies a contradiction.

Consider, now, how the constructive interpretations are treated in PRL. Each formula is justified by an object that provides the constructive meaning of the formula. Thus, a proof of A∨B yields a _justification_ that simultaneously tells which alternative is true and provides a justification of the true alternative. A proof having A∨B as a hypothesis can only be used when A∨B is constructively true. In that case, a justification of A∨B is provided as the certification of A∨B's constructive truth. In turn, the proof may use the information contained in the justification of A∨B to produce a justification certifying its conclusion. Thus, proofs denote _justification transformers_.

It should be clear how the ∨a rule uses the justifications for S and A∨B provided to it. The justification for its conclusion, F, is produced by one of the subgoals. The correct subgoal is determined by examining the selector of the A∨B justification. If, say, it selects A, then the justification transformer for S,A **pr** F is applied to the justification of S and that of A (which also is a component of the A∨B justification). Thus, the subgoal yields a justification for F which is then yielded from the ∨a rule.

Justifications for A∨B can be provided in several ways. An instance of ∨synthesis looks as follows.

S **pr** A∨B **by**

S **pr** A

This rule simply applies the justification transformer for S **pr** A to the justification for S to obtain one for A. Then it returns a justification for $A \vee B$ that always selects A and provides the justification for A. The rule of trichotomy,

$$S \text{ pr } x<0 \vee x\geq 0,$$

simply tests x, then yields an $\vee$-justification which tells which alternative is true and justifies that alternative (it happens that the justifications for relational expressions are trivial, in the sense that they need not supply any interesting information).

These notions of justifications and justification transformers are further developed in section 3.4. Chapter 4 includes a recursive data type definition for justifications which is used in an extractor that maps proofs into justification manipulating executable programs.

My ideas on extraction and constructive logic have developed, in part, from an early paper of Constable's [Constable 1971] and his recent work on PL/CV. The latter system has a constructive foundation which enables proofs to be interpreted as executable text. Constable has developed such an interpreter on paper [Constable 1978]. An implementation of this facility, integrated into the existing PL/CV system, may be forthcoming.

Chapter 3

PRL: A Logic for Correct Program Development

Chapter 2 motivated and introduced the major ideas of formal refinement. This chapter presents a formal system for program development based on those ideas.

Sections 3.1-3.3 introduce the fundamental syntactic components of propositions. Section 3.4 discusses the form and meaning of propositions, the interpretation of free names in propositions, the flow of environments down proof trees, and the constructive interpretation of propositions. Refinement rules for constructing proofs are presented in sections 3.5-3.7 with examples of their use. Section 3.8 describes the definitions and lemmas that extend the conceptual base of PRL to more abstract notions. The chapter concludes with the presentation of rules for checking that proofs satisfy the type and naming constraints set down in earlier sections.

I have used several nonstandard notations in the logic's description. The form A_x, where A is any syntactic unit and x is a list of names, has the same meaning as the unadorned form A, except that attention is being brought to the possible use of the names of x as free variables in A. There may be other free variables in A and not all names of x need occur free in A. This device, though of no formal significance, is useful for communicating the intent of various constructions presented below. The common notion of substitution is written $A_{x \leftarrow e}$ where e is a list of formulas to replace all free occurrences of corresponding names of x in A. Again, the names of x need not occur in A and, if they do, only the free occurrences are replaced. There is an important restriction, called <u>absence of capture</u>, that a substitution is valid only if no free name of e becomes bound in $A_{x \leftarrow e}$. Thus, the meaning of e in the substituted form is identical to its meaning before substitution. The refinement rules later in this chapter are applicable only when this condition is satisfied. The determination of free and bound variables for the syntactic forms of PRL is informally explained with the introduction of each form; it is made precise in section 3.9.

3.1 <u>Types and expressions</u>

An expression is a structured formula that maps each possible execution state into a single value. Evaluating an expression never has any side effects on the state; this makes the semantics of expressions tractable, in particular, it guarantees the validity of such common algebraic laws as the commutativity of addition. The criterion of an understandable and readable specification language would be hard to satisfy without these widely known properties.

The values produced by expressions are divided into an infinite number of types: booleans, integers, and a variety of sets. Booleans and integers are atomic types whose values may not be decomposed. Sets of type set(T) are unordered, nonrepeating, finite collections of values of type T, where T is integer or another set type, but not boolean (for technical reasons that are explained in section 3.6). Thus, boolean and integer are base types and **set** is a type constructor. The names for types may be generated by the following context free grammar.

type → **bool** | element type

element type → **int** | **set** (element type)

Each type has forms for denoting constants and operators for combining values. The operators are defined only for operands of certain types; they yield precisely typed results. The table below enumerates the methods for forming expressions. Within each type, the operators are listed in order of decreasing precedence (e.g., $\wedge$ binds tighter than $\vee$). Assume throughout that i,j are integer valued expressions, A,B are boolean valued expressions, T is an element type name, R,S are **set**(T) valued expressions, and e is an expression of type T. Finally, individual variables are free when considered as complete expressions and the free variables of composite forms are those of the components, unless stated otherwise.

INTEGERS

type: **int**

values: ..., -2, -1, 0, 1, 2, ...

basic forms:	..., -2, -1, 0, 1, 2, ...	constants
	variable name	the current value of the variable
	definition reference	the value returned from an instantiated definition (further explained in 3.8)

composite forms: i*j → **int** product

i+j, i-j, -i → **int** sum, difference, negation

int-list relation int-list

relation ... relation int-list → **bool**

where each relation is one of =, ≠, <, ≤, >, ≥ and each int-list is a list of integer valued expressions separated by commas. This is logically equivalent to the conjunction of the binary relations formed between each element of adjacent int-lists. For example, $1\le i,j<k,l\le n$ means $1\le i \wedge 1\le j \wedge i<k \wedge i<l \wedge j<k \wedge j<l \wedge k\le n \wedge l\le n$.

BOOLEANS

type: **bool**

values: **true, false**

basic forms: **true, false**

variable name

definition reference

composite forms: $\neg B \rightarrow$ **bool** negation

$(\exists \mathrm{iter}^d) B_d \rightarrow$ **bool** existential quantification

$(\forall \mathrm{iter}^d) B_d \rightarrow$ **bool** universal quantification

The notation iter^d is defined in section 3.2. It represents an iterator that defines the names given by the name list d. The free variables of these forms are those of the iterator together with those of B_d not appearing in d. Free occurrences in B_d of names of d become bound in the complete quantified expression.

$A \wedge B \rightarrow$ **bool** conjunction

$A \vee B \rightarrow$ **bool** disjunction

$A \Rightarrow B \rightarrow$ **bool** implication

$A \Leftrightarrow B \rightarrow$ **bool** equivalence

$A \Leftrightarrow B$ means $A=B$ and it is treated this way, but equality is always written $\Leftrightarrow$ for boolean operands.

SETS

type: **set**(T) (for any given element type T)

values: all finite sets of elements of type T

basic forms:

null set(T)	the empty set of type **set**(T)
{ generator }	the set of elements produced by the generator. The free names are those of the generator. The generator must be automatically evaluatable (defined in section 3.2), hence, the set finite and easily computed.
variable name	
definition reference	

composite forms:

$R \cap S$	→ **set**(T)	intersection
$R \cup S$	→ **set**(T)	union
$R - S$	→ **set**(T)	difference
$\# S$	→ **int**	number of elements (size)
$x \in S$	→ **bool**	element of
$x \notin S$	→ **bool**	not element of

set-list relation set-list

relation ... relation set-list → **bool**

where each relation is one of $=$, $\neq$, $\subseteq$, $\subset$ and each set-list is a list of set valued expressions separated by commas. For example, $R \cap S \subseteq R, S \subseteq R \cup S$ is a fact about union and intersection.

In the present formulation, all expressions are total functions, i.e., they are defined for any choice of input values. This simplifies the logic since the well-formedness of expressions is reduced to the question of type correctness, rather than requiring more general reasoning about the properties of possibly undefined computations. Though this restriction rules out many useful partial functions, such as integer division, it does not hinder presentation of the essential ideas of PRL. Nonetheless, as this is a central concern for any practical logic, I have spent significant effort toward finding a pleasing treatment of partial functions that introduces the least additional text into proofs. I have not yet found a satisfying solution, but my thoughts and some directions for further research are presented in chapter 5. The PL/CV project has considered several solutions to this problem in their programming logic [Constable and O'Donnell 1978].

With the type restrictions presented above, the types of all expressions may be statically determined. Illegal expressions are forbidden from specifications and proofs by easily applied type checking rules. The envisioned program development system performs this checking as proofs are interactively produced so that all intermediate developments are assured type correct without any programmer assistance. A more flexible type system, e.g., allowing heterogeneous sets through union types or a typeless semantics, would require the introduction of proof text to show the absence of type errors. Such a system might be appropriate if one had a general approach to handling errors as called for above, but it is inconsistent with the present philosophy of treating only non-erroneous expressions.

3.2 Generators and iterators

Many PRL constructs are formed from components that represent groups of values, for example, sets are formed by placing braces {...} around such a component. These components are called generators. As discussed in section 2.3, each denotes an unordered, nonrepeating, possibly infinite collection of values of a single type. Generators are total, side effect free mappings from the state whose types may be statically determined. I will formally present them shortly.

A related construct is the iterator, also introduced in section 2.3, in which a tuple of variables (the bound variables) ranges over an unordered, nonrepeating, possibly infinite collection of tuples of values. Each bound variable of the iterator has a specific statically determined type, but no relation is assumed between the types of different variables. The context of an iterator determines both the scope of its bound variables and the use of its collection of tuples. (For example, in the universally quantified formula $(\forall d{:}1 \textbf{ to } 50)B_d$, the iterator's scope extends over the quantified predicate. The truth of the formula is determined by evaluating the predicate many times with d bound to each successive generated value.) An iterator that binds a list of variables x is written iter^{x}. Thus given S of type **set**(T), $(\exists x,y{:}\in S,R{:}\subset S)\ldots$ contains an iterator, $\text{iter}^{x,y,R}$, providing the names x,y,R of types T,T,**set**(T) that range over all tuples from $S\times S\times 2^S$.

In any logic of sets or quantified expressions, some decision must be made about what may be written inside of set braces or after the quantifier. Conventionally, the choice is to require a name and a type with an optional restricting predicate (thus, $(\forall d{:}\textbf{int})B_d$ or

$\{d{:}\mathbf{int} \mid B_d\})$. I have chosen to explore an extended notation for these constructs for two reasons. First, it is important that the programming system supporting PRL be able to evaluate expressions automatically whenever possible. This task is simplified if one can easily express varieties of bounded quantifiers, bounded set formers, etc., because the system can then determine by syntactic examination whether a construct is automatically evaluatable (more below). For example,

$$\neg(\exists x{:}2 \textbf{ to } p-1)(\exists y{:}1 \textbf{ to } p)(p=x*y)$$

is computationally more tractable (to a simple automatic evaluator) than

$$\neg(\exists x,y{:}\mathbf{int})(1<x<p \wedge 1\le y\le p \wedge p=x*y).$$

Second, I have long been intrigued by Earley's work on iterators [Earley 1974a]. He seems to derive a great deal of expressive power from these sorts of constructs, in particular from pattern matching facilities. Though PRL iterators are very limited in comparison to his ideas, I wanted to begin developing a feel for them. One option for later research is to increase the power of the specification language by extending the class of iterators.

Let us now examine the methods for constructing iterators and generators. An iterator, $iter^x$, may be formed in four ways.

x : generator binds the names of x with a set of tuples in which each name of x takes on all the values of the generator. The free variables are those of the generator. For example, "i,j:1 to n" yields n^2 tuples where n is a free variable.

$iter^x$ **st** B_x binds the names of x with the subset of the tuples of $iter^x$ that satisfy the predicate. The free variables are those of $iter^x$ together with those of B_x not appearing in x. For example, "p:**int st** (∃i:**int**)(p=d*i+r)" yields an infinite class of values for p that satisfy p <u>mod</u> d=r for (given) free d and r.

$iter^x_y$ **for** $iter^y$ binds the names of x with the set of tuples formed by unioning the sets of tuples produced by $iter^x_y$ for each assignment to y yielded by $iter^y$. The free variables are those of $iter^y$ together with those of $iter^x_y$ not appearing in y. For example, "m:∈sons(n) **for** n:∈current_nodes" produces the name m and the set of sons of all the current nodes. The only free name is current_nodes.

$iter^y$, $iter^z$ (where x is y concatenated with z) binds the names of x with the set of tuples formed from the cross product of the tuples of $iter^y$ with the tuples of $iter^z$. The free variables are those of $iter^y$ together with those of $iter^z$. For example, "S:∈start_states, f:∈final_states" produces the set of all pairs of starting and final states.

The generators are more easily described. (Though I shall use the term "set of values" to describe the result of evaluating a generator, this is a distinct notion from the set expressions described in the last section. Generators must be enclosed in set brackets to produce directly manipulatable values.)

type generates the set of values of the type. There are no free variables.

exp generates the singleton set containing the value of the expression. The free variables are those of the expression.

exp **to** exp (where both are integer valued) generates the set of values in the inclusive range from the first to the second expression. The free variables are those of both expressions.

sub exp (where sub is ϵ, $\subset$, or $\subseteq$ and exp is set valued) generates the set of elements of exp, strict subsets of exp, or subsets of exp, according to the choice of sub. The free variables are those of the expression. For example, $\{\subseteq S\}$ is a set valued expression that represents the powerset of S.

$iter^x$ (where x is a single name) generates the set of values yielded by the iterator. The free variables are those of the iterator. For example, "p:**int st** prime(p)" generates the collection of prime numbers.

gen_x **for** $iter^x$ is similar to the use of **for** in iterators. It generates the set of values formed by unioning the sets produced by gen_x for each assignment to x yielded by $iter^x$. The free variables are those of $iter^x$ together with those of gen_x not appearing in x. For example, "n^2 **for** n:1 **to** 50" produces the first 50 positive squares.

gen ; gen generates the set of values formed by unioning the sets produced by the two generators. The free variables are those of both generators. For example, {1;2;3} is the set containing 1, 2, and 3.

Proving properties of constructs containing iterators and generators requires a more formal characterization of the values produced than given above. This characterization is provided by a map ∇ from iterators to predicates -- $\nabla(\text{iter}^x)$ yields a predicate on names of x (among other free variables) which is true exactly when the values of x are some tuple produced by iter^x. For example, ∇(i,j:1 **to** n) is $1 \le i,j \le n$.

∇ is defined inductively on an extension of the syntactic class of iterators. The extended domain includes the form exp-list:generator in addition to the various iterators. Each form of iterator, including the various forms of exp-list:generator, is handled as a distinct clause of the definition. This enlarged domain increases the usefulness of ∇; however, it does not modify the class of legal iterators.

∇ : extended iterator form $\rightarrow$	predicate
$exp_1,\ldots,exp_n$: gen	$\nabla(exp_1\text{:gen}) \wedge \ldots \wedge \nabla(exp_n\text{:gen})$
exp : type	**true**, the extended iterator cannot be legally formed otherwise. This sort of type information

	is carried in the environments
	discussed in section 3.4.
exp : exp'	exp=exp'
exp : exp_1 to exp_2	$\text{exp}_1 \le \text{exp} \le \text{exp}_2$
exp : sub exp'	exp sub exp', where sub is ∊, ⊂, or ⊂
exp : iter^x	$\nabla(\text{iter}^x)_{x\leftarrow \text{exp}}$, since exp is produced
	by iter^x exactly when the
	predicate characterizing iter^x
	holds with exp substituted for x.
exp : gen_x for iter^x	$(\exists \text{iter}^y)\nabla(\text{exp:gen}_{x\leftarrow y})$, where iter^y
	is iter^x with a change of
	bound variables from x to y.
	(y must not occur in exp, gen,
	or iter^x.) This name change
	is needed since exp is placed in
	the scope of the iterator.
	Any free occurrences in exp of
	variables of x must not be
	captured by the existential.
	For example,
	∇(n:(∊succ(n) for n:∊marked)) is
	is (∃y:∊marked)(n∊succ(y)).
exp : gen_1 : gen_2	$\nabla(\text{exp:gen}_1) \vee \nabla(\text{exp:gen}_2)$
iter^x st B_x	$\nabla(\text{iter}^x) \wedge B_x$, for example
	∇(s:∊S st s≤0) is s∊S∧s≤0.
iter_y^x for iter^y	$(\exists \text{iter}^y)\nabla(\text{iter}_y^x)$, for example
	∇((next:∊succ(n)) for n:∊marked)

is $(\exists n{:}\epsilon\, \mathrm{marked})(\mathrm{next}\epsilon\, \mathrm{succ}(n))$.

$\mathrm{iter}^y \cdot \mathrm{iter}^z$ $\nabla(\mathrm{iter}^y)\wedge\nabla(\mathrm{iter}^z)$

(Note: Though the syntax of iterators and generators is ambiguous (the precedence of operators is not specified), all choices of parentheses produce extensionally equivalent predicates according to ∇. Since the meaning of these constructs is formally defined by ∇ alone, the ambiguities are harmless.)

The inclusion of the form "type" for generators permits the expression of unbounded generators and iterators. The complete proof of a transition specification will eventually refine away any use of unbounded iteration, so that the system can extract a program that is guaranteed to terminate. (This restriction is needed only because the extractor is very limited -- many transition specifications using unbounded iteration do characterize terminating programs.) Thus, though proofs may reason about forms whose values are not obviously computable, by the time refinement reaches the leaves of a proof only easily computed forms remain. This property of being easily computed is called automatically evaluatable, abbreviated a.e.. It is defined as follows.

Def An expression, generator, iterator, or transition specification is a.e. iff it does not contain any generator of the form "type".

It is a purely syntactic question whether a construct is a.e.. In particular, though boolean expressions are not a.e. if they contain unbounded quantifiers, integer and set valued expressions are always a.e., as may be seen by examining section 3.1. (See section 3.8 for treatment of definition references.) Thus,

$$\{p{:}1 \textbf{ to } 50 \textbf{ st } (\forall k{:}\textbf{int})(k \text{ divides } p \Rightarrow k{=}1 \vee k{=}p)\}$$

is not a legal set expression. This is the technical meaning of the remark, in section 3.1, that sets are finite and easily computed. The elements of an a.e. generator can be enumerated by a straightforward, though inefficient, algorithm, so all sets can be computed.

The characterization of iterators by predicates necessitates a restriction on the use of names for bound variables. Consider the iterator x:x, presumably meaning that bound variable x ranges over the single current value of free variable x. ∇ characterizes this iterator by x=x, thus losing the distinction between bound and free variables. The usual sort of solution would resolve this naming conflict by uniformly renaming bound variables throughout their scope. However, as ∇ generally does not have access to the entire scope of variables defined by iterators, a restriction is made that insures no name conflicts will occur. Thus,

> no bound variable name of any iterator may
> occur as a free variable of that iterator.

This restriction may be formulated by cases:

1) in x:generator no variable of x may occur free in the generator.
2) in iter^x_y **for** iter^y no variable of y may occur free in iter^x.
3) in $\text{iter}^y, \text{iter}^z$ no variable of y may occur free in iter^z and conversely.

Thus, a:∈{a;b}, a:a, (a:∈S, S:⊂R), and (b:a **to** a+10) **for** a:b-10 **to** b are illegal, while c:∈{a;b}, b:a, (a:∈S, Q:⊂R), and (c:a **to** a+10) **for** a:b-10 **to** b are legal. In practice, this restriction is of little consequence since name conflicts can be easily avoided by the programmer. Checking for conflicts is essential, however. This is part of the static analysis described in section 3.9.

3.3 Transition specifications

The major purpose of PRL is to permit expression of the specification and refinement of programs. As discussed in section 2.3, the only visible effects of the programs considered are changes to the values of globally declared variables. These restricted programs define nondeterministic mappings (relations) from initial to final state, where states are simply associations of values with variable names. As there are no structured variables and no discernible sharing of representation by values, the logic for reasoning about values and assignment is straightforward.

The general form of a transition specification is

$$\text{var-list} \leftarrow \text{any iter}^{x}$$

where x has the same length as var-list. This means that the var-list may be set to any tuple of values produced by the iterator. After execution of the transition specification (or its refinement), the values of var-list satisfy the predicate

$$\nabla(\text{iter}^{x})_{\text{var-list},x \leftarrow \text{var}_0\text{-list},\text{var-list}}$$

where var_0-list is var-list with each variable v replaced by v_0. This predicate relates the current values of var-list to the values of the variables before they were changed (var_0-list).

The above transition form is used in formal description and analysis of the logic. However, in specifications such as $i \leftarrow$ **any** $j: 2*i$ and $x \leftarrow$ **any** $b: \in S$, it requires the introduction of meaningless bound variables. Therefore, when writing specifications there are two additional acceptable forms:

$$\text{var-list} \leftarrow \textbf{any}\ \text{gen-list}$$

and $$\text{var-list} \leftarrow \textbf{any}\ \text{gen-list}_x\ \textbf{for}\ \text{iter}^x.$$

Also, the word **any** may be dropped if desired, e.g., if the generators are single valued. These forms are interpreted as

$$\text{var}_1,\ldots,\text{var}_n \leftarrow \textbf{any}\ \ \text{var}'_1{:}\text{gen}_1,\ldots,\text{var}'_n{:}\text{gen}_n$$

and $$\text{var}_1,\ldots,\text{var}_n \leftarrow \textbf{any}\ (\text{var}'_1{:}\text{gen}_1,\ldots,\text{var}'_n{:}\text{gen}_n)\ \textbf{for}\ \text{iter}^x,$$

respectively, where the var'_i are new variables. Thus, $i \leftarrow 2*i$ and $x \leftarrow$ **any** $\in S$ are legal abbreviations for specifications.

There has been a constant tension between my desire to eliminate unnecessary auxiliary names from specifications and the need to keep the reasoning rules simple. Although auxiliary names have been largely removed from specifications, they are generally re-introduced in the refinement text. I am not aware of any satisfactory method for reasoning about variables that avoids this difficulty. Ideally, the logic would allow direct algebraic reasoning about transition specifications. Some thoughts in this direction are presented in chapter 5.

3.4 Propositions, environments, and justifications

A proposition, of the form hypothesis-list pr conclusion, is a claim that some conclusion follows from certain hypotheses. The hypotheses are boolean expressions that characterize the state of the proposition's free variables. The conclusion is a boolean expression that is claimed true or a transition specification that is claimed to specify a computable mapping from state to state.

The propositions of a hypothesis-list are necessarily written in some specific order. Nevertheless, they are interpreted as lacking order, in that they may be freely re-ordered as desired. Many refinement rules single out a specific hypothesis for manipulation, for example,

S,A^B pr C by ...

concerns the conjunction A^B. These rules may be used even when the selected hypothesis is not last in the hypothesis-list, without explicit mention of the required re-ordering. This essential informality entails a slight complication of the extraction process.

The discussion of top down development in section 2.2 recommends that meaning be assigned to propositions without regard for surrounding context, so that refinement validity is a local property of proofs. In order to achieve this, it is necessary to associate with each proposition an environment describing the types of all accessible global variables. If a proposition references other variables, or is incompatible with the types of the variables it does reference, then it is considered ill-formed.

Incorporating environments into proofs extends the form of propositions to

<env> hypothesis-list **pr** conclusion

where env is a set of "name:type" pairs, for example,

<x:**int**, S:**set**(**int**)> **pr** x ← **any** b:∈S.

The environment of the outermost proposition of a proof is formed from global variable declarations of the form "**var** name:type" (definitions and lemmas are also introduced at this level). This initial environment is then modified according to the refinement rules as it flows down the proof tree. The acceptable refinements are those whose subgoals are well-formed with respect to their inherited environments. Thus, the validity of a proof may be assured by checking that each refinement step is legal according to the rules of the next few sections and that each proposition is well-formed according to the rules of section 3.9.

I shall often omit the environment clause from propositions occurring in examples since the examples are usually simple and the environments obvious. Nonetheless, during development of proofs with machine aid I expect that environments would be at least partially displayed, as they provide information essential to correct refinement.

As explained in section 2.4, the interpretation of propositions is constructive in that a justification is associated with each proved boolean formula certifying its truth. A proof of a proposition denotes a method for transforming justifications of its hypotheses into a justification of its conclusion.

The choice of precisely what information to carry in justifications is determined by how the justified formulas are used in the proof rules. The $\vee$a rule of section 2.4 demonstrated that an $\vee$-justification must include a selector telling which alternative of the disjunction is true. The remaining varieties of justifications have been similarly developed. I will present them informally below. Section 4.2 supplies a precise discussion of justifications and their properties.

Some boolean expressions need no justification; the extraction process requires only that proofs establish their classical truth. I will say that such an expression has no <u>deep constructive meaning</u> or that its justification is <u>trivial</u>. Arithmetic relations, for example, have this property.

Transition specifications do need justifications, because after a transition is executed, the changed variables are supposed to be related to their old values by a constructively true predicate. The extracted program for a proposition of the form S **pr** x $\leftarrow$ **any** iter^d uses the justifications of the formulas of S and the current values of the free variables to compute new values for x and a justification for the predicate $\nabla(\text{iter}^d)$. That is, in addition to performing the transition, the extracted program provides a justification for a specific boolean expression. If one lets x represent its own values before the transition and d represent x's values after the transition then $\nabla(\text{iter}^d)$ describes how the new values relate to the old values. Though this justification may appear unusual, it carries precisely the information needed to permit extraction.

The justifications for each kind of formula are enumerated below. Note that those for the logical connectives mirror the constructive interpretations presented in section 2.4.

syntactic form:	is justified by:
$A \wedge B$	a pair (justification for A, justification for B)
$A \vee B$	a pair (selector, j) where if selector is "left" then j justifies A, othegwise j justifies B
$A \Rightarrow B$	a justification transformer which maps justifications of A into those for B
$A \Leftrightarrow B$	a pair (justification transformer for $A \Rightarrow B$, justification transformer for $B \Rightarrow A$)
$\neg B$	the trivial justification, since a demonstration that B leads to a contradiction provides no constructively useful information
$(\exists iter^d)B_d$	a triple (value, justification for $\nabla(iter^d)$, justification for B_d), where d=value satisfies $\nabla(iter^d) \wedge B_d$
$(\forall iter^d)B_d$	a justification transformer that maps pairs (value of d, justification of $\nabla(iter^d)$) into justifications for B_d
$x \leftarrow$ **any** $iter^d$	a justification for $\nabla(iter^d)$
true, **false**	the trivial justification (but note that a justification for **false** will never

	actually be used becauae **false** can never be proved true)
boolean variable	the trivial justification (but see section 4.2)
boolean definition reference	(see sections 3.8, 4.2, 4.6)
exp∈set-exp	the trivial justification (for reasons explained in section 3.6)
exp∉set-exp	the trivial justification (it is equivalent to ¬(exp∈set-exp))
int-exp relation ... int-exp,	
set-exp relation ... set-exp	the trivial justification, but recall that complex relational expressions are treated as conjunctions of simple ones, e.g., $1 \le i, j \le n$ means $1 \le i \le n \wedge 1 \le j \le n$.

Extraction is an inductive process. In order to carry it out, for each refinement rule one must be able to construct a justification transformer for its goal from those extracted from its subgoals. This means that a description of the justifications chosen for each formula is really an inductive property characterizing extraction. If the description is too weak, the subgoals will not provide enough information. If the description is too strong, the goal will require too much information. The reason for choosing the above varieties of justifications is that they work.

3.5 Refinement: logical rules

The logical refinement rules describe the analysis and synthesis of boolean expressions, the use of substitution, and other fundamentals of the proof system. The analysis and synthesis rules are a variation of those presented by Gentzen in his development of natural deduction systems and the sequent calculus [Kleene 1952, Constable and O'Donnell 1978, Szabo 1969]. In particular, recall from section 2.4 that proofs are trees, the subgoals of a goal are independent propositions, and there is no implicit flow of information between subgoals. Thus, the ordering of subgoals in the refinement rules is of no significance.

Let A_i, B_i be boolean expressions, C be any conclusion, and S be a list of boolean expressions (representing a set of hypotheses). Consider the rules for composite boolean expressions. These rules are named by the operator to which they apply suffixed with "a" or "s", for analysis or synthesis, respectively. Unless stated otherwise, the environments of subgoals are the same as that of the goal.

$\wedge$a $S, A_1 \wedge A_2$ **pr** C **by**
  S, A_1, A_2 **pr** C

$\wedge$s S **pr** $B_1 \wedge B_2$ **by**
  S **pr** B_1
  S **pr** B_2

The rule of $\wedge$analysis allows splitting hypothesis $A_1 \wedge A_2$ into its components for further separate analysis. The corresponding synthesis rule allows combining independent conclusions B_1, B_2 into $B_1 \wedge B_2$.

$\vee$a $\quad S, A_1 \vee A_2$ pr C by

$\quad\quad S, A_1$ pr C

$\quad\quad S, A_2$ pr C

$\vee$s $\quad S$ pr $B_1 \vee B_2$ by $\quad\quad S$ pr $B_1 \vee B_2$ by

$\quad\quad S$ pr B_1 $\quad$ also $\quad S$ pr B_2

If C may be proved from $S.A_1$ and from $S.A_2$ then it may be proved from $S.A_1 \vee A_2$ by $\vee$analysis. The justification of $A_1 \vee A_2$ determines which prcof of C is used. Conclusion $B_1 \vee B_2$ may be proved by establishing either B_1 or B_2 and using the appropriate form of $\vee$synthesis. The justification for $B_1 \vee B_2$ depends directly on which form of $\vee$synthesis is used; other rules for synthesizing disjunctions, e.g., trichotomy, produce more complex justifications.

$\Rightarrow$a $\quad S, A_1 \Rightarrow A_2$ pr C by

$\quad\quad S$ pr A_1

$\quad\quad S, A_1, A_2$ pr C

$\Rightarrow$s $\quad S$ pr $B_1 \Rightarrow B_2$ by

$\quad\quad S, B_1$ pr B_2 .

The first rule allows the splitting of hypothesis $A_1 \Rightarrow A_2$ into $A_1 . A_2$ given a proof of A_1. The justification of A_2 is obtained by applying the justification of $A_1 \Rightarrow A_2$ to that of A_1. The second rule allows the antecedent of $B_1 \Rightarrow B_2$ to be treated as part of the hypothesis-list. The justification transformer extracted from the subgoal is applied to the justification of S, but not applied to that of B_1. This leaves a transformer that justifies $B_1 \Rightarrow B_2$.

$\neg$a $\quad$ S,$\neg$B **pr** false **by**

S **pr** B

$\neg$a $\quad$ S **pr** $\neg$B **by**

S,B **pr** false

The $\neg$analysis rule says that a contradiction arises if S entails B, since both "B is provable" and "from a proof of B a contradiction may be derived" are then true. The $\neg$synthesis rule allows the conclusion "from a proof of B a contradiction may be derived".

As mentioned in section 2.4, one should note that

S **pr** B **by**

S,$\neg$B **pr** false

is not a refinement rule. It is classically valid, but the subgoal does not provide any information on why B is true, only that $\neg$B is inconsistent with S. It is not generally possible to produce a justification for B given only justifications for S and a proof of the subgoal. Thus, tbis rule is not constructively valid and is not present in PRL.

$\Leftrightarrow$a $\quad$ $S, A_1 \Leftrightarrow A_2$ **pr** C **by**

$S, A_1 \Rightarrow A_2, A_2 \Rightarrow A_1$ **pr** C

$\Leftrightarrow$a S **pr** $B_1 \Leftrightarrow B_2$ **by**

S **pr** $B_1 \Rightarrow B_2$

S **pr** $B_2 \Rightarrow B_1$.

These rules capture the obvious relationship between equivalence and implication.

The rules for quantifiers are somewhat more subtle; they deal with new names, changing environments, and substitution.

$\forall$a <env> S,$(\forall iter^x)B_x$ **pr** C **by**

<env> S **pr** $\nabla(iter^x)_{x \leftarrow exp}$

<env> S,$(\forall iter^x)B_x, B_{x \leftarrow exp}$ **pr** C

where exp is a list of a.e. expressions of the same type in <env> as x (in $iter^x$) in <env>

The $\forall$analysis rule allows introducing an instantiation of B_x into the hypothesis-list for any evaluatable expressions that satisfy the iterator. The use of ∇ in the first subgoal is a metanotation, like substitution, that disappears in any actual use of this rule. For example,

$(\forall$x:**int st** prime(x))($\neg$prime(x+7)) **pr** $\neg$prime(y) **by**

pr prime(y-7)

$(\forall$x:**int st** prime(x))($\neg$prime(x+7)),$\neg$prime(y-7+7) **pr** $\neg$prime(y)

instantiates $\neg$prime(x+7) with y-7 for x. The justification of $B_{x \leftarrow exp}$ is derived from the first subgoal and the justification of the universally quantified hypothesis.

∀a <env> S pr $(\forall \mathrm{iter}^x)B_x$ by

<env,d:T> S,$\nabla(\mathrm{iter}^x)_{x\leftarrow d}$ pr $B_{x\leftarrow d}$

where T is the type of x (in iter^x) in <env> and d∉<env> (if x is a list of names then interpret T and d accordingly)

This allows the conclusion of a universally quantified formula if the formula can be established in an environment where the bound variables are free (and thus implicitly universally quantified). The change of names is needed since names of x may occur in <env>. The justification of the conclusion is a method of providing a justification of B_x for any x satisfying the iterator; this is precisely what is supplied by the proof of the subgoal.

∃a <env> S,$(\exists \mathrm{iter}^x)B_x$ pr C by

<env,d:T> S,$\nabla(\mathrm{iter}^x)_{x\leftarrow d}$,$B_{x\leftarrow d}$ pr C

where T is the type of x (in iter^x) in <env> and d∉<env> (if x is a list of names then interpret T and d accordingly)

∃analysis allows one to assume the availability of a value for each name of d that together satisfy the existential. These values may be used in the proof of C. The needed values and justifications for the hypotheses of the subgoal are provided by the justification of the existential. As with ∀analysis, this rule illuminates the need for constructively interpreted formulas when one considers that C may represent a program basing its calculations on the value of d.

∃a <env> S **pr** (∃$iter^x$)B_x **by**

<env,d:T> S **pr** d ← **any** $iter^x$ **st** B_x

where T is the type of x (in $iter^x$) in <env>

and d∉<env> (if x is a list of names then

interpret T and d accordingly)

This rule introduces a new variable list, d, and permits the stated conclusion if one can develop a program that sets d to appropriate values. Thus, an existentially quantified formula may be viewed as a device for packaging a computing mechanism and sending it to other parts of the proof. The justification for the existential formula is built from the justification of the transition specification. As the existential must provide both a list of values and a justification of B for those values, so must the justification of the transition specification. Therefore, rules for refining the latter must permit the extraction of both code and justifications describing how calculated values satisfy their iterators. For example, extracting from a proof of

j ← **any** i:**int** **st** prime(i)∨prime(i+1)

must yield code for computing j and a justification telling whether j or j+1 is prime. Such a proof would allow synthesis of the conclusion

(∃i:**int**)(prime(i)∨prime(i+1)).

The next set of rules describe some fundamental properties of the logic. They need no refinement for their validity and may be thought of as axioms.

<u>true def</u> S **pr** **true**

false def $S, \mathbf{false}$ **pr** C

excluded middle S **pr** $B \vee \neg B$

where B is an a.e. boolean expression

assumption S, B **pr** B

≠ def S **pr** $exp_1 \neq exp_2 \Leftrightarrow \neg(exp_1 = exp_2)$

The false definition rule has the interesting property that since one can never be presented with a justification of truth of the hypothesis, one need not be concerned with producing a justification for the conclusion. The excluded middle rule is quite useful in conjunction with $\vee$analysis when the cases are determined by the truth value of an automatically evaluatable boolean expression. The ≠ definition rule is a logical (rather than, say, integer) rule because equality and inequality are applicable to any pair of expressions of like type.

consequence <env> S **pr** C **by**

<env> S **pr** B_1

<env> S, B_1 **pr** B_2

.
.
.

<env> $S, B_1, \ldots, B_{n-1}$ **pr** B_n

<env> $S, B_1, \ldots, B_n$ **pr** C

The rule of consequence permits the joining of sequential proofs into a composite whole. The justifications flow from the initial hypotheses, S, through the subgoals in sequence to produce a justification for C.

specialization <env> $S_{x\leftarrow exp}$ **pr** $C_{x\leftarrow exp}$ **by**

<env,x:T> S_x **pr** C_x

where exp is a.e. and T is the type of exp in the context of <env> (if exp is a list of expressions then interpret T accordingly)

The specialization rule allows the replacement of free variables x in S and C by any a.e. expressions of appropriate types. Such a substitution is permissible since if S_x **pr** C_x holds for any tuple of values, x, then surely it holds for the particular values of exp. In the case that C is a boolean expression, the same effect may be achieved by synthesizing $(\forall x:T)(S \Rightarrow C)$ from S **pr** C using $\forall$s and then analyzing it using $\forall$a with exp replacing x. However, this separate rule of specialization is needed if C is a transition specification, because $S \Rightarrow C$ would be ill-formed.

As an example of logical proofs, consider showing that $\neg(exp_1 \neq exp_2)$ **pr** $exp_1 = exp_2$. This holds in general, if the expressions are a.e., as follows.

$\neg(exp_1 \neq exp_2)$ **pr** $exp_1 = exp_2$ **by** consequence

$\neg(exp_1 \neq exp_2)$ **pr** $exp_1 = exp_2 \vee \neg(exp_1 = exp_2)$ **by** excluded middle

$\neg(exp_1 \neq exp_2) . exp_1 = exp_2 \vee \neg(exp_1 = exp_2)$ **pr** $exp_1 = exp_2$ **by** $\vee$a

$\neg(exp_1 \neq exp_2) . exp_1 = exp_2$ **pr** $exp_1 = exp_2$ **by** assumption

$\neg(exp_1 \neq exp_2) . \neg(exp_1 = exp_2)$ **pr** $exp_1 = exp_2$ **by** consequence

... **pr** $exp_1 \neq exp_2$ **by** consequence

... **pr** $exp_1 \neq exp_2 \Leftrightarrow \neg(exp_1 = exp_2)$ **by** $\neq$ def

$\ldots . exp_1 \neq exp_2 \Leftrightarrow \neg(exp_1 = exp_2)$ **pr** $exp_1 \neq exp_2$ **by** $\Leftrightarrow$a

$\ldots . exp_1 \neq exp_2 \Rightarrow \neg(exp_1 = exp_2) . \neg(exp_1 = exp_2) \Rightarrow$

$exp_1 \neq exp_2$ **pr** $exp_1 \neq exp_2$ **by** $\Rightarrow$a

$\ldots . exp_1 \neq exp_2 \Rightarrow \neg(exp_1 = exp_2)$

pr $\neg(exp_1 = exp_2)$ **by** assumption

$\ldots . exp_1 \neq exp_2 \Rightarrow \neg(exp_1 = exp_2) .$

$exp_1 \neq exp_2$ **pr** $exp_1 \neq exp_2$ **by** assumption

$\ldots . exp_1 \neq exp_2$ **pr** **false** **by** $\neg$a

$\ldots . exp_1 \neq exp_2$ **pr** $exp_1 \neq exp_2$ **by** assumption

$\ldots . exp_1 \neq exp_2 .$ **false** **pr** $exp_1 = exp_2$ **by** false def

The final set of logical refinement rules deal with equality. The properties of reflexivity, symmetry, and transitivity are captured as follows.

<u>reflexivity</u> S **pr** exp=exp

<u>symmetry</u> S **pr** exp_2=exp_1 **by**
S **pr** exp_1=exp_2

<u>transitivity</u> S **pr** exp_1=exp_3 **by**
S **pr** exp_1=exp_2
S **pr** exp_2=exp_3

These rules apply to all types of expressions (recall that equality for boolean expressions is written $\Leftrightarrow$). Since no ill-defined expressions can appear anywhere in PRL, there is no need to mention well-definedness in the reflexivity rule.

The fourth equality rule provides for the substitution of equals.

<u>substitution</u> $S_{x\leftarrow exp_1}$ **pr** $C_{x\leftarrow exp_1}$ **by**
$S_{x\leftarrow exp_1}$ **pr** exp_1=exp_2
$S_{x\leftarrow exp_2}$ **pr** $C_{x\leftarrow exp_2}$

This means that to prove S **pr** C one may establish exp_1=exp_2 from S and then show S' **pr** C' which is obtained from S **pr** C by changing selected occurrences of exp_1 to exp_2. Note that x is used here as a "trick" to allow selective substitution, but (unless it happens to appear in exp_1 or exp_2) it does not actually occur free in the hypothesis or conclusion.

With each refinement step of proofs, I shall sometimes write the name of the rule used and other relevant annotation immediately after the word **by**. The following proof schemes, presented in this style, show that substitution is powerful enough to encompass symmetry and transitivity.

derived symmetry:

```
S pr exp2=exp1 by substitution where C is "exp2=x"
    S pr exp1=exp2 by
        >proof required
    S pr exp2=exp2 by reflexivity
```

derived transitivity:

```
S pr exp1=exp3 by substitution where C is "x=exp3"
    S pr exp1=exp2 by
        >proof required
    S pr exp2=exp3 by
        >proof required
```

Despite these reductions, the rules of symmetry and transitivity are retained, because there is no sufficiently general facility in PRL for introducing derived proof rules. The lemma mechanism of section 3.8 provides some ability to extend the logic, but it is not powerful enough. An effective solution to this problem that allowed the introduction of derived rules and hints on their applicability (perhaps along the lines of LCF metalanguage [Gordon et. al. 1977a]) would permit more compact proofs. It could gradually lead to the semi-automatic generation of proofs according to user developed strategies.

3.6 Refinement: integers and sets

The theory of integers, though of great practical importance, does not shed much light on the issues considered in this thesis. Therefore, much of the theory is contained in a general rule that permits any valid inference about simple arithmetic relations. Several additional rules capture the remainder of the theory.

An arithmetic relation is "exp-list relation exp-list relation ... relation exp-list" or its negation, where all expressions are integer valued and the relations are =, ≠, <, ≤, >, ≥. The rule

> **arith** $S, B_1, \ldots, B_n$ pr C
>
> where C is an arithmetic relation that follows from arithmetic relations $B_1, \ldots, B_n$ alone according to some reasonable axiom system

provides for the introduction of "intuitively obvious" reasoning steps without further explanation. In an implementation of PRL this rule would probably be broken down into more easily verified rules. However, as arithmetic relations have no deep constructive meaning, the extraction technique of chapter 4 depends only on the validity of these inferences, not on their deep structure. Accordingly, a mechanism is postulated capable of checking arithmetic inferences and further levels of refinement are safely ignored.

In addition to the above rule, one needs to be able to prove disjunctive conclusions. The following three rules provide this ability. I have separated them from the general rule because non-trivial justifications must be produced for their conclusions.

<u>trichotomy</u> S pr $exp<0 \vee (exp=0 \vee exp>0)$
where exp is an integer valued expression

$\leq$<u>def</u> $S, exp_1 \leq exp_2$ pr $exp_1 < exp_2 \vee exp_1 = exp_2$
where exp_1, exp_2 are integer valued expressions

$\geq$<u>def</u> $S, exp_1 \geq exp_2$ pr $exp_1 = exp_2 \vee exp_1 > exp_2$
where exp_1, exp_2 are integer valued expressions

The justifications for conclusions of these rules are easily obtained since the expressions are integer valued, hence a.e.. Deciding which alternative of a disjunction holds is simply a matter of evaluating the expressions and performing a comparison or two.

The PL/CV project has developed a theory of integers and an implemented set of reasoning rules that are quite powerful [Constable and O'Donnell 1978, Chan 1978]. Though the rules are organized differently from those above, they include an arithmetic rule (similar to arith) permitting significant deductions to be made in one step. The well studied PL/CV rules and their implementation, modified to produce justifications for disjunctive conclusions, could be used as a reasonable theory of integers for PRL.

The set theory I have developed is simple. Since all sets are finite and easily computed, the set theory axioms are easily shown consistent by construction of a model in which they are true (see chapter 4).

Though a theory of infinite sets is of considerable interest, it is also difficult to develop. I have chosen to stay with an easily axiomatized theory for the present development. Constable has been working on a constructive type theory that is expected to handle infinite sets [Constable 1979]. Such a theory could probably be molded into a powerful set theory for PRL.

For the remainder of this section, let T be an element type, x,y be expressions of type T (hence a.e.), Q,R be expressions of type set(T), and gen be an a.e. generator of type T.

The first pair of axioms

$\in$<u>def</u>   **S pr x$\in$null set(T) $\Leftrightarrow$ false**

and

S pr x$\in$\{gen\} $\Leftrightarrow$ ∇(x:gen)

describe the basic properties of $\in$. The latter rule uses the extended domain of ∇ to characterize the elements of \{gen\}. For instance, i*j$\in$\{a;b\} is equivalent to i*j=a$\vee$i*j=b. As in this example, ∇(x:gen) sometimes requires a justification. The conclusion of the latter rule is a $\Leftrightarrow$ and thus requires two justification transformers. First, x$\in$\{gen\} has no deep constructive meaning (no information is ever needed on why it is true), so producing its justification from one of ∇(x:gen) is trivial. (This would not be the case if sets could contain booleans, since by $a \vee b \in \{\mathbf{true}\} \Leftrightarrow a \vee b = \mathbf{true} \Leftrightarrow a \vee b$, $x \in \{gen\}$ would have to carry further information. I have chosen in favor of a simpler theory by limiting the element types of sets.) Second, since sets are always a.e., $\nabla(x{:}gen)$ is a.e. so its justification may be obtained from the truth of $x \in \{gen\}$ by a recursive analysis of its structure (deferred to chapter 4). Thus, justifications for the conclusion of the second rule can be constructed. The first rule yields a simple justification for its conclusion. Since both sides of the biconditional have trivial justifications, the two justification transformers for the $\Leftrightarrow$ always yield the trivial justification.

Recall that the characterization of iterators by predicates placed a restriction on the bound variable names of legal iterators (section 3.2). For example, $x{:}\in\{x;y\}$ is ill-formed because the resulting ∇ predicate, $x=x \vee x=y$, does not correctly characterize the values of x produced by the iterator. However, that predicate does capture the meaning of $x \in \{x;y\}$ since neither instance of x is considered bound. In general, use of the ∇ function in the second $\in$def rule is correct, even though x:gen may not be legal as an iterator, because free uses of x in the generator refer to the value of x in the surrounding scope, just as does the x preceding the colon. This use of ∇ motivated its definition (section 3.2) on an extension of the class of iterators.

The remaining set theory rules are presented without individual discussion.

$\notin$<u>def</u> S pr $x \notin Q \Leftrightarrow \neg x \in Q$

$\cap$<u>def</u> S pr $x \in Q \cap R \Leftrightarrow x \in Q \wedge x \in R$

$\cup$<u>def</u> S pr $x \in Q \cup R \Leftrightarrow x \in Q \vee x \in R$

$-$<u>def</u> S pr $x \in Q-R \Leftrightarrow x \in Q \wedge x \notin R$

$\subseteq$<u>def</u> S pr $Q \subseteq R \Leftrightarrow (\forall a : \in Q)(a \in R)$

$=$<u>def</u> S pr $Q=R \Leftrightarrow Q \subseteq R \wedge R \subseteq Q$

$\subset$<u>def</u> S pr $Q \subset R \Leftrightarrow Q \subseteq R \wedge Q \neq R$

$\#$<u>def</u> S pr $\#$null set(T)=0

also S pr $x \in Q \Leftrightarrow \#(Q$ add $x)=\#Q$

and S pr $x \notin Q \Leftrightarrow \#(Q$ add $x)=\#Q+1$

Justifications for these axioms, where needed, are easily obtained since all expressions are a.e. and the alternatives of disjunctions can be directly tested.

The completeness of the PRL set theory will not be argued. It is sufficiently complete to handle the examples presented herein. As I have noted, the development of a comprehensive theory of sets and other constructed types is left to future research.

As an example of reasoning about sets, consider the description of set reduction operators for union and intersection. These operators map a set of sets to the union (intersection) of all component sets. Given S, a set of sets of objects of type T, the union, R, is characterized by

$$(\forall x{:}T)(x\epsilon R \Leftrightarrow (\exists Q{:}\epsilon S)(x\epsilon Q)).$$

The intersection is characterized by

$$(\forall x{:}T)(x\epsilon R \Leftrightarrow (\forall Q{:}\epsilon S)(x\epsilon Q)).$$

It will be convenient later to know that these descriptions uniquely define R. I shall demonstrate this by supplying a proof schema which may be specialized to the two desired proofs.

Suppose a set R is characterized by $x\epsilon R \Leftrightarrow P_x$, where P is any predicate not involving R. Clearly, P uniquely characterizes R, since if $x\epsilon R' \Leftrightarrow P_x$ then $x\epsilon R' \Leftrightarrow P_x \Leftrightarrow x\epsilon R$, and so $R=R'$. This is formalized as follows.

(∀x:T)(x∈R⇔P_x),(∀x:T)(x∈R'⇔P_x) **pr** R=R' **by** = def

... **pr** R⊂R'∧R'⊂R **by** ∧s

... **pr** R⊂R' **by** ⊂ def

... **pr** (∀a:∈R)(a∈R') **by** ∀s

...,a∈R **pr** a∈R' **by** ∀a on first hyp with a for x

...,a∈R **pr true by** true def

...,a∈R,a∈R⇔P_a **pr** a∈R' **by** ⇔a

...,a∈R,a∈R⇒P_a **pr** a∈R' **by** ⇒a

...,a∈R **pr** a∈R **by** assumption

...,P_a **pr** a∈R' **by** ∀a with a for x

...,P_a **pr true by** true def

...,P_a,a∈R'⇔P_a **pr** a∈R' **by** ⇔a

...,P_a,P_a⇒a∈R' **pr** a∈R' **by** ⇒a

...,P_a **pr** P_a **by** assumption

...,a∈R' **pr** a∈R' **by** assum

... **pr** R'⊂R **by** ⊂ def

>symmetric argument

3.7 Refinement: transition specifications

Transition specification refinement rules provide for the synthesis of programs from simpler ones using several well known constructions, such as sequential composition and unbounded iteration. The selection statement (if-then-else) of conventional languages is provided by ∨analysis. These refinement rules are theoretically adequate for implementing any computable operation, but rules for other control constructs could be included as desired (see chapter 5).

The lowest level transition specifications in a proof are implemented by the system. The rules

<u>ae</u> <env> S pr $x_1,\ldots,x_n \leftarrow$ any a_1:$exp_1,\ldots,a_n$:exp_n

where each exp is an a.e. expression

<u>choice</u> <env> S pr $x \leftarrow$ any a:$\in$exp by

<env> S pr exp$\neq$null set(T)

where x is a single variable and exp

is an expression of type set(T) in <env>

may be used to prove "obviously true" propositions (choice is "obviously true" because all sets are a.e.). Together with the following rule, they are the only ways to refine a transition specification without introducing another one as a subgoal.

<u>null</u> <env> S pr $x \leftarrow$ any iter^a by

<env> S pr $\nabla(\text{iter}^a)_{a \leftarrow x}$

This rule states that no computation need be done if x already contains a value satisfying the specification. The justification showing that x satisfies the iterator is provided by the proof of the subgoal. Examples follow.

x≥0 pr x ← abs(x) by null

x≥0 pr x=abs(x) by arith

(since x ← abs(x) is an abbreviation for

x ← any a:abs(x) and ∇(a:abs(x)) is a=abs(x))

x∈S pr x ← any b:∈S by null

x∈S pr x∈S by assumption

The next rule

<u>simplification</u> <env> S pr x ← any $iter^a$ by

<env> S pr x ← any $iter1^b$

<env,b:T> S,$\nabla(iter1^b)$ pr $\nabla(iter^a)_{a \leftarrow b}$

where T is the type of x in <env> and

b∉<env> (if x is a list of names

then interpret T and b accordingly)

captures the notion that the goal specification may be satisfied by implementing another, apparently simpler, specification. The justification for the conclusion is obtained by flowing the justification for S down through the two subgoals. For example,

x<0 pr x ← abs(x) by simplification

x<0 pr x ← -x

x<0,b=-x pr b=abs(x).

Often, a transition specification cannot be easily implemented without further use of storage. The rule

<u>var</u>   <env> S **pr** x ← **any** iter^{a} **by**

var y:T

<env,y:T> S **pr** x,y ← **any** iter^{a},b:T

where T is any type name, y∉<env>, b≠a,

and b does not occur free in iter^{a}

allows the introduction of a new local variable, y, that may be used in lower levels of refinement. The interior specification does not require that any particular value be left in y, so it is a "scratch" variable.

The rule shown below for composing n sequential transition specifications is somewhat overwhelming at first sight, hence I shall develop it gradually. First, the null and consequence rules are special cases of composition for n=0 and n=1, respectively. A further useful special case is for n=2:

<env> S **pr** x ← **any** iter^{a} **by**

<env>   S **pr** x ← **any** iter1^{b}

<env,b:T>   S,∇(iter1^{b}) **pr** $Q_{x\leftarrow b}$

<env> Q_{x} **pr** x ← **any** iter2^{c}

<env,b:T,c:T> S,∇(iter1^{b}),$\nabla(\text{iter2}^{c})_{x\leftarrow b}$ **pr** $\nabla(\text{iter}^{a})_{a\leftarrow c}$

where T is the type of x in <env>, b≠c, and b,c∉<env>

(if x is a list of names then treat T,b,c accordingly).

The second subgoal shows that Q_{x} holds after executing the first subgoal's code. The last subgoal assures that the value produced by the third subgoal's code satisfies the original specification. The justification for S flows downward producing further justifications for ∇(iter1^{b}), Q_{x}, and ∇(iter2^{c}). These then yield the justification for the conclusion. The effect of symbolic substitution on the justifications of formulas is described in chapter 4.

The full composition rule follows.

<u>composition</u>

$\langle env\rangle$ S **pr** x ← **any** iter^{a} **by**

$\langle env\rangle$ S **pr** $Q1_x$

$\langle env\rangle$ $Q1_x$ **pr** x ← **any** iter1^{a1}

$\langle env, a1{:}T\rangle$ S, $\nabla(\text{iter1}^{a1})$ **pr** $Q2_{x \leftarrow a1}$

$\langle env\rangle$ $Q2_x$ **pr** x ← **any** iter2^{a2}

⋮

$\langle env, a1{:}T, \ldots, an\text{-}1{:}T\rangle$ S, $\nabla(\text{iter1}^{a1}), \nabla(\text{iter2}^{a2})_{x \leftarrow a1},$
$\ldots, \nabla(\text{itern-1}^{an-1})_{x \leftarrow an-2}$ **pr** $Qn_{x \leftarrow an-1}$

$\langle env\rangle$ Qn_x **pr** x ← **any** itern^{an}

$\langle env, a1{:}T, \ldots, an{:}T\rangle$ S, $\nabla(\text{iter1}^{a1}), \nabla(\text{iter2}^{a2})_{x \leftarrow a1},$
$\ldots, \nabla(\text{itern}^{an})_{x \leftarrow an-1}$ **pr** $\nabla(\text{iter}^{a})_{a \leftarrow an}$

where T is the type of x in $\langle env\rangle$, all ai are disjoint, and a1,...,an$\notin\langle env\rangle$ (if x is a list of names then interpret T,a1,...,an accordingly)

The even numbered subgoals specify transitions that are composed sequentially. The odd numbered subgoals provide the reasoning that insures each precondition, Q_i, holds for its associated transition. Note that in all odd numbered (logical) subgoals the names of x denote their original values, while new values are represented by the names a1,...,an. In even numbered (transition) subgoals the names of x represent their "current" value, while a1,...,an are inaccessible. Thus, in the last subgoal, S describes x, $\nabla(\text{iter1}^{a1})$ relates a1 to x, $\nabla(\text{iter2}^{a2})_{x\leftarrow a1}$ relates a2 to a1, and so on. The conclusion, $\nabla(\text{iter}^{a})_{a\leftarrow an}$, states that the current values of names of x (denoted by an and related to an-1, an-2, ... according to the hypotheses) have the desired relation to the original values of names of x (still denoted by x). As with two-fold composition, the justification of the conclusion is the final result of starting with justification of S and proceeding down the list of subgoals acquiring derived justifications from each conclusion.

This rule is a good illustration of the need for an interactive system to support PRL. All of the odd numbered subgoals (though not their proofs) may be derived automatically from the even ones. Thus, handling the preconditions and performing substitutions could be left to the system. The user would indicate his desire to use composition, enter the even numbered subgoals, and the rest of the refinement step would be filled in.

As remarked earlier, logical constants, though not always needed in specifications, have returned to the proof system. They arise in a particularly nasty way in this composition rule. A method for reasoning about transition specifications directly, without first converting to predicates via ∇, would be beneficial.

A simple example of composition follows.

```
S≠small set(int) pr mean,var ← any m:div(sum(S),#S),
                  v:div(sum({x*x for x:∈S}),#S) -
                      div(sum(S),#S)*div(sum(S),#S) by composition
   S≠small set(int) pr S≠small set(int) by assumption
   S≠small set(int) pr mean,var ← any ml:sum(S),vl:sum({x*x for x:∈S}) b
      >proof omitted: this would probably be a loop
   S≠small set(int),ml=sum(S)∧vl=sum({x*x for x:∈S}) pr true by true def
   true pr mean,var ← any m2:div(mean,#S),
                            v2:div(var,#S)-div(mean,#S)*div(mean,#S) by
      >proof omitted: if div is an a.e. function (definition)
                      then the a.e. rule may be used here
   S≠small set(int),ml=sum(S)∧vl=sum({{x*x for x:∈S}),
    m2=div(ml,#S)∧v2=div(vl,#S)-div(ml,#S)*div(ml,#S) pr
     m2=div(sum(S),#S)∧v2=div(sum({x*x for x:∈S}),#S)
     -div(sum(S),#S)*div(sum(S),#S) by
      >proof omitted: simply use ∧a on the last two
      .               hypotheses then use substitution
```

The final rule for transition specification refinement treats unbounded iteration.

<u>iteration</u>

<env> S **pr** $x \leftarrow$ **any** $iter^{a}$ **by**

  invariant $x \leftarrow$ **any** $iter^{i}$ **as** $t_{i}\downarrow$ **while** B_{i}

<env> S **pr** $x \leftarrow$ **any** $iter^{i}$

<env,i:T> $S,\nabla(iter^{i})$ **pr** $B_{i} \vee \neg B_{i}$

<env,i:T> $S,\nabla(iter^{i}),B_{i}$ **pr** $Q_{x\leftarrow i}$

<env> Q_{x} **pr** $x \leftarrow$ **any** $iter2^{b}$

<env,i:T,b:T> $S,\nabla(iter^{i}),B_{i},\nabla(iter2^{b})_{x\leftarrow i}$ **pr**

  $\nabla(iter^{i})_{i\leftarrow b} \wedge t_{i\leftarrow b} < t_{i}$

<env,i:T> $S,\nabla(iter^{i}),t_{i}\leq 0$ **pr** $\neg B_{i}$

<env,i:T> $S,\nabla(iter^{i}),\neg B_{i}$ **pr** $\nabla(iter^{a})_{a\leftarrow i}$

where T is the type of x in <env> , b,i$\notin$<env>, t_i is integer valued, and B_i is boolean valued (if x is a list of names then interpret T and i accordingly)

In this rule the names of x denote their original values, i.e., the ones satisfying S, in all subgoals except the fourth ($x \leftarrow$ **any** $iter2^{b}$), where the names of x denote their values at the start of each iteration. The invariant transition $x \leftarrow$ **any** $iter^{i}$ describes all the values that x might contain after any number of iterations of the loop. Thus, the first subgoal (the second line of the refinement) establishes that the names of x contain some tuple of values produced by $iter^{i}$ -- this is initialization for the loop. The second subgoal shows that the predicate B_i can always be tested; the justification it produces is used to decide when to terminate the loop. The third subgoal shows that whenever B_i holds, so will the precondition for the fourth subgoal. The fourth subgoal (the loop body) changes the values of x to another tuple which, by the fifth subgoal, satisfies $iter^i$ and is smaller than the previous tuple when measured by the integer valued termination function t_i. The sixth subgoal insures that when t_i reaches zero the loop will terminate. The last subgoal shows that any tuple of $iter^i$ that satisfies $\neg B_i$ is an acceptable result for x according to the original goal. The flow of justifications for this rule is left to the next chapter.

The following example shows how the union reduction (introduced in the last section) may be computed. Given S, a set of sets of objects of type T, this loop computes a set R which is the union of the component sets of S.

```
pr R ← any Ra:set(T) st (∀x:T)(x∈Ra⟺(∃Q:∈S)(x∈Q)) by
    var v:set(set(T))
    pr R,V ← any Ra:set(T) st (∀x:T)(x∈Ra⟺(∃Q:∈S)(x∈Q)),
                  Va:set(set(T)) by iteration
        invariant R,V ← any Ri:set(T),Vi:set(set(T)) st
                  Vi⊆S∧(∀x:T)(x∈Ri⟺(∃Q:∈Vi)(x∈Q))
                  as #(S-Vi)↓ while Vi≠S
           (let inv mean Vi⊆S∧(∀x:T)(x∈Ri⟺(∃Q:∈Vi)(x∈Q)))
        pr R,V ← any Ri:set(V),Vi:set(set(T)) st inv by simpl
           pr R,V ← null set(T),null set(set(T)) by a.e.
           R'=null set(T),V'=null set(set(T)) pr
                  V'⊆S∧(∀x:T)(x∈R' ⟺(∃Q:∈V')(x∈Q)) by
              >proof omitted
        inv pr Vi≠S∨Vi=S by excluded middle
        inv,Vi≠S pr Vi⊂S by
           >proof omitted: Vi⊆S,Vi≠S⟹vi⊂S by set theory
        V⊂S pr R,V ← any (Rb:R∪Q,Vb:V∪{Q}) for Q:∈(S-V) by
           >proof omitted: introduce var Q, choose Q from S-V,
                     set V and R
        inv,Vi≠S,(∃Q:∈S-Vi)(Vb=Vi∪{Q}∧Rb=Ri∪Q) pr
                  Vb⊆S∧(∀x:T)(x∈Rb⟺(∃Q:∈Vb)(x∈Q))
                  ∧#(S-Vb)<#(S-Vi) by
           >large proof omitted: should draw on a previous
                          development of set theory
        inv,#(S-Vi)≤0 pr ¬(Vi≠S) by
           >proof omitted: again, should be based on a
                     separate development of set theory
```

$\textbf{inv},\neg(Vi{\neq}S)\ \textbf{pr}\ (\forall x{:}T)(x{\in}Ri{\Leftrightarrow}(\exists Q{:}{\in}S)(x{\in}Q))\ \textbf{by}\ \text{cons}$

$\textbf{inv},\neg(Vi{\neq}S)\ \textbf{pr}\ Vi{=}S\ \textbf{by}$

>proof as in section 3.5

$\textbf{inv},Vi{=}S\ \textbf{pr}\ (\forall x{:}T)(x{\in}Ri{\Leftrightarrow}(\exists Q{:}{\in}S)(x{\in}Q))\ \textbf{by}\ \text{subst}$

$\textbf{inv},Vi{=}S\ \textbf{pr}\ S{=}Vi\ \textbf{by}\ \text{symmetry}$

$\textbf{inv},Vi{=}S\ \textbf{pr}\ Vi{=}S\ \textbf{by}\ \text{assumption}$

$\textbf{inv},Vi{=}S\ \textbf{pr}\ (\forall x{:}T)(x{\in}Ri{\Leftrightarrow}(\exists Q{:}{\in}Vi)(x{\in}Q))\ \textbf{by}\ \text{assum}$

3.8 Abstraction

Stating and solving complex programming problems requires the formulation of complex specifications and proofs. These may be far more easily understood and manipulated if facilities are provided for naming and parameterizing their components, thus capturing useful abstractions. I shall present two mechanisms for achieving this abstraction: first definitions (of two varieties) and then lemmas.

The power of the specification language derives from the use of complex predicates and expressions. A definition associates a parameterized expression, i.e., a function, with a name. The simplest form is

$$\textbf{def}\ D(b{:}T) \equiv exp_b$$

where D is the definition name, b is a list of distinct parameter names, T describes the type of each parameter, and exp_b is the definition body. The free variables of the expression are limited to the names of b and the expression must be well-formed in the environment <b:T> (well-formedness is precisely defined in section 3.9).

In order to simplify the logic, definitions may not be recursive (directly or mutually) and must be global to all proofs (i.e., there is no notion of the scope of a definition). Relaxing the restriction on recursion would be valuable; it is a purely technical problem for future research. Whether to permit local definitions, and if so in precisely what form, is a deeper problem. The goal is to encourage modular proofs that fit well into the machine aided refinement style. This broad question is also left to further study.

Definitions may be referenced with a rule of the following style.

<env> $S_{x \leftarrow D(e)}$ **pr** $C_{x \leftarrow D(e)}$ **by**

<env> $S_{x \leftarrow exp_{b \leftarrow e}}$ **pr** $C_{x \leftarrow exp_{b \leftarrow e}}$

where exp is the defined body of D and e is any list of a.e. expressions whose types in <env> match those of T.

(As in the substitution rule, the name x represents those places in S, C where D(e) occurs.) This rule is just the substitution rule with the subgoal ... **pr** $D(e)=exp_{b \leftarrow e}$ left implicit. An equivalent system would result if definitions were referenced by the rule S **pr** $D(e)=exp_{b \leftarrow e}$, but the above form is often more useful. For example:

def diviаors(n:**int**) ≡ {i:1 **to** n **st** (∃j:1 **to** n)i*j=n}

def prime(n:**int**) ≡ divisors(n)={1;n}

pr prime(5) **by** definition of prime

pr divisors(5)={1;5} **by** definition of divisors

pr {i:1 **to** 5 **st** (∃j:1 **to** 5)i*j=5} = {1;5} **by**

>further proof text

A definition invocation, D(e), is s.e. iff the body of the definition, exp_b, is s.e.. Since all parameters to an invocation are s.e., the value of an s.e. invocation may be computed by evaluating the parameters, binding the formal names to the values, and evaluating the definition body. Thus, given that the definition bodies are available for inspection, constructs involving definition invocations can still be syntactically checked for being s.e..

Often the parameter types do not affect the meaning or well-formedness of a definition. For instance,

def flatten(R:**set**(**set**(?T)) ≡ {∈Q **for** Q:∈R}

is a polymorphic (many types) definition that maps **set**(**set**(element type)) to **set**(element type) for any element type. The variable ?T is an implicit type parameter that may be used as a type name in the body of the definition. A type parameter may occur several times in the formal parameter list, in which case any invocation must define the type parameter uniformly. Thus, the definition

$$\textbf{def}\ \mathrm{overlapping}(R{:}\mathbf{set}(?T),Q{:}\mathbf{set}(?T)) \equiv (\exists x{:}?T)(x\in R \wedge x\in Q) \wedge (\exists x{:}?T)(x\in R \wedge x\notin Q) \wedge (\exists x{:}?T)(x\notin R \wedge x\in Q)$$

may be invoked only if the two actual parameters have identical types (set of something). Note the use of the type parameter in the definition body. At each instantiation, the body of a polymorphic definition is evaluated with the type parameters replaced by actual types.

Checking polymorphic definitions for well-formedness requires extending the class of types and environments to handle type parameters. In order to limit the complexity of the extraction process, types may be abstracted only with respect to element types, i.e., a type ?T cannot be instantiated as a **bool**. (Otherwise a formal parameter of type ?T would have to carry complex justifications sometimes and trivial justifications other times. This leads to extracted code which performs different activities depending on the types of variables. In this work, I have chosen to avoid run-time manipulation of type information.) Thus, let type names be defined by

```
type → bool                    element type → int
     | element type                         | set ( element type )
                                            | ?var
```

where ?var is the class of variable names that start with "?". An environment is a set of clauses of the form

```
clause → var : type
       | ?var
```

where no var or ?var appears more than once. A type of the form ?var is legal only if it is present in the environment of the proposition in which it is used. These definitions extend those given in section 3.1.

There are several ways of treating type parameters in a polymorphic language. This area continues to be widely researched, with much debate on the best approach [Demers and Donahue 1979, Milner 1977, Liskov et.al. 1977, Wulf 1978]. I have opted for a very simple system; all type parameters in an environment are treated as representing distinct types. Thus, two types are equal exactly when their names are identical. Further, the type of any expression is uniquely determined. These properties enable the introduction of polymorphism with minimal conceptual extensions; the only changes are to the type checking rules (still to be presented in section 3.9) and the invocation of abstractions. In particular, the proof rules, ∇ function, and set theory remain correct.

Consider a definition

$$\textbf{def } D(b:T) \equiv exp_b$$

that may be polymorphic. The environment of the body is <b:T> augmented by ?var for each type name, ?var, occurring in the parameter list. The expression must be well-formed in this <u>augmented environment</u>. The rule for definition reference developed above must be modified to handle polymorphism. First, the interpretation of the word "match" (as in "e is any list of a.e. expressions whose types in <env> match those of T") must be extended.

Given an invocation with actual parameters of types $a_1,\ldots,a_n$ of a well-formed abstraction whose formal parameters are of types $f_1,\ldots,f_n$, say the types of the actuals <u>match</u> the types of the formals iff

$$\text{a) } (\forall i{:}1 \textbf{ to } n)(\text{match}(f_i,a_i) \neq \text{"fail"})$$

$$\text{and b) } (\forall i,j{:}1 \textbf{ to } n)(\text{match}(f_i,a_i)=\text{"?var}\leftarrow t_1\text{"} \wedge \text{match}(f_j,a_j)=\text{"?var}\leftarrow t_2\text{"} \Rightarrow t_1=t_2)$$

where match(f,a) $\equiv$

f=a=**bool** $\Rightarrow$ "succeed"

f=a=**int** $\Rightarrow$ "succeed"

f=**set**(t)$\wedge$a=**set**(t') $\Rightarrow$ match(t,t')

f=?var $\Rightarrow$ "?var$\leftarrow$a"

else "fail".

That is, the types match if all individual parameter types match and if the type parameters may be assigned a globally consistent interpretation. For example, "flatten" may be invoked with any actual parameter that is a set of sets of T, for any element type T. The body of "flatten" must be correct in the augmented environment <R:**set**(**set**(?T)),?T>. If "flatten" is invoked with a **set**(**set**(**set**(**int**))) then match yields "?T$\leftarrow$**set**(**int**)".

Consider a particular definition invocation and let ?var$\leftarrow$t represent the composite substitution derived from the results of match applied to each parameter. Thus, ?var$\leftarrow$t changes all uses of type parameters into the corresponding actual types supplied by the invocation. The type of a definition invocation can now be defined as the type of $\text{exp}_{?var\leftarrow t}$ in an environment which is $\langle b{:}T\rangle_{?var\leftarrow t}$ joined with any type variables occurring in t that are present in the invoking environment. For instance, if "flatten" is invoked on P of type ?T' (in the invoking environment) then the type of flatten(P) is the type of $\{\epsilon Q$ **for** $Q{:}\epsilon R\}$ in $\langle R{:}\mathbf{set}(\mathbf{set}(?T))\rangle_{?T\leftarrow ?T'}$ joined by ?T', i.e., in $\langle R{:}\mathbf{set}(\mathbf{set}(?T')),?T'\rangle$. The general rule for referencing polymorphic definitions is now expressible.

<u>**definition**</u> $\langle env\rangle\ S_{x\leftarrow D(e)}$ **pr** $C_{x\leftarrow D(e)}$ **by**

$\langle env\rangle\ S_{x\leftarrow (exp_{b},?var\leftarrow e,t)}$ **pr** $C_{x\leftarrow (exp_{b},?var\leftarrow e,t)}$

where there is a definition $D(b{:}T)\equiv exp_b$, e is any a.e. expression whose types in <env> match those of T, and "?var←t" is the composite substitution resulting from the parameter type matching process.

The form of definition that I have been discussing is useful for shortening the expressions used in specifications, but it adds no real expressive power to those specifications. Often, a value is most easily specified by a predicate uniquely characterizing it. PRL supports this second variety of definition by allowing a definition's body to be

the $iter^{x}$

where x is a single name and there is exactly one value for x produced by the iterator. This uniquely determined value is the result of invoking the definition.

Associated with each definition of this form are two parts. The first shows the constructive existence of a value satisfying the iterator. The second shows that the value is unique. These proofs are necessary to insure that syntactically legal definition invocations are well-defined deterministic expressions, i.e., that reflexivity holds for them. Thus,

$$\textbf{def } D(b{:}T) \equiv \textbf{the } iter^x_b$$

$$\langle y{:}T',env'\rangle \quad \textbf{pr } y \leftarrow \textbf{any } iter^x_b$$

$$\langle x{:}T',y{:}T',env'\rangle \; \nabla(iter^x_b),\nabla(iter^x_b)_{x\leftarrow y} \; \textbf{pr } x{=}y$$

where D,b,T are as above, x is a single name, env' is the augmented environment of <b:T>, T' is the type of x (in $iter^x_b$) in <env'>, and $y \neq x$, $y \neq b$. The type of the definition when invoked is the type of $iter^x_{?var\leftarrow t}$ in $\langle b{:}T\rangle_{?var\leftarrow t}$ joined by any type variables occurring in t that are present in the invoking environment (as explained above). Such definitions may be referenced by

<u>definition evaluation</u>

$$\langle env\rangle \; S \; \textbf{pr} \; \nabla(iter^x_b)_{x,b,?var\leftarrow D(e),e,t}$$

where e is a list of s.e. expressions whose types in <env> match those of T and "?var←t" is the result of parameter matching and

<u>definition uniqueness</u>

$\langle env\rangle\ S_{y\leftarrow D(e)}$ **pr** $C_{y\leftarrow D(e)}$ **by**

$\langle env\rangle\ S_{y\leftarrow D(e)}$ **pr** $\nabla(\mathrm{iter}_b^x)_{x,b,?var\leftarrow exp,e,t}$

$\langle env\rangle\ S_{y\leftarrow exp}$ **pr** $C_{y\leftarrow exp}$

where e,"?var←t" are as above, exp is an a.e. expression whose type in <env> is that of D(e), and y is a place holder for occurrences of D(e) in S, C.

The first rule uses the existence property to guarantee that the value produced, D(e), satisfies the iterator. The second rule uses the uniqueness property to allow replacing D(e) by any expression, exp, that satisfies the iterator. This rule could be phrased

S **pr** exp=D(e) **by**

S **pr** $\nabla(\mathrm{iter}_b^x)_{x,b,?var\leftarrow exp,e,t}$

but again it is more often applicable in the substitution form.

This form of definition provides a method for computing the values of invocations even if iter_b^x is not a.e.. Thus, an invocation of this form of definition is always considered a.e..

The justification of an invocation, D(e), of this kind of definition describes how the value returned, say x, satisfies $\nabla(\mathrm{iter}_b^x)_{b,?var\leftarrow e,t}$. This is further discussed in section 4.6.

The following example demonstrates the use of definitions. Two set reduction operators, union and intersection, are defined and used to manipulate ordered pairs of integers.

```
def unred(S:set(set(?T)))≡
    the R:set(?T) st (∀x:?T)(x∈R⇔(∃Q:∈S)(x∈Q))
        pr R' ← any R:set(?T) st (∀x:?T)(x∈R⇔(∃Q:∈S)(x∈Q)) by
            >proof by iteration as shown in section 3.7
        (∀x:?T)(x∈R⇔(∃Q:∈S)(x∈Q)),(∀x:?T)(x∈R'⇔(∃Q:∈S)(x∈Q))
                              pr R = R' by
            >proof as shown in section 3.6


def inred(S:set(set(?T)))≡
    the R:set(?T) st (∀x:?T)(x∈R⇔(∀Q:∈S)(x∈Q))
        >proofs as for unred


def pair(i:int,j:int)≡ {{i};{i;j}}


def ispair(p:set(set(int)))≡ (∃x,y:int)(p=pair(x,y))
```

```
def hd(p:set(set(int)))≡
    the h:int st ispair(p)⇒(∃t:int)(p=pair(h,t))
            ∧ ¬ispair(p)⇒h=0
        pr y ← any h:int st ispair(p)⇒(∃t:int)(p=pair(h,t))
                        ∧ ¬ispair(p)⇒h=0 by consequence
            pr ispair(p)∨¬ispair(p) by excluded middle
            ispair(p)∨¬ispair(p) pr ... by ∨a
                ispair(p) pr ... by simplification
                    ispair(p) pr y ← any h:∈inred(p) by choice
                        ispair(p) pr inred(p)≠null set(int) by cons
                .   .       ispair(p) pr (∀x:int)(x∈inred(p)
                                       ⇔(∀Q:∈p)(x∈Q)) by def eval
                .   .       ispair(p),... pr inred(p)≠null set(int) by
                                >proof omitted
                .   ispair(p),h∈inred(p) pr
                            ispair(p)⇒(∃t:int)(p=pair(h,t))
                .       ∧ ¬ispair(p)⇒h=0 by
                        >proof omitted
                ¬ispair(p) pr ... by
                    >proof omitted: set h to 0
        >uniqueness statement and proof omitted


def tl(p:set(set(int)))≡
    the t:int st ispair(p)⇒(∃h:int)(p=pair(h,t))
            ∧ ¬ispair(p)⇒t=0
        >proofs as for hd: compute tl as any t:∈(unred(p)-inred(p))
```

I will now discuss the second abstraction mechanism, the lemma, which supports modular proofs. A proposition may be named and parameterized by writing

$$\textbf{lemma}\ L\ v{:}VT(b{:}BT) \equiv \langle env\rangle\ S_{v,b}\ \textbf{pr}\ C_{v,b}$$

where L is the lemma name, v are the variable parameter names of types VT, b are the constant parameter names of types BT, the combined names of v,b are all distinct, <env> is the environment <v:VT,b:BT> augmented to include type parameters, and v:VT is present only if C is a transition specification of the form v ← any $iter_{v,b}$. Subsequent to such a (proven) lemma, one may use the rule

$$\underline{\text{lemma}}\quad \langle env\rangle\ S_{v,b,?var\leftarrow n,e,t}\ \textbf{pr}\ C_{v,b,?var\leftarrow n,e,t}\ \textbf{by}\ L\ n(e)$$

where S, C are from the lemma definition, n is a list of distinct variable names whose types in <env> match VT, e is a list of a.e. expressions whose types in <env> match BT, and "?var←t" is the composite substitution resulting from matching all the parameters.

If C is a boolean expression, the lemma is of the classical mathematical kind. If C is a transition specification, however, the lemma may be thought of as a procedure definition whose invocation behaves according to the specification C with actual parameters syntactically substituted for formal parameters. This interpretation forces distinctness of the variable actual parameters, n, since the resulting transition specification might be ill-formed otherwise.

This semantics for parameter passing reminds one of ALGOL 60's call by name mechanism. However, since there is no way to have a variable's location depend on another variable's value, the call by reference mechanism is adequate for handling v. Further, since the proof of a lemma assumes that names of v and b denote independent, distinct variables, the implementation must handle actual parameters for b with call by value, because a change to v must not affect the values of b even if the actual parameters are identical variables. These details (the creation and manipulation of justifications for lemmas) are treated in the next chapter. It is interesting that the programmer may ignore such subtleties; the rule for "procedure call" is very simple.

As an example, the following pure logical lemma outlines a proof that pairs uniquely determine their components.

```
lemma pair_uniqueness(h:int,t:int,h':int,t':int)
    pair(h,t)=pair(h',t') pr h=h'∧t=t' by def of pair(twice)
        {{h};{b;t}}={{h'};{h';t'}} pr h=h'∧t=t' by consequence
                (let R abbreviate {{h};{h;t}}
                 and R' abbreviate {{h'};{b';t'}} below)
            R=R pr R⊂R'∧R'⊂R by
                >proof omitted: use =def and reasoning about ⇔
            R=R',R⊂R∧R'⊂R pr (∀s:∈R)(s∈R')
                >proof omitted: use ⊂def and reasoning about ⇔
            R=R',R⊂R'∧R'⊂R,(∀s:∈R)(s∈R')
                                pr h=h'∧t=t' by consequence
                (∀s:∈R)(s∈R') pr h=h' by ∀s with {h} for s
                    pr {h}={h}∨{h}={h;t} by ∨s
                        pr {h}={h} by reflexivity
                    {h}∈R' pr h=h' by cons
                        {h}∈R' pr {h}={h'}∨{h}={h';t'} by
                            >proof omitted: use ∈def, ⇔ reasoning
                        {h}={h'}∨{h}={h';t'} pr h=h' by ∨s
                            {h}={h'} pr h=h' by
                                >proof omitted
                            {h}={b';t'} pr h=h' by
                                >proof omitted
                R=R',R⊂R'∧R'⊂R,(∀s:∈R)(s∈R'),h=h' pr t=t' by
                    >proof omitted
                h=h',t=t' pr h=h'∧t=t' by ∧s
                    h=h',t=t' pr h=h' by assumption
                    h=h',t=t' pr t=t' by assumption
```

The next few definitions introduce several notions about directed graphs.

Let "pair" be an abbreviation for **set(set(int))**, "seq" be an abbreviation for **set**(pair), "edge" be an abbreviation for pair, and "graph" be an abbreviation for **set**(edge). A sequence of integers n1,n2,n3,...,nk-1,nk will be represented by the set {pair(1,n1),...,pair(k,nk)}. A set may be checked for representing a sequence and the $i^{\underline{th}}$ element of a sequence may be selected, as follows.

```
def isseq(p:set(pair))≡
    #p≥1 ∧ (∀x:∊p)(ispair(x)) ∧ {hd(x) for x:∊p}={1 to #p}


def select(p:seq,i:int)≡
    the n:int st (isseq(p)∧1≤i≤#p ⇒ pair(i,n)∊p)
                 ∧ (¬(isseq(p)∧1≤i≤#p) ⇒ n=0)
        >proofs omitted
```

A graph is represented as a set of edges. A path in a graph G is a sequence of nodes which are connected by edges of G. A node. n, is reachable from another node, r, in a graph G if there is a path from r to n in G.

```
def ispath(p:seq,G:graph)≡
    isseq(p) ∧ (∀i:1 to #p-1)(pair(select(p,i),select(p,i+1))∊G)

def reachable(n:int,r:int,G:graph)≡
    (∃p:seq)(ispath(p,G) ∧ select(p,1)=r ∧ select(p,#p)=n)
```

The following lemma outlines a procedure for computing the nodes reachable in G from a selected node r.

```
lemma reach NS:set(int) (r:int,G:graph)≡
    r∈nodes(G) pr NS ← any S:{n:∈nodes(G) st reachable(n,r,G)} by
        var examined: set(int)
        r∈nodes(G) pr NS,examined ← any ....set(int) by iteration
            >proof omitted: start with NS={r},examined=∅,
             repeatedly pick (but don't delete) a node from NS
             (the reached nodes) which has not been examined,
             add it to examined, and add all its sons to NS,
             until there are no more such nodes
```

3.9 Static [illegible] and type [illegible]

Throughout this chapter, I have referred to rules for static analysis; these are presented below. The refinement rules and extraction process are only valid when all components of proofs are well-formed. Thus, a mechanism must be introduced to prohibit erroneous forms from arising.

The static analysis mechanism is presented as a set of functions that operate on syntactic objects. In addition to checking well-formedness, the functions define the free and bound variables of constructs, the scopes of names, and the types of expressions, iterators, and generators. These would be included in an interactive system in such a way that suggested refinements could be rejected as ill-formed as they were initially entered.

The environment of each line in a proof is determined by the rules of refinement. Those rules assure the validity of a proof given the well-formedness of each line. Thus, the goal is to define a boolean valued function W<env>[syntactic construct] that is true exactly when the construct is well-formed in the environment.

In order to define W, I shall introduce several auxiliary functions. T<env>[syntactic construct] yields information on the type(s) of well-formed constructs. Given W<env>[exp], the type of exp in <env> is determined by T<env>[exp]. Similarly, T<env>[iter] is a sequence of types that correspond to the bound variables of iter. Finally, T<env>[gen] is the type in <env> of the (values produced by the) generator. The types yielded by T will be strings of the grammar for type presented in the last section. An environment is a set of clauses, each of the form var:type or ?var, also as described in the last section. In any given environment, only certain strings from the type grammar are meaningful. An element type is legal in <env> iff its type parameter (if any) appears in <env>. A type is legal iff it is bool or a legal element type. N[iter] yields the sequence of names bound by the iterator, i.e., N[iter^x] yields x. F[syntactic construct] yields the set of free variable names of the construct. Lastly, E<env>[iter] provides the modification of <env> that is appropriate in the scope of iter.

The functions manipulate names, strings, integers, sets, and sequences. Strings and names may be compared using = and ≠. Integers and sets are treated much as in PRL; set expressions are sometimes formed using generators. Sequences may be created with sequence brackets [...], subscripted (e.g., S(1) is the first element of S), concatenated (||), their size determined (#), and their content checked (ε). The notation used is flexible; I will explain any unusual constructions where they first appear.

<u>Propositions</u>

W<env>[boolexp_1,...,boolexp_n pr conclusion] ≡

W<env>[boolexp_1] ∧...∧ W<env>[boolexp_n] ∧ W<env>[conclusion]

A proposition is well-formed iff all its components are. The other functions need not be defined for propositions.

<u>Transition specifications</u>

W<env>[x_1,...,x_n ← any iter] ≡

W<env>[iter] ∧ #I:[iter]=n ∧

(∀ i:1 to n)(x_iεenv ∧ T<env>[x_i]=T<env>[iter](i))

A transition specification is well-formed iff its iterator is well-formed, the iterator binds the proper number of names, all the assigned variables are accessible, and the types of the assigned variables are the same as those of the iterator. The predicate "nameεenv" is taken to mean that name occurs in some name:type clause of <env>.

Iterators

For any iterator, E<env>[iter] is the environment created by the iterator for use throughout its scope. For example, in $(\forall x{:}\mathbf{int})P_x$ the scope of x:**int** is P_x and the environment is the original one augmented by the clause x:**int**. E may be defined using T and N without deeper examination of the form of iter.

$$E\langle env\rangle[iter] \equiv env - \{\text{"n:anything"} \textbf{ for } n{:}\in N[iter]\}$$
$$\cup \{\text{"n:t"} \text{ where } n=N[iter](i),\ t=T\langle env\rangle[iter](i) \textbf{ for } i{:}1 \textbf{ to } \#N[iter]\}$$

From here on each syntactic form will be listed followed by the text of all functions defined on that form. The environment parameter of each function is implicitly <env>. I shall also limit my comments to difficult or unusual constructions, as English is botb less concise and less accurate than the formalism.

$[x_1,\ldots,x_n : gen]$

$N \equiv [x_1;x_2;\ldots;x_n]$

$F \equiv F[gen]$

$T \equiv [T\langle env\rangle[gen] \textbf{ for } i{:}1 \textbf{ to } n]$

$W \equiv W\langle env\rangle[gen] \wedge (\forall i{:}1 \textbf{ to } n)(x_i \notin F[gen])$

The type of the iterator is a sequence of length n whose every element is the type of the generator. The iterator is well-formed only if no x_i occurs free in the generator (see section 3.2).

[iter st boolexp]

$N \equiv N[iter]$

$F \equiv F[iter] \cup (F[boolexp]-N[iter])$

$T \equiv T<env>[iter]$

$W \equiv W<env>[iter] \wedge W<E<env>[iter]>[boolexp]$

In the body of F, N[iter] is undergoing implicit conversion from a sequence to a set, i.e., it really means {n:∈N[iter]}. In W, the environment for checking well-formedness of the boolean expression is that yielded by E<env>[iter].

$[iter_1 \text{ for } iter_2]$

$N \equiv N[iter_1]$

$F \equiv F[iter_2] \cup (F[iter_1]-N[iter_2])$

$T \equiv T<E<env>[iter_2]>[iter_1]$

$W \equiv W<env>[iter_2] \wedge W<E<env>[iter_2]>[iter_1]$

$\quad \wedge (N[iter_1] \cap F[iter_2]=\emptyset)$

$[iter_1 , iter_2]$

$N \equiv N[iter_1] \,||\, N[iter_2]$

$F \equiv F[iter_1] \cup F[iter_2]$

$T \equiv T<env>[iter_1] \,||\, T<env>[iter_2]$

$W \equiv W<env>[iter_1] \wedge W<env>[iter_2]$

$\quad \wedge (N[iter_1] \cap F[iter_2]=\emptyset) \wedge (N[iter_2] \cap F[iter_1]=\emptyset)$

<u>Generators</u>

[type]

$F \equiv \emptyset$

$T \equiv$ type

$W \equiv$ type is a legal type

[exp]

$F \equiv F[\mathrm{exp}]$

$T \equiv T\langle \mathrm{env}\rangle[\mathrm{exp}]$

$W \equiv W\langle \mathrm{env}\rangle[\mathrm{exp}]$

$[\mathrm{exp}_1$ **to** $\mathrm{exp}_2]$

$F \equiv F[\mathrm{exp}_1] \cup F[\mathrm{exp}_2]$

$T \equiv$ **int**

$W \equiv W\langle \mathrm{env}\rangle[\mathrm{exp}_1] \wedge W\langle \mathrm{env}\rangle[\mathrm{exp}_2]$

$\wedge\ T\langle \mathrm{env}\rangle[\mathrm{exp}_1]=T\langle \mathrm{env}\rangle[\mathrm{exp}_2]=$**int**

$[\in \mathrm{exp}]$

$F \equiv F[\mathrm{exp}]$

$T \equiv t$ **where** $T\langle \mathrm{env}\rangle[\mathrm{exp}]=$**set**$(t)$

$W \equiv W\langle \mathrm{anv}\rangle[\mathrm{exp}]$

$\wedge\ (\exists t$:legal element type$)(T\langle \mathrm{env}\rangle[\mathrm{exp}]=$**set**$(t))$

[⊂ exp] , [⊂ exp]

$F \equiv F[exp]$

$T \equiv T\langle env\rangle[exp]$

$W \equiv W\langle env\rangle[exp]$

$\wedge\ (\exists t\text{:legal element type})(T\langle env\rangle[exp]=set(t))$

[iter]

$F \equiv F[iter]$

$T \equiv T\langle env\rangle[iter](1)$

$W \equiv W\langle env\rangle[iter] \wedge \#N[iter]=1$

[gen for iter]

$F \equiv F[iter] \cup (F[gen]-N[iter])$

$T \equiv T\langle E\langle env\rangle[iter]\rangle[gen]$

$W \equiv W\langle env\rangle[iter] \wedge W\langle E\langle env\rangle[iter]\rangle[gen]$

$[gen_1 \; ; \; gen_2]$

$F \equiv F[gen_1] \cup F[gen_2]$

$T \equiv T\langle env\rangle[gen_1]$

$W \equiv W\langle env\rangle[gen_1] \wedge W\langle env\rangle[gen_2]$

$\wedge\ T\langle env\rangle[gen_1]=T\langle env\rangle[gen_2]$

Boolean expressions

$[\![\mathbf{true}]\!]$, $[\![\mathbf{false}]\!]$

$F \equiv \emptyset$

$T \equiv \mathbf{bool}$

$W \equiv \mathbf{true}$

$[\![\neg\ exp]\!]$

$F \equiv F[\![exp]\!]$

$T \equiv \mathbf{bool}$

$W \equiv W\langle env\rangle[\![exp]\!] \wedge T\langle env\rangle[\![exp]\!]=\mathbf{bool}$

$[\![exp_1\ boolop\ exp_2]\!]$ where $boolop \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

$F \equiv F[\![exp_1]\!] \cup F[\![exp_2]\!]$

$T \equiv \mathbf{bool}$

$W \equiv W\langle env\rangle[\![exp_1]\!] \wedge W\langle env\rangle[\![exp_2]\!]$

$\wedge\ T\langle env\rangle[\![exp_1]\!]=T\langle env\rangle[\![exp_2]\!]=\mathbf{bool}$

$[\![(\forall iter)exp]\!]$, $[\![(\exists iter)exp]\!]$

$F \equiv F[\![iter]\!] \cup (F[\![exp]\!]-N[\![iter]\!])$

$T \equiv \mathbf{bool}$

$W \equiv W\langle env\rangle[\![iter]\!] \wedge W\langle E\langle env\rangle[\![iter]\!]\rangle[\![exp]\!]$

$\wedge\ T\langle E\langle env\rangle[\![iter]\!]\rangle[\![exp]\!]=\mathbf{bool}$

<u>Integer expressions</u>

[con] where con$\in${...,-2,-1,0,1,2,...}

$F \equiv \emptyset$

$T \equiv$ **int**

$W \equiv$ **true**

[exp_1 intop exp_2] where intop$\in${*,+,-}

$F \equiv F[exp_1] \cup F[exp_2]$

$T \equiv$ **int**

$W \equiv W\langle env\rangle[exp_1] \wedge W\langle env\rangle[exp_2]$

$\wedge\ T\langle env\rangle[exp_1]=T\langle env\rangle[exp_2]=$**int**

[- exp]

$F \equiv F[exp]$

$T \equiv$ **int**

$W \equiv W\langle env\rangle[exp] \wedge T\langle env\rangle[exp]=$**int**

[$exp_{1,1}, exp_{1,2}, \ldots, exp_{1,m_1}$ relop ...

relop $exp_{n,1}, \ldots, exp_{n,m_n}$] where each relop$\in${=,≠,<,≤,>,≥}

$F \equiv \{\in F[exp_{i,j}]$ **for** $j{:}1$ **to** m_i **for** $i{:}1$ **to** $n\}$

$T \equiv$ **bool**

$W \equiv (\forall i{:}1$ **to** $n)(\forall j{:}1$ **to** $m_i)(W\langle env\rangle[exp_{i,j}]$

$\wedge\ T\langle env\rangle[exp_{i,j}]=$**int**)

<u>Set expressions</u>

[null set(t)]

F ≡ ∅

T ≡ **set**(t)

W ≡ t is a legal element type

[{gen}]

F ≡ F[gen]

T ≡ **set**(t) **where** t=T<env>[gen]

W ≡ W<env>[gen] ∧ T<env>[gen] is a legal element type

∧ gen is a.e.

gen may be shown a.e. by straightforward syntactic examination as described in section 3.2.

[exp_1 setop exp_2] where setop∈{∩,∪,-}

F ≡ F[exp_1] ∪ F[exp_2]

T ≡ T<env>[exp_1]

W ≡ W<env>[exp_1] ∧ W<env>[exp_2] ∧ (∃t:legal element type)

(T<env>[exp_1]=T<env>[exp_2]=**set**(t))

[# exp]

F ≡ F[exp]

T ≡ **int**

W ≡ W<env>[exp]

∧ (∃t:legal element type)(T<env>[exp]=**set**(t))

$[exp_1 \in exp_2]$

$F \equiv F[exp_1] \cup F[exp_2]$

$T \equiv$ **bool**

$W \equiv W\langle env\rangle[exp_1] \wedge W\langle env\rangle[exp_2] \wedge (\exists t:\text{legal element type})$

$(T\langle env\rangle[exp_2]=\mathbf{set}(t) \wedge T\langle env\rangle[exp_1]=t)$

$[exp_{1,1},exp_{1,2},\dots,exp_{1,m_1}$ relop ...

relop $exp_{n,1},\dots,exp_{n,m_n}]$ where relop$\in\{=,\neq,\subset,\subseteq\}$

$F \equiv \{\in F[exp_{i,j}]$ **for** $j:1$ **to** m_i **for** $i:1$ **to** $n\}$

$T \equiv$ **bool**

$W \equiv (\exists t:\text{legal element type})(\forall i:1\ \mathbf{to}\ n)(\forall j:1\ \mathbf{to}\ m_i)$

$(W\langle env\rangle[exp_{i,j}] \wedge T\langle env\rangle[exp_{i,j}]=\mathbf{set}(t))$

<u>Other expressions</u>

$[variable]$

$F \equiv \{variable\}$

$T \equiv t$ **where** "variable:t"$\in$env

$W \equiv$ variable$\in$env

Let match(f:type,a:type) be a compile time function as in section 3.8:

match(f,a) $\equiv$

f=a=**bool** $\Rightarrow$ "succeed"

f=a=**int** $\Rightarrow$ "succeed"

f=**set**(t)$\wedge$a=**set**(t') $\Rightarrow$ match(t,t')

f=?var $\Rightarrow$ "?var$\leftarrow$a"

else "fail".

[D(exp_1,....,exp_n)] where D is the name of a definition

def D(b_1:t_1,....,b_n:t_n) $\equiv$ exp

F $\equiv$ {ϵF[exp_i] **for** i:1 **to** n}

T $\equiv$ T<env'>[$exp_{?var\leftarrow t}$]

where ?var$\leftarrow$t is the substitution formed by combining the substitutions yielded by match(t_i,T<env>[exp_i]) for i:1 to n and <env'> is <b_1:t_1,....,b_n:t_n>$_{?var\leftarrow t}$ joined with type names taken from <env> for any type parameters in t (as explained in section 3.8).

W $\equiv$ ($\forall$i:1 **to** n)(W<env>[exp_i] $\wedge$ exp_i is a.e.

$\wedge$ match(t_i,T<env>[exp_i])$\neq$"fail")

$\wedge$ ($\forall$i,j:1 **to** n)

(match(t_i,T<env>[exp_i])="?var$\leftarrow$ta"

$\wedge$ match(t_j,T<env>[exp_j])="var$\leftarrow$tb"

$\Rightarrow$ ta=tb)

$[D(exp_1,\ldots,exp_n)]$ where D is the name of a definition

def $D(b_1{:}t_1,\ldots,b_n{:}t_n) \equiv$ **the** iter

where $\#N[iter]=1$

$F \equiv \{\epsilon F[exp_i]$ **for** i:1 **to** n$\}$

$T \equiv T\langle env'\rangle[iter_{?var\leftarrow t}](1)$

where env' and ?var←t are as above

$W \equiv$ same as W for above form of definition

Chapter 4

The Extraction of Programs from Proofs

4.1 Refinement logic metatheory

There are a number of important questions that may be asked of any system of formal reasoning. The study of these questions, the metatheory, certainly includes three fundamental concerns: independence, completeness, and consistency. This section discusses the meanings of these metatheoretic properties for refinement logics. The remainder of the chapter demonstrates the consistency of PRL.

A set of axioms and rules of inference is said to be *independent* if no element can be removed from the theory without decreasing the class of theorems, i.e., provable propositions. In this case, there is no redundancy in the foundation of the theory.

Independence is a useful property of a logic if one's main concern is the corresponding metatheory, because by holding the logic to essentials, the volume of metatheoretic analysis is minimized. If, however, one is interested in actual proofs in the theory then independence is not as important as having a useful, easily applied system. In PRL, where there is only a limited mechanism for introducing derived inference rules, there are redundant rules built into the logic (e.g. symmetry and transitivity), but they are sufficiently common and useful to justify their presence. As with programming languages, the clarity and elegance of user text is the guiding consideration; often this leads to a clear and elegant compiler (metatheory), but this is not the primary concern.

Completeness in a classical logic means that for any proposition A (with no free variables), either A or ¬A is provable. The intent is that every true proposition be provable within the system (truth may be defined by assigning an interpretation to all propositions of the language). Sufficiently powerful classical logics lack completeness, since for any machine M either "M halts" or "M diverges" is true, but by undecidability results, in general neither can be proven from any finite set of axioms. Thus, one is forced to consider relative completeness results of the form "if this fragment of the theory were complete then so would be the entire theory". Interesting results for programming logics can be obtained in this framework [Clarke 1979, Gorelick 1975, Owicki 1975]; in particular, with appropriate assumptions, Hoare's axiom system is complete for simple sequential programming languages [Cook 1975].

In a constructive system, such as PRL, there is no obvious model theory, i.e., no definition of truth other than provability, so the completeness question must take a different form. A general notion might be "whenever a formula is provable in any 'reasonable' axiom system is it also provable in the system under study?". If the answer is "yes" then the system might be considered "reasonably" complete. Hence, the idea of "reasonably provable" takes the place of truth. Of course, "reasonable" axiom systems are too vaguely specified for formal study. However, one might examine specific well-known logics in place of all "reasonable" ones. For example, one could establish that PRL is complete relative to Hoare's logic by showing that any precondition/postcondition pair that accurately describes an asserted program in Hoare's logic can be phrased as a transition specification and refined in PRL. That is, the specification language and refinement rules are powerful enough to develop any asserted program that is expressible in Hoare's logic. Though I have not performed such a proof, I see no significant barriers to doing so (of course, the logic may have to be extended if it turns out too weak, but I would expect only minor technical extensions).

There is one further notion of completeness that is not formalizable, yet is nonetheless important. This is whether the language of propositions is sufficiently powerful to characterize concisely all interesting program segments and computations. Clearly, this is not the case, as such notions as input/output, abstract data types, and CLU iterators (among many) are not supported. In addition, the rules for refining specifications could be extended in many ways. Much work remains to develop a complete system in this informal sense.

<u>Consistency</u> means that no formal contradictions can be derived in the logic. Classically, this is usually shown by building a model of the theory that provides an interpretation for each proposition and showing that every provable formula is true in the model. Given that the model assigns opposite truth values to A and ¬A (for all propositions A), the theory must not contain contradictions, for otherwise both a proposition and its negation would be true in the

model. The proof that all provable formulas are true proceeds by induction on the length of derivations: one shows that the axioms are true and the rules of inference preserve truth.

In PRL, many propositions claim the constructive existence of programs, so proving consistency means showing how to extract programs from proofs. Therefore, I demonstrate consistency in this chapter by presenting an extractor (compiler) that recursively analyzes any proof and produces an implementation for its goal. Where no implementation is needed, I simply show why the truth of a conclusion follows from the truth of its hypotheses. For example, in set theory there is no deep constructive meaning to $x \epsilon S$, but it is necessary to show that the the theory is consistent. The explicit description of an extractor proves an essential property of the logic and justifies the view of PRL as a language for expressing programs.

Once the notion of extraction is conceived, one must decide what sorts of objects will result from its application. As was explained in section 2.4, because PRL has a unified set of proof rules, justifications are needed to provide the constructive content of each proven formula. One can imagine logics for which this is not the case, logics in which formulas have no constructive content and hence need no justifications. Extractors for these logics would produce straightforward code for each program specification, combining these code segments according to the refinement rules used. Pure logical proofs would have no effect on the extraction process, only rules such as composition and iteration would build new code (and they would ignore proofs of purely logical subgoals). This type of extraction is precisely what happens when a conventional compiler reads an asserted program, compiling the code while ignoring assertions. The assertions may thus appear to the programmer as an unnecessary burden.

In contrast, when one constructs a proof in a logic whose boolean formulas have constructive meaning, the whole text of the proof enters into the extraction process. Important parts of the final program come from proofs of disjunctions and existentials. The extractor may be applied only to complete valid proofs, hence, the "purely logical" proof text takes on immediate significance, as it is needed not only for correct code, but for any code at all.

The method for extraction (presented below) developed slowly. In my early work, I believed that justifications could be compile time objects describing how to combine code fragments, while final programs would manipulate only those values and variables clearly used in proofs. For many proofs there were "obvious" programs that I desired the extractor produce. Though the desire was justified and the belief intuitively correct, many difficulties arose in my attempts to extract "obvious" programs directly.

Consider the $\vee$analysis rule where the goal is a boolean expression: $A \vee B$ pr C. Each subgoal produces a method of converting a justification of its hypothesis into a justification of the conclusion. The overall goal must do the same, but the justification of its conclusion depends on whether A or B is true, which may depend on the run time result of some test. Thus, the particular justification produced for the conclusion, though it always has the same general form, depends on events occurring during execution.

Upon realizing this difficulty, I still wanted justifications to be handled only at compile time. I tried to develop a compile time method for unifying the subgoal justifications of an $\vee$analysis into a single justification for its conclusion. This led to increasingly complex compile time data structures for representing justifications and similarly complex methods of extraction. When further difficulties developed with other rules, I decided to consider the run time manipulation of justifications, even though this seemed to imply inefficient extracted programs.

With this approach, the extraction process immediately became tractable, if subtle. I was able to write the extractor as a large recursive function. When I applied it to examples, it was clear that several well known code optimization techniques could so improve the extracted programs that the extract-optimize combination often produced precisely the sort of implementation that I had intuitively believed possible. Thus, I present below the extractor for run time manipulation of justifications and the optimization techniques that make this approach feasible.

4.2 The extraction of programs from proofs

Given a proof of a proposition, $B_1,\ldots,B_n$ **pr** C, the *extractor* X produces a procedure that maps justifications of the hypotheses into a justification of the conclusion. These procedures are expressed in a conventional PASCAL-like programming language, L, that is described below. The choice of a high level target language is to aid the reader; an actual implementation of PRL would likely produce assembly or machine code directly. Nonetheless, while L is intended to hide many messy details of compilation, it has been kept sufficiently simple that most interesting aspects of extraction are confined to X.

The extractor is a large recursive function from proofs to L-code. A clause of X exists for each refinement rule describing how to generate code for any proof that begins with that rule in terms of the code generated for each subgoal. Thus, one may generate code for a large proof tree by applying X at the leaves and successively working up the tree using the appropriate clauses of X at eacb stage. Almost no other information transfer or checking is done during extraction, but given a well formed proof, the resulting L program will be similarly well formed.

The general form of a clause of X is

$$X[\![\text{refinement rule}]\!] \equiv \text{rhs}$$

where refinement rule is a pattern representing a complete proof that begins with the refinement rule and rhs is either a procedure constant of L or a choice of procedure constants based on information obtained from the static analysis rules of section 3.9. The procedure constants of the rhs often contain instances of X applied to subgoals. These are to be textually replaced by their values, i.e., the procedure constants produced by X when applied to proofs of the respective subgoals. The rhs also contains names, expressions, iterators, etc., present in the refinement rule. These are to be textually replaced by those actually present in the instance of the refinement rule being processed. Thus, X may be viewed as a complicated sort of macro definition in which parameters are obtained from each refinement step by pattern matcbing.

At this point, the reader may wish to examine the next section for examples.

L is a conventional assignment based programming language. It includes an **if-then-else-fi** conditional statement, a case-of-esac selection statement, and a **do**-exit-**od** loop statement, all with standard semantics. Assignment is completely defined by Hoare's assignment axiom because there are no partial variables (such as array elements), nor any aliasing. All variables must be declared with a fixed type. Scopes are determined statically. L does include two unusual features requiring detailed discussion: procedure values and recursively defined data types.

The types of L include boolean, integer, character string, procedure, and those that may be defined with recursive data type definitions [Hoare 1975]. Booleans and integers have commonly available operations. Strings are treated flexibly, but all operations could be easily reduced to substring, concatenation, and so on. Procedure values are explained in detail shortly; they may be formed from the text of a procedure and an environment by the **bind** operator and may be applied to actual parameters. Data type definitions allow types to be defined as a discriminated union of records, for example,

LIST = NULL() + PAIR(HD:VALUE, TL:LIST)

VALUE = BOOL(B:**bool**) + INT(I:**int**) + SET(L:LIST)

Given a variable R of type LIST, it must be either NULL or a PAIR. If R is a PAIR then its second component could be selected by R.TL. A new value of type LIST could be formed as PAIR(INT(3),R), where INT produces a VALUE from the integer 3. Thus, NULL, PAIR, BOOL, INT, and SET are constructors, while HD, TL, B, I, and L are selectors. Equality testing and assignment are defined for all types, both primitive and defined.

A procedure constant is written

proc (X **from** Y): var decls; stmts **end**

where X is a list of typed parameter names that are called by reference and Y is a list of typed parameter names that are called by value. Procedure constants may read and modify global variables according to static scoping rules. A procedure constant, P, that does not modify global variables may be converted into an assignable value by writing **bind**(P). A bound procedure includes a copy of the environment and state active at binding, any references to read global variables are resolved in this fixed environment. Thus, a bound procedure is similar to a LISP FUNARG; it is a pure value whose meaning remains invariant as it is passed around the program.

The type of a procedure is written **proc**(types **from** types). Variables may be declared procedure valued. They may be assigned any bound procedure value whose parameter types equal corresponding parameter types of the procedure variable's type.

Given a procedure constant or bound procedure, P, the invocation of P may be written

P(vars **from** exps)

where vars is a list of distinct variable names accessible and modifiable in the current environment, while exps is a list of expressions. The types of vars and exps must equal the types of the procedure's corresponding formal parameters.

For any proved proposition, $S_1,\ldots,S_n$ pr C, the extractor will produce a procedure of type **proc**(JUST **from** $JUST^n$) where JUST is the type of a run time structure that represents justifications. This procedure produces a justification of C from justifications of $S_1,\ldots,S_n$. If C is a transition specification, x ← **any** $iter^a$, then the procedure will change the global variables named by x and yield a justification for $\nabla(iter^a)$, where a represents the new values of x. Thus, extracted procedures perform all the needed computation on the state: justifications passively describe why formulas are true (though they may contain bound procedures which can be invoked to perform various side-effect free computations).

The type JUST is defined recursively below.

```
JUST =  NIL()
     +  AND(J1:JUST, J2:JUST)
     +  OR(SEL:bool, J:JUST)
     +  IMPL(JT:proc(JUST from JUST))
     +  EQUIV(JT1:proc(JUST from JUST), JT2:proc(JUST from JUST))
     +  SOME(VALS:LIST, J1:JUST, J2:JUST)
     +  ALL(JT:proc(JUST from LIST,JUST))
     +  TRANS(J:JUST)
```

Each kind of formula has an associated form of JUST. A formula may be thought of as "type" information describing the structure of its justification, e.g., $A \wedge B$ describes an AND whose components are described by A and B, respectively.

A given true formula may be justified in a variety of ways. For instance, there may be many values satisfying an existentially quantified predicate. In order to understand the behavior of X, one needs a precise description of the relationship between objects of type JUST and the formulas they justify. This description is provided by J:formula x state → set of JUST, where the formula is constructively true in the state. I shall say that j justifies F in state s iff $j \in J[f](s)$.

A justification is a pure value; it makes no reference to external variables. Thus, once created, a justification may be passed around a program without its value changing. A formula, however, does refer to variables, so its meaning depends at any time on the state of the computation at that time. For example, the justification chosen for $x<0 \vee x\geq 0$ depends on the value of x, but the two choices, OR(**true**,NIL()) and OR(**false**,NIL()), do not themselves depend on x. Thus, J yields the set of justifications that certify a formula in a given state.

The definition of J is by cases, formalizing the presentation of section 3.4. The definition uses the notation s[d←x] to denote the variation of state s in which variables listed in d have values listed in x.

J[A∧B](s) = {AND(j1,j2) | j1∈J[A](s), j2∈J[B](s)}

J[A∨B](s) = {OR(b,j) | (b=**true**∧A is true in s∧j∈J[A](s))
∨ (b=**false**∧B is true in s∧j∈J[B](s))}

J[A⇒B](s) = {IMPL(P) | (∀j)(j∈J[A](s) ⇒ j'∈J[B](s)
where j' is the value of j'' yielded by
the invocation P(j'' **from** j))} .

since P is bound, no state need
be specified at its invocation

$J[A \Leftrightarrow B](s) = \{EQUIV(P1,P2) \mid IMPL(P1) \in J[A \Rightarrow B](s)$
$\wedge\ IMPL(P2) \in J[B \Rightarrow A](s)\}$

$J[\neg B](s) = \{NIL()\}$

$J[(\exists iter^d)B_d](s) = \{SOME(x,j1,j2) \mid \nabla(iter^d)$ is true in $s[d \leftarrow x]$
$\wedge\ B_d$ is true in $s[d \leftarrow x]$
$\wedge\ j1 \in J[\nabla(iter^d)](s[d \leftarrow x])$
$\wedge\ j2 \in J[B_d](s[d \leftarrow x])\}$

$J[(\forall iter^d)B_d](s) = \{ALL(P) \mid (\forall x$ of types corresponding to $d)(\forall j)$
$(j \in J[\nabla(iter^d)](s[d \leftarrow x]) \Rightarrow j' \in J[B_d](s[d \leftarrow x])$
where j' is the value of j'' yielded
by the invocation $P(j''$ **from** $j))\}$,
since P is bound, no state need
be specified at its invocation

$J[x \leftarrow$ **any** $iter^d](s) = \{TRANS(j) \mid (\exists v$ of types corresponding to $d)$
$(\nabla(iter^d)$ is true in $s[d \leftarrow v]$
$\wedge\ j \in J[\nabla(iter^d)](s[d \leftarrow v]))\}$

$J[$simple def ref$](s) = J[$instantiated body$](s)$

$J[$bool variable$](s) = \{NIL()\}$, but see discussion below
of extended justifications

$J[$other bool exp$](s) = \{NIL()\}$, varieties enumerated in section 3.4

Given the preceding description of which justifications certify which formulas, one can approach the problem of establishing the correctness of the extractor. I will not formally demonstrate X's correctness, but will sketch how it could be done and will note several lemmas of importance in such an undertaking. This discussion should aid the reader in understanding the extractor.

A proposition, S **pr** F, denotes a justification transformer. A justification transformer for S **pr** F maps a state and justifications for S into one of many possible final states and justifications for F. Let M (for meaning) map propositions to the justification transformers they denote. Thus,

M: proposition $\rightarrow$ (state x JUST*) $\rightarrow$ set of (state x JUST)

M may be defined by cases:

M[S1,...,Sn **pr** B](s,j1,...,jn)

= {<s,j> | j∈J[B](s)},

where B is a boolean expression and (∀i)(ji∈J[Si](s)).

M[S1,...,Sn **pr** x ← **any** $iter^d$](s,j1,...,jn)

= {<s',j> | (∃v of types corresponding to d)

($\nabla(iter^d)$ is true in s[d←v] ∧ s'=s[x←v]

∧ j∈J[$\nabla(iter^d)$](s[d←v]))} ,

where (∀i)(ji∈J[Si](s)).

The extractor, X, is correct iff

(∀proofs: S1,...,Sn **pr** F **by** ...)(∀s:state)(∀j1,...,jn:JUST)

(X[S1,...,Sn **pr** F **by** ...] is a procedure P(j' **from** j1,...,jn) s.t.

[(∀i)(Si true in s ∧ ji∈J[Si](s))

⇒ the invocation P(j' **from** j1,...,jn) in state s

yields a state s' and a justification j' such that

<s',j'>∈M[S1,...,Sn **pr** F](s,j1,...,jn)]).

One would show the correctness of the extractor by induction on the structure of proofs. Each clause of X's definition would require an argument that correct extraction from the subgoals resulted in correct extraction for the goal. There are two important properties needed in such an analysis. One concerns justifications; the other concerns justification transformers.

The first property, _free name independence_, states that any justification for a true formula B_x in state s also justifies $B_{x \leftarrow d}$ in state $s[d \leftarrow x]$, where d is a list of names not occurring in B_x. Intuitively, this holds because $B_{x \leftarrow d}$ has the same structure as B_x. Since boolean variable names are justified by any JUST (usually NIL()), since other variable names have no justification, and since justifications themselves do not refer to external variables, the justifications for B_x and $B_{x \leftarrow d}$ are identical. This property may be formally proved by induction on the structure of formulas (with much use of the definition of J).

The second property, _correct transformation of extended justifications_, states that an extracted procedure for S_x pr F_x should map a justification of $S_{x \leftarrow exp}$ to one for $F_{x \leftarrow exp}$. If x is not boolean then S_x and $S_{x \leftarrow exp}$ have identical justifications, similarly for F_x and $F_{x \leftarrow exp}$. Thus, in this case, the property clearly holds. If x is boolean then a justification for $B_{x \leftarrow exp}$ is called an _extended justification_ for B_x. Extracted procedures correctly transform extended justifications because any structure that justifies a boolean variable in S_x is manipulated as an indivisible unit (the "type" information of S_x prohibits any examination of deeper levels of the justification).

Further, the manipulations of justifications for boolean variables makes no assumption about the form of those justifications -- what goes in comes out. This property is required to support the validity of the substitutions used in the rules of specialization and $\forall$analysis, among others. It may be established during the proof of correctness of the extractor, by induction on the structure of proofs. Equivalently, it may be considered as one of the correctness criteria for X.

This ends my discussion of properties of X. The reader will likely find it helpful to refer back to this text while reading sections 4.3-4.6. In particular, the description of J is most useful. Before I present the defining clauses of X, however, there are three remaining topics to cover.

First, in order to reduce the volume of text present in the definition of X (sections 4.3-4.6), the names H and G and all their variations are implicitly assumed of type JUST, while the name L and all its variations are implicitly assumed of type LIST. H names represent justifications for hypotheses of propositions; G names represent justifications of conclusions. The symbol H* is an abbreviation for H1,...,Hn; it represents the justifications of hypotheses for S, as in S **pr** C.

Second, as discussed in section 3.4, the hypotheses of a proposition may be freely reordered and the hypothesis of interest in a refinement step need not occur last in the hypothesis list. The extractor must compensate for this freedom by reordering formal and actual parameters to correspond to the refinement text. I shall not mention this further in the extractor description, but it will arise in the examples at the end of the chapter.

Finally, the extractor's handling of automatically evaluatable expressions must be examined. Translating a.e. expressions into L is easily achieved with conventional compiling techniques. Integers and booleans are directly representable. Sets may be implemented using the LIST type with trivial $O(n^2)$ algorithms for the set operators. Though better representations and algorithms would undoubtedly be used in a PRL implementation, they are not of interest here. The details of translating a.e. expressions are similarly omitted; I assume a function V: a.e. expression → equivalent L expression.

Producing justifications for a.e. expressions, though no harder than evaluation, is of greater interest. Define J_{ae} : true a.e. boolean expression → L expression evaluating to a justification, by cases.

$J_{ae}[A \wedge B]$ = AND($J_{ae}[A]$,$J_{ae}[B]$)

$J_{ae}[A \vee B]$ = **if** V[A] **then** OR(**true**,$J_{ae}[A]$)

else OR(**false**,$J_{ae}[B]$)

$J_{ae}[A \Rightarrow B]$ = IMPL(**bind**(**proc**(G **from** H): G ← $J_{ae}[B]$ **end**))

since given H, A must be true, hence B must be true (and a.e.), so $J_{ae}[B]$ is the desired result

$J_{ae}[A \Leftrightarrow B]$ = EQUIV(**bind**(**proc**(G **from** H): G ← $J_{ae}[B]$ **end**),

bind(**proc**(G **from** H): G ← $J_{ae}[A]$ **end**))

reasoning as for ⇒, above

$J_{ae}[\neg B]$ = NIL()

$J_{ae}[(\exists iter^d)B_d]$ = SOME(d,$J_{ae}[\nabla(iter^d)]$,$J_{ae}[B_d]$)

where d=choose(V[$iter^d$])

This means to choose any tuple of values of $iter^d$ (assume V produces an appropriate representation), call this tuple d, and yield the SOME justification where the inside uses of J_{ae} are evaluated in a state witb d available. Since the ∃ is true, V[$iter^d$] cannot be empty.

$J_{ae}[(\forall iter^d)B_d]$ = ALL(**bind**(**proc**(G **from** L,H):

var d:VALUE;

d ← elements of L;

G ← $J_{ae}[B_d]$

end))

Given L and justification H for $\nabla(iter^d)$, since $(\forall iter^d)B_d$ is true it must be that $B_{d \leftarrow \text{elements of } L}$ is true (where "d ← elements of L" means to assign the i^{th} name of d the i^{th} value of L for all i), hence $J_{ae}[B_d]$ yields the desired result when evaluated in an environment where d exists with the proper values.

J_{ae}[simple def ref] = J_{ae}[instantiated body]

J_{ae}[other bool exp] = NIL()

4.3 Extraction: logical rules

Each rule of section 3.5 is reproduced below together with the associated result yielded by the extractor. Often the rhs of X is a single procedure constant, but for some rules a condition based on static analysis information is used to select between alternatives. Brief explanations follow the extraction rules, where needed.

true definition

X[S pr true] ≡

proc(G from H*): G ← NIL() end

Given justifications H* for S, i.e., $H_1, H_2, \ldots, H_n$, the justification for **true** is simply NIL().

false definition

X[S,false pr C] $\equiv$

proc(G **from** H*,H): G ← NIL() **end**

Since a justification, H, for **false** can never be produced, this procedure cannot be invoked. Nonetheless, this rule may be used in proofs of loop termination where no justifications need be produced.

excluded middle

X[S pr B∨¬B**]** $\equiv$

proc(G **from** H*): **if** V[B] **then** G ← OR(**true**,J_{ae}[B])
else G ← OR(**false**,NIL()) **fi end**

If B is true in the current state then yield an OR whose J component justifies B otherwise yield an OR whose J component justifies ¬B (i.e., is NIL()).

assumption

X[S,B pr B] $\equiv$

proc(G **from** H*,H): G ← H **end**

If B is a boolean variable with a non-trivial justification, this rule allows the justification to pass across the **pr** for subsequent synthesis into more complex conclusions. Passing the justification through without modification is essential to the validity of the claim of the last section that extracted procedures correctly handle extended justifications.

$\wedge$<u>analysis</u>

```
X[S.A₁∧A₂ pr C by
      S.A₁.A₂ pr C] ≡
proc(G from H*,H): X[S.A₁.A₂ pr C](G from H*,H.J1,H.J2) end
```

Given H justifying $A_1 \wedge A_2$, break it apart and produce G by applying the procedure constant of the subgoal to H* and tbe two component justifications.

$\wedge$<u>synthesis</u>

```
X[S pr B₁∧B₂ by
      S pr B₁
      S pr B₂] ≡
proc(G from H*): var G1,G2;
                 X[S pr B₁](G1 from H*);
                 X[S pr B₂](G2 from H*);
                 G ← AND(G1,G2)
                 end
```

Independently compute justifications for B_1, B_2 and yield tbeir AND combination.

$\vee$<u>analysis</u>

```
X[S.A₁∨A₂ pr C by
      S.A₁ pr C
      S.A₂ pr C] ≡
proc(G from H*,H): if H.SEL then X[S.A₁ pr C](G from H*,H.J)
                             else X[S.A₂ pr C](G from H*,H.J) fi end
```

Test the selector and produce G from the appropriate subgoal's justification transformer.

^{v}<u>synthesis</u>

```
X[S pr B₁∨B₂ by
      S pr B₁] ≡
proc(G from H*): var G';
                X[S pr B₁](G' from H*);
                G ← OR(true,G')
                end

also

X[S pr B₁∨B₂ by
      S pr B₂] ≡
proc(G from H*): var G';
                X[S pr B₂](G' from H*);
                G ← OR(false,G')
                end
```

⇒analysis

```
X[S,A1⇒A2 pr C by
      S pr A1
      S,A1,A2 pr C] ≡
proc(G from H*,H): var G1,G2;
                  X[S pr A1](G1 from H*);
                  H.JT(G2 from G1);
                  X[S,A1,A2 pr C](G from H*,G1,G2)
                  end
```

Obtain G1 justifying A_1, apply the bound procedure H.JT to G1 obtaining G2 justifying A_2, and use H*,G1,G2 to set G. H.JT cannot change the state so G1 justifies A_1 throughout.

⇒synthesis

```
X[S pr B1⇒B2 by
      S,B1 pr B2] ≡
proc(G from H*): G ← IMPL(bind(proc(G' from H):
                      X[S,B1 pr B2](G' from H*,H) end)) end
```

G.JT is a procedure with the values of global variables and of H* bound in. When it is given a justification for B_1 that is valid in the "frozen" state, it returns a justification for B_2 that is similarly valid.

¬<u>analysis</u>

```
X[S,¬B pr false by
      S pr B] ≡
proc(G from H*,H): G ← NIL() end
```

¬<u>synthesis</u>

```
X[S pr ¬B by
      S,B pr false] ≡
proc(G from H*): G ← NIL() end
```

⇔<u>analysis</u>

```
X[S,A1⇔A2 pr C by
      S,A1⇒A2,A2⇒A1 pr C] ≡
proc(G from H*,H): var G1,G2;
                   G1 ← IMPL(H.JT1);
                   G2 ← IMPL(H.JT2);
                   X[S,A1⇒A2,A2,⇒A1 pr C](G from H*,G1,G2)
                   end
```

$\Leftrightarrow$synthesis

X[S **pr** $B_1 \Leftrightarrow B_2$ **by**

S **pr** $B_1 \Rightarrow B_2$

S **pr** $B_2 \Rightarrow B_1$] $\equiv$

proc(G **from** H*): **var** G1,G2;

X[S **pr** $B_1 \Rightarrow B_2$](G1 **from** H*);

X[S **pr** $B_2 \Rightarrow B_1$](G2 **from** H*);

G $\leftarrow$ EQUIV(G1.JT,G2.JT)

end

$\forall$analysis

X[<env> S,$(\forall iter^x)B_x$ **pr** C **by**

<env> S **pr** $\nabla(iter^x)_{x \leftarrow exp}$

<env> S,$(\forall iter^x)B_x$,$B_{x \leftarrow exp}$ **pr** C] $\equiv$

proc(G **from** H*,H): **var** G1,G2;

X[S **pr** $\nabla(iter^x)_{x \leftarrow exp}$](G1 **from** H*);

H.JT(G2 **from** V[exp],G1);

X[S,$(\forall iter^x)B_x$,$B_{x \leftarrow exp}$ **pr** C](G **from** H*,H,G2)

end

Given a justification for the quantified formula and G1 justifying $\nabla(iter^x)_{x \leftarrow exp}$, a justification for $B_{x \leftarrow exp}$ can be obtained. A subtle point concerns the use here of H.JT. The expected input to H.JT is a justification of $\nabla(iter^x)$, but G1 may be an extended justification (section 4.2), for it also justifies exp (if exp is a boolean expression). Further, the desired result must justify $B_{x \leftarrow exp}$, not only B_x. However, because H.JT correctly transforms extended justifications (see ∀a), supplying a deeper justification to H.JT causes it to produce a deeper result, justifying $B_{x\leftarrow exp}$ in exactly the required manner.

<u>∀synthesis</u>

X[<env> S **pr** $(\forall iter^{x})B_{x}$ **by**

<env,d:T> S,$\nabla(iter^{x})_{x\leftarrow d}$ **pr** $B_{x\leftarrow d}$] ≡

proc(G **from** H*): G ← ALL(**bind**(**proc**(G' **from** L,H):

var d:VALUE;

d ← elements of L;

X[S,$\nabla(iter^{x})_{x\leftarrow d}$ **pr** $B_{x\leftarrow d}$](G' **from** H*,H)

end)) **end**

The values of L must be assigned to the names of d since they may be examined by the implementation of the subgoal. G' justifies $B_{x\leftarrow d}$ in an environment where d has the values of L, thus by the free name independence property discussed in section 4.2, G' justifies B_{x} in an environment and state where x has the values of L.

<u>∃analysis</u>

X[<env> S,$(\exists iter^{x})B_{x}$ **pr** C **by**

<env,d:T> S,$\nabla(iter^{x})_{x\leftarrow d}$,$B_{x\leftarrow d}$ **pr** C] ≡

proc(G **from** H*,H): **var** d:VALUE;

d ← elements of H.VALS;

X[S,$\nabla(iter^{x})_{x\leftarrow d}$,$B_{x\leftarrow d}$ **pr** C](G **from** H*,H.J1,H.J2)

end

H.J1 and H.J2 justify $\nabla(\text{iter}^x)$ and B_x when x has the values of H.VALS, so they also justify $\nabla(\text{iter}^x)_{x+d}$ and B_{x+d} when d has the values of H.VALS. Since C does not refer to d, G justifies C even when d is deallocated at procedure exit.

<u>∃synthesis</u>

```
X[<env> S pr (∃iter^x)B_x by
      <env,y:T> S pr y ← any iter^x st B_x] ≡
proc(G from H*): var G';
                 var y:VALUE;
                 X[S pr y ← any iter^x st B_x](G' from H*);
                 G ← SOME(y,G'.J.J1,G'.J.J2)
                 end
```

G' is an AND justification since it justifies $\nabla(\text{iter}^x$ **st** $B_x)$. The components justify $\nabla(\text{iter}^x)$ and B_x, respectively, in a state where x has the values of y.

<u>consequence</u>

X[<env> S **pr** C **by**

<env> S **pr** B_1

<env> S,B_1 **pr** B_2

.
.
.

<env> S,B_1,....,B_{n-1} **pr** B_n

<env> S,B_1,......,B_n **pr** C] ≡

proc(G **from** H*): **var** G1,G2,....,Gn;

X[S **pr** B_1](G1 **from** H*);

X[S,B_1 **pr** B_2](G2 **from** H*,G1);

.
.
.

X[S,B_1,....,B_{n-1} **pr** B_n](Gn **from** H*,G1,....,Gn-1);

X[S,B_1,....,B_n **pr** C](G **from** H*,G1,....,Gn)

end

<u>specialization</u>

X[<env> $S_{x\leftarrow exp}$ **pr** $C_{x\leftarrow exp}$ **by**

<env,x:T> S_x **pr** C_x] ≡

proc(G **from** H*): **var** x:VALUE;

x ← V[exp];

X[S_x **pr** C_x](G **from** H*)

end

Since X[S_x **pr** C_x] has the property of correctly transforming extended justifications, it produces the desired result when given H* justifying $S_{x \leftarrow exp}$ (as discussed in section 4.2).

reflexivity

```
X[S pr exp=exp] ≡
if T<env>[exp]=bool then
    proc(G from H*): G ← EQUIV(bind(proc(G' from H): G' ← H end),
                               bind(proc(G' from H): G' ← H end)) end
else
    proc(G from H*): G ← NIL() end
```

If the expression is boolean, say A, then build an EQUIV with two procedures for A⟺A. Otherwise, the expression requires only the trivial justification.

symmetry

```
X[S pr exp2=exp1 by
      S pr exp1=exp2] ≡
if T<env>[exp1]=bool then
    proc(G from H*): var G';
                    X[S pr exp1=exp2](G' from H*);
                    G ← EQUIV(G'.JT2,G'.JT1)
                    end
else
    proc(G from H*): G ← NIL() end
```

<u>transitivity</u>

```
X[S pr exp1=exp3 by
      S pr exp1=exp2
      S pr exp2=exp3] ≡
if T<env>[exp1]=bool then
    proc(G from H*):
            var G1,G2;
            X[S pr exp1=exp2](G1 from H*);
            X[S pr exp2=exp3](G2 from H*);
            G ← EQUIV(bind(proc(G' from H):
                                var G'';
                                G1.JT1(G'' from H);
                                G2.JT1(G' from G'')
                                end )),
                      bind(proc(G' from H):
                                var G'';
                                G2.JT2(G'' from H);
                                G1.JT2(G' from G'');
                                end )) )
            end
else
    proc(G from H*): G ← NIL() end
```

<u>Substitution</u>

The rule of substitution is, by far, the most difficult from whicb to extract. When an expression of a formula is replaced by another expression of completely different form, the justification of the original formula must be internally restructured to justify the new formula. This is accomplished by replacing justifications of all occurrences of the original expression by justifications of the new expression. A recursive L procedure named SUB is used to perform this replacement at run time. However, because the recursion is on the structure of formulas, every invocation of SUB could be expanded in-line at compile time, given appropriate concern for parameter passing and local names.

X[<env> $S_{x \leftarrow exp_1}$ **pr** $C_{x \leftarrow exp_1}$ **by**

<env> $S_{x \leftarrow exp_1}$ **pr** $exp_1 = exp_2$

<env> $S_{x \leftarrow exp_2}$ **pr** $C_{x \leftarrow exp_2}$] ≡

if T<env>[exp_1]=**bool then**

proc(G **from** H1,...,Hn):

var G1,...,Gn,G',G'';

X[$S_{x \leftarrow exp_1}$ **pr** $exp_1 = exp_2$](G' **from** H1,...,Hn);

for i:1 **to** n **do**

SUB(Gi **from** Si_x,Hi,G'.JT1,G'.JT2)

od;

X[$S_{x \leftarrow exp_2}$ **pr** $C_{x \leftarrow exp_2}$](G'' **from** G1,...,Gn);

SUB(G **from** C_x,G'',G'.JT2,G'.JT1)

end

else

proc(G **from** H*): X[$S_{x \leftarrow exp_2}$ **pr** $C_{x \leftarrow exp_2}$](G **from** H*) **end**

where SUB(G **from** F,J,JT1,JT2) takes a (boolean or transition) formula F with free occurrences of x, a justification J of $F_{x \leftarrow e1}$, JT1 mapping justifications of e1 to justifications of e2, JT2 performing the inverse mapping, and sets G to a justification of $F_{x \leftarrow e2}$ (for any e1,e2).

If exp_1 is not boolean then the subgoal performs the desired transformation. If exp_1 is boolean then $exp_1 = exp_2$ provides a pair of justification transformers which are used to turn eacb Hi into a justification Gi of $Si_{x \leftarrow exp_2}$ and G'' (justifying

$C_{x\leftarrow exp_2}$) into a justification of $C_{x\leftarrow exp_1}$. The procedure SUB, satisfying the above stated properties, is defined as follows.

```
SUB ≡ proc(G from F,J,JT1,JT2):
         var G1,G2;
         if F doesn't contain x free then
            G ← J
         else case F of
            X          →  JT1(C from J)
            A∧B        →  SUB(G1 from A,J.J1,JT1,JT2);
                          SUB(G2 from B,J.J2,JT1,JT2);
                          G ← AND(G1,G2)
            A∨B        →  if J.SEL then
                             SUB(G1 from A,J.J,JT1,JT2);
                          else
                             SUB(G1 from B,J.J,JT1,JT2);
                          fi
                          G ← OR(J.SEL,G1)
            A⇒B        →  G ← IMPL(bind(proc(G' from H):
                                   var G1,G2;
                                   SUB(G1 from A,H,JT2,JT1);
                                   J.JT(G2 from G1);
                                   SUB(G' from B,G2,JT1,JT2)
                                   end ))
            A⇔B        →  G ← EQUIV(bind(proc(G' from H):
                                   var G1,G2;
                                   SUB(G1 from A,H,JT2,JT1);
                                   J.JT1(G2 from G1);
```

```
                    SUB(G' from B,G2,JT1,JT2)
                    end )),
                  bind(proc(G' from H):
                    var G1,G2;
                    SUB(G1 from B,H,JT2,JT1);
                    J.JT2(G2 from G1);
                    SUB(G' from A,G2,JT1,JT2)
                    end ))
    (∃iter^a)B_a  →  SUB(G1 from ∇(iter^a),J.J1,JT1,JT2);
                     SUB(G2 from B_a,J.J2,JT1,JT2);
                     G ← SOME(J.VALS,G1,G2)
    (∀iter^a)B_a  →  G ← ALL(bind(proc(G' from L,H):
                         var G1,G2;
                         SUB(G1 from ∇(iter^a),H,JT2,JT1);
                         J.JT(G2 from L,G1);
                         SUB(G' from B_a,G2,JT1,JT2)
                         end ))
    a←any iter^b  →  SUB(G1 from ∇(iter^b),J.J,JT1,JT2);
                     G ← TRANS(G1)
    else          →  G ← NIL()
  end case
fi
end
```

4.4 Extraction: integers and sets

The general rule of arithmetic allows any valid reasoning about simple arithmetic relations. Since these relations have no constructive meaning, the extraction rule for arithmetic is trivial.

arith

$X[S,B_1,\ldots,B_n \textbf{ pr } C] \equiv$

proc(G **from** H*): G ← NIL() **end**

The other rules of arithmetic have marginally more interesting extractions.

trichotomy

X[S **pr** exp<0 v (exp=0 v exp>0)] ≡

proc(G **from** H*): G ← **if** V[exp]<0

then OR(**true**,NIL())

else OR(**false**,OR(V[exp]=0,NIL())) **end**

≤def

$X[S,exp_1 \le exp_2 \textbf{ pr } exp_1 < exp_2 \vee exp_1 = exp_2] \equiv$

proc(G **from** H*,H): G ← $OR(V[exp_1] < V[exp_2], NIL())$ **end**

≥def

$X[S,exp_1 \ge exp_2 \textbf{ pr } exp_1 = exp_2 \vee exp_1 > exp_2] \equiv$

proc(G **from** H*,H): G ← $OR(V[exp_1] = V[exp_2], NIL())$ **end**

Consider, now, a model for set theory in which sets are represented as non-repeating LISTs of VALUEs. The null set of any type is represented by the list NULL(). x∈S is tested by searching the representation of S for an element that equals x (where two sets are considered equal iff every element of one equals some element of the other, and vice versa). The set forming notation, {generator}, can be implemented by describing how each form of a.e. generator produces an object of type LIST. Finally, all set operations can be described in terms of operations on LIST representations. In this model, it is easy to extract from the set refinement rules.

```
∈def
    X[S pr x∈null set(T) ⇔ false] ≡
        proc(G from H*): G ← EQUIV(
                              bind(proc(G' from H): G' ← NIL() end),
                              bind(proc(G' from H): G' ← NIL() end))
                         end
```

Since x∈null set(T) cannot be true in the model, the justification produced for **false** will never be used. Similarly, no justification can ever be supplied for **false**, so that produced for x∈null set(T) will not be used.

```
∈def
    X[S pr x∈{gen} ⇔ ∇(x:gen)] ≡
        proc(G from H*): G ← EQUIV(bind(proc(G' from H):
                                        G' ← J_ae[∇(x:gen)] end),
                                   bind(proc(G' from H):
                                        G' ← NIL() end ))
                         end
```

Given a complete description of the LIST representation for {gen}, one can show that ∇ properly characterizes the elements in the representation. Thus, in the model, $x \in \{gen\}$ is equivalent to $\nabla(x:gen)$. Given H justifying $x \in \{gen\}$, one can conclude the truth of $\nabla(x:gen)$, hence $J_{ae}[\nabla(x:gen)]$ produces the desired justification. In the other direction H justifies $\nabla(x:gen)$, thus $x \in \{gen\}$ is true. Recall (from section 4.2) that its justification is simply NIL().

The remaining set axioms are clearly true in the suggested model. Further, since all set expressions are a.e., extractors for the remaining set axioms may be constructed as shown above, using J_{ae} to produce justifications as needed. I leave this construction to the reader.

4.5 Extraction: transition specifications

The procedures extracted from transition specifications modify their environment in addition to producing justifications. Recall that the justification produced for $x \leftarrow$ any $iter^a$ is TRANS(J) where J justifies $\nabla(iter^a)$ in an environment and state in which a is the new value of x and x is the old value.

as

X[<env> S **pr** $x_1,\dots,x_n$ ← **any** a_1:$exp_1,\dots,a_n$:exp_n] ≡
    proc(G **from** H*):
        G ← TRANS(AND(...AND($J_{ae}[a_1=exp_1]$,$J_{ae}[a_2=exp_2]$),...,
                $J_{ae}[a_n=exp_n]$));
        $x_1,\dots,x_n$ ← V[exp_1],...,V[exp_n]
        end

The procedure evaluates the n expressions and assigns them to global variables $x_1,\dots,x_n$. The justification produced is for $\nabla(a_1:exp_1,\dots,a_n:exp_n)$, that is, for $a_1=exp_1\wedge\dots\wedge a_n=exp_n$, which is associated left to right. For each non-boolean expression, the corresponding component of G is NIL(). For the boolean expressions, $a_i=exp_i$ is treated as $a_i\Leftrightarrow exp_i$ and an appropriate justification produced. G is computed before the new values of x are assigned, since the components of G may reference the original values of x.

choice

X[<env> S **pr** x ← **any** a:∈exp **by**
        <env> S **pr** exp≠**null set**(T)] ≡
    proc(G **from** H*):
        x ← V[exp].HD;
        G ← TRANS(NIL())
        end

Once the subgoal is established, exp is known non-empty, i.e., V[exp] is not NULL(). Thus, the HD component exists and may be assigned to x. The justification for ∇(a:ϵexp) is NIL() since ∇(a:ϵexp)=aϵexp.

null

```
X[<env> S pr x ← any iter^a by
      <env> S pr ∇(iter^a)_{a←x}] ≡
   proc(G from H*):
      var G';
      X[S pr ∇(iter^a)_{a←x}](G' from H*);
      G ← TRANS(G')
      end
```

Since G' justifies $\nabla(\text{iter}^a)_{a\leftarrow x}$, by free name independence, G' justifies $\nabla(\text{iter}^a)$ in any state where a has the same value as x. Therefore, since the new value of x equals the old value of x, G' justifies $\nabla(\text{iter}^a)$ in a state where a is the new value of x and x is the old value.

<u>simplification</u>

```
X[<env> S pr x ← any iter^a by
      <env>     S pr x ← any iterl^b
      <env,b:T> S.∇(iterl^b) pr ∇(iter^a)_{a←b}] ≡
    proc(G from H*):
        var G1,G2;
        var x',b:VALUE;
        x' ← x;
        X[S pr x ← any iterl^b](G1 from H*);
        b ← x;
        x ← x';
        X[S.∇(iterl^b) pr ∇(iter^a)_{a←b}](G2 from H*,G1.J);
        x ← b;
        G ← TRANS(G2)
        end
```

G1 justifies $\nabla(\text{iterl}^b)$ in a state where b is the new value of x and x is the old value, that is, in the state active when the second subgoal's procedure is invoked. G2 justifies $\nabla(\text{iter}^a)_{a\leftarrow b}$, or again by free name independence, $\nabla(\text{iter}^a)$ when a has the same value as b (which is the new value of x) and x has its current value (which is the old value of x). Thus, after restoring the new value to x, G is the desired result.

```
var
    X[<env> S pr x ← any iter^a by
                    var y:T
         <env,y:T> S pr x,y ← any iter^a,b:T] ≡
       proc(G from H*):
          var G';
          var y:VALUE;
          X[S pr x,y ← any iter^a,b:T](G' from H*);
          G ← G'.J.J1
          end
```

G' is a TRANS(AND(-,-)) whose J.J1 component justifies $\nabla(\text{iter}^a)$ in the appropriate state. The declaration of y is accessed according to conventional static scoping rules from the interior extracted procedure.

<u>composition</u>

```
X[<env> S pr x ← any iter^a by
      <env>      S pr Q1_x
      <env>      Q1_x pr x ← any iter1^a1
      <env,a1:T> S,∇(iter1^a1) pr Q2_{x←a1}
      <env>      Q2_x pr x ← any iter2^a2
                 .
                 .
                 .
      <env,a1:T,...,an-1:T> S,∇(iter1^a1),∇(iter2^a2)_{x←a1},
                 ...,∇(itern-1^{an-1})_{x←an-2} pr Qn_{x←an-1}
      <env>      Qn_x pr x ← any itern^an
      <env,a1:T,...,an:T> S,∇(iter1^a1),∇(iter2^a2)_{x←a1},
                 ...,∇(itern^an)_{x←an-1} pr ∇(iter^a)_{a←an}] ≡
   proc(G from H*):
      var G1,G1',...,Gn,Gn',G';
      var a0,a1,...,an:VALUE;
      a0 ← x;
      X[S pr Q1_x](G1 from H*);
      X[Q1_x pr x ← any iter1^a1](G1' from G1);
      a1 ← x;  x ← a0;
      X[S,∇(iter1^a1) pr Q2_{x←a1}](G2 from H*,G1'.J);
      x ← a1;
      X[Q2_x pr x ← any iter2^a2](G2' from G2);
      a2 ← x;  x ← a0;
          .
          .
          .
      an ← x;  x ← a0;
```

$X[S, \nabla(\text{iter1}^{a1}), \nabla(\text{iter2}^{a2})_{x \leftarrow a1},$

$\quad \ldots, \nabla(\text{itern}^{an})_{x \leftarrow an-1}$ **pr** $\nabla(\text{iter}^{a})_{a \leftarrow an}]$

$\quad\quad$ (G' **from** H*,G1'.J,G2'.J,...,Gn'.J);

$x \leftarrow an;$

$G \leftarrow TRANS(G')$

end

Eacb ai records the value of x after executing the extracted procedure for Qi_x **pr** $x \leftarrow$ **any** iteri^{ai}. The original (old) value of x is a0. The first subgoal produces G1 justifying $Q1_x$. The second subgoal produces G1' with G1'.J justifying $\nabla(\text{iter1}^{a1})$ in a state where a1 has the new value of x and x has its original value. The next two assignments create this state so that G2 justifies $Q2_{x \leftarrow a1}$. After the next assignment, by free name independence, G2 justifies $Q2_x$ and G2' is created with G2'.J justifying $\nabla(\text{iter2}^{a2})$ in the appropriate state. This process continues until G' is produced justifying $\nabla(\text{iter}^{a})_{a \leftarrow an}$. The final assignments set x and G to the desired results. By free name independence, G.J justifies $\nabla(\text{iter}^{a})$ when a equals an (the new value of x) and x has its original value. In programs extracted from real proofs, adequate information on the intermediate values of x is often supplied by the Gi justifications alone, so the intermediate ai variables are never accessed. In these cases, many of the variables and assignments introduced by this rule may be eliminated using the optimizations described in section 4.7.

<u>iteration</u>

```
X[<env> S pr x ← any iter^a by
            invariant x ← any iter^i as t_i↓ while B_i
    <env>          S pr x ← any iter^i
    <env,i:T>      S,∇(iter^i) pr B_i∨¬B_i
    <env,i:T>      S,∇(iter^i),B_i pr Q_{x←i}
    <env>          Q_x pr x ← any iter2^b
    <env,i:T,b:T>  S,∇(iter^i),B_i,∇(iter2^b)_{x←i} pr
                              ∇(iter^i)_{i←b} ∧ t_{i←b}<t_i
    <env,i:T>      S,∇(iter^i),t_i≤0 pr ¬B_i
    <env,i:T>      S,∇(iter^i),¬B_i pr ∇(iter^a)_{a←i}] ≡
  proc(G from H*):
    var Gi,Gi',Gb,Gtest,Gq,G';
    var xo,i,b:VALUE;
    xo ← x;
    X[S pr x ← any iter^i](Gi from H*);
    Gi ← Gi.J;
    i ← x;
    x ← x0;
    do /* invariant ∇(iter^i) is justified by Gi in tbe present
          state, i has the new value of x, and x bas its
          original value */
       X[S,∇(iter^i) pr B_i∨¬B_i](Gtest from H*,Gi);
       exit if ¬Gtest.SEL;
       X[S,∇(iter^i),B_i pr Q_{x←i}](Gq from H*,Gi,Gtest.J);
       x ← i;
       X[Q_x pr x ← any iter2^b](Gb from Gq);
```

```
        b ← x;
        x ← x0;
        X[S,∇(iter^i),B_i,∇(iter2^b)_{x←i} pr ∇(iter^i)_{i←b}∧t_{i←b}<t_i]
                        (Gi' from H*,Gi,Gtest.J,Gb.J);
        Gi ← Gi'.J1;
        i ← b
    od
    X[S,∇(iter^i),¬B_i pr ∇(iter^a)_{a←i}](G' from H*,Gi,Gtest.J);
    x ← i;
    G ← TRANS(G')
    end
```

The original value of x is kept in x0. After the initialization transition and subsequent three assignments, i has the current value of x and x has its original value; thus, Gi justifies $\nabla(\text{iter}^i)$ and the invariant holds initially. The body of the loop produces Gtest which tells whether B_i or $\neg B_i$ is true and the loop is exited if $\neg B_i$. Otherwise Gq justifying $Q_{x\leftarrow i}$ is created, x is restored to its "current" value, and with Gq now justifying Q_x (by free name independence) Gb is produced with J component justifying $\nabla(\text{iter2}^b)_{x\leftarrow i}$ in a state where b is the new value of x. This state is achieved by the next two assignments. Then Gi' is created and Gi is set to the part justifying $\nabla(\text{iter}^i)$ in a state where i has the value of b, i.e., the newest iteration result. The last assignment creates this state so the invariant holds again. When the loop is finished, $\neg B_i$ is justified by Gtest.J and i,x have the new,original values of x, respectively. G' justifies $\nabla(\text{iter}^a)_{a \leftarrow i}$, thus after x ← i, x has the desired final value and G.J justifies $\nabla(\text{iter}^a)$ in a state where a has the new value of x (i.e., i) and x has the old value. Note that the proof of termination is necessary to insure that the extracted loop terminates, but it does not enter into the extracted code.

4.6 Extraction: definitions and lemmas

Extraction for abstractions is heavily based on the ideas used in the rules of specialization and substitution. Since all actual parameters of abstraction invocations are a.e., their binding to formal parameters is similar to the binding that occurs in the specialization rule.

Consider an instance of the simple form of definitions:

$$\textbf{def } D(b:T) \equiv exp_b.$$

If exp is a.e. then any invocation, D(e), is also a.e.. In this case, V and J_{ae} must be extended to cover this form of a.e. expression. V[D(e)] is defined as $V[exp_{b,?var \leftarrow e,t}]$ and J_{ae}[D(e)] is defined as $J_{ae}[exp_{b,?var \leftarrow e,t}]$, where "?var←t" is the composite substitution resulting from parameter matching (described in section 3.8). If exp is not a.e. then no invocation of D will ever be directly evaluated, rather, proofs involving D(e) will reduce the calculation to a combination of a.e. calculations (just as with other non-a.e. expressions). However, justifications of D(e) may be manipulated whether D(e) is a.e. or not; as D(e) is considered an abbreviation of $exp_{b,?var \leftarrow e,t}$, the justification of D(e) is identical to that of

$exp_{b,?var\leftarrow e,t}$. Thus, the extraction rule for simple definition reference is trivial.

<u>definition</u>

$X[S_{x\leftarrow D(e)}$ **pr** $C_{x\leftarrow D(e)}$ **by**

$S_{x\leftarrow exp_{b,?var\leftarrow e,t}}$ **pr** $C_{x\leftarrow exp_{b,?var\leftarrow e,t}}$] ≡

proc(G **from** H*):

$X[S_{x\leftarrow exp_{b,?var\leftarrow e,t}}$ **pr** $C_{x\leftarrow exp_{b,?var\leftarrow e,t}}$](G **from** H*)

end

Next, consider an instance of the more complex form of definitions:

def D(b:T) ≡ **the** $iter_b^x$

<y:T',env'> **pr** y ← **any** $iter_b^x$

<x:T',y:T',env'> $\nabla(iter_b^x),\nabla(iter_b^x)_{x\leftarrow y}$ **pr** x=y

Since this definition provides a means of computing the result for all invocations, it is always a.e.. In addition to providing the value satisfying the iterator, the definition must yield a justification supporting that claim. This justification is provided by the first proof of the definition body. Thus, for each complex definition, construct an L function D_L, as follows:

```
fcn D_L(b:VALUE) returns (VAL:VALUE,J:JUST):
    var G;
    var y:VALUE;
    X[pr y ← any iter_b^x](G from);
    return (y,G)
    end.
```

The value function, V, may be extended to complex definitions: V[D(e)] means D_L(V[e]).VAL. The J component of D_L is used in the definition reference rules presented shortly.

An invocation of a complex definition is not an abbreviation for the substituted body. The justification for D(e) is simply NIL(). This difference between simple and complex definitions arises because all needed justifications are provided in full by the latter, while the former (not necessarily being a.e.) rely on justifications of D(e) provided by the user of the definition reference rule.

The rules of definition reference for complex definitions are shown below.

<u>definition evaluation</u>

$X[S \textbf{ pr } \nabla(iter_b^x)_{x,b,?var \leftarrow D(e),e,t}] \equiv$

proc(G **from** H*):

G ← D_L(V[e]).J

end

Since G justifies $\nabla(iter_b^x)_{b \leftarrow e}$, by free name independence (as used in specialization) and the observation that type names never affect the code extracted from a proof, G also justifies $\nabla(iter_b^x)_{x,b,?var \leftarrow D(e),e,t}$.

<u>definition uniqueness</u>

```
X[S_{y+D(e)} pr C_{y+D(e)} by
        S_{y+D(e)} pr ∇(iter_b^x)_{x,b,?var+exp,e,t}
        S_{y+exp} pr C_{y+exp}] ≡
    if T&lt;env&gt;[D(e)]=bool then
        proc(G from H1,....,Hn):
            var G1,....,Gn,Ga,Gb,Gc,Gd;
            var b,x,y:VALUE;
            X[S_{y+D(e)} pr ∇(iter_b^x)_{x,b,?var+exp,e,t}]
                    (Ga from H1,....,Hn);
            b ← V[e];
            x ← V[exp];
            y ← D_L(V[e]).VAL;
            G_b ← D_L(V[e]).J;
            X[∇(iter_b^x),∇(iter_b^x)_{x+y} pr x=y](Gc from Ga,Gb);
            for i:1 to n do
                SUBy(Gi from Si_y,Hi,Gc.JT2,Gc.JT1)
            od;
            X[S_{y+exp} pr C_{y+exp}](Gd from G1,....,Gn);
            SUBy(G from C_y,Gd,Gc.JT1,Gc.JT2)
            end
    else
        proc(G from H*):
            X[S_{y+exp} pr C_{y+exp}](G from H*)
            end
```

This extractor is very similar to that for substitution; there are two main differences. First, SUBy is a version of SUB that looks for occurrences of y rather than x. Second, producing Gc justifying exp=D(e) requires more work, as follows. The first subgoal is used to create Ga justifying $\nabla(\mathrm{iter}_b^x)_{x,b,?var\leftarrow exp,e,t}$. Then Gb is created justifying $\nabla(\mathrm{iter}_b^x)_{x,b,?var\leftarrow D(e),e,t}$. Finally, a built-in use of specialization produces Gc from the definition's uniqueness subproof (the second one), by creating a state where b is e, x is exp, and y is D(e).

Extraction for lemmas is a good bit simpler than for definitions. For each lemma

$$\textbf{lemma}\ L\ v{:}VT(b{:}BT) \equiv S_{v,b}\ \textbf{pr}\ C_{v,b}$$

a L procedure is created:

$$L_L \equiv \textbf{proc}(G, v{:}VALUE\ \textbf{from}\ H^*, b{:}VALUE){:}$$
$$X[S_{v,b}\ \textbf{pr}\ C_{v,b}](G\ \textbf{from}\ H^*)$$
$$\textbf{end}.$$

This is the only instance of a procedure in which the variables of v might be both referenced and assigned to by the body. The procedure is used, as in specialization, in the following extractor for lemma references.

lemma

$$X[S_{v,b,?var\leftarrow n,e,t} \text{ pr } C_{v,b,?var\leftarrow n,e,t} \text{ by } L\ n(e)] \equiv$$

```
proc(G from H*):
    L_L(G,n from H*,V[e])
    end
```

Since the invocation of L_L is call by reference on n and call by value on V[e], the lemma body (computing from inside L_L) can treat v, b as variables and fixed values, respectively, as it assumes it can. As mentioned earlier, the substitution ?var←t is of no consequence since type names have no effect on the code extracted from proofs.

4.7 Example extractions and optimization

Two example extractions are presented below together with the application of several code optimization techniques to the extracted procedures. (See [Aho and Ullman 1973] for an overview of code optimization techniques.) Unfortunately, the extreme tedium of extracting by hand (and reading extractions) has forced me to severly restrict the size and number of examples considered. Though most of the extraction rules are not demonstrated, this section is still quite long. As I have mentioned previously, PRL can only be feasible with machine support; it is not designed for paper reasoning. Extraction is an area where this is most apparent.

The need for optimization is motivated by the first example, a simple proof in pure logic.

Example 1: $(\exists x{:}\mathbf{int})P_x \Rightarrow Q$ **pr** $(\forall x{:}\mathbf{int})(P_x \Rightarrow Q)$

This is a refinement schema where P_x and Q are arbitrary boolean expressions. Each proposition of the refinement is labeled for reference from the extraction.

1) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q$ **pr** $(\forall x{:}\mathbf{int})(P_x \Rightarrow Q)$ **by** $\forall$a
2) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}$ **pr** $P_d \Rightarrow Q$ **by** $\Rightarrow$s
3) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d$ **pr** Q **by** $\Rightarrow$a reordered
4) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d$ **pr** $(\exists x{:}\mathbf{int})P_x$ **by** $\exists$s
5) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d$ **pr** $y \leftarrow$ **any** $x{:}$**int** **st** P_x **by** cons
6) . $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d$ **pr** $y \leftarrow$ **any** b:d by ae
7) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d.b{=}d$ **pr** $\mathbf{true} \wedge P_b$ **by** $\wedge$s
8) . $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d.b{=}d$ **pr** **true** **by** true def
9) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d.b{=}d$ **pr** P_b **by** subst
10) . $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d.b{=}d$ **pr** b=d **by** assum
11) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d.b{=}d$ **pr** P_d **by** assum reord
12) $(\exists x{:}\mathbf{int})P_x \Rightarrow Q.\mathbf{true}.P_d.(\exists x{:}\mathbf{int})P_x.Q$ **pr** Q **by** assum

The extracted procedure is displayed with each line annotated with the labels of refinements of which it is a part. This information is similar to the nesting level numbers provided by some compilers.

I shall not provide a step by step description of the extraction process, but the motivated reader will see that it is a purely mechanical application of the extraction rules.

```
1               proc(G from H1): G ← ALL(bind(proc(G' from L,H):
1                   var d:VALUE;
1                   d ← L.HD;
1 2                 proc(G from H1,H2): G ← IMPL(bind(proc(G' from H):
1 2 3                   proc(G from H,H1,H2):
1 2 3                       var G1,G2;
1 2 3 4                     proc(G from H1,H2,H3):
1 2 3 4                         var G';
1 2 3 4                         var y:VALUE;
1 2 3 4 5                       proc(G from H1,H2,H3):
1 2 3 4 5                           var G1,G2;
1 2 3 4 5                           var y',b:VALUE;
1 2 3 4 5                           y' ← y;
1 2 3 4 5 6                         proc(G from H1,H2,H3):
1 2 3 4 5 6                             G ← TRANS(NIL());
1 2 3 4 5 6                             y ← d
1 2 3 4 5 6                             end
1 2 3 4 5                                 (G1 from H1,H2,H3);
1 2 3 4 5                           b ← y;
1 2 3 4 5                           y ← y';
1 2 3 4 5 7                         proc(G from H1,H2,H3,H4):
1 2 3 4 5 7                             var G1,G2;
1 2 3 4 5 7 8                           proc(G from H1,H2,H3,H4):
1 2 3 4 5 7 8                               G ← NIL()
1 2 3 4 5 7 8                               end
1 2 3 4 5 7                                   (G1 from H1,H2,H3,H4);
1 2 3 4 5 7 9                           proc(G from H1,H2,H3,H4):
```

```
1 2 3 4 5 7 9 11                proc(G from H1,H2,H,H3):
1 2 3 4 5 7 9 11                    G ← H
1 2 3 4 5 7 9 11                    end
1 2 3 4 5 7 9                         (G from H1,H2,H3,H4)
1 2 3 4 5 7 9                   end
1 2 3 4 5 7                       (G2 from H1,H2,H3,H4);
1 2 3 4 5 7                 G ← AND(G1,G2)
1 2 3 4 5 7                 end
1 2 3 4 5                     (G2 from H1,H2,H3,G1.J);
1 2 3 4 5               y ← b;
1 2 3 4 5               G ← TRANS(G2)
1 2 3 4 5               end
1 2 3 4                   (G' from H1,H2,H3);
1 2 3 4             G ← SOME(y,G'.J.J1,G'.J.J2)
1 2 3 4             end
1 2 3                 (G1 from H,H1,H2);
1 2 3           H.JT(G2 from G1);
1 2 3 12        proc(G from H1,H2,H3,H4,H):
1 2 3 12            G ← H
1 2 3 12            end
1 2 3                 (G from H,H1,H2,G1,G2)
1 2 3           end
1 2               (G' from H1,H2,H)
1 2           end ))
1 2         end
1             (G' from H1,H)
1         end ))
```

```
l           end
```

Clearly, the above procedure is subject to massive improvement; the most obvious suggestion is to expand procedure invocations in-line with suitable renaming of variables to avoid conflicts. This optimization results in the following code.

```
proc(G from H1):
    G ← ALL(bind(
        proc(G' from L.H):
            var d:VALUE;
            d ← L.HD;
            G' ← IMPL(bind(
                proc(G'a from Ha):
                    var G1,G2;
                    var G'b;
                    var y:VALUE;
                    var G1a,G2a;
                    var y',b:VALUE;
                    y' ← y;
                    G1a ← TRANS(NIL());
                    y ← d;
                    b ← y;
                    y ← y';
                    var G1b,G2b;
                    G1b ← NIL();
                    G2b ← Ha;
                    G2a ← AND(G1b,G2b);
                    y ← b;
                    G'b ← TRANS(G2a);
                    G1 ← SOME(y,G'b.J.J1,G'b.J.J2);
                    H1.JT(G2 from G1);
                    G'a ← G2;
                    end ))
```

```
        end))
    end
```

From this it is apparent that eliminating chain assignments, propagating known expressions, removing dead variables, and removing dead code could substantially improve the extracted procedure. These optimizations yield the text below.

```
proc(G from H1):
    G ← ALL(bind(
        proc(G' from L,H):
            G' ← IMPL(bind(
                proc(G'a from Ha):
                    var G1;
                    G1 ← SOME(L.HD,NIL(),Ha);
                    H1.JT(G'a from G1)
                    end ))
            end ))
    end
```

This procedure nicely captures the semantics of the original formula, $(\exists x{:}\mathbf{int})P_x \Rightarrow Q$ **pr** $(\forall x{:}\mathbf{int})(P_x \Rightarrow Q)$, providing a justification of the conclusion in a clean, direct manner. This is the sort of "obvious" extracted code I spoke of in section 4.1. It seems that the code produced by the extractor is so simple and highly structured that these basic optimizations are extremely effective.

The second example is the proof of a specification for an absolute value program. It demonstrates the extraction of procedures that change global variables.

Example 2: **pr** x ← **any** a:abs(x)

```
 1)  pr x ← any a:abs(x)  by cons
 2)      pr x<0∨(x=0∨x>0)  by trichotomy
 3)      x<0∨(x=0∨x>0) pr x ← any a:abs(x)  by ∨a
 4)          x<0 pr x ← any a:abs(x)  by cons
 5)              x<0 pr x ← any b:-x  by ae
 6)              x<0,b=-x pr b=abs(x)  by arith
 7)          x=0∨x>0 pr x ← any a:abs(x) by null
 8)              x=0∨x>0 pr x=abs(x)  by ∨a
 9)                  x=0 pr x=abs(x)  by arith
10)                  x>0 pr x=abs(x)  by arith
```

This proof yields the following extracted code (again, by mechanical application of the extraction rules).

```
1          proc(G from):
1              var Gl;
1 2            proc(G from):
1 2                G ← if x<0 then OR(true,NIL())
1 2                            else OR(false,OR(x=0,NIL()))
1 2                end
1                    (Gl from);
1 3            proc(G from H):
1 3                if H.SEL then
1 3 4                  proc(G from Hl):
1 3 4                      var Gl,G2;
1 3 4                      var x',b:VALUE;
1 3 4                      x' ← x;
1 3 4 5                    proc(G from Hl):
1 3 4 5                        G ← TRANS(NIL());
1 3 4 5                        x ← -x
1 3 4 5                        end
1 3 4                            (Gl from Hl);
1 3 4                      b ← x;
1 3 4                      x ← x';
1 3 4 6                    proc(G from Hl,H2):
1 3 4 6                        G ← NIL()
1 3 4 6                        end
1 3 4                            (G2 from Hl,Gl.J);
1 3 4                      x ← b;
1 3 4                      G ← TRANS(G2)
1 3 4                      end
```

Eliminating chain assignments, propogating known expressions, removing dead variables, and removing dead code:

```
proc(G from):
    var G1;
    G1 ← if x<0 then OR(true,NIL())
                else OR(false,OR(x=0,NIL()));
    if G1.SEL then
        x ← -x;
        G ← TRANS(NIL())
    else
        var G';
        if G1.J.SEL then
            G' ← NIL()
        else
            G' ← NIL()
        fi;
        G ← TRANS(G')
    fi
    end
```

Analyzing the **if** statements with fundamental data flow analysis techniques, this can be reduced further:

```
proc(G from):
    if x<0 then
        x ← -x
    fi;
    G ← TRANS(NIL())
    end
```

Thus, optimizing the extracted procedure is again extremely worthwhile and yields the "obvious" implementation for the original specification.