Extracting Efficient Code From Constructive Proofs

James T. Sasaki
Ph.D. Thesis

TR 86-757
May 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

EXTRACTING EFFICIENT PROGRAMS FROM CONSTRUCTIVE PROOFS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

James Toshio Sasaki

June, 1986

EXTRACTING EFFICIENT PROGRAMS FROM CONSTRUCTIVE PROOFS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

James Toshio Sasaki

June, 1986

Extracting Efficient Programs From Constructive Proofs

James Toshio Sasaki, Ph.D.

Cornell University 1986

Extraction is a technique for producing verified programs. A proof of $\forall x{:}T\,.\,\exists y{:}T'\,.\,F$ corresponds to a function f of type $T \rightarrow T'$ that maps every x of type T to a y of type T' such that F is true. If the proof is constructive, then f is recursive. The semantics of extracted code involves the manipulation of ***justifications***, which are pieces of evidence for the truth of formulas. The raw extracted code for the formula above is actually a function γ that maps x to a pair (y, γ'), where γ' is a justification that provides evidence for the truth of F.

This thesis presents various ways to improve the efficiency of extracted programs. The first way uses traditional code optimizations. Though very helpful, they are no panacea. The second way involves small changes to its underlying semantics. Certain formulas, called ***singleton*** formulas, have no interesting justifications; if F is such a formula, no justification for it needs to be built, which simplifies the extracted code.

The third way to improve extracted code is to add call-by-reference parameters to it. As originally defined, extracted code passes arguments by value, which leads to inefficient code for mutable objects like arrays: passing an array by value requires making a copy of it.

Adding call-by-reference parameters entails adding a state to the semantics of extracted code, which in turn leads to various semantic and syntactic design issues, like aliasing and side-effects. To account for these changes, the constructive logic used to build proofs is modified. A proof of quicksort illustrates the functional, assignment-free, side-effect-free style of proof promoted by the new logic.

To relieve the user of some of the mental overhead involved in using the new logic, an ***array inferencing*** algorithm is presented. The algorithm allows users to get code that uses arrays from proofs that reason about and manipulate lists in restricted ways. In this way, users can view the use of arrays as an optimization.

Biographical Sketch

James Toshio Sasaki was born and raised in Uptown, a small neighborhood near Lake Michigan, on the north side of Chicago. He went to the Illinois Institute of Technology, where he received a B.A. with high honors in Computer Science in 1976. He joined a consulting firm, SEI Information Technology, where he worked in programming, analysis, and design, taught part-time, and did some technical writing. Although he rather enjoyed his job, he eventually decided to leave it to go to graduate school at Cornell, so he left Chicago and moved to Ithaca, New York. This was something of a shock: Chicago is a large rectilinearly-grid city built on flat ground, and Ithaca is a small town with some very crooked streets built on some very icy slopes. Nonetheless, he received an M.S. in Computer Science in 1984. He lives in hope that one day the Cubs will win the pennant.

Acknowledgements

First of all, I must thank my advisor, Joe Bates, since without his ideas and energy, the PRL project would never have existed. I appreciate the effort he went through to help me finish. Also, I want to thank Bob Constable. His encouragement and optimism were helpful and appreciated. In addition, I want to thank the other members of the PRL project, especially Mark Bromley and Todd Knoblock, who slogged through some early drafts of the thesis.

Sometimes Ithaca is green and sunny, and sometimes it is gray and rainy, but surely without friends it would often be big and lonely. My thanks to all of you. I'm particularly grateful to Chris Buckley and Ellen Voorhees and the rest of the Friday night crowd for many enjoyable bacon pizzas and bridge games. Ellen has earned special thanks: for many days she has listened to stories of justification and extraction, and heard about the never-ending battle with LaTeX. (Over lunch, yet.) Thank you, Ellen. My thanks also to Doug Moore and Janet Walz, who also heard more than they could ever want to about formatting constructive proofs with LaTeX. For introducing me to folk dancing, I owe a great debt to Vivian Sewelson. Through her I met many friends. My thanks go to all of them, with special thanks to Anna Doepker, Sarah Gowin, Carol Roberts, Anna Simola, and Marie Wendt.

A final note to Matthew Greising, who started all of this by getting me to join the computer club at Lane Tech High School, lo these many years ago: all is forgiven.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This cabinet is formed of gold
And pearl and crystal shining bright,
And within it opens into a world
And a little lovely moony night.
— *William Blake*

This thesis is about a technique called *extraction*. Extraction was defined by Joe Bates in his Ph.D. dissertation, *A Logic for Correct Program Development* [Bat]. It produces verified programs through manipulation of constructive proofs of program specifications. This thesis discusses the extraction process and gives techniques for improving the efficiency of the programs it produces.

1.1 Background

Writing programs is a difficult task. More precisely, writing *correct* programs is a difficult task: people are fallible, so programmers can only understand programs of limited complexity, and users often do not know what they want their programs to do. Even when users specify their needs precisely and formally (no mean feat in itself), the programming systems they receive almost always contain bugs.

One approach to the problem of writing correct programs is program verification. A typical example of this approach is that used by the Stanford Pascal Verifier [ILL]. Here, programmers annotate their programs with specifications and loop invariants. From these the program verifier generates *verification conditions*, formulas that have the property that the program is correct if and only if all the verification conditions are true. The verification conditions are processed by the *simplifier*, a theorem prover that transforms the verification conditions to simpler (though equivalent) conditions. Verification conditions that do not simplify to true are presented to the user for proof. This method has the disadvantage that the program and its proof of correctness are decoupled — the programming logic is encoded into the actions of the verification condition generator, so running the verifier yields a set of sometimes-mysterious formulas to be proved.

The PL/CV system [CoJ,CJE,COD] avoids this problem by making the programming logic explicit and by combining the program and its proof into one object. Users write their own proofs of correctness and annotate their programs with them. A proof checker inspects the program and its proof to verify that the proof is correct and fits the given program. Since proof checking is a separate step, PL/CV has the disadvantage that programmers can write programs without regard to correctness. So, as desirable as verified programs are, they are hard to obtain with program verifiers.

It is better, of course, to develop programs ***along with*** their proofs of correctness. The work of David Gries in *The Science of Programming* [Gri] is an example of this philosophy. Throughout his book, Gries develops correct programs by reasoning about their specifications. Again, the programs are annotated with predicates, but here the predicates are developed along with the programs, not afterwards. In this way, the proof of correctness is put on a more equal footing with the program.

In his dissertation Bates approaches the problem in a different way, one suggested by Constable [Con71], Bishop [Bis70], and Martin-Lōf [ML79]. Instead of writing a program and then proving it correct, he advocates taking a formal specification of some desired result and manipulating it in ways that automatically yield a program that meets the specification. For example, developing a program to sort an array of integers takes two steps.

The first step is to write the specification, which informally might be stated as

> *For every array of integers **a** there exists an array of integers **b**, where **b** is a sorted, permuted copy of **a**.*

More formally, it might be

$$\mathsf{All}\ a:\mathsf{array}\ .\ \mathsf{Some}\ b:\mathsf{array}\ .\ \mathit{Permuted}(a,b)\ \&\ \mathit{Sorted}(b).$$

The second step is to prove the specification (in this case, probably by induction on the length of a). From the proof, a program that sorts arrays is built automatically.

In such a system, a proof of correctness serves two purposes. It not only shows that the specification is true, it also is used to form the implementing program, the program that meets the specification. Since this approach uses proofs for two purposes, it is not surprising that these proofs require special properties. In his dissertation, Bates defined a logic he called PRL ("pearl"), short for Program Refinement Logic. He made the PRL logic ***constructive*** so that PRL proofs could be translated into the required programs: he omitted rules like the law of the excluded middle and proof by contradiction because these rules do not correspond to any program-building schemes.[1]

[1] More precisely, they do not correspond to any recursive program-building schemes.

The heart of PRL is the mapping of constructive proofs to programs. Bates called this mapping *extraction* and discussed many of its aspects. He gave a solid semantic basis for the extracted programs (they manipulate *justifications*, pieces of evidence for formulas), he defined extraction for his version of PRL, and he gave small examples of extracted code.[2]

These small examples made it clear that the size and running time of extracted code needed to be improved. Bates used code optimizations to improve the extracted code, but he was not able to study the problem in detail. The extraction of *efficient* code is the topic of this thesis. Three techniques are used to improve extracted code: code optimizations, changes to the notions of extraction and justification, and the introduction of call-by-reference parameters.

1.2 A comparison of the PRL systems

This section compares the different versions of PRL. Besides the version of PRL presented by Bates in his thesis (called θ PRL from here on), three other versions of PRL have been proposed: μ PRL ("micro-pearl"), λ PRL, and ν PRL. λ PRL is the version used in the thesis. This section is included primarily for historical reasons: knowledge of θ PRL, μ PRL, and ν PRL is

[2]Large examples were not available because Bates did not implement his PRL logic, knowing that such an implementation would require extensive programming.

not necessary for understanding the efficiency of λ PRL extracted code. Readers uninterested in this historical information are encouraged to skip to the next section, which contains a short overview of λ PRL.

θ PRL, the first PRL system, has already been discussed. μ PRL, the second PRL system [BaC81], is different from θ PRL in two ways: it has an applicative, state-free logic, and its range of types is much larger. It omits the assignment-statement-like "transition specifications" of θ PRL, which simplifies the system, but it has a much more involved type structure, related to the systems of Martin-Lőf [ML73,ML79,ML84]. The result is a richer and cleaner system.

μ PRL was to have been a prototype implementation of a system with a much simpler type structure than the ultimate PRL's: instead of having an infinite hierarchy of universes of types, μ PRL had only one full universe, U_1, and part of the next, U_2. The name "micro" was somewhat misleading, however: the work involved in designing the logic was still large enough to detract seriously from the other parts of the μ PRL project. In the end no implementation of μ PRL was ever done. Instead, a simpler system, λ PRL, was developed and implemented.

The logic used in λ PRL is applicative, like the logic used in μ PRL, but unlike μ PRL, λ PRL has a very simple type system: it contains only integers and lists of integers. Since λPRL has such simple types, its syntax is patterned after the predicate calculus, unlike the other PRL systems.

λPRL is implemented in Franz Lisp and ML under Unix [GMW, Prl]. It also runs on Symbolics Lisp machines with ZetaLisp. The implemented system is fairly large (about 35,000 lines) and includes a macro language processor, a small library handler, an ML-based tactics interface, and editors and parsers for proofs, programs, and commands. The optimizing extractor, which I wrote, is about a hundred pages long. It includes many but not all of the optimizations described in this thesis. My hope is that more of them will find their way into the implementation. Alternatively, when the next generation of PRL systems are designed, these optimizations should be built in.

νPRL is the latest version of PRL.[3] Work on νPRL began as soon as λPRL became stable. The underlying implementation of νPRL is the same as λPRL's, but νPRL has a completely different logic, more like μPRL's than λPRL's but even more powerful. Instead of one or two levels of universes of types, νPRL has an infinite hierarchy of universes of types. Each universe is an infinite collection of types, a collection built up from primitive types using type constructors like dependent product and sum.

[3] λPRL was named λPRL because it was obtained by trimming back μPRL. νPRL was named νPRL because ν comes after μ, because ν sounds something like "new," and because "NUPRL" could be an acronym for Nearly Ultimate PRL [CoB].

The types that are members of higher universes specify the types of operations on lower universes. For example, if f maps integers to members of U_1, then the type of f is $\mathbf{int} \rightarrow U_1$. The type $\mathbf{int} \rightarrow U_1$ is a member of U_2.

νPRL is currently being used to develop large mathematical theories, including parts of finite state automata theory, regular expression theory, and number theory. Especially interesting is the work of Howe [How]: using the work of Bishop [Bis67] as a guide, he proves large segments of constructive analysis.

Lack of space forces me to omit more discussion of νPRL. Fortunately, more information on PRL and related topics is available: *Implementing Mathematics with the Nuprl Proof Development System* [Con86] describes the νPRL logic, semantics, and programming system in detail. Harper has covered the development of the νPRL logic [Har]. Stansifer has studied a topic closely related to PRL, the representation of constructive theories in programming languages [Sta]. Bates and Constable have explored the relationship between proofs and programs [BaC85, Con82].

1.3 An overview of λPRL

The λPRL logic is organized as a refinement-style sequent calculus. In other words, the hypotheses available at each point in the proof are listed explicitly, and proofs are arranged in trees, top-down. For example, if F and G are formulas, the canonical proof of F & G from F and G is

$F, G \gg F \,\&\, G$ by intro
- $F, G \gg F$ by hyp 1
- $F, G \gg G$ by hyp 2.

Each of the three lines is a ***sequent*** or ***goal***. In the first line, "$\gg$" ("arrow" or "turnstile") is the ***sequent arrow***. It separates the hypotheses F and G from the conclusion F & G. The keyword intro specifies the proof rule being used (***and-introduction*** in this case). λPRL has about two dozen rules; ***and-introduction*** says that to prove F & G it is sufficient to prove F and G separately.

The second and third lines of the proof are indented with bullets to show that they are both subproofs of the first line.[4] They both use the ***hypothesis*** rule to prove their conclusions. Neither of these subproofs require subproofs themselves, so the proof as a whole has no doubly-indented lines.

λPRL also includes a programming language. Functions over integers and lists of integers can be explicitly defined with the help of built-in operations like addition, subtraction, and so on. Like the logic, the programming language used in λPRL is restricted: the only kinds of routines definable are

[4] The implemention of λPRL displays proofs sequent by sequent, so the actual format is slightly different. It does not use bullets to present the goals of the subproofs.

total functions that cannot have side effects and cannot use loops, general recursion, or higher-order functions (functions of functions, functions that yield functions, and so on). Primitive recursion *is* available, so functions like factorial are definable. The programming part of λPRL is connected to the logic through proof of correctness rules, so

$$\gg fact(0) = 1$$

and

$$n > 0 \gg fact(n) = n * fact(n-1)$$

both follow immediately from the definition of factorial.

The restrictions on explicitly defined functions are not great problems because there is another way to define functions: by extraction from constructive proofs. For example, let Th be a λPRL theorem (a proof that has been given the name Th), and let the goal of Th be

$$\gg \text{All } x : \text{int} \,.\, \text{Some } y : \text{int} \,.\, F(x, y)$$

where F is a formula. Once the complete proof of Th has been typed in, the user can request that a function be extracted from Th. The result is an integer to integer function f that maps any x to a y such that $F(x, y)$ holds. This property of f is known to the system: proofs that refer to f can assert that $F(t, f(t))$ holds for any integer term t.

Extraction, as defined for θ PRL, produces very large programs. (They are only as large as the proofs from which they are extracted, but the proofs themselves are very big.) Bates recognized this, and noted that code optimizations seemed to reduce the size of the programs significantly. Without an implementation of θ PRL, he was unable to get large examples of extracted code,[5] so he was unable to investigate questions dealing with the size and speed of extracted code.

Such questions are crucial: a system that produces verified code is useful, but not if the code it produces is too large or runs too slowly. As one of the original implementors of λ PRL, I was able to study the situation more closely and look for ways to improve the size and efficiency of extracted code.

1.4 Objectives

The time and space requirements of extracted programs should be properties of the underlying proof, not artifacts introduced by the extraction process. In this thesis, I claim that with suitable modifications and optimizations, extraction produces programs that are relatively efficient in this sense. In addition, the sizes of the programs can be reduced to reasonable levels.

[5] Extracting code from θ PRL theorems by hand is too tedious to be useful.

Any program specification that actually can be implemented has many different implementations. For example, the specification for sorting in section 1.1 can be implemented by bubble sort, quicksort, and so on. The basic differences between these implementations are a deep property of the proofs from which they are extracted. For example, a bubble sort proof (a proof of sorting that extracts to a bubble sort) looks very little like a quicksort proof. They both assert that any arbitrary segment of an array can be sorted, but the approaches they take to sort the segment are completely different.

The aim of this thesis is to modify extraction so that it does not produce programs that have significant extraction-induced overhead. For example, a typical bubble sort proof should extract to an $O(n^2)$ bubble sort program, and a typical quicksort proof should extract to an $O(n \log n)$ expected-time quicksort.

Three techniques are used to speed up extracted code in this thesis. The first, suggested by Bates [Bat], is the use of traditional code optimizations. While helpful, they are no panacea. Structural modifications to the underlying notions of extraction and justification are much more useful. This second technique reduces the size of the unoptimized code enormously at very little extra cost. The third technique introduces more radical changes to proofs and their semantics to allow for call-by-reference parameters. (The naive semantics for λPRL uses call-by-value parameters.) Using call-by-reference allows for more latitude in the allowable primitive data structures. In particular, it allows the extension of λPRL to include arrays that can be modified in place, and this is done in a sample proof of quicksort.

These techniques yield extracted code that is natural-looking and runs in reasonable time and space. In addition, as a side effect of the structural modifications and code optimizations, it becomes possible to extract code from incomplete proofs. This code has the property that it either executes correctly or halts with an error message. Extraction from incomplete proofs is not studied thoroughly here, but its possibility shows that future PRL systems should be able to deal with a continuum of verified code, from totally verified through various stages of partial verification.

1.5 Organization

This thesis has seven chapters. Chapter 2 contains a detailed explanation of the syntax and semantics of λPRL, including definitions and examples of of justification, extraction, sequents, and proofs. The chapter gives a proof of binary search as an example of a typical λPRL proof.

Chapter 3 presents improvements that come from code optimizations and simple structural changes to justifications and extraction. Though

dramatic improvements are obtained, these techniques are not the final answer to the problem of generating efficient extracted code.

Chapter 4 discusses the problems caused by adding call-by-reference parameters to λPRL. For concreteness, the problem of adding arrays to the logic is studied. Though it is necessary to add a state to the semantics, it is not necessary to add a state to the syntax. Instead, the status of array identifiers is tracked in the proofs using new fields added to the environments of the proofs. A proof of quicksort is given as an example of the kind of proof style supported by these changes.

Chapter 5 presents the array-handling version of λPRL formally: it redefines the notion of justification and gives the proof rules for a new version of the logic. The chapter also proves that the extraction process works correctly: extraction takes a proof and produces code that justifies the conclusion of the proof.

Chapter 6 gives a proof annotation algorithm that can be used to relieve the user of some of the burden of remembering the restrictions on array usage. The algorithm takes proofs entered by users and annotates them to determine which values of type list-of-integer can be implemented as array-of-integer. Users can have a list implemented as an array simply by using the list consistently in array-like operations (like subscripting).

The final chapter, chapter 7, contains conclusions and a discussion of future problems.

Some acquaintance with mathematical logic is necessary to read this thesis, but no great depth of knowledge is required: logic at the level of Enderton [End] or the first three chapters of Crossley et al. [Cros] is sufficient. Some knowledge of denotational semantics is also necessary: Gordon [Gor] is a good text. More advanced references for logic include Ershov and Palyutin [ErP], which is broad, concise, and surprisingly inexpensive, Bell and Machover [BeM], and Schoenfield [Sch]. For proof theoretic issues, the works by Kleene [Kle52, Kle67] are excellent. Advanced references for denotational semantics include de Bakker and Stoy [deB,Sto].

Chapter 2

Naive λPRL

Some asked how pearls did grow, and where?

— Robert Herrick

This chapter contains a description of naive (unoptimized) λPRL, with an emphasis on the notions of justification and extraction. No previous knowledge of λPRL [Prl], νPRL [Con86,CoB], or other PRL systems [Bat] is assumed.

Justification and extraction are the heart of λPRL. A *justification* for a formula is a piece of evidence for its truth. When justifications are used to give semantics to proofs, constructive proofs denote computable ways to manipulate justifications. (Nonconstructive proofs denote possibly-uncomputable ways to manipulate justifications, so they are not as interesting.) *Extraction* is the mapping of constructive proofs to executable code;

when run, extracted code manipulates justifications. Extraction preserves meaning and computability, so extracted code is guaranteed to run, and run correctly.

Section 1 below gives the syntax of λPRL terms, formulas, and explicitly defined procedures. Section 2 defines *justification* and shows its equivalence to validity. Section 3 describes the syntax and semantics of proof goals ("sequents") and shows how they are connected together by proof rules into proof trees. Section 4 contains detailed descriptions of the proof rules and their extracted code. Section 5 presents an example of a large λPRL proof, binary search.

2.1 Terms, formulas, and explicitly defined procedures

This section gives the syntax of terms, formulas, and explicitly defined procedures. (Implicitly defined, or "extracted," procedures are defined in section 2.4.)

Definition 2.1 Terms are built in the usual way, with integers, procedure applications like $f(x)$ and the usual operators, $+, -, *, /$. Lists of integers are also available: $(1,2,3)$ is a list of length three and $()$ is the empty list. Dot serves as infix cons: 1.2.3.() equals $(1,2,3)$. The built-in procedures hd

and tl (head and tail) serve as car and cdr: $hd((1,2,3))$ is 1 and $tl((1,2,3))$ is $(2,3)$. All procedures in λPRL are total: $n/0$ is 0, $hd(())$ is 0, and $tl(())$ is $()$.

There are three kinds of procedures in λPRL: built-in, implicitly defined, and explicitly defined. Implicitly defined ("extracted") procedures are defined in section 2.4. Explicitly defined functions must be total and may use only primitive recursion [MaY], so though many interesting functions are definable, functions like Ackermann's are not. In practice, this is not a problem because such functions can be defined implicitly. In this chapter, all parameters are passed by value. (Reference parameters are added in chapters 4 and 5, when modifiable arrays are added to the logic.)

Definition 2.2 The definition of explicitly defined procedures used here is a simplified version of the definition used by the actual λPRL system [Prl].[1] In addition, the syntax here has been modified for clarity.[2] A procedure f of type $\mathrm{int} \times T_1 \times \cdots T_n \rightarrow T'$ can be declared with

$$\mathrm{proc}\ (x : \mathrm{int}, x_1 : T_1, \ldots, x_n : T_n) : T' .$$
$$\mathrm{if}\ x = 0 \rightarrow t_1 \;\square\; x < 0 \rightarrow t_2 \;\square\; x > 0 \rightarrow t_3\ \mathrm{fi}.$$

[1] In the actual implementation of λPRL, it is possible to define blocks of mutually primitive recursive procedures. The definitions can also have multiple base cases.

[2] For example, the actual λPRL syntax for the factorial function definition below is $fact(x\mathrm{:int})\mathrm{:int} = x$ => 0; 0 => 0; x => $x * fact(x-1)$.

The term t_1 may not contain calls of f. In contrast, t_2 may contain calls of f, but only of the form $f(x+1, t'_1, \ldots, t'_n)$, and t_3 may contain calls of f, but only of the form $f(x-1, t'_1, \ldots, t'_n)$. The only identifiers that may appear in the t'_i are x or the x_j. More generally, no nonlocal identifiers can be used in the body of f. For example of an explicitly defined function, here is the factorial function, *fact*:

$$\text{proc}\ (x : \text{int}) : \text{int}\ .$$
$$\text{if } x = 0 \rightarrow 0 \ [\,]\ x < 0 \rightarrow 0 \ [\,]\ x > 0 \rightarrow x * fact(x-1) \text{ fi}.$$

Primitive recursion on lists of integers is also allowed: f can have the form

$$\text{proc}\ (x : \text{list}, x_1 : T_1, \ldots, x_n : T_n) : T'\ .$$
$$\text{if } x = () \rightarrow t_1 \ [\,]\ x \neq () \rightarrow t_2 \text{ fi}.$$

The term t_1 may not contain calls of f; t_2 may contain calls of f, but only of the form $f(tl(x), t'_1, \ldots, t'_n)$.

Definition 2.3 Formulas are defined as follows. If t and t' are terms, then $t = t'$, $t < t'$, and so on are *atomic* formulas. In addition, true and false are atomic formulas. If F and G are formulas, then $F \,\&\, G$ is a *conjunction*, $F \mid G$ is a *disjunction*, $F \Rightarrow G$ is an *implication*, and $\neg F$ is a *negation*. F and G are the *conjuncts* of $F \,\&\, G$ and the *disjuncts* of $F \mid G$. F is the

antecedent of $F \Rightarrow G$ and G is the *consequent* of $F \Rightarrow G$. A *literal* is an atomic formula or the negation of an atomic formula.

There are two kinds of *quantified* formulas: *existentials* and *universals*. The syntax for existentials is Some $x_1 : T_1, \ldots, x_n : T_n \,.\, F$ where $n \geq 1$. The x_i are identifiers and the T_j are types: int, list, and (starting with chapter 4), array.

The x_i above are *bound* or *quantified* in the formula. They are also said to be *bound by* or *quantified by* the *quantifier*, Some. For convenience, if adjacent identifiers have the same type, they can be combined. For example, Some x, y : int; x : list . F means Some x : int; y : int; x : list . F.

Universals have the same definition as existentials except that All is used as the quantifier instead of Some. The subformula that appears in a quantified formula is called the *body* of the quantified formula.

In increasing order of binding strength, the formula operators are All and Some (a tie), $\Rightarrow$, $\mid$, $\&$, and $\neg$. Thus, All $x : T \,.\, F_1 \mid F_2 \Rightarrow F_3 \,\&\, F_4$ means All $x : T \,.\, ((F_1 \mid F_2) \Rightarrow (F_3 \,\&\, F_4))$.

2.2 Justifications and validity

A *justification* for a formula is a certain kind of evidence for the truth of the formula. A formula is *justified* if there exists a justification for it.

The validity theorem below shows that a formula is classically true iff it has justifications, so the notions of being justified and being valid are equivalent. Justifications have a structure that is used λPRL to give an interesting computational semantics to proofs and proof goals. Readers acquainted with other writings on PRL [Bat, Con86, CoB, Prl] should be warned that the definition here of justification does not include constructions or operations. (Thus, the treatment here is more like Constable's approach in [Con85].) The differences between this approach and earlier explanations is discussed at the end of the section.

Notation. u, v, w, x, y, z are identifiers, t and t' are terms, F and G are formulas, γ is a justification, and J is the set of all justifications. J^2 is $\mathsf{J} \times \mathsf{J}$, J^3 is $\mathsf{J} \times \mathsf{J} \times \mathsf{J}$, and so on. The trivial justification (defined below) is *triv*. T is a syntactic type: int (integer) or list (list of integers). Domains, like the set of integers Z, are sans serif and begin with a capital letter; values of domains are lower case sans serif letters, like v and w. In this section and in the next, "function" means "not necessarily computable function." In section 2.4, constructive λPRL proofs will be brought in, and from that point, only computable functions will be discussed.

Definition 2.4 $\gamma \Vdash F, env$ means that γ justifies F in the environment env.[3] (An environment maps the free identifiers of formulas and terms to their values.) The definition of $\Vdash$ is given by induction on the structure of F.

- ***F is atomic.***

  Say F is of the form $t = t'$. $\gamma \Vdash t = t', env$ iff γ is the ***trivial*** justification, *triv*, and the values of t and t' are equal in the given environment. The trivial justification is an atomic, unstructured object and provides no evidence for the equality of t and t', but none is needed: in λPRL, the types of all terms are known, and all procedures and operations are total recursive, so the equality of t and t' is easy to verify.

  For F of the form $t < t'$, $t \leq t'$, and so on, justification is defined similarly: $triv \Vdash t < t', env$ iff t is less than t' in env; $triv \Vdash t \leq t', env$ iff t is less than or equal to t' in env; and so on. Finally, F might be true or false; true is justified by *triv*, and false has no justification.

- ***F is a conjunction.***

  $\langle \gamma_1, \gamma_2 \rangle \Vdash F_1 \,\&\, F_2, env$ iff $\gamma_1 \Vdash F_1, env$ and $\gamma_2 \Vdash F_2, env$.

[3] The symbol $\Vdash$ is used in set theory for forcing. Its use here is not meant to connect justification with forcing.

- ***F is a disjunction.***

  $(\tau,\gamma') \Vdash F_1 \mid F_2, env$ iff τ is 1 or 2 and $\gamma' \Vdash F_\tau, env$.

- ***F is an implication.***

  $\gamma \Vdash F_1 \Rightarrow F_2, env$ iff γ is a function from justifications of F_1 to justifications of F_2. That is, for all $\gamma' \Vdash F_1, env$, we have $\gamma(\gamma') \Vdash F_2, env$.

- ***F is a negation.***

  $\gamma \Vdash \neg G, env$ iff $\gamma \Vdash G \Rightarrow \mathsf{false}, env$.

- ***F is an existential.***

  $(\mathsf{v}_1,\ldots,\mathsf{v}_n,\gamma') \Vdash \mathsf{Some}\ x_1{:}T_1,\ldots,x_n{:}T_n\,.\,F, env$, where $\mathsf{v}_1,\ldots,\mathsf{v}_n$ are values of types $T_1,\ldots,T_n$ respectively,[4] iff $\gamma' \Vdash F, env'$, where env' is

  $$env\,[x_1,\ldots,x_n \leftarrow \mathsf{v}_1,\ldots,\mathsf{v}_n]$$

  which is the update of env with v_i for x_i, for each i.

- ***F is a universal.***

  $\gamma \Vdash \mathsf{All}\ x_1{:}T_1,\ldots,x_n{:}T_n.G, env$ iff γ is a function from $T_1\times\cdots\times T_n$ to justifications for G. That is, if $\mathsf{v}_1,\ldots,\mathsf{v}_n$ are values of types $T_1, \ldots, T_n$ respectively, then $\gamma(\mathsf{v}_1,\ldots,\mathsf{v}_n) \Vdash G, env'$, ($env'$ as above for existentials).

[4] That is, each v_i is a member of the domain denoted by T_i.

The validity theorem claims that justification and truth are equivalent notions in λPRL.[5]

Notation. $\models F, env$ means that formula F is classically true or *valid* in the given environment, in the standard model of integers and finite lists of integers.

Theorem 2.1 *(Justification validity)*
For all formulas F and environments env: $\models F, env$ iff there exists a justification γ such that $\gamma \Vdash F, env$.

Proof

By induction on the structure of F. □

The definition of justification will be modified in chapters 4 and 5, where techniques for improving λPRL programs are discussed. The validity theorem will be extended in those chapters in order to preserve the equivalence of justification and validity.

λPRL uses justifications to provide evidence for all kinds of formulas, but the most important justifications are those for formulas of the form $\mathsf{All}\ldots\mathsf{Some}\ldots F$; they are used to derive verified functions.

[5] Here, and from now on, "Justification" will sometimes mean "The abstract quality of being justified," in the same way that "Truth" can mean "The abstract quality of being true." Thus, the validity theorem claims that being justified and being true are equivalent notions for formulas.

Definition 2.5 An $\forall\exists$ (pronounced "ā-ē") formula is one of the form

$$\text{All } x_1 : T_1, \ldots, x_n : T_n \,.\, \text{Some } y_1 : T'_1, \ldots, y_m : T'_m \,.\, F.$$

F is the *matrix* of the $\forall\exists$ formula [BeM,ErP,Sch].

Definition 2.6 Given an environment *env* and an existential formula

$$F = \text{Some } x_1 : T_1, \ldots, x_n : T_n \,.\, G$$

witnesses for F in *env* are values of the right types for the x_i such that the body G becomes true. That is, if $v_1, \ldots, v_n$ are values of types $T_1, \ldots, T_n$ respectively, then the v_i are witnesses for F iff

$$\models G, env\,[x_1, \ldots, x_n \leftarrow v_1, \ldots, v_n]\,.$$

Any justification γ for F must have the form $\langle v_1, \ldots, v_n, \gamma' \rangle$. *The witnesses of* γ are the v_i.

Consider an $\forall\exists$ formula All $x : T$. Some $y : T'$. G, with justification γ in environment *env*. By the definition of justification, γ must be a function of type $T \rightarrow (T' \times \mathsf{J})$, where J is the set of all justifications. (By definition, γ maps values of type T to justifications for Some $y : T'$. G, which have the type $T' \times \mathsf{J}$.) If v is a value of type T then $\gamma(v)$ must be a pair (w, γ') where w is a value of type T' and $\gamma' \Vdash G, env\,[x \leftarrow v]\,[y \leftarrow w]$.

Define $f : T \rightarrow T'$ by $f(v) = w$ where $\gamma(v) = (w, \gamma')$. Let $G\,[y/f(x)]$ be G with all free occurrences of the identifier y replaced by the term $f(x)$, renaming bound identifiers if necessary. By the validity theorem, $\models G, env\,[x \leftarrow v]\,[y \leftarrow w]$, so $\models G\,[y/f(x)], env\,[x \leftarrow v]$. Thus f is a function that maps x's to y's such that G is true,[6] and the validity theorem verifies that f has this behavior.

$\forall\exists$ formulas are important in λPRL because they are used as specifications for verified programs: All $x : T$ specifies the input, Some $y : T'$ specifies the output, and the matrix G specifies the relationship between them.

For a larger, more concrete example of a specification, let us look at binary search. The type array will be needed, along with the length function on arrays, and array subscripting. (All these and in-place array updating will be added to λPRL in chapters 4 and 5, so this anticipates the development a bit.) The arrays are zero-origin arrays of integers.

To improve readability, some shorthand is used in the formulas. (For example, $a[i-1] < x \le a[i]$ means $a[i-1] < x \;\&\; x \le a[i]$.) To simplify the specification, a total recursive integer function *searchable* is used; *searchable* takes an array a and returns 1 if the array is sorted, nonempty, and without duplicates. It returns 0 if not. $|a|$ is the length of a, so the elements of a are $a[0], a[1], \ldots, a[|a|-1]$.

If array a is sorted and of length n, and if x is a value within the bounding values of the array, then the search finds an i, $0 < i < n$, such

[6] Such functions are called *skolem* functions [End].

that $a[i-1] < x \le a[i]$. More formally, a specification for binary search is

$$\begin{aligned} &\textsf{All } a:\textsf{array}, x:\textsf{int}\,.\,\textsf{Some } i:\textsf{int}\,.\\ &\quad searchable(a) = 1 \,\&\, a[0] < x \le a[|a|-1] \\ &\qquad \Rightarrow a[i-1] < x \le a[i] \,\&\, 0 < i < |a|. \end{aligned} \tag{2.1}$$

(Note that $a[0]$ is used as a sentinel value.)

A justification for (2.1) is a function $\gamma : \mathsf{A} \times \mathsf{Z} \to \mathsf{Z} \times (\mathsf{J}^3 \to \mathsf{J}^4)$ where A is the set of arrays of integers.[7] The $\mathsf{J}^3 \to \mathsf{J}^4$ component corresponds to the matrix of (2.1): the matrix is an implication whose antecedent is a conjunction of three formulas (hence the J^3) and whose consequent is a conjunction of four formulas (hence the J^4). Note that it is also correct (but less informative) to say that γ is of type $\mathsf{A} \times \mathsf{Z} \to \mathsf{Z} \times \mathsf{J}$.

Say γ justifies (2.1) in $\emptyset$, the empty environment. (Since (2.1) has no free identifiers, the environment is irrelevant.) If a and x are values of type array and integer respectively, then $\gamma(\mathsf{a},\mathsf{x})$ is some pair (i, γ'), where i is an integer and γ' is a justification (more precisely, an implication-justifying function of type $\mathsf{J}^3 \to \mathsf{J}^4$). The function $binsearch : \mathsf{A} \times \mathsf{Z} \to \mathsf{Z}$ defined by $\gamma(\mathsf{a},\mathsf{x}) = (binsearch(\mathsf{a},\mathsf{x}), \gamma')$ is guaranteed to meet the specification (2.1).

Let P be a program, in some language, that denotes the function γ. Note that (2.1) does not force P to be a ***binary*** search: the specification of the speed of P is not expressible in λPRL. In section 2.5, a particular P is

[7] Recall that J^3 is $\mathsf{J} \times \mathsf{J} \times \mathsf{J}$ and J^4 is $\mathsf{J} \times \mathsf{J} \times \mathsf{J} \times \mathsf{J}$.

obtained, through careful work, that *is* a binary search. In PRL, programs are obtained by manipulating proofs, and the form of a proof determines the speed of the corresponding program, so users can develop programs of different speeds by writing different proofs. The binary search program is obtained by using a proof that halves the length of the search interval at each step. A linear search program would be obtained by using a proof that simply shortens the interval one element at a time.

The rest of this section discusses the relationships between justification for λPRL (as explained here), justification for other PRL systems, and Kleene's notion of realizability. Readers unfamiliar with one or more of these should skip to the next section.

Justification is presented here as a purely semantic or model-theoretic notion that does not include constructions or operations. In contrast, other explanations of PRL [Con86,Bat,CoB] have included the notion of construction, so that (for example) an all justification is thought of as a construction that takes members of the quantified type to justifications for the body. Constructions were not used here partly for technical reasons (it is not necessary to use constructions to understand justification in λPRL), partly to simplify the presentation (constructive notions can be delayed until the proof system is explained), but mostly for pedagogical reasons: justifications are modified in later chapters to include call-by-reference parameters, and these modified justifications are hard enough to explain without the extra complication of constructions.

Constructions are also used in Kleene realizability [Kle45,Kle52]. Realizability is essentially the same as justification in ν PRL [Con86,CoB] and θ PRL [Bat]. All these systems are frameworks for understanding intuitionistic truth (not classical truth). Their differences lie mainly in how they encode operations: Kleene uses Gödel numbers, and ν PRL and θ PRL use programs over a language.

The difference between λ PRL as explained here and these other systems is that here justifications are not necessarily computable whereas Kleene realizers and ν PRL and θ PRL justifications are. For example, in [Kle52], Kleene mentions that the formula $\neg\neg F \Rightarrow F$ is not apparently realizable for arbitrary F (because it is not clear how to construct a realizer for it). In contrast, since $\neg\neg F \Rightarrow F$ is classically valid, it does have a (possibly uncomputable) justification in λ PRL. Though the semantics presented here for λ PRL allows for uncomputable justifications, only computable justifications are actually generated. This is made clear in section 2.4, when constructive λ PRL proofs are brought in and extraction is explained.

2.3 Proof goals, proofs, and their semantics

λ PRL proofs are proofs of ***goals***. Goals, also known as ***sequents***, have the form

$$[x_1 : T_1, \ldots, x_m : T_m]\ H_1, \ldots, H_n \gg F$$

where F and the H_i are formulas, the x_i are identifiers, and the T_i are types. The double > symbol, "$\gg$", is the ***sequent arrow*** and is pronounced "arrow" or "prove" or sometimes "turnstile." The H_i are the ***hypotheses***, and F is the ***conclusion***. The H_i are the only hypotheses available to help prove F; if the list $H_1, \ldots, H_n$ is empty, F must be proved without hypotheses. The list of bindings $x_1 : T_1, \ldots, x_m : T_m$ is the ***proof environment***. It specifies the types of the free identifiers of the conclusion and hypotheses. For example, for the goal $\gg x = y - 1$, the environment should include the bindings $x : \mathsf{int}, y : \mathsf{int}$. All free identifiers of the hypotheses and conclusion must be declared in the environment, so if the environment is empty, the hypotheses and conclusion may have no free identifiers.

Notation. H is a hypothesis; $\mathcal{H}$ is a list of hypotheses $H_1, \ldots, H_n$. $\mathcal{E}$ is a proof environment, so goals are generally indicated by $[\mathcal{E}]\ \mathcal{H} \gg F$.

Definition 2.7 $\models [\mathcal{E}]\ \mathcal{H} \gg F, env$ means that the goal $[\mathcal{E}]\ \mathcal{H} \gg F$ is (***classically***) ***true*** or ***valid*** in env. The goal is valid iff the truth in env of the members of $\mathcal{H}$ implies the truth in env of the conclusion, F.

By the validity theorem, a formula is true iff it has justifications, so if $[\mathcal{E}]\ \mathcal{H} \gg F$ is true in *env*, then if all the members of $\mathcal{H}$ have justifications, then so does F. Therefore, for any true goal $[\mathcal{E}]\ \mathcal{H} \gg F$, there exists a ***justification transformer*** Γ, a function of type $\mathsf{J}^n \to \mathsf{J}$ (recall that $\mathcal{H}$ has n members), and this transformer maps justifications for the members of $\mathcal{H}$ to a justification for F.

Definition 2.8 $\Gamma \Vdash [\mathcal{E}]\ \mathcal{H} \gg F, env$ means that Γ is a ***justification transformer*** that ***justifies*** or ***validates*** $[\mathcal{E}]\ \mathcal{H} \gg F$ in *env*. That is, Γ maps justifications for the members of $\mathcal{H}$ to a justification for F. From the discussion above, it should be clear that $\models [\mathcal{E}]\ \mathcal{H} \gg F, env$ iff there exists a Γ such that $\Gamma \Vdash [\mathcal{E}]\ \mathcal{H} \gg F, env$.

In λPRL, proof goals are connected by ***proof rules*** into refinement style ***proof trees***. A λPRL proof consists of a goal (the ***main goal***), a proof rule, and zero or more ***subproofs***. Each subproof is a proof in its own right, with a goal, rule, and so on, so proof trees have a recursive structure. The ***subgoals*** of the main proof are the goals of the subproofs.

A historical note: the refinement style of logic used in λPRL was designed by Bates for his thesis [Bat]. The style used is a top-down variant of Gentzen's ***sequent calculus*** [ErP,Gen,Kle52]. Bates actually developed the logic from Gentzen's natural deduction system [Pra65], and this deriviation motivated the names of the rules given below.

In the implementation of λPRL, proofs are built interactively through refinement: a user enters the main goal and then adds in proof rules. Each application of a proof rule to a goal generates some number of subgoals for the goal. Some rules generate no subgoals at all. When this happens, the goal is proved. If subgoals are actually generated, then the process of adding proof rules is repeated on the generated subgoals (and their descendents) until all the leaves of the proof tree have been proved. A proof tree with unproved leaves is ***incomplete*** or ***partial***; one with no unproved leaves is ***complete***.

PRL proofs were designed to be explored interactively [Prl, Con86], and they are nearly impossible to write down on paper. Two formats are used in this thesis to present proofs: long proofs are telescoped into tables, and short proofs are linearized. (The table format is explained later in the chapter, where it is used in a proof of binary search.) Short proofs are linearized by listing their nodes in preorder. Bullets are used to indicate the depth of nodes in the tree. For example, consider the following incomplete proof:

$\mathcal{H} \gg (F_1 \mathbin{\&} F_2) \mathbin{\&} (F_3 \mathbin{\&} F_4)$ by intro

• $\mathcal{H} \gg F_1 \mathbin{\&} F_2$ by intro

$$\begin{array}{ll}
\bullet\bullet\ \mathcal{H} \gg F_1 & \\
\bullet\bullet\ \mathcal{H} \gg F_2 & \qquad (2.2)\\
\bullet\ \mathcal{H} \gg F_3 \,\&\, F_4 \quad \text{by intro} & \\
\bullet\bullet\ \mathcal{H} \gg F_3 & \\
\bullet\bullet\ \mathcal{H} \gg F_4. &
\end{array}$$

The main goal of the proof is $\mathcal{H} \gg (F_1 \,\&\, F_2) \,\&\, (F_3 \,\&\, F_4)$. Using the *and-introduction* rule (*and-intro*, for short) on the main goal generates two subgoals, $\mathcal{H} \gg F_1 \,\&\, F_2$ and $\mathcal{H} \gg F_3 \,\&\, F_4$. (Introduction rules break down conclusions.) The *and-intro* rule is used on the two subgoals to produce the four (sub)subgoals, $\mathcal{H} \gg F_i$, $(i = 1, \ldots, 4)$. These four goals are unproved, since they are missing proof rules, so the proof as a whole is incomplete.

The λPRL logic includes many other rules besides the *and-intro* rule used above. Syntactically, a proof rule transforms a goal into a (possibly empty) list of subgoals. All the rules have keywords (like intro and elim) that specify the kind of transformation. Some rules also require parameters: terms, identifiers, or hypothesis numbers (1 for the first hypothesis, 2 for the second, and so on). Details are given in the next section, which contains descriptions of the various proof rules.

Semantically, each proof rule is a second-order justification transformer (a justification-transformer transformer), since a proof rule takes justification transformers for the subgoals and yields a justification transformer for

the (main) goal. For example, in the proof (2.2) above, the *and-intro* rule had the form

$$\begin{array}{ll}
[\mathcal{E}]\ \mathcal{H} \gg F_1 \,\&\, F_2 \quad \text{by intro} & \\
\bullet\ [\mathcal{E}]\ \mathcal{H} \gg F_1 & \qquad (2.3)\\
\bullet\ [\mathcal{E}]\ \mathcal{H} \gg F_2. &
\end{array}$$

Here the main goal and the subgoals are all validated by transformers of type $J^n \rightarrow J$. The rule as a whole corresponds to a second-order transformation of type $(J^n \rightarrow J)^2 \rightarrow (J^n \rightarrow J)$. As will be seen in the next section, the rule corresponds to a *computable* second-order transformer. (It takes a $\Gamma_1 \Vdash [\mathcal{E}]\ \mathcal{H} \gg F_1, env$ and a $\Gamma_2 \Vdash [\mathcal{E}]\ \mathcal{H} \gg F_2, env$ and *computably* yields a $\Gamma \Vdash [\mathcal{E}]\ \mathcal{H} \gg F_1 \,\&\, F_2, env$.) All of the λPRL proof rules are computable, and this is vitally important in the next section when extraction is defined.

By itself, an instance of the *and-intro* rule is an incomplete proof, and like the *and-intro* rule, incomplete proofs are second-order transformers: they take transformers for the incomplete leaves of the proof and yield a transformer for the main goal of the proof. Complete proofs need no transformer as input but still yield a transformer as output, so they are simply first order transformers. For example, proof (2.2) corresponds to a function of type $(J^n \rightarrow J)^4 \rightarrow (J^n \rightarrow J)$, since it has four incomplete leaves with goals $[\mathcal{E}]\ \mathcal{H} \gg F_i$. If it were completed, the proof would correspond to a justification transformer for $[\mathcal{E}]\ \mathcal{H} \gg (F_1 \ \&\ F_2) \ \&\ (F_3 \ \&\ F_4)$, a function of type $\mathsf{J}^n \rightarrow \mathsf{J}$. (Or more precisely, $\mathsf{J}^n \rightarrow (\mathsf{J} \times \mathsf{J})^2$.)

2.4 Proof rules and extraction

This section contains descriptions of the proof rules and defines the notions of extraction and extracted code.

As was seen in the previous section, complete λPRL proofs correspond to justification transformers; a proof π of $\mathcal{H} \gg F$ corresponds to some Γ of type $\mathsf{J}^n \rightarrow \mathsf{J}$. *Extraction* maps λPRL proofs to programs, programs whose meanings are justification transformers for the goals of the proof. If P is the extracted code for π, then the meaning of P is a justification transformer Γ that validates $\mathcal{H} \gg F$. The form of the λPRL logic guarantees that extracted code is total recursive, so extracted code manipulates computable (representations of) justifications, in computable ways.

In λPRL as implemented, extracted code is written in Lisp [All]. (Franz Lisp or Zetalisp, depending on the machine.) Here, an expression language is used instead of Lisp. In addition to the terms and formulas described in section 2.1, it uses the following:

$$\text{close}\ (x_1, \ldots, x_n) \,.\, e$$
$$\text{proc}\ (x_1 : T_1, \ldots, x_n : T_n) : T' \,.\, e$$

$$\text{if } b_1 \rightarrow e_1 \ \square \cdots \square \ b_n \rightarrow e_n \text{ fi}$$
$$\text{let } x = e_1 \text{ in } e_2 \text{ ni}$$
$$\text{let } (x_1, \ldots, x_n) = e_1 \text{ in } e_2 \text{ ni}$$
$$\text{trivial.}$$

In addition, the language allows general lists (lists of things other than integers). In the closure expression, the x_i are the identifiers to be closed over or saved. In the procedure expression, if the body e has free identifiers, they are evaluated dynamically in the environment active at the time the procedure is used. Of course, if a procedure is within a closure, then the free identifiers closed over are effectively statically scoped. In the second let expression, e_1 should yield a list of length n; for each i, x_i is bound to the i'th component of the list. The constant trivial is discussed below. Some abbreviations are used to improve readability. For example,

$$\text{let } x_1 = e_1 ; x_2 = e_2 \text{ in } e_3 \text{ ni}$$

means

$$\text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } e_3 \text{ ni ni}$$

and

$$\text{proc}\ (x, y : T, z : T') : T'' \,.\, e$$

means

$$\text{proc}\ (x : T, y : T, z : T') : T'' \,.\, e.$$

Up to this point, terms have represented integers and lists. In extracted code, they can also represent justifications. The simplest example is trivial, the constant whose value is the trivial justification, *triv*. In procedures, the keyword just indicates a justification, so for example in proc $(x : \mathrm{int}) : \mathrm{just} . e$, e must be a justification-valued expression. That is, e must be of type just. The justification-valued expressions are defined as follows. Below, X is an identifier of type just, ξ is an identifier of type int or list, and x is an identifier of unspecified type. Similarly, E is an expression of type just, ϵ is an expression of type int or list, and e is an expression of unspecified type. The justification-valued expressions are:

X

trivial

close $(x_1, \ldots, x_n) . E$

let $x = e$ in E ni

let $(x_1, \ldots, x_n) = e$ in E ni

if $b_1 \rightarrow E_1 \, \square \cdots \square \, b_n \rightarrow E_n$ fi

(E_1, E_2)

$(\epsilon_1, \ldots, \epsilon_n, E)$

proc $(\xi_1 : T_1, \ldots, \xi_n : T_n) : \mathrm{just} . E$

proc $(X_1, \ldots, X_n : \mathrm{just}) : \mathrm{just} . E$

The last four cases show how justifications are represented: (E_1, E_2) represents an and-justification, (ϵ, E) represents an or-justification (where ϵ must be of type int), $(\epsilon_1, \ldots, \epsilon_n, E)$ represents a some-justification,

$$\mathrm{proc}\ (\xi_1 : T_1, \ldots, \xi_n : T_n) : \mathrm{just} . E$$

represents an all-justification, and

$$\mathrm{proc}\ (X_1, \ldots, X_n : \mathrm{just}) : \mathrm{just} . E$$

represents an implication-justification when $n = 1$. (The general case $n > 1$ is used for building the justification-transforming procedures used for lemmas.) In addition, the procedures that represent all-justifications and implication-justifications are always surrounded by close clauses that close over all the free identifiers of the procedure. This is done to ensure that these procedures are effectively statically closed.

Definition 2.9 A *concrete justification* is an expression of type just. Let M be the obvious meaning function for the language and let e be a concrete justification; the justification *represented by* e in environment env, $\mathsf{M}(e)(env)$, is the abstract value denoted by e in env.

Definition 2.10 If e is a concrete justification, then $e \Vdash F, env$ means $\mathsf{M}(e)(env) \Vdash F, env$. That is, "$e$ justifies F in env" means that the justification γ represented by e in env justifies F in env. If env is understood, it is sometimes omitted: "e justifies F" means $e \Vdash F, env$.

Warning. I will almost always leave out the "concrete" in "concrete justification," and I will generally shorten phrases like "e represents a justification for F" to "e justifies F." For example, "J is a justification identifier for F" means that J is a concrete justification identifier (an identifier of type just), where the abstract value represented by $env(J)$ justifies F in env. Concrete justifications are programs and do not directly justify formulas; the justifications they represent do.

Notation. Pgm is the set of all programs. Pf is the set of all λPRL proofs. π is a proof. The *extraction function* X is of type $\mathsf{Pf} \times \mathsf{Ids}^* \rightarrow \mathsf{Pgm}$, where Ids^* is the set of all finite lists of identifiers.[8]

Definition 2.11 *(Extraction)*

Let π be a proof and let $(J_1, \ldots, J_n)$ be a list of justification identifiers for its hypotheses. The *extraction function* X maps π and $(J_1, \ldots, J_n)$ to a program, the *extracted program* or *extracted code* for π. If π is a proof of

$$[x_1 : T_1, \ldots, x_m : T_m]\ H_1, \ldots, H_n \gg F$$

and $J_1, \ldots, J_n$ are justification identifiers, then $\mathsf{X}[\![\pi, (J_1, \ldots, J_n)]\!]$ is an expression such that if $J_i \Vdash H_i, env$ (for each i), then

$$\mathsf{X}[\![\pi, (J_1, \ldots, J_n)]\!] \Vdash F, env.$$

[8] This differs slightly from Bates [Bat], where the identifiers are not included. As explained in chapter 3, this definition avoids many of the β-reductions done by Bates in his first optimization pass.

The expression may contain the J_i and the x_i free. Since X yields an expression with free identifiers, if an actual justification-transformer is needed,

$$\mathsf{close}\ (x_1, \ldots, x_m)\ .\ \mathsf{proc}\ (J_1, \ldots, J_n)\ .\ \mathsf{X}[\![\pi, (J_1, \ldots, J_n)]\!]$$

is used. For example, the hypothesis rule π_{hyp} has the form

$$[x_1 : T_1, \ldots, x_m : T_m]\ H_1, \ldots, H_n \gg H_i \quad \text{by hyp } i$$

and $\mathsf{X}[\![\pi_{hyp}, (J_1, \ldots, J_n)]\!]$ is the expression J_i. A justification transformer for π_{hyp} would be

$$\mathsf{proc}\ (J_1, \ldots, J_n)\ .\ J_i.$$

(No close is necessary, since the procedure makes no nonlocal references.)

Notation. J, K, L, and M are justification identifiers. $\mathcal{J}$ is a list of justification identifiers, $J_1, \ldots, J_n$. Typically, J_i is a justification identifier for hypothesis H_i. $\mathsf{X}[\![\pi, J_1, \ldots, J_n]\!]$ means $\mathsf{X}[\![\pi, (J_1, \ldots, J_n)]\!]$. $\mathsf{X}[\![\pi, \mathcal{J}]\!]$ means $\mathsf{X}[\![\pi, J_1, \ldots, J_n]\!]$.

X is defined by induction on proofs. The extracted code for a proof π is built using the extracted code for the subproofs of π; the exact combination used depends on the proof rule used in π. For example, let π be

$$[\mathcal{E}]\ \mathcal{H} \gg F \,\&\, G \quad \text{by intro}$$

- $[\mathcal{E}]\ \mathcal{H} \gg F \quad \ldots$
- $[\mathcal{E}]\ \mathcal{H} \gg G \quad \ldots$

and let σ_1 and σ_2 be its subproofs. Because this is a proof by ***and-intro***, $\mathsf{X}[\pi, \mathcal{J}]$, which is the extracted code for π, is $(\mathsf{X}[\sigma_1, \mathcal{J}], \mathsf{X}[\sigma_2, \mathcal{J}])$. The justification of F & G is built by pairing the justifications for F and G. The ***and-intro*** rule actually generates subgoals, so it corresponds to an inductive case of the definition of X. The base cases of X come from rules like ***hyp***, which generate no subgoals.

It is time to talk about computability and constructivity. Extracted code will be recursive. ("Recursive" will always mean "total recursive.") For the code to be recursive, the base cases of X must produce recursive code and the inductive cases of X must combine code recursively. Since the cases of X's definition correspond to proof rules of the logic, this means that each rule must correspond to a recursive operation. Thus, rules that would correspond to nonrecursive operations must be barred. For example, the law of the excluded middle

$$[\mathcal{E}]\ \mathcal{H} \gg F \mid \neg F$$

is not in the logic, since there is no recursive way to decide which of arbitrary F and $\neg F$ are true.[9]

[9] A special case of excluded middle *is* allowable. Since integer and list terms are recursively evaluable in λPRL, if F is a literal over such terms (that is, if it is an atomic formula or the negation of an atomic formula), then F and $\neg F$ can be decided simply by evaluating them, so a justification for $F \mid \neg F$ can be built recursively. This special case of the excluded middle is included in the logic in the decision rule.

λPRL is a ***constructive*** logic: its proof rules can be understood as specifying computable, recursive constructions. Because of the justification-based semantics being used, these constructions are applied to concrete justifications. For example, the ***and-intro*** rule corresponds to a pairing operation: to build the justification for F & G, the justifications for F and G are paired up.

Constructive logic proper is a fascinating area, but it is outside the scope of this thesis. Here are some references for people interested in learning more about constructive logic. Beeson [Bee] gives some historical background and presents some different philosophies of constructive mathematics. (In addition, he presents material on various technical topics, including the type theories of Martin-Löf [ML79], which influenced the work of Bates and Constable on PRL.) Dummett [Dum] also presents material on constructive logics. Troelstra [Tro] is an encyclopedic reference.

The rest of this section is devoted to the proof rules. There are many rules, unfortunately, so this section is quite long. The rules described here are slightly streamlined versions of the actual rules: inessential variants of rules have been dropped, and some rules have been restructured to make them clearer.[10] The rules are broken down into the following categories:

[10] For those already familiar with the rules as actually implemented [Prl], here are the changes: the n-ary operators were removed from the propositional rules, the optional identifiers were made mandatory in the quantifier rules, the extra parameters were eliminated from the induction rules, the implicit sequence rule application was removed from the

propositional calculus rules, quantifier rules, induction rules, and miscellaneous rules.

Propositional calculus rules

The propositional calculus rules (and the quantifier rules below) are variations of Gentzen's sequent calculus rules [ErP, Gen, Pra65]. They allow for the proof and use of formulas which have as their main connective &, |, ⇒, and ¬. In particular, the *introduction rules* allow them to be proved and the *elimination rules* allow them to be broken down when they appear as hypotheses.

For example, when the formula $F \mathbin{\&} G$ is conclusion of a goal, the *and-intro* rule is used to prove it. When it appears as a hypothesis, the *and-elim* rule is used to break it down. As seen earlier in the chapter, the and *and-intro* rule generates two subgoals, $\mathcal{H} \gg F$ and $\mathcal{H} \gg G$. The *and-elim* rule generates only one subgoal: if $\mathcal{H}, F \mathbin{\&} G, \mathcal{H}' \gg F'$ is the main goal then $\mathcal{H}, \mathcal{H}', F, G \gg F'$ is the subgoal.

Notation. Given a goal $[\mathcal{E}]\ \mathcal{H} \gg F$, if $\mathcal{E}$ is $x_1 : T_1, \ldots, x_m : T_m$, then close $(\mathcal{E})$ means close $(x_1, \ldots, x_m)$. If the environment of a goal is omitted, it should be assumed to be $\mathcal{E}$.

---

function definition rule, the simplify rule was dropped, and the arith, equality, and division rules were lumped together into the decision rule.

Notation. $\mathcal{H}$ and $\mathcal{H}'$ are lists of hypotheses $H_1, \ldots, H_{n-1}$ and $H_{n+1}, \ldots, H_q$ respectively. $\mathcal{J}$ and $\mathcal{J}'$ are corresponding lists of justification identifiers, $J_1, \ldots, J_{n-1}$ and $J_{n+1}, \ldots, J_q$. Warning: this changes the original meaning of $\mathcal{H}$ and $\mathcal{J}$ from lists of length n to lists of length $n-1$.

Notation. Below, π_{rule} refers to the specified proof rule. For example, $\pi_{and\text{-}intro}$ refers to the *and-intro* proof rule. Within the description of each rule, σ is a subproof of π_{rule}. If π_{rule} has only one subproof, then σ refers to it. If π_{rule} has more than one subproof, then σ_1 refers to the first one, σ_2 to the second, etc.

The *and-intro* rule splits the conclusion into two subformulas, one for each of the two subgoals. In the extracted code, the justifications of the subformulas are paired to form the justification for the conclusion. The $\pi_{and\text{-}intro}$ proof rule is

$$\mathcal{H} \gg F \mathbin{\&} G \quad \text{by intro}$$

- $\mathcal{H} \gg F$
- $\mathcal{H} \gg G$

and $\mathsf{X}[\![\pi_{and\text{-}intro}, \mathcal{J}]\!]$ is $(\mathsf{X}[\![\sigma_1, \mathcal{J}]\!], \mathsf{X}[\![\sigma_2, \mathcal{J}]\!])$.

The *and-elim* rule splits the hypothesis into its conjuncts. The extracted code for the rule unpairs the justification of the hypothesis to get justifications for the conjuncts. The $\pi_{and\text{-}elim}$ proof rule is

$$\mathcal{H},\ F \,\&\, G,\ \mathcal{H}' \gg F' \quad \text{by elim } n$$
$$\bullet\ \mathcal{H},\ \mathcal{H}',\ F,\ G \gg F'$$

and $\mathsf{X}[\![\pi_{and\text{-}elim}, J, J_n, J']\!]$ is

$$\text{let } (K_1, K_2) = J_n \text{ in } \mathsf{X}[\![\sigma, J, J', K_1, K_2]\!] \text{ ni.}$$

Before presenting the other twenty proof rules,[11] perhaps it is time for a small example of extracted code. Let π be the proof

$$F \,\&\, G \gg G \,\&\, F \quad \text{by elim 1}$$
$$\bullet\ F, G \gg G \,\&\, F \quad \text{by intro}$$
$$\bullet\bullet\ F, G \gg G \quad \text{by hyp 2}$$
$$\bullet\bullet\ F, G \gg F \quad \text{by hyp 1.}$$

Also, let σ_1 be the first and only subproof of π, let $\sigma_{1,1}$ be the first subproof of σ_1, and let $\sigma_{1,2}$ be the second subproof of σ_1. A justification γ for $F \,\&\, G$ is a pair of justifications (γ', γ'') for F and G respectively, so intuitively the extracted code for π should take γ, unpair it into γ' and γ'', and return (γ'', γ'). Let J be a justification identifier for $F \,\&\, G$, and let K and L be new justification identifiers, then

$$\begin{aligned} \mathsf{X}[\![\pi, J]\!] &= \text{let } (K, L) = J \text{ in } \mathsf{X}[\![\sigma_1, K, L]\!] \text{ ni} \\ &= \text{let } (K, L) = J \text{ in } (\mathsf{X}[\![\sigma_{1,1}, K, L]\!], \mathsf{X}[\![\sigma_{1,2}, K, L]\!]) \text{ ni} \end{aligned}$$

[11] Sigh.

$$= \text{let } (K, L) = J \text{ in } (L, K) \text{ ni}$$

which agrees with the intuition of how $\mathsf{X}[\![\pi, J]\!]$ should work.

The reader may want to try proving $F \,\&\, G \gg G \,\&\, F$ by introducing $G \,\&\, F$ first and then using an elimination and hypothesis rule in each branch. The code for the resulting proof (like the proof itself) is slightly more complicated than the code above: it is

$$(\text{let } (K, L) = J \text{ in } L \text{ ni}, \text{ let } (K, L) = J \text{ in } K \text{ ni}).$$

To continue on with the rules, the *or-intro* rule allows one to prove a disjunction by explicitly proving one of the disjuncts. In the extracted code, the justification of the disjunct is paired together with a tag indicating which disjunct was proved. The result justifies the disjunction. The $\pi_{or\text{-}intro}$ proof rule is

$$\mathcal{H} \gg F_1 \mid F_2 \quad \text{by intro } k$$
$$\bullet\ \mathcal{H} \gg F_k$$

where k is 1 or 2 and $\mathsf{X}[\![\pi_{or\text{-}intro}, J]\!]$ is $(k, \mathsf{X}[\![\sigma, J]\!])$.

The *or-elim* rule splits the proof into two cases. The first case contains the left disjunct as a new hypothesis; the second case contains the right disjunct. The extracted code for the rule checks the tag of the or justification and executes the code for the appropriate case: if the tag is 1, the left disjunct is true, so the code for the first subproof is executed; if the tag is 2, the right disjunct is true, so the code for the second subproof is executed. The $\pi_{or\text{-}elim}$ proof rule is

$$\mathcal{H},\ F \mid G,\ \mathcal{H}' \gg F' \quad \text{by elim } n$$
$$\bullet\ \mathcal{H},\ \mathcal{H}',\ F \gg F'$$
$$\bullet\ \mathcal{H},\ \mathcal{H}',\ G \gg F'$$

and $\mathrm{X}[\![\pi_{or\text{-}elim}, J, J_n, J']\!]$ is

```
let ⟨k, K⟩ = J_n
in
   if k = 1 → X[σ1, J, J', K]
   [] k = 2 → X[σ2, J, J', K]
   fi
ni.
```

The *impl-intro* rule lets one assume the antecedent of an implication in order to prove its consequent. In the extracted code for the rule, the code for the proof of the consequent is abstracted to get a procedure that takes a justification for the antecedent and yields a justification for the consequent. The $\pi_{impl\text{-}intro}$ proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F \Rightarrow G \quad \text{by intro}$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H}, F \gg G$$

and $\mathrm{X}[\![\pi_{impl\text{-}intro}, J]\!]$ is

$$\text{close } (\mathcal{E}, J)\,.\,\text{proc } (K : \text{just}) : \text{just}\,.\,\mathrm{X}[\![\sigma, J, K]\!].$$

The body of the procedure may contain references to the x_i in $\mathcal{E}$ or to the justification identifiers in J, so the extracted code closes over these identifiers so that their values will be available at extracted code runtime. The resulting closed procedure is a pure value in the sense that it justifies $F \Rightarrow G$ in any runtime environment in which $F \Rightarrow G$ is true.

The *impl-elim* rule breaks down the eliminated implication into its antecedent and consequent. The first subproof proves the antecedent. The second subproof continues the proof of the main goal's conclusion, using the antecedent and consequent as new hypotheses. The $\pi_{impl\text{-}elim}$ proof rule is

$$\mathcal{H},\ F \Rightarrow G,\ \mathcal{H}' \gg F' \quad \text{by elim } n$$
$$\bullet\ \mathcal{H},\ \mathcal{H}' \gg F$$
$$\bullet\ \mathcal{H},\ \mathcal{H}',\ F,\ G \gg F'$$

and $\mathrm{X}[\![\pi_{impl\text{-}elim}, J, J_n, J']\!]$ is

$$\text{let } K = \mathrm{X}[\![\sigma_1, J, J']\!];\ L = J_n(K) \text{ in } \mathrm{X}[\![\sigma_2, J, J', K, L]\!] \text{ ni.}$$

In the extracted code for the rule, the code for the first subproof is used to produce a justification for the antecedent, which is then used (with the implication justification) to produce a justification for the consequent. The implication justification is a procedure that takes a justification for the antecedent and yields a justification for the consequent, so the appropriate procedure application yields the desired justification.

The *not-intro* rule resembles the *impl-intro* rule. (Negation is treated as meaning "implies false.") Its extracted code is similar to the code for the *impl-intro*. The $\pi_{not\text{-}intro}$ proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg \neg F \quad \text{by intro}$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H},\ F \gg \text{false}$$

and $\mathbf{X}[\![\pi_{not\text{-}intro}, \mathcal{J}]\!]$ is

$$\text{close}\ (\mathcal{E}, \mathcal{J})\ .\ \text{proc}\ K\ .\ \mathbf{X}[\![\sigma, \mathcal{J}, K]\!].$$

The *not-elim* rule is similar to the *impl-elim* rule, because of the treatment of negation. The $\pi_{not\text{-}elim}$ proof rule is

$$\mathcal{H},\ \neg F,\ \mathcal{H}' \gg G \quad \text{by elim } n$$
$$\bullet\ \mathcal{H},\ \mathcal{H}' \gg F$$
$$\bullet\ \mathcal{H},\ \mathcal{H}',\ F,\ \text{false} \gg G$$

and $\mathbf{X}[\![\pi_{not\text{-}elim}, \mathcal{J}, \mathcal{J}_n, \mathcal{J}']\!]$ is

$$\text{let } K = \mathbf{X}[\![\sigma, \mathcal{J}, \mathcal{J}']\!];\ L = \mathcal{J}_n(K) \text{ in } \mathbf{X}[\![\pi_{not\text{-}elim}, \mathcal{J}, \mathcal{J}', K, L]\!] \text{ ni}.$$

The *false-intro* rule states that false is provable given contradictory hypotheses, F and $\neg F$. Its extracted code produces a justification for false by taking the justification of F and using it as input to the justification for $\neg F$. (Since $\neg F$ is the same as $F \Rightarrow$ false, the result is a justification for false.) Of course, the code for this rule is never meant to be run, since to run it would require actually having justifications for contradictory hypotheses. (In practice, the code for this rule can be optimized away.) The $\pi_{false\text{-}intro}$ proof rule is

$$\mathcal{H} \gg \text{false} \quad \text{by intro } F$$
$$\bullet\ \mathcal{H} \gg F$$
$$\bullet\ \mathcal{H} \gg \neg F$$

and $\mathbf{X}[\![\pi_{false\text{-}intro}, \mathcal{J}]\!]$ is $\mathbf{X}[\![\sigma_2, \mathcal{J}]\!](\mathbf{X}[\![\sigma_1, \mathcal{J}]\!])$.

The *false-elim* rule is similar to the *false-intro* rule in that the extracted code for it is never run because to run it one would need a justification for false. The extracted code yields trivial as its justification for the conclusion, and though this might be wrong in general, it is acceptable here because the code will never be run. The extracted code is supposed to produce a justification of the conclusion under the assumption that the hypotheses have justifications. Since not all the hypotheses can be true, not all of them can have justifications, so it does not matter that the justification being returned does not actually justify the conclusion; any justification would be as good as trivial. The $\pi_{\mathit{false\text{-}elim}}$ proof rule is

$$\mathcal{H},\ \mathsf{false},\ \mathcal{H}' \gg F \quad \text{by elim } n$$

and $\mathsf{X}[\![\pi_{\mathit{false\text{-}elim}}, \mathcal{J}, \mathcal{J}_n, \mathcal{J}']\!]$ is trivial.

Quantifier rules

The quantifier rules permit the proof and use of existential and universal formulas. All four of these rules depend heavily on the proof environment: the *all-elim* and *some-intro* rules use terms whose identifiers must appear in the environment, and the *some-elim* and *all-intro* rules introduce new identifiers into the environment.

Notation. $\mathcal{X}$ is $x_1, \ldots, x_m$, $\mathcal{Y}$ is $y_1, \ldots, y_m$, and $\mathcal{T}$ is $t_1, \ldots, t_m$.

The *all-intro* rule proves $\mathsf{All}\ x_1 : T_1, \ldots, x_m : T_m \,.\, F$ by proving F with the values of the bound identifiers unspecified. Below, the identifiers $y_1, \ldots, y_m$ must be *new* (not contained in the environment $\mathcal{E}$). The y_i are substituted for the x_i in the proof of the body, so the proof has the y_i implicitly universally quantified. The extracted code for the rule takes the code for the subproof and abstracts it with respect to the new identifiers. The resulting procedure is an all justification: it maps values for the new

identifiers to a justification for the body. The $\pi_{\mathit{all\text{-}intro}}$ proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg \mathsf{All}\ x_1 : T_1, \ldots, x_m : T_m \,.\, F \quad \text{by intro } \mathcal{Y}$$
$$\bullet\ [\mathcal{E}, y_1 : T_1, \ldots, y_m : T_m]\ \mathcal{H} \gg F\,[\mathcal{Y}/\mathcal{X}]$$

and $\mathsf{X}[\![\pi_{\mathit{all\text{-}intro}}, \mathcal{J}]\!]$ is

$$\mathsf{close}\,(\mathcal{E}, \mathcal{J})\,.\,\mathsf{proc}\,(y_1 : T_1, \ldots, y_m : T_m) : \mathsf{just}\,.\,\mathsf{X}[\![\sigma, \mathcal{J}]\!].$$

Like the *impl-intro* rule, the abstracted code closes over $\mathcal{E}$ and J so that the values of their identifiers will be available at runtime. The resulting closed procedure is a pure value that justifies the universal in any runtime environment in which the universal is true.

The *all-elim* rule lets one justify a particular instance of the body of the eliminated universal. The terms in the rule specify the particular values with which to instantiate. The terms must be type-correct, and their identifiers must all appear in $\mathcal{E}$. The $\pi_{\mathit{all\text{-}elim}}$ proof rule is

$$\mathcal{H},\ \mathsf{All}\ x_1 : T_1, \ldots, x_m : T_m \,.\, F,\ \mathcal{H}' \gg G \quad \text{by elim } n, \mathcal{T}$$
$$\bullet\ \mathcal{H},\ \mathsf{All}\ x_1 : T_1, \ldots, x_m : T_m \,.\, F,\ \mathcal{H}',\ F\,[\mathcal{T}/\mathcal{X}] \gg G$$

and $\mathsf{X}[\![\pi_{\mathit{all\text{-}elim}}, \mathcal{J}, \mathcal{J}_n, \mathcal{J}']\!]$ is

$$\mathsf{let}\ K = \mathcal{J}_n(\mathcal{T})\ \mathsf{in}\ \mathsf{X}[\![\sigma, \mathcal{J}, \mathcal{J}_n, \mathcal{J}', K]\!]\ \mathsf{ni}.$$

The t_i can use identifiers that would be bound by quantifiers embedded in F. If so, $F\,[\mathcal{T}/\mathcal{X}]$ must have those bound identifiers renamed.

Unlike the other *elim* rules, the *all-elim* rule does not remove the eliminated formula from the list of hypotheses. Thus, universals can be instantiated many times. The extracted code for the rule calls the all justification with the given terms as parameters. The result is a justification for the instantiation.

The *some-intro* rule proves an existential by proving a specific instance of its body. The $\pi_{some\text{-}intro}$ proof rule is

$$\mathcal{H} \gg \text{Some } x_1 : T_1, \ldots, x_m : T_m \,.\, F \quad \text{by intro } \mathcal{T}$$
$$\bullet\ \mathcal{H} \gg F[\mathcal{T}/\mathcal{X}]$$

and $\mathsf{X}[\pi_{some\text{-}intro}, \mathcal{J}]$ is $(\mathcal{T}, \mathsf{X}[\sigma, \mathcal{J}])$. The terms specified in the rule give the witness values for the some justification. The terms must be type-correct, and their identifiers must all appear in $\mathcal{E}$.

The *some-elim* rule lets one conclude that the body of the eliminated existential holds for some fixed values, which are denoted by new identifiers. In the extracted code, the new identifiers are given values by binding them to the witness values of the some justification. The $\pi_{some\text{-}elim}$ proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ \text{Some } x_1 : T_1, \ldots, x_m : T_m \,.\, F,\ \mathcal{H}' \gg G \quad \text{by elim } n, \mathcal{Y}$$
$$\bullet\ [\mathcal{E},\ y_1 : T_1, \ldots, y_m : T_m]\ \mathcal{H},\ \mathcal{H}',\ F[\mathcal{Y}/\mathcal{X}] \gg G$$

and $\mathsf{X}[\pi_{some\text{-}elim}, J, J_n, J']$ is

$$\text{let } (\mathcal{Y}, K) = J_n \text{ in } \mathsf{X}[\sigma, J, J', K] \text{ ni}.$$

If any of the y_i would be bound by quantifiers embedded in F, then $F[\mathcal{Y}/\mathcal{X}]$ must have those bound identifiers renamed.

Before going on to the induction rules, here is another example of a λPRL proof and its extracted code. The goal of the proof is

$$[\,]\ (\text{Some } x : T \,.\, F) \Rightarrow G \gg \text{All } x : T \,.\, (F \Rightarrow G).$$

Let π be

$$[\,]\ (\text{Some } x : T \,.\, F) \Rightarrow G \gg \text{All } x : T \,.\, (F \Rightarrow G) \quad \text{by intro } x$$
$$\bullet\ [x : T]\ (\text{Some } x : T \,.\, F) \Rightarrow G \gg F \Rightarrow G \quad \text{by intro}$$
$$\bullet\bullet\ [x : T]\ (\text{Some } x : T \,.\, F) \Rightarrow G,\ F \gg G \quad \text{by elim 1}$$
$$\bullet\bullet\bullet\ [x : T]\ F \gg \text{Some } x : T \,.\, F \quad \text{by intro } x$$
$$\bullet\bullet\bullet\bullet\ [x : T]\ F \gg F \quad \text{by hyp 1}$$
$$\bullet\bullet\bullet\ [x : T]\ F,\ \text{Some } x : T \,.\, F,\ G \gg G \quad \text{by hyp 3.}$$

Bending the notation a bit, let the five descendent subproofs of π be σ_1, σ_2, σ_3, σ_4, and σ_5, respectively, and let J be a justification identifier, then

$$\begin{aligned}
\mathsf{X}[\pi, J] &= \text{close}\,(J)\,.\,\text{proc}\,(x : T) : \text{just}\,.\,\mathsf{X}[\sigma_1, J] \\
\mathsf{X}[\sigma_1, J] &= \text{close}\,(x, J)\,.\,\text{proc}\,(K : \text{just}) : \text{just}\,.\,\mathsf{X}[\sigma_2, J, K] \\
\mathsf{X}[\sigma_2, J, K] &= \text{let } L = \mathsf{X}[\sigma_3, K]; M = J(L) \text{ in } \mathsf{X}[\sigma_5, K, L, M] \text{ ni} \\
\mathsf{X}[\sigma_3, K] &= (x, \mathsf{X}[\sigma_4, K]) \\
\mathsf{X}[\sigma_4, K] &= K \\
\mathsf{X}[\sigma_5, K, L, M] &= M.
\end{aligned}$$

Combining the code, $\mathsf{X}[\![\pi, J]\!]$ is

$$\begin{array}{l} \text{close}\ (J)\ .\ \text{proc}\ (x : T) : \text{just}\ . \\ \quad \text{close}\ (x, J)\ .\ \text{proc}\ (K : \text{just}) : \text{just}\ . \\ \quad\quad \text{let}\ L = (x, K); M = J(L)\ \text{in}\ M\ \text{ni}. \end{array}$$

Induction rules

The induction rules are similar to the *all-intro* rule, since they prove universals. Here, however, the type being quantified over is important: the extracted code for the rules depends critically on the structure of the integers (for the integer induction rule) and on the structure of lists (for the list induction rule).

The integer induction rule proves All x : int . F by induction. The base case is a parameter k, and the inductive cases step toward k by ones. (The λPRL rules as implemented allow for more base cases and larger step sizes [Prl].) The extracted code for the rule runs recursively. The justification for $F[0/x]$ is built the base case; in the inductive cases, a recursive call provides a justification for $F[x-1/x]$ or $F[x+1/x]$ (as appropriate), which is then used to build a justification for F. The $\pi_{int\text{-}ind}$ proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg \text{All}\ x : \text{int}\ .\ F \quad \text{by ind}\ k$$

- $[\mathcal{E},\ x : \text{int}]\ \mathcal{H},\ x < k,\ F[x+1/x] \gg F$

- $[\mathcal{E}]\ \mathcal{H} \gg F[k/x]$
- $[\mathcal{E},\ x : \text{int}]\ \mathcal{H},\ x > k,\ F[x-1/x] \gg F$

where k is an integer constant. $\mathsf{X}[\![\pi_{int\text{-}ind}, J]\!]$ is

$$\text{close}\ (\mathcal{V})\ .\ \text{proc}\ (x)\ .\ f(x, \mathcal{V})$$

where $\mathcal{V}$ is the set of closure variables (J plus the identifiers of $\mathcal{E}$), and f is

```
proc (x, V) .
    if x < k →
        let K1 = trivial; L1 = f(x + 1, V)
        in X[σ1, J, K1, L1]
        ni
    [] x = k → X[σ2, J]
    [] x > k →
        let K2 = trivial; L2 = f(x − 1, V)
        in X[σ3, J, K2, L2]
        ni
    fi
```

Strictly speaking, f does not need the identifiers of $\mathcal{V}$ as parameters, but it simplifies the implementation somewhat, since f then has no nonlocal identifiers.

The list induction rule is similar to the integer induction rule. Here, the base case is the empty list, (), and the inductive case proves $F[y\,.\,z/x]$ with the help of $F[z/x]$. ($y\,.\,z$ is the cons of the integer y to the head of the list z.) The extracted code for the rule works recursively, of course, building a justification for $F[()/x]$ as the base case, and using a recursive call to get a justification for $F[tl(x)/x]$. The $\pi_{list\text{-}ind}$ proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg \text{All } x\text{:list} . F \quad \text{by ind } y\,.\,z$$

- $[\mathcal{E}]\ \mathcal{H} \gg F[()/x]$
- $[\mathcal{E}, y\text{:int}, z\text{:list}]\ \mathcal{H},\ F[z/x] \gg F[y\,.\,z/x]$

and $\mathsf{X}[\![\pi_{list\text{-}ind}, \mathcal{J}]\!]$ is

$$\text{close}\,(\mathcal{V})\,.\,\text{proc}\,(x)\,.\,f(x, \mathcal{V})$$

where $\mathcal{V}$ is the set of closure variables ($\mathcal{J}$ plus the identifiers of $\mathcal{E}$), and f is

```
proc (x, V) .
  if x = () → X[σ1, J]
  □ x ≠ () →
    let y = hd(x); z = tl(x); K = f(z, V)
    in X[σ2, J, K]
    ni
  fi.
```


Miscellaneous rules

The rest of the rules are presented here. Some, like decision and function definition, are logical rules that did not fit into the categories above. Others, like sequence and theorem, are structural rules.

The *decision* rule is used to prove literals and disjuncts of literals. (A literal is an atomic formula or the negation of an atomic formula.) For purposes of explaining the extracted code, it is not important what precise formulas can be proved via *decision*. Speaking generally, the rule knows about the ring of integers, the relationship between *hd*, *tl*, and ., (car, cdr, and cons respectively), some facts about division, and equality under uninterpreted function symbols.[12]

There are two cases to the *decision* rule. In the first case, $\pi_{decision}$ is

$$\mathcal{H} \gg F \quad \text{by decision}$$

where F is a literal. In this case, $\mathsf{X}[\![\pi_{decision}, \mathcal{J}]\!]$ is trivial. In the second case, $\pi_{decision}$ is

$$\mathcal{H} \gg F_1 \mid F_2 \quad \text{by decision}$$

where each F_i is a literal. In this case, $\mathsf{X}[\![\pi_{decision}, \mathcal{J}]\!]$ is

$$\text{if } F_1 \rightarrow (1, \text{trivial})\ \square\ F_2 \rightarrow (2, \text{trivial})\ \text{fi}.$$

[12] The *decision* rule is actually implemented as the *arith*, *division*, and *equality* rules [CJE,Joh,Koz,Nel,Prl].

The validity of the rule assures that at runtime, at least one of the F_i will be true, and the computability of the extracted code assures that both of the F_i will be decidable.

The explicit procedure definition rule connects the programming part of λPRL (the explicitly defined procedures) to the proof logic. If a procedure f is defined by

$$\text{proc}\ (x : \text{int}) : T'\ .\ \text{if}\ x = 0 \rightarrow e_1\ \square\ x < 0 \rightarrow e_2\ \square\ x > 0 \rightarrow e_3\ \text{fi}$$

where e_1, e_2, and e_3 are terms, then $f(0) = e_1$ is provable, and given a term t, $f(t) = e_2[t/x]$ is provable if $t < 0$ is provable, and $f(t) = e_3[t/x]$ is provable if $t > 0$ is provable. More generally, the proof rule $\pi_{expl\text{-}def}$ is

$$\mathcal{H} \gg F \quad \text{by def}\ f(t_1, \ldots, t_k)\ dir$$

where the direction *dir* is up, down, or omitted.

If the direction is omitted, then t_1 must be the constant zero and $\pi_{expl\text{-}def}$ must be

$$\mathcal{H} \gg f(0, t_2, \ldots, t_k) = e_1' \quad \text{by def}\ f(0, t_2, \ldots, t_k)$$

where e_1' is e_1 with the t_i's substituted for the formal parameters of f. If the direction is up, then $\pi_{expl\text{-}def}$ must be

$$\mathcal{H} \gg f(t_1, \ldots, t_k) = e_2' \quad \text{by def}\ f(t_1, \ldots, t_k)\ \text{up}$$

- $\mathcal{H} \gg t_1 > 0$.

The down direction is similar. In all three cases, the extracted code for the rule, $\mathsf{X}[\![\pi_{expl\text{-}def}, \mathcal{J}]\!]$, is trivial.

With the ***hypothesis*** rule, one can prove the conclusion if it appears as a hypothesis. The extracted code simply returns the justification for the hypothesis. The π_{hyp} proof rule is $\mathcal{H} \gg H_i$ by hyp i and $\mathsf{X}[\![\pi_{hyp}, \mathcal{J}]\!]$ is J_i.

The ***sequence*** rule permits the addition of a new hypothesis to the list of hypotheses. The first subproof proves the new hypothesis (the ***sequence*** or ***cut*** formula), and the second allows its use. From a logical viewpoint, this rule is not always necessary: as shown by Gentzen [Gen, Pra65], any proof that does not use induction can be normalized into a proof that, in addition, does not use the sequence rule. It is impractical to omit the sequence rule, however, since without it the proofs of sequence formulas may have to be repeated over and over. This can even be seen from the proof in Prawitz [Pra65], in which proofs have their sizes increase exponentially when sequence rules are removed from them. The π_{seq} proof rule is

$$\mathcal{H} \gg F \quad \text{by seq}\ G$$

- $\mathcal{H} \gg G$
- $\mathcal{H}, G \gg F$

and $\mathsf{X}[\![\pi_{seq}, \mathcal{J}]\!]$ is

$$\text{let}\ K = \mathsf{X}[\![\sigma_1, \mathcal{J}]\!]\ \text{in}\ \mathsf{X}[\![\sigma_2, \mathcal{J}, K]\!]\ \text{ni}.$$

Definition 2.12 A λPRL *theorem* is a named λPRL proof.

The *theorem* rule is a purely structural and (logically speaking) unnecessary rule that is essential in practice. It allows large theorems to be broken up in into smaller sections, where each section is a separate theorem. The π_{thm} proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by theorem } Th$$

where $[\mathcal{E}']\ H_{i_1}, \ldots, H_{i_m} \gg F$ is the goal of theorem Th, for some sequence $i_1, \ldots, i_m$. In addition, the bindings of $\mathcal{E}'$ must all appear in $\mathcal{E}$. Let *Th_code* be the name of the (second-order) justification transformer for theorem Th. That is, let *Th_code* be bound to

$$\text{proc}\ (K_1, \ldots, K_m : \text{just}) : \text{just} \,.\, \mathrm{X}[\![\pi_{Th}, K_1, \ldots, K_m]\!]$$

where π_{Th} is the proof of Th. The extracted code for π_{thm}, $\mathrm{X}[\![\pi_{thm}, \mathcal{J}]\!]$, is

$$Th\_code(J_{i_1}, \ldots, J_{i_m}).$$

Definition 2.13 Let Th be a λPRL theorem of

$$[\,]\ \gg \text{All}\ x_1 : T_1, \ldots, x_m : T_m \,.\, \text{Some}\ y : T' \,.\, F.$$

An *extracted* or *implicitly defined procedure* for Th is a procedure f defined by

$$\text{proc}\ (x_1 : T_1, \ldots, x_m : T_m) : T' \,.\, \text{let}\ \langle y, J \rangle = \mathrm{X}[\![\pi]\!](x_1, \ldots, x_m)\ \text{in}\ y\ \text{ni}$$

where π is the proof of Th. An implicitly defined procedure may not be defined from Th if Th does not have the form above.

In the extracted procedure definition rule, if the procedure f was extracted from a theorem whose goal was

$$[\,]\ \gg \text{All}\ x : T \,.\, \text{Some}\ y : T' \,.\, F$$

then the rule allows one to conclude $F[t, f(t)/x, y]$ for any t. In other words, the rule asserts that an extracted procedure satisfies its specification. The extracted code for the rule runs the code for the specified theorem on the given terms, throws away the witness values, and returns the justification for the matrix F. Recall that $\mathcal{X}$ is $x_1, \ldots, x_m$ and $\mathcal{T}$ is $t_1, \ldots, t_m$. The $\pi_{extr\text{-}def}$ proof rule is

$$\mathcal{H} \gg F[\mathcal{T}, f(\mathcal{T})/\mathcal{X}, y] \quad \text{by def}\ f(\mathcal{T})$$

where f was extracted from theorem Th, which had the goal

$$\gg \text{All}\ x_1 : T_1, \ldots, x_m : T_m \,.\, \text{Some}\ y : T' \,.\, F.$$

The extracted code, $\mathrm{X}[\![\pi_{extr\text{-}def}, \mathcal{J}]\!]$, is

$$\text{let}\ \langle y, K \rangle = Th\_code(\mathcal{T})\ \text{in}\ K\ \text{ni}$$

where *Th_code* is the second-order justification transformer extracted from theorem Th.

Extractor correctness

For the extractor to be correct, it must produce justifications for the conclusions of the theorems whose proofs are being extracted from.

Theorem 2.2 *(Extractor correctness)*

Let π be a proof of $[\mathcal{E}]\ \mathcal{H} \gg F$. Given an environment *env*, if $J_i \Vdash H_i, env$ for each i, then $X[\pi, \mathcal{J}] \Vdash F, env$.

Proof

By induction on the structure of π. □

This theorem is extended in chapter 5, after the logic is modified to include call-by-reference parameters with in-place modification of arrays. A full proof is included there.

2.5 Binary search

This section contains the proof of binary search promised earlier in the chapter. With this example the reader may get a better feel for how real λPRL proofs work and how a real program is developed with λPRL.

As previously mentioned, λPRL proofs were designed to be screen-edited, not written down on paper, and even though uninteresting parts of the binary search are omitted, the proof is quite long.

To compress the text of the proof, a tabular format is used here: the proof is represented by a table of goals in which formulas have been replaced by their indices in a list of formulas. For example, the goal $2 \gg 1$ indicates that formula 1 is the conclusion and formula 2 is the hypothesis. References to hypotheses are also made using indices. For example, **elim** 2 means that formula 2 is being eliminated, not the second hypothesis.

The specification for binary search was given in section 2.2 as

$$\textsf{All } a : \textsf{array}, x : \textsf{int} \,.\, \textsf{Some } i : \textsf{int} \,.\, F_1(a, x, i) \tag{2.4}$$

where $F_1(a, x, i)$ is

$$\begin{aligned} &searchable(a) = 1 \ \ \&\ \ a[0] < x \leq a[|a| - 1] \\ &\quad \Rightarrow a[i-1] < x \leq a[i] \ \ \&\ \ 0 < i < |a|. \end{aligned}$$

Recapping the discussion there, a is a zero origin array, and assuming that a is sorted, nonempty, and has no duplicates, and assuming that $a[0] < x \leq a[|a| - 1]$, the i returned is in the range $1..|a| - 1$ and $a[i-1] < x \leq a[i]$. ($a[0]$ is used as a sentinel value.)

To prove the binary search predicate (2.4), it is generalized: 0 and $|a|$ become left and right endpoints l and r, and the predicate is proved by induction on $r - l$ (called n below). The resulting predicate is

$$\begin{aligned} &\textsf{All } n : \textsf{int} \,.\, \textsf{All } a : \textsf{array}, x, l, r : \textsf{int} \,.\, \textsf{Some } i : \textsf{int} \,. \\ &\qquad F_2(n, a, x, l, r, i) \end{aligned} \tag{2.5}$$

where $F_2(n, a, x, l, r, i)$ is

$$(searchable(a) = 1 \ \& \ a[l] < x \leq a[r-1] \ \& \ 0 \leq l < r \leq |a|$$
$$\& \ r - l \leq n)$$
$$\Rightarrow a[i-1] < x \leq a[i] \ \& \ l < i < r.$$

The proof of (2.4) from (2.5) is shown in figure 2.1. (The line numbers there are used for reference here in the text.) The first table of figure 2.1 shows the proof structure: in lines 1 and 2, formula (2.5) is added to the hypotheses, in lines 4 and 5, $|a|$ is used for n, and 0 and $|a|$ are used for l and r. The resulting i is used as the i to be returned in lines 6 and 7. The rest of the proof is omitted, but easy. The relevant part of the goal of line 8 is $6 \gg 7$. Substituting from the formula table of figure 2.1, this is

$$(searchable(a) = 1 \ \& \ a[0] < x \leq a[|a|-1] \ \& \ 0 \leq 0 < |a|$$
$$\& \ |a| - 0 \leq |a|)$$
$$\Rightarrow a[i-1] < x \leq a[i] \ \& \ 0 < i < |a|$$
$$\gg searchable(a) = 1 \ \& \ a[0] < x \leq a[|a|-1]$$
$$\Rightarrow a[i-1] < x \leq a[i] \ \& \ 0 < i < |a|.$$

This is provable because $searchable = 1$ implies that $0 < |a|$.

The extracted code for this theorem[13] is shown in figure 2.2. In it, σ_8 refers to the omitted subproof of line 8. The code for σ_8 is irrelevant,

[13]The figure shows the second-order justification transformer extracted from the theorem. The phrase "extracted code for a theorem" will sometimes be used this way.

Line	Depth	Env	Sequent	Rule
1	0		$\gg 1$	seq 2
2	1		$\gg 2$	theorem *binsearch_lemma*
3	1		$2 \gg 1$	intro
4	2	a, x	$2 \gg 3$	elim 2, $|a|$
5	3	a, x	$2, 4 \gg 3$	elim 4, $a, x, 0, |a|$
6	4	a, x	$2, 4, 5 \gg 3$	elim 5, i
7	5	a, x, i	$2, 4, 6 \gg 3$	intro i
8	6	a, x, i	$2, 4, 6 \gg 7$	*(omitted)*

Above, Depth gives the depth of the given line within the proof.

	Formulas
1	All a : array, x : int . Some i : int . $F_1(a, x, i)$
2	All n : int . All a : array, x, l, r : int . Some i : int . $F_2(n, a, x, l, r, i)$
3	Some i : int . $F_1(a, x, i)$
4	All a : array, x, l, r : int . Some i : int . $F_2(|a|, a, x, l, r, i)$
5	Some i : int . $F_2(|a|, a, x, 0, |a|, i)$
6	$F_2(|a|, a, x, 0, |a|, i)$
7	$F_1(a, x, i)$

Figure 2.1: The proof of the binary search theorem.

Binsearch_code:

```
proc () : just .
    let J1 = Binsearch_lemma_code()
    in close (J1) . proc (a : array, x : int) : just .
           let J2 = J1(|a|);
               J3 = J2(a, x, 0, |a|);
               (i, J4) = J3
           in (i, X[σ8, J1, J2, J4])
           ni
    ni
```

Figure 2.2: Extracted code for the binary search theorem.

in a sense which will be made clear in the next chapter, when simple code optimizations are discussed.

The proof of (2.5) is shown in figures 2.3 and 2.4. The proof is by induction on n. Much of the proof (lines 2 through 13) deals with impossible cases: if n is less than 2, or $r - l$ is less than 2, then the antecedent

$$searchable(a) = 1 \;\&\; a[l] < x \leq a[r-1] \;\&\; 0 \leq l < r \leq |a| \;\&\; r - l \leq n$$

is contradictory. In such cases, the i returned is not important; the proof uses 0 for i.

Lines 14 through 26 are the heart of the proof. In lines 14 and 15, the segment $a[l+1..r-1]$ is exactly one element long, so $r - 1$ is used for i.

Line	Depth	Env	Sequent	Rule
1	0		$\gg 1$	ind 0
2	1	n	$2,3 \gg 4$	intro a, x, l, r
3	2	n, a, x, l, r	$2,3 \gg 5$	intro 0
4	3	n, a, x, l, r	$2,3 \gg 6$	*(omitted)*
5	1	n	$\gg 7$	intro a, x, l, r
6	2	n, a, x, l, r	$\gg 8$	intro 0
7	3	n, a, x, l, r	$\gg 9$	*(omitted)*
8	1	n	$10, 11 \gg 4$	intro a, x, l, r
9	2	n, a, x, l, r	$10, 11 \gg 5$	seq 12
10	3	n, a, x, l, r	$10, 11 \gg 12$	decision
11	3	n, a, x, l, r	$10, 11, 12 \gg 5$	elim 12
12	4	n, a, x, l, r	$10, 11, 13 \gg 5$	intro 0
13	5	n, a, x, l, r	$10, 11, 13 \gg 6$	*(omitted)*
14	4	n, a, x, l, r	$10, 11, 14 \gg 5$	intro $r - 1$
15	5	n, a, x, l, r	$10, 11, 14 \gg 16$	*(omitted)*
16	4	n, a, x, l, r	$10, 11, 15 \gg 5$	seq 17
17	5	n, a, x, l, r	$10, 11, 15 \gg 17$	decision
18	5	n, a, x, l, r	$10, 11, 15, 17 \gg 5$	elim 17
19	6	n, a, x, l, r	$10, 11, 15, 18 \gg 5$	elim $11, a, x, (l+r)/2, r$
20	7	n, a, x, l, r	$10, 11, 15, 18, 20 \gg 5$	elim $20, i$
21	8	n, a, x, l, r, i	$10, 11, 15, 18, 21 \gg 5$	intro i
22	9	n, a, x, l, r, i	$10, 11, 15, 18, 21 \gg 22$	*(omitted)*
23	6	n, a, x, l, r	$10, 11, 15, 19 \gg 5$	elim $11, a, x, l, (l+r)/2+1$
24	7	n, a, x, l, r	$10, 11, 15, 19, 23 \gg 5$	elim $23, i$
25	8	n, a, x, l, r, i	$10, 11, 15, 19, 24 \gg 5$	intro i
26	9	n, a, x, l, r, i	$10, 11, 15, 19, 24 \gg 22$	*(omitted)*

Figure 2.3: The proof of the binary search lemma.

	Formulas
1	All n : int . All a : array, x, l, r : int . Some i : int . $F_2(n, a, x, l, r, i)$
2	$n < 0$
3	All a : array, x, l, r : int . Some i : int . $F_2(n+1, a, x, l, r, i)$
4	All a : array, x, l, r : int . Some i : int . $F_2(n, a, x, l, r, i)$
5	Some i : int . $F_2(n, a, x, l, r, i)$
6	$F_2(n, a, x, l, r, 0)$
7	All a : array, x, l, r : int . Some i : int . $F_2(0, a, x, l, r, i)$
8	Some i : int . $F_2(0, a, x, l, r, i)$
9	$F_2(0, a, x, l, r, 0)$
10	$n > 0$
11	All a : array, x, l, r : int . Some i : int . $F_2(n-1, a, x, l, r, i)$
12	$r - l < 2 \mid r - l = 2 \mid r - l > 2$
13	$r - l < 2$
14	$r - l = 2$
15	$r - l > 2$
16	$F_2(n, a, x, l, r, r-1)$
17	$a[(l+r)/2] < x \mid a[(l+r)/2] \geq x$
18	$a[(l+r)/2] < x$
19	$a[(l+r)/2] \geq x$
20	Some i : int . $F_2(n-1, a, x, (l+r)/2, r, i)$
21	$F_2(n-1, a, x, (l+r)/2, r, i)$
22	$F_2(n, a, x, l, r, i)$
23	Some i : int . $F_2(n-1, a, x, l, (l+r)/2+1, i)$
24	$F_2(n-1, a, x, l, (l+r)/2+1, i)$

Figure 2.4: The formulas for the proof of the binary search lemma.

The relevant part of the incomplete leaf in line 15 is 14 $\gg$ 16, which is

$$r - l = 2 \gg F_2(n, a, x, l, r, r-1).$$

Substituting for formula F_2, this is

$$\begin{aligned} &r - l = 2 \gg \\ &\quad (searchable(a) = 1 \ \& \ a[l] < x \leq a[r-1] \\ &\qquad \& \ 0 \leq l < r \leq |a| \ \& \ r - l \leq n) \\ &\qquad \Rightarrow a[(r-1)-1] < x \leq a[r-1] \ \& \ l < r-1 < r. \end{aligned}$$

In lines 17 and 18, the midpoint $a[(l+r)/2]$ is compared to x.[14] In lines 19 through 22, $a[(l+r)/2]$ is less than x, so $(l+r)/2$ is used as the new left endpoint when the inductive hypothesis is applied, and the i that results is used for the i to be returned. The relevant part of the incomplete leaf in line 22 is 15, 18, 21 $\gg$ 22, which is

$$\begin{aligned} &r - l > 2, \ a[(l+r)/2] < x, \ F_2(n-1, a, x, (l+r)/2, r, i) \\ &\quad \gg F_2(n, a, x, l, r, i). \end{aligned}$$

Here, the F_2 hypothesis states that if $(l+r)/2..r$ is a reasonable part of a to search (that is, if $a[(l+r)/2] < x \leq a[r-1]$, and so on), then i gives the location of x ($a[i-1] \leq x \leq a[i]$), and i is between $(l+r)/2$ and r.

[14] Division in λPRL truncates toward zero, which is why this case split is not made until it is known that $r - l > 2$.

The conclusion states that if $l..r$ is a reasonable part of a to search, then i gives the location of x, and i is between l and r.

Lines 23 through 26 are similar except that $a[(l+r)/2]$ is greater than or equal to x, so $(l+r)/2+1$ is used as the right endpoint, not the left. ($(l+r)/2+1$ is used for r instead of $(l+r)/2$ because x is supposed to be less than or equal to $a[r-1]$, not $a[r]$.)

Figures 2.5 and 2.6 give the extracted code for the binary search lemma. In them, σ_i refers to the omitted subproof of line i, for various i. The extracted code is not efficient: if n is $|a|$ then $f(n)(a,x,0,n)$ takes $O(n)$ time, not $O(\log n)$ time.[15] This slowdown happens because the call $f(n-1)$ appears far too early; optimally, it should be done only if $r-l>2$, since that is the only time $f(n-1)$ is actually used. In addition, the code is large: the omitted parts of the proofs generate nontrivial amounts of extracted code. The next chapters address techniques for reducing the size and running time of extracted code.

[15] It is necessary to assume that the omitted subproofs generate extracted code that takes a constant time to execute. In practical terms, this means that none of the omitted subproofs should use inductive proofs, which is a reasonable assumption to make, even though it is false in the current implementation. The implementation's *decision* rule needs to be incorporated into the *arith* rule so that PRL will not need an inductive proof to know, for example, that $r-(l+r)/2 = r-l-(r-l)/2$.

Binsearch_lemma_code:

```
        proc () : just . close () . proc (n : int) : just . f(n)
where f is
    proc (n : int) : just .
      if n < 0 →
        let J1 = trivial; J2 = f(n + 1)
        in close (n, J1, J2) . proc (a : array, x, l, r : int) : just .
              (0, X[σ4, J1, J2])
        ni
      [] n = 0 →
        close (n) . proc (a : array, x, l, r : int) : just . (0, X[σ7])
      [] n > 0 →
        let J3 = trivial; J4 = f(n − 1)
        in close (n, J3, J4) . proc (a : array, x, l, r : int) : just .
              let J5 =
                  if r − l < 2 → (1, trivial)
                  [] r − l = 2 → (2, trivial)
                  [] r − l > 2 → (3, trivial)
                  fi;
                  (J6, J7) = J5
              in if J6 = 1 → (0, X[σ13, J3, J4, J7])
                  [] J6 = 2 → (r − 1, X[σ15, J3, J4, J7])
                  [] J6 = 3 →
                     Code for nontrivial case. (See figure 2.7).
                  fi
              ni
        fi
```

Figure 2.5: Extracted code for the trivial cases of the binary search lemma.

$$
\begin{array}{l}
\mathsf{let}\ J_8 = \\
\quad \mathsf{if}\ a[(l+r)/2] < x \rightarrow (1, \mathsf{trivial}) \\
\quad \Box\ a[(l+r)/2] \geq x \rightarrow (2, \mathsf{trivial}) \\
\quad \mathsf{fi}; \\
\quad (J_9, J_{10}) = J_8 \\
\mathsf{in} \\
\quad \mathsf{if}\ J_9 = 1 \rightarrow \\
\qquad \mathsf{let}\ J_{11} = J_4(a, x, (l+r)/2, r); \\
\qquad\qquad (i, J_{12}) = J_{11} \\
\qquad \mathsf{in}\ (i, \mathsf{X}[\![\sigma_{22}, J_3, J_4, J_7, J_{10}, J_{12}]\!]) \\
\qquad \mathsf{ni} \\
\quad \Box\ J_9 = 2 \rightarrow \\
\qquad \mathsf{let}\ J_{13} = J_4(a, x, l, (l+r)/2 + 1); \\
\qquad\qquad (i, J_{14}) = J_{13} \\
\qquad \mathsf{in}\ (i, \mathsf{X}[\![\sigma_{26}, J_3, J_4, J_7, J_{10}, J_{14}]\!]) \\
\qquad \mathsf{ni} \\
\quad \mathsf{fi} \\
\mathsf{ni}
\end{array}
$$

Figure 2.6: Extracted code for the nontrivial case of the binary search lemma.

Chapter 3

Simple optimizations

This chapter begins the study of methods for improving extracted code. Although in many instances the code is already good, there is still room for improvement.

The first section of this chapter lists some simple code optimizations, techniques that are easy to apply or guaranteed to be useful. (Preferably both.) Verifying a conjecture by Bates [Bat], these optimizations do much to improve the code, reducing both its size and running time. They are, however, no panacea: further-reaching optimizations become available as soon as the nature of justification is taken into account.

In the second section, justifications are changed slightly to enlarge the set of formulas with trivial justifications. The change is negligible semantically, but carrying it through the proof system yields an extractor that produces smaller and faster extracted code in (in practice) less time. Pragmatically, this is the most important simple optimization one can apply.

More complicated techniques, and larger, deeper changes to the notions of justification and extraction will be seen in the next chapter.

3.1 Code optimizations

Bates pointed out in his dissertation [Bat] that θPRL extracted code improved greatly when he applied traditional code optimization techniques to it.[1] λPRL code also improves, but not as much, since it is already shorter.

Definition 3.1 The *size* of a proof π is the number of proof nodes (sequents) it contains. Its *area* is its size plus the number of hypotheses in each node, summed over all nodes, plus its size.

Let us assume that quantified formulas only bind a constant number of identifiers, not a number proportional to the proof's size or area. Given this, both the θPRL and λPRL extraction functions take time linear in the area of π. θPRL extracted code is linear in the area of π, and λPRL

[1] Recall θPRL is the name used here for the logic presented by Bates in his dissertation.

code is linear in the size of π. (These can easily be proved by induction on the structure of proofs, using the definition of extraction.)[2]

λPRL code is shorter than θPRL code because of its style of extraction: in λPRL the extraction function produces an expression whose value justifies the conclusion of a goal, but in θPRL the extractor produces a procedure constant that denotes a justification transformer for the goal. That is, it takes a proof π and produces a procedure

$$\mathsf{proc}\,(J\!:\!\mathsf{just})\!:\!\mathsf{just}\,.\,e$$

where e is essentially λPRL's $\mathbf{X}[\pi, J]$. If the code for π is to be used by the parent of π, then in λPRL some identifiers $K = K_1, \ldots, K_n$ are chosen and $\mathbf{X}[\pi, K]$ is used. In θPRL, the parent of π uses a program P

$$(\mathsf{proc}\,(J\!:\!\mathsf{just})\!:\!\mathsf{just}\,.\,e)(K).$$

Since e may contain assignment statements in θPRL, P cannot simply be β-reduced to $e[K/J]$; more involved transformations are required. Since the β-reduction is already built into λPRL extraction, it does not need

[2] The number of identifiers in close expressions is ignored here. Procedures for all-justifications and implication-justifications are effectively statically scoped, so if the language being used to implement extracted code uses static scoping, then close expressions are basically unnecessary. If only dynamic binding is available, then in practice close expressions are restricted so that they only include the nonlocal identifiers of the procedure. In this case, the close expressions can be calculated by a linear pass over the extracted code, so the relation between λPRL and θPRL extracted code remains the same modulo this linear pass.

to be done as an optimization step, so naive λ PRL code is significantly optimized relative to θ PRL code.

For example, consider the proof (from chapter 2) of $F \,\&\, G \gg G \,\&\, F$:

$F \,\&\, G \gg G \,\&\, F$ by elim 1
• $F, G \gg G \,\&\, F$ by intro
•• $F, G \gg G$ by hyp 2
•• $F, G \gg F$ by hyp 1.

Its justification transforming extracted code is

$$\text{proc}\ (J : \text{just}) : \text{just}\,.\ \text{let}\ (K, L) = J\ \text{in}\ (L, K)\ \text{ni}.$$

If the proof had been extracted in a θ PRL style, the code would have been

```
proc (J1 : just) : just .
    let (J2, J3) = J1
    in (proc (J4, J5 : just) : just .
            ((proc (J6, J7 : just) : just . J7)(J4, J5)
            ,(proc (J8, J9 : just) : just . J8)(J4, J5)
            )
       )(J2, J3)
    ni.
```


Although λ PRL code is much shorter and faster than θ PRL code, there is still much that can be done to further improve it. The focus in this section is source-to-source transformation of the code, using traditional code optimization techniques. Not only are such transformations fruitful, they also make the extracted code easier to read. Readability is not crucial at the moment, since extracted code is not shown to users in the current implementation of λ PRL, but if future PRL systems display the extracted code to users, users will find it easier to read simplified code.

Here is a list of simple optimizations. For simplicity, assume that α-renaming has been done so that no identifier bound by a let or proc is rebound. Below, "$\Longrightarrow$" is written for "may be replaced by."

- *Constant folding.*

$$\text{let}\ x = t\ \text{in}\ e\ \text{ni} \Longrightarrow e\,[t/x]$$

if t is a constant term.

- *Inline procedure expansion.*

$$(\text{proc}\ (x_1 : T_1, \ldots, x_n : T_n) : T\,.\,e)(e_1, \ldots, e_n)$$
$$\Longrightarrow \text{let}\ x_1 = e_1; \cdots; x_n = e_n\ \text{in}\ e\ \text{ni}.$$

- *Unused binding removal.*

$$\text{let}\ x = e\ \text{in}\ e_2\ \text{ni} \Longrightarrow e_2$$

if e_2 contains no occurence of x.

- *Collapsing binding chains.*

$$\text{let } x = e_1 \text{ in} \ldots \text{let } y = x \text{ in } e_2 \text{ ni} \ldots \text{ni}$$
$$\Longrightarrow \text{let } x = e_1 \text{ in} \ldots e_2\,[x/y] \ldots \text{ni.}$$

- *Substitution for identifiers used only once.*

$$\text{let } x = e_1 \text{ in } e_2 \text{ ni} \Longrightarrow e_2\,[e_1/x]$$

if e_2 has only one occurrence of x. The restriction prevents $e_2\,[e_1/x]$ from being larger than the let expression. Recursive extracted code is generated by creating a separate procedure for each inductive proof, so it is not possible to produce slower code by replacing x by e_1 inside the body of a recursive procedure called by e_2. (Note that the let expression might itself be part of a recursive procedure called by e_2. Even in this case, the resulting code is no slower than the original code.) If extracted code is extended to include loops, then this optimization should not be done if the occurrence of x is inside a loop body.

- *Let decoupling.*

$$\text{let } (x_1, \ldots, x_n) = (e_1, \ldots, e_n) \text{ in } e \text{ ni}$$
$$\Longrightarrow \text{let } x_1 = e_1; \cdots; x_n = e_n \text{ in } e \text{ ni.}$$

Recall that enough α-renaming has been assumed to know that no x_i occurs in $e_{i+1}, \ldots, e_n$.

- *Closure minimization.*

$$\text{close } () \,.\, e \Longrightarrow e$$
$$\text{close } (\mathcal{V}) \,.\, e \Longrightarrow \text{close } (\mathcal{V}') \,.\, e$$

where $\mathcal{V}'$ is the subset of $\mathcal{V}$ of identifiers used by e.

- *Dead code removal.* Guarded commands $b_i \rightarrow g_i$ where b_i is false can be removed from if statements.

- *Guard combination.* If an if expression has two guarded commands with the same command, they can be combined. For example, $b_i \rightarrow g$ and $b_j \rightarrow g$ would become $b_i \,]\, b_j \rightarrow g$.

- *If statement removal.*

$$\text{if } b \rightarrow g \text{ fi} \Longrightarrow g.$$

This works because at extracted code runtime at least one of the guards of an if must be true. This can be seen by looking at the only two cases where if expressions are generated: the induction rules, the *decision* rule (whose validity guarantees that one of the guards will succeed), and the *or-elim* rule (where the eliminated formula's tag must specify one of the disjuncts).

For an example of the use of some of these optimizations, take the code for the binary search theorem from figure 2.2. Unoptimized, it is

```
proc () : just .
  let J1 = Binsearch_lemma_code()
  in close (J1) . proc (a : array, x : int) : just .
      let J2 = J1(|a|);
        J3 = J2(a, x, 0, |a|);
        (i, J4) = J3
      in (i, X[σ8, J1, J2, J4])
      ni
  ni.
```

The result of optimization depends on which J_i are used by $\mathsf{X}[\![\sigma_8, J_1, J_2, J_4]\!]$. As discussed in chapter 2, the only relevant hypothesis of σ_8 is the one justified by J_4, so let us assume that J_4 is the only identifier used by the code for σ_8. In this case, the binary search code optimizes to

```
proc () : just . proc (a : array, x : int) : just .
  let (i, J4) = Binsearch_lemma_code()(|a|)(a, x, 0, |a|) in (i, ...) ni
```

where the ellipses stand for the result of optimizing $\mathsf{X}[\![\sigma_8, J_1, J_2, J_4]\!]$.

More complicated optimizations are also available. These techniques all use more global context.

- *Downward movement of bindings*. Here, the idea is to move **let** bindings down into the code through **if** statements. Consider

```
let x = e1
in ...if b1 → g1 □ ··· □ bn → gn fi...
ni.
```

If x does not occur in the ellipsed code or in the b_i and only occurs in one of the g_i, say g_1, then

```
let x = e1
in ...if b1 → g1 □ ··· □ bn → gn fi...
ni
⟹
...
if b1 → let x = e1 in g1 ni
□ b2 → g2
□ ···
□ bn → gn
fi
...
```

- *Upward code motion.* If e_1 is not an identifier and occurs in e_2 more than once, then

$$e_2 \Longrightarrow \text{let } x = e_1 \text{ in } e_2\,[x/e_1]\text{ ni}$$

where x is a new identifier. if necessary, the bound identifiers of e_2 should be renamed so that no free identifier of e_1 is captured.[3]

This optimization should not be done if the occurrences of e_1 fall in some but not all branches of an if embedded in e_2. For example, if e does not contain an occurrence of e_1, then

$$\text{if } b_1 \to e_1 \;\square\; b_2 \to e_1 \;\square\; b_3 \to e \text{ fi}$$

should *not* be replaced by

$$\begin{array}{l}\text{let } x = e_1\\ \text{in if } b_1 \to x \;\square\; b_2 \to x \;\square\; b_3 \to e \text{ fi}\\ \text{ni.}\end{array}$$

If this replacement were done, then e_1 would be calculated unnecessarily if b_3 were true.

- *Decision unchaining.* This optimization is intended to be used in *case splits*, which are combinations of the *decision* and *or-elim* rules. This

[3] A simple way to avoid this kind of problem is to rename bound identifiers before optimization so that no two binding operators bind the same identifier. A let x in one part of the code would force the renaming of every other let x in the code.

optimization requires a small change to the extractor. Consider the proof

$$\begin{array}{l}H \gg F \quad \text{by seq } F_1 \mid \cdots \mid F_m\\ \bullet\; H \gg F_1 \mid \cdots \mid F_m \quad \text{by decision}\\ \bullet\; H, F_1 \mid \cdots \mid F_m \gg F \quad \text{by elim } n\\ \bullet\bullet\; H, F_1 \gg F\\ \dots\\ \bullet\bullet\; H, F_m \gg F.\end{array}$$

(An m-ary version of *or-elim* is being used here, in contrast to the binary version defined in chapter 2.)

The unaltered, unoptimized extracted code for this proof is

$$\begin{array}{l}\text{let } J = \text{if} \ldots \;\square\; F_i \to \langle i, \text{trivial}\rangle \;\square\; \ldots \text{fi}; (K, L) = J\\ \text{in if} \ldots \;\square\; K = i \to P_i \;\square\; \ldots \text{fi}\\ \text{ni}\end{array}$$

where each P_i is the code for $H, F_i \gg F$. Let $decide(F_1, \ldots, F_m)$ be a new construct of extracted code, short for if $F_1 \to 1 \;\square\; \ldots \;\square\; F_m \to m$ fi, and make the extracted code for the *decision* rule

$$X[\![\pi_{decision}, J]\!] = (decide(F_1, \ldots, F_m), \text{trivial}).$$

Decision unchaining has the form

$$\begin{array}{l} \mathsf{let}\ J = (\mathit{decide}(F_1,\ldots,F_n),\mathsf{trivial}); (K,L) = J \\ \mathsf{in}\ \ \ldots\mathsf{if}\ldots \,\square\, K = i \rightarrow g_i \,\square \ldots \mathsf{fi}\ldots \\ \mathsf{ni} \\ \Longrightarrow \\ \mathsf{let}\ L = \mathsf{trivial} \\ \mathsf{in}\ \ \ldots\mathsf{if}\ldots \,\square\, F_i \rightarrow g_i \,\square \ldots \mathsf{fi}\ldots \\ \mathsf{ni}. \end{array}$$

- *Inductive procedure optimizations.* The induction rules generate code of the form proc $(x, \mathcal{V}) . f(x, \mathcal{V})$ where $\mathcal{V}$ is the set of closure identifiers and f is bound to proc $(x, \mathcal{V}) . e$. ($\mathcal{V}$ is J plus the identifiers of $\mathcal{E}$; the body e depends on the rule.)

  Let $\mathcal{V}'$ be the identifiers (not including x) that appear free in e; f can be bound to proc $(x, \mathcal{V}') . e$ and the induction rule's extracted code can be changed to proc $(x) . f(x, \mathcal{V}')$. (If $\mathcal{V}'$ is empty, the rule's extracted code can simply be f.)

When the optimizations above are included in the list of optimizations, more code can be optimized cleanly. For example, consider the term induction scheme shown in figures 3.1 and 3.2. Ideally, it might have the

form

$$\begin{array}{l} \gg \mathsf{All}\ n{:}\mathsf{int} . \mathsf{All}\ x{:}\mathsf{int} . (0 \le x \le n \Rightarrow F) \quad \mathsf{by\ ind}\ t \\ \bullet\ x = 0 \gg F \\ \bullet\ 0 < x \gg 0 \le t < x \\ \bullet\ 0 < x \le n,\ F[t/x] \gg F. \end{array}$$

Above, F is a formula with no free occurrences of n and t is a term in x with no free occurrences of n. For simplicity, no list of extra hypotheses $\mathcal{H}$ is included above, though it would be easy to add.

Incomplete proofs like the one in the figures are useful because they can be used as derived rules: the goal of the rule is the goal of the proof, and the subgoals of the rule are the incomplete leaves of the proof. (In the implementation, this is done using LCF tactics [CKB,Pau].) The user must give a name to the new rule, and the proof editor must avoid showing the text of the proof that makes up the rule. Once these are done, the rule can be used like any other.[4]

[4] Unfortunately, the rule derived from figures 3.1 and 3.2 is not ideal. The incomplete leaves of the proof have extra hypotheses, and the goal $x = 0 \gg F$ is repeated. The first problem would be solvable if a new rule were defined that allowed users to throw away arbitrary hypotheses. The second problem can be overcome by using the sequence rule, but it is not done here for simplicity's sake.

Line	Depth	Sequent	Rule
1	0	$\gg 1$	ind 0
2	1	$2, 3 \gg 4$	intro x
3	2	$2, 3 \gg 5$	intro
4	3	$2, 3, 6 \gg 7$	seq 8
5	4	$2, 3, 6 \gg 8$	decision
6	4	$2, 3, 6, 8 \gg 7$	elim 8
7	1	$\gg 9$	intro x
8	2	$\gg 10$	intro
9	3	$11 \gg 7$	seq 12
10	4	$11 \gg 12$	decision
11	4	$11, 12 \gg 7$	*(omitted)*
12	1	$13, 14 \gg 4$	intro x
13	2	$13, 14 \gg 5$	intro
14	3	$13, 14, 6 \gg 7$	seq 15
15	4	$13, 14, 6 \gg 15$	decision
16	4	$13, 14, 6, 15 \gg 7$	elim 15
17	5	$13, 14, 6, 12 \gg 7$	*(omitted)*
18	5	$13, 14, 6, 16 \gg 7$	elim 14, t
19	6	$13, 14, 6, 16, 17 \gg 7$	elim 17
20	7	$13, 14, 6, 16 \gg 18$	seq 20
21	8	$13, 14, 6, 16 \gg 20$	*(omitted)*
22	8	$13, 14, 6, 16, 20 \gg 18$	decision
23	7	$13, 14, 6, 16, 18, 19 \gg 7$	*(omitted)*

Figure 3.1: The term induction proof scheme.

	Formulas
1	All n : int . All x : int . $(0 \leq x \leq n \Rightarrow F)$
2	$n < 0$
3	All x : int . $(0 \leq x \leq n + 1 \Rightarrow F)$
4	All x : int . $(0 \leq x \leq n \Rightarrow F)$
5	$0 \leq x \leq n \Rightarrow F$
6	$0 \leq x \leq n$
7	F
8	false
9	All x : int . $(0 \leq x \leq 0 \Rightarrow F)$
10	$0 \leq x \leq 0 \Rightarrow F$
11	$0 \leq x \leq 0$
12	$x = 0$
13	$n > 0$
14	All x : int . $(0 \leq x \leq n - 1 \Rightarrow F)$
15	$x = 0 \,[\, x > 0$
16	$x > 0$
17	$0 \leq t \leq n - 1 \Rightarrow F[t/x]$
18	$0 \leq t \leq n - 1$
19	$F[t/x]$
20	$0 \leq t < x$

Figure 3.2: The formulas for the term induction proof scheme.

As an aside, the term induction scheme is a slight generalization of the following useful proof scheme:

$$\gg \text{All } x\!:\!\text{int} \,.\, 0 \le x \Rightarrow F \quad \text{by cov-ind } t$$

- $x = 0 \gg F$
- $0 < x \gg 0 \le t < x$
- $0 < x,\ F[t/x] \gg F.$

This scheme can be obtained from term induction by using n for x.

The term induction proof is straightforward. The main goal

$$\text{All } n\!:\!\text{int} \,.\, \text{All } x\!:\!\text{int} \,.\, (0 \le x \le n \Rightarrow F)$$

is proved by induction on n. If $n < 0$, the proof is trivial. If n is 0 then x is 0, leading to the goal $x = 0 \gg F$. Similarly, if $n > 0$ but x is 0, $x = 0 \gg F$ is used again. Finally, if $n > 0$ and $x > 0$, then the term t is used with the inductive hypothesis. This requires that $0 \le t < x$ be proved but leads to the goal $F[t/x] \gg F$.

The unoptimized extracted code for this proof is shown in figure 3.3; the optimized code is shown in figure 3.4. The optimized code was obtained by applying the following transformations, in order: decision unchaining, substitution for identifiers used only once, unused binding removal, and closure minimization.

The code for the term induction proof is close () . proc (n : int) : just . $f(n)$ where f is a new name, bound to

```
proc (n : int) : just .
  if n < 0 →
    let J1 = trivial; J2 = f(n + 1)
    in close (n, J1, J2) . proc (x : int) : just .
         close (n, x, J1, J2) . proc (J3 : just) : just .
           let J4 = trivial in trivial ni
    ni
  [] n = 0 →
    close (n) . proc (x : int) : just .
      close (n, x) . proc (J5 : just) : just .
        let J6 = trivial in X[σ11, J5, J6] ni
  [] n > 0 →
    let J7 = trivial; J8 = f(n − 1)
    in close (n, J7, J8) . proc (x : int) : just .
         close (n, x, J7, J8) . proc (J9 : just) : just .
           let J10 = (decide(x = 0, x > 0), trivial);
               (J11, J12) = J10
           in if J10 = 1 → X[σ17, J7, J8, J9, J12]
              [] J10 = 2 →
                let J13 = J8(t);
                    J16 =
                      let J15 = X[σ21, J7, J8, J9, J12]
                      in trivial
                      ni;
                    J16 = J13(J14)
                in X[σ23, J7, J8, J9, J12, J13, J14, J16]
                ni
              fi
    ni          ni
  fi
```

Figure 3.3: The unoptimized extracted code for term induction.

The code for the term induction proof is f, where f is bound to

```
proc (n : int) : just .
   if n < 0 →
      proc (x : int) : just . proc (J3 : just) : just . trivial
   [] n = 0 →
      proc (x : int) : just . proc (J5 : just) : just . X[σ11, J5, trivial]
   [] n > 0 →
      proc (x : int) : just . close (x) . proc (J9 : just) : just .
         if x = 0 → X[σ17, trivial, (J8), J9, trivial]
         [] x > 0 →
            let J16 = f(n − 1)(t)(trivial)
            in  X[σ23, trivial, (J8), J9, trivial, (J13), (J14), J16]
            ni
         fi
   fi
```

Above, calls to X refer to the optimized extracted code for the indicated subproofs. Identifiers replaced by trivial are shown as trivial arguments to X. Identifiers deleted from the code, like J_8, are shown as parenthesized arguments to X. J_{14} is assumed to be used in the code for the proof of σ_{21}. J_8 and J_{13} are assumed not to be used.

Figure 3.4: The optimized extracted code for term induction.

The most important difference between the optimized and unoptimized code is the movement of the recursive call $f(n-1)$. In the unoptimized code, f is called whenever $n > 0$, even if $x = 0$. To see how this can be a problem, consider what happens if n equals x originally and t always yields a value less than $x - 1$. As recursive calls are made, x approaches 0 faster than n does. Eventually, x becomes 0 but n is some $n' > 0$, so n' (essentially useless) recursive calls must be made before the base case $n = 0$ is reached. These calls are made even though $x = 0$ is tested, since the call comes before the test.

In the optimized code, $f(n-1)$ is called only after it is known that $x > 0$. If $x = 0$ the base case code is called without making the n' recursive calls, so the optimized code is faster. For example, if t is $x/2$ (assuming integer division with truncation) and x and n start out at 2^k, the optimized code makes k recursive calls, stopping when $x = 1/2 = 0$; the unoptimized code makes 2^k recursive calls.

This optimized term induction code allows course-of-values induction to be used to express sublinear algorithms like binary search and other algorithms with similar recursive call structures. Without this optimization, course-of-values inductive proofs do not generate extracted code that runs in the right time bounds.

It should be noted that the problem of calculating the inductive justification too early is a result of the way extraction was defined for the induction rules. To avoid the problem, one could redefine extraction so that, for example, the calculation is done whenever the inductive hypothesis is eliminated. This would lead, however, to the dual of the original problem: if the inductive hypothesis were eliminated more than once, the calculation would have to be redone, possibly at great expense. The solution to this new problem would be to use upward code motion.

3.2 Singleton formulas

This section is concerned with *singleton justification removal*. Singleton formulas (defined below) have the property that their justifications, when they exist, have no interesting structure, so code that produces justifications for such formulas can be thrown out, and code that uses these justifications can be simplified. This optimization is important in practice: it reduces the size of the extracted code greatly, and can also reduce its running time.

A surprising feature of naive extracted code is how much of it is unnecessary: much of it is unused, and much that *is* used is used only to calculate unused values. For example, code that calculates a justification for an atomic formula is unnecessary because it cannot have any computational effect: an atomic formula has only one justification, *triv*, so it is

the only possible value for such a piece of code. Similarly, if G is atomic, then code that calculates a justification for $F \Rightarrow G$ is unnecessary: it is a procedure that takes a justification for F and produces the useless result *triv*. A large class of these *singleton* formulas is defined below.

An easy local optimization is to replace these useless pieces of code by appropriate constants. For example, if expression e justifies a singleton formula, replace it with the constant trivial. If e justifies $F \Rightarrow S$, where S is a singleton, replace it with proc (J : just) : just . trivial.

On the other hand, one can do better by using the following global optimization: if a formula has only one justification, then don't bother creating or using the justification. This not only eliminates the code to calculate such justifications, it also removes the need to have justification-valued identifiers for such formulas. In particular, it becomes unnecessary to pass their justifications as arguments to implication justifications or to the justification transformers for lemmas.

This global optimization of the code cannot be fully achieved with dead code analysis techniques [Ken] because they cannot detect the uselessness of theorems.[5] For example, consider a theorem whose conclusion is atomic. Although the extracted code for this theorem will never be computationally useful, dead code analysis can never discover this; it can discover that every

[5] Recall that a PRL theorem is a named proof.

current use of the theorem is useless, but it has no way of knowing that every future use will also be useless.

It might seem fairly harmless not to use this optimization: consider a theorem *Th* whose goal is $[\mathcal{E}] \gg F$, where F is atomic — doesn't any proof of *Th* optimize to proc () : just . trivial? Unfortunately, unless inter-procedural analysis is being done, the answer is no. Let Th' be a proof of $[\mathcal{E}] \gg G \Rightarrow F$, and consider the following proof π of *Th*:

$[\mathcal{E}] \gg F$ by seq $G \Rightarrow F$

• $[\mathcal{E}] \gg G \Rightarrow F$ by thm Th'

• $[\mathcal{E}]\ G \Rightarrow F \gg F$ by elim 1

•• $[\mathcal{E}] \gg G$ by ...

•• $[\mathcal{E}]\ G,\ F \gg F$ by hyp 2

The best optimization of $\mathbf{X}[\![\pi]\!]$ yields proc () : just . Th'_*code*()(trivial). More generally, the best possible code depends on the structure of the goal of *Th* and its lemmas.

It might still seem that introducing a new global optimization is unnecessary. Dead code analysis is well understood and useful, so it might as well be included in the optimizer anyway, no? True, it should be included, but this new optimization should be included also because it is inexpensive and *very* productive. An optimizer willing to expend resources to improve code should use this new optimization first and follow it up with dead code

⟨*single*⟩::=true | false | ⟨*atomic*⟩ | ⟨*single*⟩ & ⟨*single*⟩
| ¬⟨*formula*⟩ | ⟨*formula*⟩ ⇒ ⟨*single*⟩
| All ⟨*typed ids*⟩ . ⟨*single*⟩

⟨*nonsingle*⟩::= Some ⟨*typed ids*⟩ . ⟨*formula*⟩
| All ⟨*typed ids*⟩ . ⟨*nonsingle*⟩
| ⟨*formula*⟩ & ⟨*nonsingle*⟩ | ⟨*nonsingle*⟩ & ⟨*formula*⟩
| ⟨*formula*⟩ "|" ⟨*formula*⟩ | ⟨*formula*⟩ ⇒ ⟨*nonsingle*⟩

Figure 3.5: Grammars for singleton and non-singleton formulas.

analysis; a cheaper optimizer should simply skip the dead code analysis. More will be said on this at the end of the section.

Definition 3.2 Grammars for *singleton* and *non-singleton* formulas are given in figure 3.5.

Singleton formulas have at most one justification. Non-singleton formulas may have more. For example, **true** is a singleton formula with only one justification, and **true | true** is a non-singleton formula with two justifications. On the other hand, **false** is a singleton formula with no justifications, and **false | false** is a non-singleton formula with no justifications.

Once singleton formulas are defined, the notions of justification and extraction can be modified slightly to improve the extracted code.

Definition 3.3 The notion "Justification γ *justifies* (or *validates*) formula F in environment *env*" is defined by induction on the structure of F. The notation $\gamma \Vdash F, env$ will continue to be used. Below, v is a value of type T, S and S' are singleton formulas, and N and N' are non-singleton formulas.

- ***F is a singleton.***

  There is only one case: $triv \Vdash F, env$ iff F is true in *env*.

- ***F is a conjunction.***

  There are two non-singleton cases. First, $(\gamma_1, \gamma_2) \Vdash N_1 \,\&\, N_2, env$ iff each γ_i justifies N_i in *env*. Second, $\gamma \Vdash N \,\&\, S, env$ iff $\gamma \Vdash N, env$ and $triv \Vdash S, env$. ($\gamma \Vdash S \,\&\, N, env$ is defined similarly.)

- ***F is a disjunction.***

  Again, there are two cases. First, if at least one of F_1 and F_2 is non-singleton, then $(r, \gamma) \Vdash F_1 \mid F_2, env$ iff r is 1 or 2 and $\gamma \Vdash F_r, env$. Second, $\gamma \Vdash S_1 \mid S_2, env$ if γ is 1 or 2 and $triv \Vdash S_\gamma, env$.

- ***F is an implication.***

  Here, the two cases depend on the kind of antecedent. First, $\gamma \Vdash N_1 \Rightarrow N_2, env$ iff it maps justifications of N_1 to justifications of N_2. (That is, if $\gamma' \Vdash N_1, env$, then $\gamma(\gamma') \Vdash N_2, env$.) Second, $\gamma \Vdash S \Rightarrow N, env$ if $triv \Vdash S, env$ implies $\gamma \Vdash N, env$.

- ***F is an existential.***

  Once more, there are two cases. First, $\langle \mathsf{v}, \gamma \rangle \Vdash$ Some $x{:}T\,.\,N, env$ iff $\gamma \Vdash N, env[x \leftarrow \mathsf{v}]$. Second, $\mathsf{v} \Vdash$ Some $x{:}T\,.\,S, env$ iff $triv \Vdash S, env[x \leftarrow \mathsf{v}]$.

- ***F is a universal.***

  Here there is only one case: $\gamma \Vdash$ All $x{:}T\,.\,N, env$ iff γ is a function from T to justifications for N. (That is, iff for all v of type T, $\gamma(\mathsf{v}) \Vdash N, env[x \leftarrow \mathsf{v}]$.)

It is easy to verify that the above definition of justification still satisfies the justification validity theorem.

Theorem 3.1 *(Justification validity, for non-singleton justifications.)*

For all formulas F and environments *env*, $\models F, env$ iff there exists a justification γ such that $\gamma \Vdash F, env$.

Proof

By induction on the structure of F. The proof is slightly more complicated than the proof of the validity theorem in chapter 2, since there are more cases here. For example, if F is $F_1 \,\&\, F_2$, then there are four cases, depending on which F_i (if any) are singleton. $\square$

The extraction function, X, will now map proofs of *non-singleton* goals to programs: if a proof π has the goal $\mathcal{H} \gg F$ then X is defined on π only if

F is non-singleton. Further, if $J_1 \ldots J_n$ are the program variables passed to X, then the only J_i that will appear in the extracted code are those which justify non-singleton H_i. It would be reasonable to redefine X so that the only J_i required as parameters are those for these non-singleton H_i, but, for explanatory purposes, this will not be done. Instead, trivial will be substituted for J_i as a parameter of X whenever the hypothesis H_i is a singleton, and J_i will not be used. For example, if π is a proof of the goal

$$[\mathsf{int}\ b, c, d]\ \mathsf{Some}\ a : \mathsf{int} \,.\, a = b,\ c = d \gg N$$

then its extracted code is $\mathsf{X}[\![\pi, J_1, \mathsf{trivial}]\!]$.

When extraction is redefined this way, it becomes slightly more complicated, since some rules now have more cases. A list of the necessary changes is given in the appendix. They are straightforward.

For an example of a rule whose extraction definition becomes more complicated, consider the sequence rule: it now splits into two cases, depending on whether or not the sequence formula is singleton. Let π be the proof

$$\mathcal{H} \gg G \quad \text{by seq } F$$

- $\mathcal{H} \gg F$
- $\mathcal{H},\ F \gg G$

and let σ_1 and σ_2 be its two subproofs. If F is singleton, then $\mathsf{X}[\![\pi, J]\!]$ is $\mathsf{X}[\![\sigma_2, J, \mathsf{trivial}]\!]$; if F is non-singleton, it is

$$\mathsf{let}\ K = \mathsf{X}[\![\sigma_1, J]\!]\ \mathsf{in}\ \mathsf{X}[\![\sigma_2, J, K]\!]\ \mathsf{ni}.$$

In contrast, some rules become easier to extract from. For example, since justifications are not being built for singleton formulas, no *intro* rule for a singleton formula generates any extracted code, and with the exception of the *impl-elim* rule, no *elim* rule for a singleton formula generates code either. (The *impl-elim* rule can be an exception because when $N \Rightarrow S$ is eliminated, the antecedent N can be used in the rest of the proof, so code must be extracted to build the justification for it. Of course, no justification for S is necessary in any case.)

In addition, note that no code at all is extracted from the proof of a singleton goal. If the proof is small, then not much is gained, but quite often the proof is large, and then much *is* gained.

For an example of the use of singleton justification removal, consider the binary search lemma of the previous chapter. Its goal was

$$\mathsf{All}\ n : \mathsf{int} \,.\, \mathsf{All}\ a : \mathsf{array}, x : \mathsf{int} \,.\, \mathsf{Some}\ i : \mathsf{int} \,.\, F_2(n, a, x, l, r, i)$$

where $F_2(n, a, x, l, r, i)$ was

$$\begin{aligned}(&searchable(a) = 1 \ \&\ a[l] < x \le a[r-1] \ \&\ 0 \le l < r \le |a| \\ &\quad \&\ r - l \le n) \\ &\Rightarrow a[i-1] < x \le a[i] \ \&\ l < i < r\end{aligned} \tag{3.1}$$

and the extracted code for its proof was given in figures 2.5 and 2.6. The matrix of the conclusion is a singleton formula, so no code need be extracted to justify it. In fact, formula (3.1) is the implication of two singleton formulas, so even its elimination produces no extracted code.

When singleton formula removal is incorporated into the extractor, the omitted subproofs of the lemma generate no extracted code. Figure 3.6 shows the result of extracting code for the lemma, with singleton formulas removed. Optimized code for the lemma is shown in figure 3.7. The optimized code was obtained by using decision unchaining, substitution for identifiers used only once, unused binding removal, downward movement of bindings, upward code motion, guard combination, and then closure minimization.[6]

[6] Decision unchaining must be slightly modified when singleton formulas are removed:

$$\left.\begin{array}{l}\text{let } J = decide(F_1, \ldots, F_m); K = J \\ \text{in } \ldots\text{if} \ldots \,[]\, K = i \rightarrow P_i \,[]\, \ldots \text{fi} \ldots \\ \text{ni}\end{array}\right\} \Longrightarrow \ldots\text{if} \ldots \,[]\, F_i \rightarrow P_i \,[]\, \ldots \text{fi} \ldots.$$

Binsearch lemma_code:

proc () : just . close () . proc (n : int) : just . f(n)

where f is

```
proc (n:int):just.
   if n < 0 →
      let J1 = f(n+1)
      in close (n, J1) . proc (a:array, x, l, r:int) : just . 0
      ni
   [] n = 0 → close (n) . proc (a:array, x, l, r:int) : just . 0
   [] n > 0 →
      let J2 = f(n−1)
      in close (n, J2) . proc (a:array, x, l, r:int) : just .
           let J3 = decide(r − l < 2, r − l = 2, r − l > 2); J4 = J3
           in if J4 = 1 → 0
              [] J4 = 2 → r − 1
              [] J4 = 3 →
                 let J5 = decide(a[(l+r)/2] < x, a[(l+r)/2] ≤ x);
                     J6 = J5
                 in if J6 = 1 →
                       let J7 = J2(a, x, (l+r)/2, r); i = J7 in i ni
                    [] J6 = 2 →
                       let J8 = J2(a, x, l, (l+r)/2+1); i = J8 in i ni
                    fi
                 ni
              fi
      ni  ni
   fi
```

Figure 3.6: Extracted code for the binary search lemma, with singleton formulas removed.

The code for the binary search lemma is proc () . f'', where f'' is

```
proc (n : int) : just .
  if n < 0 | n = 0 → proc (a : array, x, l, r : int) : just . 0
  [] n > 0 →
    proc (a : array, x, l, r : int) : just .
      let r' = r − l
      in
        if r' < 2 → 0
        [] r' = 2 → r − 1
        [] r' > 2 →
          let J2 = f''(n − 1); l' = (l + r)/2; a' = a[l']
          in if a' < x → J2(a, x, l', r)
             [] a' ≥ x → J2(a, x, l, l' + 1)
             fi
          ni
        fi
      ni
  fi
```

Figure 3.7: Optimized extracted code for the binary search lemma.

It may not be obvious how much smaller the optimized code of figure 3.7 is than the unoptimized extracted code of figures 2.5 and 2.6. The unoptimized code is actually much larger than shown in the figures, since some of the omitted subproofs of the lemma's proof are relatively large. When singleton formulas are removed, none of the omitted subproofs generate extracted code, since all of them have singleton conclusions. As discussed in chapter 2, this code is assumed to run in constant time, but is still sizable. To emphasize this point, figures 3.8 and 3.9 and shows one of the subproofs omitted from figure 2.3, and figure 3.10 shows the code that would have been extracted from it if singleton formulas were not removed.

It is difficult to make a more general statement about the general utility of singleton justification removal, since it depends basically on the number and size of singleton goal subproofs, but in my own experience, this is an extremely useful optimization. An extractor that does singleton formula removal, substitution for identifiers only used once, unused binding removal, and closure minimization does a serviceable job of generating code: the extractor is simple and small, and (assuming that users put some thought into designing their specifications), generates efficient code under most circumstances.

A further advantage to singleton formula removal is its low cost. It takes time linear in the size of a formula to decide whether or not it is a singleton,

Line	Depth	Sequent	Rule
22.0	9	$10, 11, 15, 18, 21 \gg 22$	intro
22.1	10	$10, 11, 15, 18, 21, 25 \gg 26$	elim 25
22.2	12	$10, 11, 15, 18, 21, 27\text{–}33 \gg 26$	elim 21
22.3	13	$10, 11, 15, 18, 27\text{–}33 \gg 34$	intro
22.4	13	$10, 11, 15, 18, 27\text{–}33 \gg 27$	hyp 27
22.5	13	$10, 11, 15, 18, 27\text{–}33 \gg 18$	hyp 18
22.6	13	$10, 11, 15, 18, 27\text{–}33 \gg 29$	hyp 29
22.7	13	$10, 11, 15, 18, 27\text{–}33 \gg 35$	decision
22.8	13	$10, 11, 15, 18, 27\text{–}33 \gg 36$	decision
22.9	13	$10, 11, 15, 18, 27\text{–}33 \gg 32$	hyp 32
22.10	13	$10, 11, 15, 18, 27\text{–}33 \gg 37$	decision
22.11	12	$10, 11, 15, 18, 27\text{–}33, 34, 26 \gg 26$	hyp 26

The lines above fit between lines 22 and 23 of figure 2.3. All the sequents have the environment $[n, a, x, l, r]$. For simplicity, the *and-elim* of line 22.1 is shown as one n-ary elimination instead of a series of binary eliminations. (Similar for the *and-intro* of line 22.3.) In the current implementation, all the *decision* proofs above actually require inductive proofs. For example,

$$r - l \le n,\ r - l > 2 \gg r - (l + r)/2 \le n - 1$$

requires reasoning like

$$\begin{aligned} r - (l + r)/2 &= r - (l + l + r - l)/2 \\ &= r - (l + l)/2 - (r - l)/2 \\ &= r - l - (r - l)/2 \le r - l - 1 \le n - 1. \end{aligned}$$

Each step requires a *sequence* rule and a subproof involving a decision procedure or an inductive proof.

Figure 3.8: An omitted subproof of the binary search lemma.

	Formulas
10	$n > 0$
11	All a : array, x, l, r : int . Some i : int . $F_2(n-1, a, x, l, r, i)$
15	$r - l > 2$
18	$a[(l+r)/2] < x$
21	$F_2(n-1, a, x, (l+r)/2, r, i)$
22	$F_2(n, a, x, l, r, i)$
25	*searchable* $= 1$ & $a[l] < x \le a[r-1]$ & $0 \le l < r \le \lvert a \rvert$ & $r - l \le n$
26	$a[i-1] < x \le a[i]$ & $l < i < r$
27	*searchable*$(a) = 1$
28	$a[l] < x$
29	$x < a[r-1]$
30	$0 \le l$
31	$l < r$
32	$r \le \lvert a \rvert$
33	$r - l \le n$
34	*searchable* $= 1$ & $a[(l+r)/2] < x \le a[r-1]$ & $0 \le (l+r)/2 < r \le \lvert a \rvert$ & $r - (l+r)/2 \le n - 1$
35	$0 \le (l+r)/2$
36	$(l+r)/2 < r$
37	$r - (l+r)/2 \le n - 1$

The numbering above is continued from that of figure 2.4. Recall that $F_2(n, a, x, l, r, i)$ above is $25 \gg 26$.

Figure 3.9: The formulas for an omitted subproof of the binary search lemma.

$$
\begin{array}{l}
\mathsf{close}\ (n,a,x,l,r,i)\ .\ \mathsf{proc}\ (J_{15} : \mathsf{just}) : \mathsf{just}\ .\\
\quad \mathsf{let}\ (J_{16}, J_{17}, J_{18}, J_{19}, J_{20}, J_{21}, J_{22}) = J_{15};\\
\qquad J_{23} = (J_{16}, J_{10}, J_{18}, \mathsf{trivial}, \mathsf{trivial}, J_{21}, \mathsf{trivial});\\
\qquad J_{24} = J_{11}(J_{23})\\
\quad \mathsf{in}\ J_{24}\\
\quad \mathsf{ni}
\end{array}
$$

Figure 3.10: Optimized extracted code for an omitted subproof of the binary search lemma.

so if care is taken, then during the extraction of a theorem one spends time linear in the sizes of the hypotheses, conclusion, and sequence formulas looking for singletons. Since extraction time is reduced by throwing out useless subproofs, the net time saved depends on how much of the proof is thrown out. In practice, with this optimization, if the goal of a nontrivial $\forall\exists$ theorem has a singleton matrix, its extraction generally takes less than linear time: most of the proof is computationally irrelevant.

3.3 Extraction from incomplete proofs

The use of singleton formulas and code optimizations make it easy to extract useful code from incomplete proofs, if the appropriate proof style is used. Let π be the proof of a theorem Th. If $\mathcal{H} \gg S$ is a subgoal of π, where S is singleton, then no code is extracted for the proof of $\mathcal{H} \gg S$. If

all one is interested in is the the extracted code for Th then proofs of all singleton subgoals can be omitted from π. (Note, for example, that all the omitted subproofs of the binary search lemma were of singleton subgoals.) Thus, as far as extraction is concerned, proofs for the singleton subgoals might as well not exist; indeed they need never be supplied. Of course, if singleton goals are never proved then they might not be true, and if they are not true then the extracted code would be incorrect.

One might ask why PRL users might be interested in extracting code from incomplete proofs. After all, if the proof is incomplete, then the code might be incorrect, and wasn't the whole point of PRL to produce correct code? The answer is yes, the point was to produce correct code, but in practice users want a balance between the risk that an unproved subgoal is false and the tedium of proving an obviously true subgoal.

Given that people are interested in extracting code from incomplete proofs, it should be mentioned that there are techniques for maintaining correctness when it is allowed. For example, if singleton justification removal is not being used, then the simplest technique is to extract, from an unproved subgoal $\mathcal{H} \gg F$, a piece of code that prints the error message "$\mathcal{H} \gg F$ unproved" and halts. Extracted code for the theorem as a whole has the property that either it runs to completion and produces a correct result, or it halts with an error message. It never runs to completion and produces an incorrect result.

Unfortunately, the code halts earlier than necessary, since the error will be noticed when the justification for F is being built, not when it is used. Code motion and symbolic substitution can be used to delay the execution of the error message, but perfect solutions to this problem seem to require call-by-demand semantics for extracted code.

Though this topic is interesting, my work in it is incomplete, so I will leave it for future work and return to the topic of generating efficient λ PRL code.

Chapter 4

Adding arrays to λPRL

This chapter discusses how extracted code can be improved by replacing slow operations on immutable objects with fast operations on mutable objects. For concreteness, a specific case is studied: arrays.

Since arrays are passed by reference, not by value, the semantics of λ PRL must be expanded, but adding call-by-reference parameters complicates matters: the notion of justification is harder to explain, and aliasing and side effects cause problems. To solve these problems, a proof style is imposed that restricts aliasing and side effects but still allows proofs and specifications to be written in a natural way.

4.1 Complications caused by adding reference parameters

As defined so far, λPRL code passes parameters by value. Using value parameters makes it easy to define and discuss justification and extraction, but it also makes the code inefficient on large, modifiable structures like linked lists and arrays. This chapter discusses the problems introduced when singly-dimensioned arrays of integers are added to λPRL as a new primitive data structure. I believe this treatment generalizes to include other modifiable data structures, but I have not worked this out in detail.

One reason to concentrate on adding arrays is that reasoning about their properties is straightforward [NeO], and this makes it easier to focus on the problems caused by adding call-by-reference parameters. So, though array is being added as a new type, with its own primitive functions, I will not give axioms for reasoning about arrays; let us assume that appropriate axioms will be added to the *decision* rule.[1]

To start with, let *mkarray*, *length*, *select*, and *update* be new built-in primitive routines, where *mkarray*(n) returns a zeroed array of length n, *length*(a) returns the length of the array a, *select*(a, i) returns the ith ele-

[1] Alternatively, arrays can be reasoned about as though they were lists: the primitive operations on arrays can be coded as explicitly defined procedures, and their relevant properties can be deduced from the definitions. If this is done, arrays simply become an alternative way to implement certain lists. This is discussed more in chapter 6.

ment of a, and *update*(a, i, e) returns an array a' similar to a except that $a[i]$ equals e. Like Dijkstra and Gries [Dij,Gri], I take the view that modifying an array yields a whole new array, since properties of the old array are not necessarily properties of the new array. Thus a and *update*(a, i, e) denote two distinct objects, though for efficient execution they should be stored at the same location.

Notation. $|a|$ means *length*(a), $a[i]$ means *select*(a, i), and $(a; i : e)$ means *update*(a, i, e).

Call-by-value semantics requires that the *update* function copy most of a to return its result, so $(a; i : e)$ requires $O(|a|)$ time and space to run. In the usual semantics for $(a; i : e)$, a is passed by reference: the *location* of a is passed, not its value. In addition, a must be a modifiable parameter, since otherwise all of a except for $a[i]$ must be copied to produce the result array. Thus the desired meaning for *update* is that it is a function that takes two integers i and e and the location of an array a; it modifies a so that $a[i]$ is e and returns the location of a as its result.

Adding call-by-reference parameters to the syntax and semantics is not much of a problem alone, but the possibility of modified arrays raises a set of questions:

- When is an all-justification (a justification for a universal) allowed to modify an array parameter? For example, if γ is a justification for All a : array . F, when may $\gamma(a)$ modify a?

- May an implication-justification modify an array embedded inside the justification for its antecedent? For example, if γ justifies the implication (Some a : array . F) $\Rightarrow G$, and (a', γ') justifies Some a : array. F, may $\gamma((a', \gamma'))$ modify a'?

- May all-justifications or implication-justifications modify arrays that are *not* part of their inputs? That is, if γ justifies a universal or an implication, may evaluation of γ have side effects?

- How should modified values be handled in the logic? For example, let π be the proof

$$\mathcal{H} \gg \text{Some } a' : \text{array} . F \quad \text{by intro } (a; i : e)$$
$$\bullet\ \mathcal{H} \gg F\left[(a; i : e)/a'\right].$$

Let σ be the subproof of π. The extracted code for π, $\mathsf{X}[\![\pi, \mathcal{J}]\!]$, is $((a; i : e), \mathsf{X}[\![\sigma, \mathcal{J}]\!])$. Assuming at run time $(a; i : e)$ is evaluated before $\mathsf{X}[\![\sigma, \mathcal{J}]\!]$, the value of a no longer exists when $\mathsf{X}[\![\sigma, \mathcal{J}]\!]$ is evaluated. Clearly $\mathsf{X}[\![\sigma, \mathcal{J}]\!]$ may not refer to a; what does this imply about the proof rules used in σ?

There is another aspect of call-by-reference parameter passing that causes difficulties: ***aliasing***. (Two array expressions ***alias*** if they refer to the same location [Hec].) The only built-in function that aliases is ***update***: a and $(a; i : e)$ alias. Programs can alias by using ***update***, by using other code that aliases, or by being extracted from certain proofs. For example, the natural proof of All a : array . Some a' : array . $a = a'$ aliases a and a'.

Aliasing causes difficulties because of the way it can interact with the problem of accessing modified values: if two or more expressions are aliased then if any of them may not be accessed, none of them may be accessed. The need to track and control aliasing raises some questions:

- May arrays being passed to all or implication-justifications alias? For example, if γ justifies All a, a' : array . F, may one call $\gamma(b, b')$, where b and b' are stored at the same location?

- May the witnesses of some-justifications alias? For example, suppose $\langle b, b', \gamma\rangle$ justifies Some a, a' : array . F; may b and b' alias? For another example, if (b, γ) justifies Some a : array . F and (b', γ') justifies Some a' : array . G, may b and b' alias?

- When may existential arrays alias with universal arrays? Which ones? For example, say γ justifies

$$\text{All } a, a' : \text{array} . \text{Some } b, b' : \text{array} . F.$$

May γ alias b to a or a'? May it alias b' to a or a'? Are the answers to these two questions connected? Note that other, more complicated instances exist: consider

$$\text{All } a, a' : \text{array} . (F \Rightarrow \text{Some } b, b' : \text{array} . G).$$

To handle these problems, aliasing and side effect restrictions will be added along with call-by-reference parameters. These restrictions will promote functional, side-effect-free programs and proofs. To show that this style, though restrictive, is still useful, the next section gives a sample proof and discusses the ways it modifies and aliases arrays.

4.2 Quicksort

The aliasing and side effect restrictions to be imposed will strongly affect the way proofs can be built. This section presents a sample array-modifying proof to illustrate the proof style being used and to show that interesting predicates can be proved using this style. The proof is also a concrete example to keep in mind when the new logic is presented formally.

The proof, quicksort, is presented in detail. Readers who do not want to study quicksort so closely are encouraged to read only the first few pages before skipping to the next section, where the design of the new logic is discussed.

The quicksort theorem and its goal

A quicksort proof is a proof whose extracted code is a quicksort function. Its conclusion is

$$\begin{aligned}&\text{All } a : \text{array} . \text{Some } b : \text{array} .\\ &\quad \mathit{Permuted\_arrays}(a, b) = 1 \ \&\ \mathit{Sorted\_array}(b) = 1.\end{aligned}$$

Permuted_arrays and *Sorted_array* are primitive recursive user-defined characteristic functions. Their exact definitions can be ignored here, since they have no effect on the extracted code.[2] Intuitively, $\mathit{Permuted\_arrays}(a, b)$ is 1 iff a and b are permutations of each other, and $\mathit{Sorted\_array}(b)$ is 1 iff b is sorted.

Since the sort should be done in place, a and b should be aliased and a should be variable. (That is, modifiable.) I will indicate these explicitly through the use of two new keywords, **aliases** and **var**. (These fields are not entered by the user, they are inferred by the extractor when it inspects the proof.) Adding these fields to the conclusion yields

$$\begin{aligned}&\text{All var } a : \text{array} . \text{Some var } b : \text{array} .\\ &\quad \mathit{Permuted\_arrays}(a, b) = 1 \ \&\ \mathit{Sorted\_array}(b) = 1\\ &\quad (\text{aliases } (a, b)).\end{aligned} \tag{4.1}$$

[2] Equalities are singletons, so no code is extracted for them.

The array b is marked var to indicate that it can be modified by later computations. (There will also be unmodifiable arrays; they and their relationship to modifiable arrays are discussed in the next section.)

The main conclusion of quicksort follows from the sorting lemma, which asserts that any arbitrary segment of an array can be sorted. The lemma's conclusion has the form

$$\begin{aligned}&\textsf{All } s:\textsf{int}\,.\,\textsf{All var } a:\textsf{array}, l, r:\textsf{int}\,.\,\textsf{Some var } b:\textsf{array}\,.\\ &\quad \mathit{Slice}(a,l,r,s) = 1 \Rightarrow\\ &\qquad\quad \mathit{Permuted}(a,b,l,r) = 1 \;\&\; \mathit{Sorted}(b,l,r) = 1\\ &\quad (\textsf{aliases } (a,b)).\end{aligned} \tag{4.2}$$

$\mathit{Slice}(a,l,r,s)$ is 1 iff $a[l..r]$ is a segment of a of length $\leq s$, that is, $0 \leq l \leq r < |a|$ and $r - l + 1 \leq s$. (Recall that these are zero-origin arrays.) $\mathit{Sorted}(b,l,r)$ is 1 iff the segment $b[l..r]$ is sorted. $\mathit{Permuted}(a,b,l,r)$ is 1 iff $a[l..r]$ and $b[l..r]$ are permutations of each other and a and b agree elementwise outside of $l..r$.

Two details of the form of (4.2), the sorting lemma conclusion, are worth discussing here. The first detail is the quantification of s: it is quantified separately from a, l, and r. This is done because the goal is proved by induction on s and the induction rule only allows conclusions to quantify one variable. The second detail is the quantification of a and b: they are quantified next to each other in an $\forall\exists$ ("ā-ē") formula. Returning a

modified array (that is, having a Some y array variable alias with a modified All x variable) will be restricted to $\forall\exists$ formulas, so most variants of (4.2), like

$$\textsf{All var } a:\textsf{array}, l, r:\textsf{int}\,.\,\textsf{All } s:\textsf{int}\,.\,\textsf{Some var } b:\textsf{array}\ldots$$

and

$$\textsf{All var } a:\textsf{array}\,.\,\textsf{All } s, l, r:\textsf{int}\,.\,\textsf{Some var } b:\textsf{array}\ldots$$

are not allowed. In addition, goals like

$$[\textsf{var } a:\textsf{array}] \gg \textsf{All } s:\textsf{int}\,.\,\textsf{All } l, r:\textsf{int}\,.\,\textsf{Some var } b:\textsf{array}\ldots$$

are not generally allowed. The reason for making this restriction is discussed in the next section, but basically the restriction is made to avoid generating extracted code that contains procedures with side effects. Side effects are prohibited because tracking and controlling them would impose unpleasant and unwieldy changes to the logic and semantics.

To see how one of these prohibited variants of (4.2) uses side effects, let γ justify

$$\begin{aligned}&\textsf{All var } a:\textsf{array}\,.\,\textsf{All } s, l, r:\textsf{int}\,.\,\textsf{Some var } b:\textsf{array}\ldots\\ &\quad(\textsf{aliases } (a,b)).\end{aligned}$$

For any particular a, the justification γ takes the location of a and returns a function γ' that takes s, l, and r and returns the location of b. (If singleton formulas are not being removed, a justification for the ellipsed matrix is also returned.) Since a and b are aliased, the function γ' has the side effect of modifying a.

I will now return to the proof of the quicksort theorem. The proof of the main goal using the sorting lemma is easy and is presented in detail, with the focus on how a is modified to produce b. For reference, the proof is also given in figure 4.1.

To save space, some abbreviations are made: types are suppressed, (aliases (a, b)) is mostly omitted, and the matrices of (4.1) and (4.2) are abbreviated by $F_1(a, b)$ and $F_2(s, a, l, r, b)$ respectively. Thus the conclusion of the quicksort theorem is

$$\textsf{All var } a \,.\, \textsf{Some var } b \,.\, F_1(a, b)$$

where $F_1(a, b)$ asserts that b is a sorted version of a, and the conclusion of the sorting lemma is

$$\textsf{All } s \,.\, \textsf{All var } a; l, r \,.\, \textsf{Some var } b \,.\, F_2(s, a, l, r, b)$$

where $F_2(s, a, l, r, b)$ asserts that b is like a except that $b[l..r]$ is sorted, where $l..r$ is a segment of length $\leq s$. The semicolon in var $a; l, r$ is used to separate the var a from l and r, so var $a; l, r$ means var a : array, l, r : int.

The first two lines of the quicksort proof use the sequence and theorem rules to bring in (4.2) as a new hypothesis. In the proof segments below,

Line	Depth	var	log	Sequent	Rule
1	0			$\gg 1$	seq 2
2	1			$\gg 2$	thm *sorting*
3	1			$2 \gg 1$	intro a
4	2	a		$2 \gg 3$	elim 2, $\|a\|$
5	3	a		$2, 4 \gg 3$	elim 4, a, 0, $\|a\| - 1$
6	4		a	$2, 4, 5 \gg 3$	elim 5, b
7	5	b	a	$2, 4, 6 \gg 3$	intro b
8	6		a, b	$2, 4, 6 \gg 7$	*(omitted)*

	Formulas
1	All var a : array . Some var b : array . $F_1(a, b)$
2	All s : int . All var a : array, l, r : int . Some var b : array . $F_2(s, a, l, r, b)$
3	Some var b : array . $F_1(a, b)$
4	All var a : array, l, r : int . Some var b : array . $F_2(\|a\|, a, l, r, b)$
5	Some var b : array . $F_2(\|a\|, a, 0, \|a\| - 1, b)$
6	$F_2(\|a\|, a, 0, \|a\| - 1, b)$
7	$F_1(a, b)$

Figure 4.1: The proof of the quicksort theorem.

line numbers are shown in parentheses to the left.

(1) $\gg$ All var a . Some var b . $F_1(a,b)$

by seq All s . All var $a; l, r$. Some var b . $F_2(s,a,l,r,b)$

(2) • $\gg$ All s . All var $a; l, r$. Some var b . $F_2(s,a,l,r,b)$

by thm *sorting*

In the next line, All var a is introduced. Note how the new environment is marked [var a], short for [var a : array], to indicate that a is modifiable.

(3) All $s \ldots \gg$ All var a . Some var b . $F_1(a,b)$ by intro a

(4) • [var a] All $s \ldots \gg$ Some var b . $F_1(a,b)$

In lines 3 through 5, s, a, l, and r are eliminated by $|a|$, a, 0, and $|a|-1$ respectively. Because there are two nested All quantifiers, this is done in two steps. Note that after s is eliminated with $|a|$, a is still marked var: taking the length of a does not modify it, so the value of a is still available. After the second elimination, however, the value of a is no longer available, so the new environment is [log a], short for [logical a : array]. Here logical is a new keyword used to distinguish variables whose values no longer exist or may not be accessed by the extracted code. Logical identifiers can be mentioned in the proof, but not in any way that would require them to appear in extracted code.

(3) [var a] All $s \ldots \gg$ Some var b . $F_1(a,b)$ by elim 1, $|a|$

(4) • [var a] All $s \ldots$, All var $a; l, r \ldots \gg$ Some var $b \ldots$

by elim $2, a, 0, |a|-1$

(5) •• [log a] All $s \ldots$, All var $a; l, r \ldots$,

Some var b . $F_2(|a|, a, 0, |a|-1, b) \gg$ Some var $b \ldots$

The witness b is sorted. In lines 5 through 7, it is eliminated from its hypothesis and introduced into the conclusion. Once b is introduced, it is marked logical to keep it from being modified by the proof of $F_1(a,b)$.

(5) [log a] $\ldots, \ldots$, Some var b . $F_2(|a|, a, 0, |a|-1, b)$

$\gg$ Some var b . $F_1(a,b)$ by elim 3, b

(6) • [log a; var b] $\ldots, \ldots, F_2(|a|, a, 0, |a|-1, b)$

$\gg$ Some var b . $F_1(a,b)$ by intro b

(7) •• [log a, b] $\ldots, \ldots, F_2(|a|, a, 0, |a|-1, b) \gg F_1(a,b)$

The last goal is trivial. Expanding it, and ignoring its first two hypotheses, it is

[log a, b]

$Slice(a, 0, |a|-1, |a|) \Rightarrow$

$Permuted(a, b, 0, |a|-1) = 1 \;\&\; Sorted(b, 0, |a|-1) = 1$

$\gg Permuted\_arrays(a,b) = 1 \;\&\; Sorted\_array(b) = 1.$

To prove this goal, it is not necessary to use a or b in computational ways, for example, in *all-elim* or *some-intro* rules.[3] This is fortunate, since both a and b are both logical arrays: the value of a logical array cannot be accessed, so logical arrays cannot be used computationally.

Proof of the sorting lemma

The sorting lemma is proved by induction on s, the limit on the length of the segment to be sorted. The full proof is given in figures 4.2 and 4.3. The nontrivial case, $s > 1$, is presented here in detail.

Since this is to be a quicksort proof, the segment of a to be sorted is first partitioned into left and right subsegments where each element of the left subsegment is less than each element of the right. This is done using a *partitioning* lemma, whose conclusion is

$$
\begin{array}{l}
\text{All } s:\text{int}, \text{var } a:\text{array}, l, r:\text{int} \,.\, \text{Some var } b:\text{array}, m:\text{int} \,. \\
\quad Slice(a,l,r,s) = 1 \Rightarrow \\
\quad (l \le m \le r \;\&\; Permuted(a,b,l,r) = 1 \qquad\qquad (4.3) \\
\qquad \&\; Partitioned(b,l,m-1,m,r,0) = 1) \\
\quad (\text{aliases } (a,b)).
\end{array}
$$

[3] In section 3.3, *computational* meant something more restrictive. Here, and for the remainder of the thesis, a *computational use* of an identifier in a proof is one that appears in the extracted code for the proof. Note that if the extractor is ignoring singleton formulas, a logical array can be used when eliminating a singleton universal.

The matrix of (4.4) asserts first that a and b are equal elementwise except on $l..r$; $a[l..r]$ and $b[l..r]$ are permutations of each other. Second, using the *Partitioned* predicate, (4.4) asserts that the elements of $b[l..m-1]$ are each $< b[m]$ and that the elements of $b[m..r]$ are each $\ge b[m]$. Using an array picture, as in Gries [Gri], b looks like this:

	l	m	r	
b		$< b[m]$	$\ge b[m]$	

In general, $Partitioned(b,l_1,r_1,l_2,r_2,s) = 1$ asserts that the elements of $b[l_1..r_1]$ are each $< b[l_2]$ and that the elements of $b[l_2..r_2]$ are each $\ge b[l_2]$. It also implies that the length of the segment between r_1 and l_2 is $\le s$. Pictorially, $Partitioned(b,l_1,r_1,l_2,r_2,s) = 1$ means

		l_1	r_1	l_2	r_2	
b		$< b[l_2]$		$\ge b[l_2]$		

Let $F_3(s,a,l,r,b,m)$ abbreviate the matrix of (4.4), the conclusion of the partitioning lemma. The first few lines of the proof of the sorting lemma are concerned with base cases and uninteresting cases. (For example, the case $s < 1$ is not interesting.) In the next part of the proof the conclusion of the partitioning lemma is added in as a new hypothesis. The interesting

Line	D	var	log	Sequent	Rule
1	0			$\gg 1$	ind 1
2	1			$2,3 \gg 4$	intro a, l, r
3	2	a		$2,3 \gg 5$	intro a
4	3		a	$2,3 \gg 6$	*(omitted)*
5	1			$\gg 7$	intro a, l, r
6	2	a		$\gg 8$	intro a
7	3		a	$\gg 9$	*(omitted)*
8	1			$10, 11 \gg 4$	intro a, l, r
9	2	a		$10, 11 \gg 5$	seq 12
10	3	a		$10, 11 \gg 12$	thm *partitioning*
11	3	a		$10, 11, 12 \gg 5$	elim $12, r-l+1, a, l, r$
12	4		a	$10, 11, 12, 13 \gg 5$	elim $13, b_0, m$
13	5	b_0	a	$10, 11, 12, 14 \gg 5$	elim $10, b_0, l, m-1$
14	6		a, b_0	$10, 11, 12, 14, 15 \gg 5$	elim $15, b_1$
15	7	b_1	a, b_0	$10, 11, 12, 14, 16 \gg 5$	elim $10, b_1, m+1, r$
16	8		a, b_0, b_1	$10, 11, 12, 14, 16, 17 \gg 5$	elim $17, b_2$
17	9	b_2	a, b_0, b_1	$10, 11, 12, 14, 16, 18 \gg 5$	intro b_2
18	10		a, b_0, b_1, b_2	$10, 11, 12, 14, 16, 18 \gg 19$	*(omitted)*

D is the depth of the given line within the proof.

Figure 4.2: The proof of the sorting lemma.

	Formulas
1	All s : int . All var a : array, l, r : int . Some var b : array . $F_2(s, a, l, r, b)$
2	All var a : array, l, r : int . Some var b : array . $F_2(s+1, a, l, r, b)$
3	$s < 1$
4	All var a : array, l, r : int . Some var b : array . $F_2(s, a, l, r, b)$
5	Some var b : array . $F_2(s, a, l, r, b)$
6	$F_2(s, a, l, r, a)$
7	All var a : array, l, r : int . Some var b : array . $F_2(1, a, l, r, b)$
8	Some var b : array . $F_2(1, a, l, r, b)$
9	$F_2(1, a, l, r, a)$
10	All var a : array, l, r : int . Some var b : array . $F_2(s-1, a, l, r, b)$
11	$s > 1$
12	All s : int, var a : array, l, r : int . Some var b : array, m : int . $F_3(s, a, l, r, b, m)$
13	Some var b : array, m : int . $F_3(r-l+1, a, l, r, b, m)$
14	$F_3(r-l+1, a, l, r, b_0, m)$
15	Some var b : array . $F_2(s-1, b_0, l, m-1, b)$
16	$F_2(s-1, b_0, l, m-1, b_1)$
17	Some var b : array . $F_2(s-1, b_1, m+1, r, b)$
18	$F_2(s-1, b_1, m+1, r, b_2)$
19	$F_2(s, a, l, r, b_2)$

Figure 4.3: The formulas for the proof of the sorting lemma.

part of the proof begins with line 11, where the partitioning lemma is used: the segment length s is eliminated with $r-l+1$ (the length of the segment $l..r$), and a, l, and r are eliminated with themselves.[4] The resulting array is called b_0 and its middle index is called m. Since a was modified, it is now logical; b_0, however, is var.

(11) [var a] All var $a; l, r$. Some var b ... (*inductive hyp*),

$s > 1$, All s, var $a; l, r$. Some var $b; m$ ... (*partitioning*),

$\gg$ Some var b . $F_2(s, a, l, r, b)$ by elim $3, r-l+1, a, l, r$

(12) •• [log a] ..., ..., ..., Some var $b; m$ ...

$\gg$ Some ... by elim $4, b_0, m$

(13) ••• [var b_0; log a] ..., ..., ..., $F_3(|a|, a, l, r, b_0, m)$

$\gg$ Some ...

In lines 13 and 14, the inductive hypothesis is used to sort the left subsegment $b_0[l..m-1]$. As a result, b_0 becomes logical. The resulting array is called b_1.

(13) [var b_0; log a] All var $a; l, r$ ..., ..., ..., ... $\gg$ Some ...

by elim $1, b_0, l, m-1$

[4] In lines 11 and 12 above, by elim 3 and 4 mean that the third and fourth hypotheses are being eliminated. Keep in mind that the figures use a different notation: in figure 4.2, lines 11 and 12 say by elim 12 and 13 because formulas 12 and 13 of figure 4.3 are being eliminated.

(14) • [log a, b_0] ..., ..., ..., ..., Some var b ...

$\gg$ Some ... by elim $5, b_1$

(15) •• [log a, b_0] ..., ..., ..., ..., $F_2(s-1, b_0, l, m-1, b_1)$

$\gg$ Some ...

In lines 15 and 16, the inductive hypothesis is used again, this time to sort the right subsegment, $b_1[m+1..r]$. Since it is modified, b_1 becomes logical. The resulting array b_2 is var.

(15) [var b_1; log a, b_0] All var $a; l, r$ ..., ..., ..., ..., ...

$\gg$ Some ... by elim $1, b_1, m+1, r$

(16) • [log a, b_0, b_1] ..., ..., ..., ..., ..., Some var b ...

$\gg$ Some ... by elim $6, b_2$

(17) •• [var b_2; log a, b_0, b_1] ..., ..., ..., ..., ...,

$F_2(s-1, b_1, m+1, r, b_2) \gg$ Some ...

In line 17 (not shown), b_2 is introduced for Some var b as the sorted array being returned. The rest of the proof of the sorting lemma is concerned with showing that b_2 is indeed sorted. The proof is omitted here.

Proof of the partitioning lemma

The proof of the partitioning lemma is straightforward and is shown in figure 4.4. The conclusion of the partitioning lemma is

Line	Depth	var	log	Sequent	Rule
1	0			$\gg 1$	intro s, a, l, r
2	1	a		$\gg 2$	seq 3
3	2	a		$\gg 3$	decision
4	2	a		$3 \gg 2$	elim 3
5	3	a		$4 \gg 2$	intro a, l
6	4		a	$4 \gg 6$	*(omitted)*
7	3	a		$5 \gg 2$	seq 7
8	4	a		$5 \gg 7$	thm *subpartitioning*
9	4	a		$5,7 \gg 2$	elim $7, s-1$
10	5	a		$5,7,8 \gg 2$	elim $8, a, l, l-1, r, r$
11	6		a	$5,7,8,9 \gg 2$	elim $9, b, m$
12	7	b	a	$5,7,8,10 \gg 2$	intro b, m
13	8		a, b	$5,7,8,10 \gg 2$	*(omitted)*

	Formulas
1	All s:int, var a:array; l, r:int . Some var b:array, m:int . $F_3(s, a, l, r, b, m)$ ·
2	Some var b:array, m:int . $F_3(s, a, l, r, b, m)$
3	$s \leq 0 \mid s > 0$
4	$s \leq 0$
5	$s > 0$
6	$F_3(s, a, l, r, a, l)$
7	All s:int . All var a:array, l_1, r_1, l_2, r_2:int . Some var b:array, m:int . $F_4(s, a, l_1, r_1, l_2, r_2, b, m)$
8	All var a:array, l_1, r_1, l_2, r_2:int . Some var b:array, m:int . $F_4(s-1, a, l_1, r_1, l_2, r_2, b, m)$
9	Some var b:array, m:int . $F_4(s-1, a, l, l-1, r, r, b, m)$
10	$F_4(s-1, a, l, l-1, r, r, b, m)$
11	$F_3(s, a, l, r, b, m)$

Figure 4.4: The proof of the partitioning lemma.

$$\text{All } s, \text{var } a; l, r \,.\, \text{Some } b; m \,.\, F_3(s, a, l, r, b, m) \quad (\text{aliases } (a, b))$$

where $F_3(s, a, l, r, b, m)$ is

$$Slice(a, l, r, s) = 1 \Rightarrow (l \leq m \leq r \;\&\; Permuted(a, b, l, r) = 1 \;\&$$
$$Partitioned(b, l, m-1, m, r, 0) = 1).$$

The partitioning lemma asserts that an arbitrary array segment can be partitioned into two parts, one less than the other. Pictorially, the result array b is

		l	m	r
b		$< b[m]$	$\geq b[m]$	

The partitioning lemma is proved with the help of the *subpartitioning* lemma, whose conclusion is

$$\text{All } s \,.\, \text{All var } a; l_1, r_1, l_2, r_2 \,.\, \text{Some } b; m \,.\, F_4(s, a, l_1, r_1, l_2, r_2, b, m)$$
$$(\text{aliases } (a, b))$$

where $F_4(s, a, l_1, r_1, l_2, r_2, b, m)$ is

$$Partitioned(a, l_1, r_1, l_2, r_2, s) = 1 \Rightarrow$$
$$r_1 < m \leq l_2 \;\&\; Permuted(a, b, r_1 + 1, l_2 - 1) = 1 \;\&$$
$$Partitioned(b, l_1, m-1, m, r_2, 0) = 1.$$

The subpartitioning lemma asserts that if an array segment is *partially* partitioned (that is, the elements of $a[l_1 .. r_1]$ are each $< a[l_2]$, and the elements of $a[l_2+1 .. r_2]$ are each $\geq a[l_2]$), then the middle part of the segment, $a[r_1+1 .. l_2-1]$, can be partitioned. Pictorially, the subpartitioning lemma takes an array a of the form

	l_1	r_1		l_2 r_2	
a		$< a[l_2]$		$\geq a[l_2]$	

and returns an array b of the form

	l_1	r_1 m	l_2 r_2	
b		$< b[m]$	$\geq b[m]$	

.

If l_1 is l and r_2 is r then this b can be returned by the partitioning lemma, so the proof in figure 4.4 uses the subpartitioning lemma on the trivial partial partition $a[l .. l-1]$ and $a[r .. r]$ and returns the completely partitioned b.

Proof of the subpartitioning and partition thinning lemmas

The proof of the subpartitioning lemma is shown in figures 4.5 and 4.6. It goes by induction on s, the limit on the length of the unpartitioned

Line	D	var	log	Sequent	Rule
1	0			$\gg 1$	ind 0
2	1			$2,3 \gg 4$	intro a, l_1, r_1, l_2, r_2
3	2	a		$2,3 \gg 5$	intro a, l_2
4	3		a	$2,3 \gg 6$	*(omitted)*
5	1			$\gg 7$	intro a, l_1, r_1, l_2, r_2
6	2	a		$\gg 8$	intro a, l_2
7	3		a	$\gg 9$	*(omitted)*
8	1			$10,11 \gg 4$	intro a, l_1, r_1, l_2, r_2
9	2	a		$10,11 \gg 5$	seq 12
10	3	a		$10,11 \gg 12$	decision
11	4	a		$10,11,12 \gg 5$	elim 12
12	5	a		$10,11,13 \gg 5$	intro a, l_2
13	6		a	$10,11,13 \gg 15$	*(omitted)*
14	5	a		$10,11,14 \gg 5$	seq 16
15	6	a		$10,11,14 \gg 16$	thm *partition_thinning*
16	6	a		$10,11,14,16 \gg 5$	elim $16, l_2 - r_1, a, l_1, r_1, l_2, r_2$
17	7		a	$10,11,14,16, 17 \gg 5$	elim $17, b_0, l_2', r_1'$
18	8	b_0	a	$10,11,14,16, 18 \gg 5$	elim $10, b_0, l_1, r_1', l_2', r_2$
19	9		a, b_0	$10,11,14,16,18, 19 \gg 5$	elim $19, b, m$
20	10	b	a, b_0	$10,11,14,16,18, 20 \gg 5$	intro b, m
21	10		a, b_0, b	$10,11,14,16,18, 20 \gg 21$	*(omitted)*

D is the depth of the given line within the proof.

Figure 4.5: The proof of the subpartitioning lemma.

	Formulas
1	All s : int . All var a : array, l_1, r_1, l_2, r_2 : int . Some var b : array, m : int . $F_4(s, a, l_1, r_1, l_2, r_2, b, m)$
2	All var a : array, l_1, r_1, l_2, r_2 : int . Some var b : array, m : int . $F_4(s+1, a, l_1, r_1, l_2, r_2, b, m)$
3	$s < 0$
4	All var a : array, l_1, r_1, l_2, r_2 : int . Some var b : array, m : int . $F_4(s, a, l_1, r_1, l_2, r_2, b, m)$
5	Some var b : array, m : int . $F_4(s, a, l_1, r_1, l_2, r_2, b, m)$
6	$F_4(s, a, l_1, r_1, l_2, r_2, a, l_2)$
7	All var a : array, l_1, r_1, l_2, r_2 : int . Some var b : array, m : int . $F_4(0, a, l_1, r_1, l_2, r_2, b, m)$
8	Some var b : array, m : int . $F_4(0, a, l_1, r_1, l_2, r_2, b, m)$
9	$F_4(0, a, l_1, r_1, l_2, r_2, a, l_2)$
10	All var a : array, l_1, r_1, l_2, r_2 : int . Some var b : array, m : int . $F_4(s-1, a, l_1, r_1, l_2, r_2, b, m)$
11	$s > 0$
12	$l_2 = r_1 + 1 \mid l_2 \neq r_1 + 1$
13	$l_2 = r_1 + 1$
14	$l_2 \neq r_1 + 1$
15	$F_4(s, a, l_1, r_1, l_2, r_2, a, l_2)$
16	All s : int, var a : array, l_1, r_1, l_2, r_2 : int . Some var b : array, l'_2, r'_1 : int . $F_5(s, a, l_1, r_1, l_2, r_2, b, l'_2, r'_1)$
17	Some var b : array, l'_2, r'_1 : int . $F_5(l_2 - r_1, a, l_1, r_1, l_2, r_2, b, l'_2, r'_1)$
18	$F_5(l_2 - r_1, a, l_1, r_1, l_2, r_2, b_0, l'_2, r'_1)$
19	Some var b : array, m : int . $F_4(s-1, b_0, l_1, r'_1, l'_2, r_2, b, m)$
20	$F_4(s-1, b_0, l_1, r'_1, l'_2, r_2, b, m)$
21	$F_4(s, a, l_1, r_1, l_2, r_2, b, m)$

Figure 4.6: The formulas for the proof of the subpartitioning lemma.

middle segment $a[r_1 + 1 .. l_2 - 1]$. If the middle segment is empty then a is already partitioned, so a is returned for b, with l_2 for m.

If the middle segment is not empty, it is made a little smaller by using the *partition_thinning* lemma. The lemma makes the middle segment of a one element shorter and returns a with new r_1 and l_2 indices, as appropriate. The new array is called b_0 in figure 4.5, and its r_1 and l_2 indices are called r'_1 and l'_2. Since the middle segment of b_0 is strictly shorter than that of a, the inductive hypothesis can be used on b_0 to complete the partition.

This leaves the partition thinning lemma to be proved. Its conclusion is

$$\text{All } s; \text{var } a; l_1, r_1, l_2, r_2 \text{ . Some var } b; l'_2, r'_1 \text{ .}$$
$$F_5(s, a, l_1, r_1, l_2, r_2, b, l'_2, r'_1)$$
$$(\text{aliases } (a, b))$$

where $F_5(s, a, l_1, r_1, l_2, r_2, b, l'_2, r'_1)$ is

$$(Partitioned(a, l_1, r_1, l_2, r_2, s) = 1 \;\&\; r_1 < l_2 - 1) \Rightarrow$$
$$(r_1 \le r'_1 \le l'_2 \le l_2 \;\&\; l'_2 - r'_1 < l_2 - r_1$$
$$\& \; Partitioned(b, l_1, r'_1, l'_2, r_2, s - 1) = 1$$
$$\& \; Permuted(a, b, r_1 + 1, l_2 - 1)).$$

Pictorially, given a of the form

		l_1 $\qquad$ r_1		l_2 $\qquad$ r_2	
a		$< a[l_2]$		$\geq a[l_2]$	

the lemma returns b, r_1', and l_2' of the form

		l_1 $\qquad$ r_1	r_1' $\quad$ l_2'	l_2 $\qquad$ r_2	
b		$< b[l_2']$		$\geq b[l_2']$	

Except on $r_1+1..l_2-1$, where they are permutations of each other, a and b are elementwise equal.

The proof of the partition thinning lemma is shown in figure 4.7. In the trivial case, $r_1 \geq l_2 - 1$. In this case, the middle subsegment to be thinned, $a[r_1+1..l_2-1]$, is empty, so a is returned for b and r_1 and l_2 are returned for r_1' and l_2'.

If the middle subsegment is nonempty, $a[l_2-1]$ is compared against $a[l_2]$ and moved to the left or right subsegment as appropriate: if $a[l_2-1] < a[l_2]$, then $a[l_2-1]$ and $a[r_1+1]$ are swapped and r_1+1 and l_2 are returned for r_1' and l_2'. If $a[l_2-1] \geq a[l_2]$ then $a[l_2-1]$ and $a[l_2]$ are swapped and r_1 and l_2-1 are returned for r_1' and l_2'. The *swap* routine is not described here, but it has the obvious semantics: $swap(a,i,j)$ swaps $a[i]$ and $a[j]$ and returns the location of a.

Line	Depth	var	log	Sequent	Rule
1	0			$\gg 1$	intro s, a, l_1, r_1, l_2, r_2
2	1	a		$\gg 2$	seq 3
3	2	a		$\gg 3$	decision
4	2	a		$3 \gg 2$	elim 3
5	3	a		$4 \gg 2$	intro a, l_2, r_1
6	4		a	$4 \gg 6$	*(omitted)*
7	3	a		$5 \gg 2$	seq 7
8	4	a		$5 \gg 7$	decision
9	5	a		$5,7 \gg 2$	elim 7
10	5	a		$5,8 \gg 2$	intro $swap(a, r_1+1, l_2-1)$, l_2, r_1+1
11	6		a	$5,8 \gg 10$	*(omitted)*
12	5	a		$5,9 \gg 2$	intro $swap(a, l_2, l_2-1)$, l_2-1, r_1
13	6		a	$5,8 \gg 11$	*(omitted)*

	Formulas
1	All s : int, var a : array, l_1, r_1, l_2, r_2 : int . Some var b : array, l_2', r_1' : int . $F_5(s, a, l_1, r_1, l_2, r_2, b, l_2', r_1')$
2	Some var b : array, l_2', r_1' : int . $F_5(s, a, l_1, r_1, l_2, r_2, b, l_2', r_1')$
3	$l_2 - r_1 \leq 1 \mid l_2 - r_1 > 1$
4	$l_2 - r_1 \leq 1$
5	$l_2 - r_1 > 1$
6	$F_5(s, a, l_1, r_1, l_2, r_2, a, l_2, r_1)$
7	$a[l_2-1] < a[l_2] \mid a[l_2-1] \geq a[l_2]$
8	$a[l_2-1] < a[l_2]$
9	$a[l_2-1] \geq a[l_2]$
10	$F_5(s, a, l_1, r_1, l_2, r_2, swap(a, r_1+1, l_2-1), l_2, r_1+1)$
11	$F_5(s, a, l_1, r_1, l_2, r_2, swap(a, l_2, l_2-1), l_2-1, r_1)$

Figure 4.7: The proof of the partition thinning lemma.

Pictorially, given a of the form

		l_1 … r_1		l_2 … r_2	
a		$< a[l_2]$		$\geq a[l_2]$	

the lemma returns b, r_1', and l_2' of the form

		l_1 … r_1	r_1' … l_2'	l_2 … r_2	
b		$< b[l_2']$		$\geq b[l_2']$	

Except on $r_1+1..l_2-1$, where they are permutations of each other, a and b are elementwise equal.

The proof of the partition thinning lemma is shown in figure 4.7. In the trivial case, $r_1 \geq l_2 - 1$. In this case, the middle subsegment to be thinned, $a[r_1+1..l_2-1]$, is empty, so a is returned for b and r_1 and l_2 are returned for r_1' and l_2'.

If the middle subsegment is nonempty, $a[l_2-1]$ is compared against $a[l_2]$ and moved to the left or right subsegment as appropriate: if $a[l_2-1] < a[l_2]$, then $a[l_2-1]$ and $a[r_1+1]$ are swapped and r_1+1 and l_2 are returned for r_1' and l_2'. If $a[l_2-1] \geq a[l_2]$ then $a[l_2-1]$ and $a[l_2]$ are swapped and r_1 and l_2-1 are returned for r_1' and l_2'. The *swap* routine is not described here, but it has the obvious semantics: $swap(a,i,j)$ swaps $a[i]$ and $a[j]$ and returns the location of a.

Line	Depth	var	log	Sequent	Rule
1	0			$\gg 1$	intro s, a, l_1, r_1, l_2, r_2
2	1	a		$\gg 2$	seq 3
3	2	a		$\gg 3$	decision
4	2	a		$3 \gg 2$	elim 3
5	3	a		$4 \gg 2$	intro a, l_2, r_1
6	4		a	$4 \gg 6$	*(omitted)*
7	3	a		$5 \gg 2$	seq 7
8	4	a		$5 \gg 7$	decision
9	5	a		$5,7 \gg 2$	elim 7
10	5	a		$5,8 \gg 2$	intro $swap(a, r_1+1, l_2-1)$, l_2, r_1+1
11	6		a	$5,8 \gg 10$	*(omitted)*
12	5	a		$5,9 \gg 2$	intro $swap(a, l_2, l_2-1)$, l_2-1, r_1
13	6		a	$5,8 \gg 11$	*(omitted)*

	Formulas
1	All s : int, var a : array, l_1, r_1, l_2, r_2 : int . Some var b : array, l_2', r_1' : int . $F_5(s, a, l_1, r_1, l_2, r_2, b, l_2', r_1')$
2	Some var b : array, l_2', r_1' : int . $F_5(s, a, l_1, r_1, l_2, r_2, b, l_2', r_1')$
3	$l_2 - r_1 \leq 1 \mid l_2 - r_1 > 1$
4	$l_2 - r_1 \leq 1$
5	$l_2 - r_1 > 1$
6	$F_5(s, a, l_1, r_1, l_2, r_2, a, l_2, r_1)$
7	$a[l_2-1] < a[l_2] \mid a[l_2-1] \geq a[l_2]$
8	$a[l_2-1] < a[l_2]$
9	$a[l_2-1] \geq a[l_2]$
10	$F_5(s, a, l_1, r_1, l_2, r_2, swap(a, r_1+1, l_2-1), l_2, r_1+1)$
11	$F_5(s, a, l_1, r_1, l_2, r_2, swap(a, l_2, l_2-1), l_2-1, r_1)$

Figure 4.7: The proof of the partition thinning lemma.

This finishes the proof of quicksort. The optimized extracted code for the quicksort theorem and its lemmas is given in figures 4.8 and 4.9. The code runs in the right time and space: if n is $|a|$, then the code runs in $O(n \log n)$ expected time, $O(n^2)$ worst case time, and constant space. If arrays were passed by value, then the running time would be greater by a factor of n because of the extra time and space needed to copy the arrays. The running space would equal the running time. The optimized extracted code was obtained by using the following optimizations: decision unchaining, substitution for identifiers used only once, unused binding removal, downward movement of bindings, upward code motion, guard combination, inductive procedure optimizations, and closure minimization.

A review

Quicksort was presented here to illustrate the style of proof supported by the (yet to be formally described) modified justifications and rules. The most important notion was that modifying an array yields an new array to be reasoned about: it is aliased with the old array (stored at the same location), but it is a new array conceptually. Whenever an object is modified there are three key pieces of information that must be specified: the name of the old object, the name of the new object, and the relationship between them. The most straightforward way to specify these pieces of information


```
Quicksort_code:
  proc () : just .
    proc (var a : array) : var array .
      Subsorting_code()(|a|)(a, 0, |a| − 1)

Subsorting_code:
  proc () : just . f, where f is bound to

  proc (s : int) : just
    if s < 1 | s = 1 ⇒
      proc (var a : array, l, r : int) : var array . a
    [] s > 1 →
      close (s) . proc (var a : array, l, r : int) : var array .
        let J = f(s − 1);
            (b0, m) = Partitioning_code()(r − l + 1, a, l, r)
        in J(J(b0, l, m − 1), m, r)
        ni
    fi

Partitioning_code:
  proc () : just .
    proc (s : int, var a : array, l, r : int) : var array × int .
      if s ≤ 0 → (a, l)
      [] s > 0 →
        let (b, m) = Subpartitioning_code()(s − 1)(a, l, l − 1, r, r)
        in (b, m)
        ni
      fi
```

Figure 4.8: Optimized extracted code for the quicksort theorem and the sorting and partitioning lemmas.

```
Subpartitioning_code:
    proc () : just . g, where g is bound to

    proc (s : int) : just .
       if s < 0 | s = 0 →
          proc (var a : array, l1, r1, l2, r2 : int) : var array × int . (a, l2)
       [] s > 0 →
          close (s) .
             proc (var a : array, l1, r1, l2, r2 : int) : var array × int .
                if l2 = r1 − 1 → (a, l2)
                [] l2 ≠ r1 − 1 →
                   let (b0, l2', r1') = Partition_thinning_code()
                                              (l2 − r1, a, l1, r1, l2, r2);
                       (b, m) = g(s − 1)(b0, l1, r1', l2', r2)
                   in (b, m)
                   ni
                fi
       fi

Partition_thinning_code:
    proc () : just .
       proc (s : int, var a : array, l1, r1, l2, r2 : int) : var array × int × int .
          if l2 − r1 ≤ 1 → (a, l2, r1)
          [] l2 − r1 > 1 →
             if a[l2 − 1] < a[l2] → (swap(a, r1 + 1, l2 − 1), l2, r1 + 1)
             [] a[l2 − 1] ≥ a[l2] → (swap(a, l2, l2 − 1), l2 − 1, r1)
             fi
          fi
```

Figure 4.9: Optimized extracted code for the subpartitioning and partition thinning lemmas.

is with an $\forall\exists$ formula, like All var a : array . Some var b : array . F, and this is done consistently in the proof.[5] For simplicity, this is also the only style supported.

4.3 Designing a call-by-reference array logic

This section discusses the modifications to be made to the logic and semantics in order to add call-by-reference array parameters. The discussion here is in general terms. The detailed description of the new logic is in the next chapter.

My goal in doing this design was to develop a logic compatible with λ PRL that lets users reason about arrays straightforwardly. The difficulties involved are subtle and involved, so to keep the logic simple enough to be usable, I have cut back on its expressiveness more than I originally wanted to. Nonetheless, because of proofs like quicksort and its lemmas, I believe that the logic is expressive enough to satisfy users' needs.

[5]More precisely, All var a : array . Some var b : array . $(F \Rightarrow G)$ was used, where F was a precondition and G was a postcondition: F described a, and G described b relative to a. All var a : array . $(F \Rightarrow$ Some var b : array . $G)$ would be more appropriate but it uses side effects, as described in the next section, so All var a : array . Some var b : array . $(F \Rightarrow G)$ was used instead.

Referencing modified values

Recall that arrays here are not automatically recopied, so if an array variable a is modified at extracted code runtime, no further reference to its value is allowable afterwards.[6] Note, though, that references to the value stored at the location bound to a *can* be allowed: if a is modified to become b then a may not be referenced but b may; if b is modified to c, then neither a nor b may be referenced but c may, and so on.

There are other, easy to overlook, ways to reference arrays that also must be guarded against. Let γ justify Some var a:array . G, call it F. If the witness embedded in γ is modified, then γ no longer justifies F, so γ may not be referred to by any extracted code run after the modification. In the proof, F may not be eliminated or used in a hypothesis rule in any subproof whose extracted code might be run after the modification of γ.

For another way to reference arrays, let F be All $x:T\,.\,G$, where T is any type, and let J be a justification identifier whose value at runtime justifies F. The value of J is a closed procedure, a procedure bundled together with values for its free identifiers. If one of these free identifiers is an array variable a, then a call of J might reference a, so if a is modified, J must not be called anytime afterwards. In the proof, F may not be

[6] Here, and in general from now on, "reference" means "computational reference," so purely logical, noncomputational, references like $\gg a = a$ by decision can be ignored.

eliminated in any subproof whose extracted code might be run after the modification of a.

This closure-capture problem also comes up with lemmas, since lemmas are extracted as justification-transforming closed procedures: if an array variable a is part of the top level environment of a lemma, the extracted code for the lemma can reference a, so the lemma must not be called in any context where the value of a is not available.

Using local properties of proofs

Users are not shown extracted code in λPRL, so the question "What pieces of extracted code may not refer to identifier a or J?" must be translated into "What parts of the proof may not use identifier a or formula F?" A basic design decision was to have the restrictions on any particular rule instance depend as much as possible on the goal being proved by the instance: its hypotheses, conclusion, and environment. For example, in

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by seq } G$$

- $[\mathcal{E}]\ \mathcal{H} \gg G$
- $[\mathcal{E}]\ \mathcal{H}, G \gg F$

the identifiers and formulas that can be used by the proof of $[\mathcal{E}]\ \mathcal{H} \gg G$ depend on $\mathcal{E}$, $\mathcal{H}$, F, G, and its appearance as the first subproof of a sequence rule. Other proofs of $[\mathcal{E}]\ \mathcal{H} \gg G$ may have other restrictions, depending on the contexts in which they appear.

To understand the reasons for this design decision, it is helpful to understand its alternative. If the array restrictions on a rule instance are not a property of its local context, they must be a property of its global context. Thus the other design involves taking a complete proof and — given the point of the proof that made it necessary to ban some identifier or formula — using flow analysis techniques as in Kennedy [Ken] to verify that no subproof that can be executed after that point uses the forbidden identifier or formula.[7]

For example, consider the following proof, in which a is referenced in the first arm of a sequence proof and modified in the second.

(1) $[\mathsf{var}\ a] \gg \mathsf{Some\ var}\ b\,.\,F$ by seq $\mathsf{All}\ x\!:\!T\,.\,G$

(2) $\bullet\ [\mathsf{var}\ a] \gg \mathsf{All}\ x\!:\!T\,.\,G$ by intro x

(3) $\bullet\bullet\ [\mathsf{var}\ a] \gg G$ (... references a ...)

... other lines omitted ...

(4) $\bullet\ [\mathsf{var}\ a]\ \mathsf{All}\ x\!:\!T\,.\,G \gg \mathsf{Some\ var}\ b\,.\,F$ (4.4)

by intro $(a;i\!:e)$

[7] One proof can be executed before another if its extracted code can be executed before the other's. Note that because of recursion it is possible for a proof to execute before itself. It is also possible to have two proofs both of which are executed before each other.

(5) $\bullet\bullet\ [\log\ a]\ \mathsf{All}\ x\!:\!T\,.\,G \gg F[(a;i\!:e)/a]$

... must not eliminate x ...

The extracted code for this proof is

let K = close (a) . proc $(x:\mathsf{int})$: just .

(... references a ...)

in $((a;i\!:e)$, ... must not call K ...)

ni.

The reference of a in line 3 of the proof is part of the proof of the **All**, so in the extracted code, a is referenced inside the body of K. At runtime, a is actually modified by calculating $(a;i\!:e)$ from line 4, so the code for lines 5 and on must not call K, else the code for line 3 would be evaluated with the wrong value of a. Thus, this proof and its extracted code illustrates the closure-capture problem mentioned earlier in the section.

When all the relevant parts of the proof are available as above, it is not impossible to trace through the proof looking for possible conflicts in how the array identifiers or witnesses are used. In practice, however, this is undesirable for three reasons.

The first reason concerns the difficulty of the analysis, not only for the system, which must check the proofs, but also for the users, who must write them. The problem is that an operation in some given part of the proof can be affected by an operation in a very distant part of the proof, in nonobvious ways. In proof (4.4) above, the uses of a are close together: the reference of a is done in a proof node that is a nephew of the node that modifies a. More generally, the reference could have been made deep in the first arm of the sequence proof, and the modification could have been deep in the second arm. Users unfamiliar with the proof would find it hard to figure out why the modification of a invalidates the sequence formula.

The second reason that local properties of proofs are better than global properties is closely related to the first: the way proofs are built in the PRL systems [Prl,Con86] makes it difficult for users to compare distant parts of a proof. Proofs are built top-down, using a structure-oriented editor similar to the Cornell Program Synthesizer [TeR]. The proof editor displays only one node at a time: a goal, a rule instance, and the subgoals generated by the rule. In addition, the editor's proof traversal commands make it tedious to go from one part of a proof to another, distant part of the proof. Thus, the editor makes it hard for users to get global context information about a proof, and this clashes badly with a technique that requires this information.

The third reason that a design that uses local context is better is that it helps incremental proof development: you need less context to decide what rules can be applied to a given goal. If the goal's global context were

important, you would have to inspect the whole proof before changing a node.

Restriction of side effects

The notion that local properties of the proof are preferable to global properties motivates another design decision: procedures should be free of side effects. That is, procedures may modify their array parameters but not other arrays. There are four kinds of procedures.

- All-justifications map integers, lists, and arrays to justifications for bodies.
- Implication-justifications map justifications for antecedents to justifications for consequents.
- Lemma justification-transformers map justifications for hypotheses to justifications for conclusions.
- Explicitly defined procedures map integers, lists, and arrays to other integers, lists, and arrays.

Explicitly defined procedures are already required to have no non-local references, so the decision to ban side effects does not affect them. It does, however, affect the other three kinds of procedures.

For simplicity's sake, I have applied this restriction strongly: array witnesses embedded inside existential justifications may not be modified. For example, if γ justifies (Some var $a.F) \Rightarrow G$ and $\langle a', \gamma' \rangle$ justifies Some var $a.F$, then $\gamma(\langle a', \gamma' \rangle)$ may not modify a'. As a further simplification, this rule applies even if F is singleton. Banning modification of embedded variables simplifies matters because in general, array witnesses can be buried deep inside justifications. Consider, for example,

$$(G_1 \,\&\, (G_2 \,\&\, (\ldots \,\&\, (G_n \,\&\, \text{Some var } a \,.\, F)\ldots))) \Rightarrow G_{n+1}.$$

The restriction on modifying embedded variables limits expressiveness but not functionality: (Some var $a \,.\, F) \Rightarrow G$ can be rewritten as All var $a \,.\, (F \Rightarrow G)$ and

$$(G_1 \,\&\, (G_2 \,\&\, (\ldots \,\&\, (G_n \,\&\, \text{Some var } a \,.\, F)\ldots))) \Rightarrow G_{n+1}$$

can be rewritten as

$$\text{All var } a \,.\, ((G_1 \,\&\, (G_2 \,\&\, (\ldots \,\&\, (G_n \,\&\, F)\ldots))) \Rightarrow G_{n+1}).$$

(Renaming must be done if a occurs in G or in the G_i.)

Side effects are prohibited because allowing them results in proofs where variables and hypotheses can be killed in locally unpredictable ways. For example, consider

(1) [var a] $\gg F$ by seq All $x : T \,.\, G$

(2) • [var a] $\gg$ All $x : T \,.\, G$ by intro x

(3) •• [var a] $\gg G$ ... uses and modifies a ... (4.5)

(4) • [var a] All $x : T \,.\, G \gg F$ by elim 1,t

(5) •• [log a] $G[t/x] \gg F$.

The code for line 4, the elimination of x with t, runs the procedure that justifies the All. The body of the procedure modifies a, so a must be logical in line 5. Note also that All $x : T \,.\, G$ is not a hypothesis in line 5. Normally it would still be a hypothesis after the *all-elim* in line 4, but here it was removed because it cannot be re-eliminated: the procedure that justifies it no longer has the right value of a.

A procedure without side effects cannot modify an array closed over in an all-justification or embedded in an some-justification, so a call of it cannot invalidate the justification of any hypothesis. Similarly, the call cannot destroy the value of any free variable, except for the arguments of the call.

Banning side effects makes rules like *all-elim* and *impl-elim* easy to design. For example, the subgoal of the *all-elim* rule has all the hypotheses and free variables of its main goal, except for the variables used in the rule. None of the other free variables and none of the hypotheses have to be removed from the subgoal because none of them are affected by the call of the all-justification.

Imported values and the "value" keyword

The import problem is related to the side effect problem: a procedure has a side effect when it modifies a non-local variable; a procedure *imports* a non-local variable if it simply references its value. Proof (4.4) was an example of how an imported array can cause a problem. In it, All $x:T.G$ was proved using a non-local reference to a; after a was modified, x could not be eliminated, since the value of a no longer existed. Proof (4.5) suffered from a similar problem: its universal had to be removed from line 5 because it imported a modified variable.

To keep imported variables from invalidating justifications, they are disallowed: procedures may not import variables.[8] If a procedure wishes to reference the value of a var array, it must access the array as a parameter. As with the side effect restriction, this affects lemmas, all-justifications, and implication-justifications. For example, in a proof of [var a] $\gg$ All $x:T.G$ by *all-intro*, a is made logical so that the all-justification being built will

[8] This restriction is similar to the Russell import rule, which states that an Operation Denotation may not import var identifiers from the surrounding scope [BDD,DoD]. The import rule here is made for roughly the same reason it is made in Russell. In Russell, it is made to insure that syntactically identical Operation Denotations denote the same value. Here, it is made to insure that extracted code for lemmas, all-justifications, and implication-justifications is not invalidated if a non-local variable is changed.

not close over it:

[var a : array] $\gg$ All $x:T.G$ by intro x

• [log a : array, $x:T$] $\gg G$.

Suppose that All $x:T.G$ is being used as a sequence formula to prove F. To prove All $x:T.G$ using a free variable a, a must be made a parameter and then eliminated with itself, as in

[var a] $\gg F$ by seq All var a : array, $x:T.G$

• [var a] $\gg$ All var a : array, $x:T.G$ by intro a', x

•• [log a, var a'] $\gg G$ ... references a' ...   (4.6)

• [var a] All var a : array, $x:T.G \gg F$ by elim $1, a, t$

•• [log a] All $\ldots, G[t/x] \gg F$.

Note that a is logical in the last line of the proof. Locally, this makes sense: a was used to eliminate an All var, and in general such an elimination can modify a. In this *particular* instance, the elimination does not change a, but there is no way to know this locally.

This kind of operation, which uses the value of a variable array without modifying it, comes up too often to be excluded or misinterpreted by the new logic. For example, surely the new logic must include the *length* operation, which returns the length of a without modifying it. To accommodate this new kind of operation, the keyword value, val for short, is introduced.

A value array expression is one whose value may be referenced but never modified. For example, in $a[i]$, which stands for $select(a, i)$, a is a value parameter, so calling $select(a, i)$ does not modify a.

An all-justification cannot modify its value parameters. Since value parameters are not modified, there is no harm in supplying variable arguments for them, so a variable array may used to eliminate a value array,[9] as in

$$[\mathsf{var}\ a]\ \ \mathsf{All\ val}\ a' \ldots \gg \ldots \quad \text{by elim } 1, a.$$

On the other hand, a value array may not be used to eliminate a variable array, as in

$$[\mathsf{val}\ a']\ \ \mathsf{All\ var}\ a \ldots \gg \ldots \quad \text{by elim } 1, a'$$

because the all-justification being called might modify a (and hence a').

Now that value parameters have been introduced, proof (4.6) above can be redone. The proof of the All can look at the *value* parameter a, and the elimination of a can be given a variable array. The proof becomes

$[\mathsf{var}\ a]\ \gg F$ by seq $\mathsf{All\ val}\ a{:}\mathsf{array}, x{:}T\,.\,G$

- $[\mathsf{var}\ a]\ \gg \mathsf{All\ val}\ a{:}\mathsf{array}, x{:}T\,.\,G$ by intro a', x
  - •• $[\log a, \mathsf{val}\ a']\ \gg G \ldots$ references $a' \ldots$

[9]The situation is actually somewhat more complicated: to eliminate All val $a'\,.\,F$ with var a, F must not contain an All or an implication whose justification might capture the value of a', else the import rule could be violated, and F must not contain a Some val b' that aliases with a', else b' would have a value that might be modified later.

- $[\mathsf{var}\ a]\ \ \mathsf{All\ val}\ a{:}\mathsf{array}, x{:}T\,.\,G \gg F$ by elim $1, a, t$
  - •• $[\mathsf{var}\ a]\ \ \mathsf{All\ val}\ a{:}\mathsf{array}, x{:}T\,.\,G,\ G[t/x] \gg F.$

The All val a does not have to be removed from the last line of the proof now, since its elimination does not invalidate its justification at runtime.

Problems caused by aliasing

The final design topic is aliasing. The quicksort proof of the previous section used a form of aliasing that did not affect the computation: in each theorem an input array a was modified to become an output array b, perhaps through intermediaries b_0, b_1, and so on. The aliasing of a and b was an *effect* of the computation; it did not *affect* the computation. Aliasing can indeed affect computations. I will show how this can happen and discuss why it is undesirable in a λPRL setting.

Aliasing does not always affect computations, of course. For example, consider the formula

$$\mathsf{All\ var}\ a, b\,.\,\mathsf{Some\ var}\ a', b'\,.\,F \quad (\mathsf{aliases}\ (a, a'), (b, b')) \qquad (4.7)$$

where F specifies that a' and b' have the form

a'	$a[0]$	$b[1]$	$a[2]$	$b[3]$	...
b'	$b[0]$	$a[1]$	$b[2]$	$a[3]$	...

.

A justification for (4.7) takes a and b and swaps $a[1]$ with $b[1]$, $a[3]$ with $b[3]$, and so on.

As a specification, (4.7) still makes sense even if a and b are aliased: a, b, a', and b' are all stored at the same location, and a' and b' are both

a'	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$\dots$

.

For a specification not achievable if the inputs are aliased, let f and g be functions of type $\mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$ and consider

$$\text{All var } a, b \,.\, \text{Some var } a', b' \,.\, G \quad (\text{aliases } (a, a'), (b, b')) \tag{4.8}$$

where G specifies that a' and b' have the form

a'	$f(a[0], b[0])$	$f(a[1], b[1])$	$f(a[2], b[2])$	$\dots$
b'	$g(a[0], b[0])$	$g(a[1], b[1])$	$g(a[2], b[2])$	$\dots$

.

Assume that for some integer x, $f(x, x)$ does not equal $g(x, x)$, and let a and b be aliased arrays with $a[0] = b[0] = x$. It is now impossible to have a' and b' alias and still satisfy G, since

$$a'[0] = f(a[0], b[0]) = f(x, x) \neq g(x, x) = g(a[0], b[0]) = b[0].$$

There are two basic ways out of this problem: require that a and b be distinct, or modify the specification of the result to include the possibility

that a and b might be aliased.[10] Let aliased be a new field, attached to input identifiers, that specifies the pairs of inputs that *might* be aliased. Equation (4.8) has no aliased field so it requires that a and b be distinct. The unachievable variant of (4.8) is

$$\begin{aligned} &\text{All var } a, b \; (\text{aliased } (a, b)) \,.\, \text{Some var } a', b' \,.\, G \\ &\quad (\text{aliases } (a, a'), (b, b')). \end{aligned} \tag{4.9}$$

Actually, (4.9) is unachievable only if the aliases field is taken to mean "always aliases." I intend it to mean "may alias," so you *could* satisfy (4.9) by never aliasing a' to a and b' to b, but this just avoids the issue.

The alternative is to change the specification. To do this, it is necessary to use the aliased and aliases field more actively, as in

$$\begin{aligned} &\text{All var } a, b\colon \text{array} \,.\, \text{Some } a', b'\colon \text{array}. \\ &\quad G \;\&\; (\neg \text{aliased } (a, b) \Rightarrow (\text{aliases } (a, a'), (b, b'))). \end{aligned}$$

Making aliased and aliases first-class notions, actual formulas, would be a very serious step: new rules dealing with the locations of objects would have to be added to the logic. This would be difficult but not impossible. For example, Boehm [Boe84,Boe85] and Cartwright and Oppen [CaO] have discussed logics of aliasing.

[10] Also, the issue can be sidestepped by translating aliasing properties into array segment properties. Instead of having two arrays $a[0..n]$ and $b[0..n]$ that might alias, you have two segments of some larger array, say $a'[l_1..l_1+n]$ and $a'[l_2..l_2+n]$. The question "Do a and b alias?" becomes "Does l_1 equal l_2?" General overlapping of arrays can also be discussed.

Aliasing is a second-class notion in the new logic for two reasons. The first reason is that λPRL is for studying objects at a high, abstract level, and turning it into a logic that reasons explicitly about memory locations would change its focus, something I want to avoid here. The second reason is pragmatic. None of the new fields var, val, log, and aliases are entered by the user; they are inferred by the extractor. If aliasing were a first-class notion, the inferencing algorithms would be too complex.

Restrictions on aliasing

In the new logic, aliasing of identifiers is purely an effect of how certain operations like *update* are implemented. To keep aliasing from affecting the provability of predicates, procedure parameters are prohibited from aliasing. For example, if All var $a, b \,.\, F$ is eliminated with a' and b', then a' and b' may not alias.

This restriction also applies to procedure calls that appear in extracted code. For example, in an *all-elim* with $f(a, a')$, the variables a and a' may not alias.

The aim is to avoid having free variables in a proof environment alias. (Precisely, the aim is for them not to alias at extracted code runtime.) If variables were aliased at runtime then changing the value of any of them would change the value of all of them, so that none of them could be

referenced. Thus anytime a variable was modified in the proof, all its potential runtime aliases would have to be marked **logical** (in addition to the variable itself). If there were no restrictions on how expressions could alias, identifying potential aliases would be very difficult; forbidding procedure arguments from aliasing eliminates the problem: in any proof environment, if two variables or expressions are syntactically distinct, then they do not alias.[11]

There are two ways to make expressions alias in the new logic. One is with procedures; for example, $update(a, i, e)$ aliases with a. The other is with $\forall\exists$ formulas. In the quicksort proof of the previous section, a became aliased with b when a was used to eliminate a formula like

$$\text{All var } a \,.\, \text{Some var } b \,.\, F \quad (\text{aliases } (a, b))$$

as in

$$\begin{array}{lll} [\text{var } a] \ \text{All var } a \,.\, \text{Some var } b \,.\, F \ (\text{aliases } (a,b)) \gg G & \text{by elim } 1, a & \\ [\text{log } a] \ \text{All } \ldots, \text{Some var } b \,.\, F \gg G & \text{by elim } 2, b & (4.10) \\ [\text{var } b, \text{log } a] \ \text{All } \ldots, F \gg G. & & \end{array}$$

Introducing and eliminating $\forall\exists$ formulas are the only ways that identifiers can become directly aliased in the new logic. They are also the only

[11] Strictly speaking, keeping procedure arguments distinct does not eliminate the problem alone; other parts of the new logic, like the import rule, the lack of side effects, and the form of the rules, are also important.

way that an identifier can become aliased with the witness of an existential. In general, a proof of

$$[\mathsf{var}\ a]\ \mathcal{H} \gg \mathsf{Some\ var}\ b\,.\,F$$

may *not* alias b to a. The exception is when such a goal is part of a proof of an $\forall\exists$ formula, like

$$\mathcal{H} \gg \mathsf{All\ var}\ a\,.\,\mathsf{Some\ var}\ b\,.\,F \quad \text{by intro } a$$
$$\bullet\ [\mathsf{var}\ a]\ \mathcal{H} \gg \mathsf{Some\ var}\ b\,.\,F.$$

To emphasize the use of $\forall\exists$ formulas, the new logic uses a derived rule, *skolem*, to describe the elimination of $\forall\exists$ formulas.[12] Proof (4.10) above becomes

$$[\mathsf{var}\ a]\ \mathsf{All\ var}\ a\,.\,\mathsf{Some\ var}\ b\,.\,F\ (\mathsf{aliases}\ (a,b))$$
$$\gg G \quad \text{by skolem } 1,a,b$$
$$[\mathsf{var}\ b, \log\ a]\ \mathsf{All}\ \ldots, F \gg G.$$

$\forall\exists$ formulas are used this way for two reasons. First, restricting the aliasing of variable existential witnesses to $\forall\exists$ formulas promotes using local properties of a proof: to modify an array, you do two quick eliminations or one skolem. If more complicated formulas were used, there could be an

[12]The rule is called skolem because $\forall\exists$ formulas are also called Skolem formulas [BeM, End,Sch]. Derived rules are easy to define in PRL with LCF-like tactics [CKB,Pan].

arbitrarily long distance between the ***all-elim*** and the ***some-elim***: consider a hypothesis like

$$\mathsf{All\ var}\ a\,.\,(F_1\ \&\ (F_2\ \&\ \ldots(F_n\ \&\ \mathsf{Some\ var}\ b\,.\,G)\ldots)).$$

A second reason for using $\forall\exists$ formulas is simplicity: the allowable combinations of All and Some that could possibly alias can be put into $\forall\exists$ normal form. Say that Some var $b\,.\,F$ is embedded in the body of a universal, All var $a\,.\,G$. Because of the side effect and import rules, b cannot alias with a if the existential is in the body of some intervening universal, as in

$$\mathsf{All\ var}\ a \ldots \mathsf{All} \ldots \mathsf{Some\ var}\ b\,.\,F\ldots$$

Similarly, the existential may not be part of the conclusion of an implication, as in

$$\mathsf{All\ var}\ a\,.\,(\ldots \Rightarrow \mathsf{Some\ var}\ b\,.\,F\ldots)$$

Thus, the existential must be a disjunct or conjunct, like

$$\mathsf{All\ var}\ a\,.\,(\ldots\ \&\ \mathsf{Some\ var}\ b\,.\,F\ldots).$$

Such formulas can be easily converted into $\forall\exists$ form, possibly after renaming. For the formula above, the result would be

$$\mathsf{All\ var}\ a\,.\,\mathsf{Some\ var}\ b\,.\,(\ldots\ \&\ F\ldots).$$

To conclude, there are two basic aliasing restrictions. First, free variables may not alias. (In particular, procedure parameters may not alias.)

Second, existential variables may alias only with universal variables, and only in $\forall\exists$ formulas. These two restrictions interact when quantifiers act over multiple arrays. For example, given a formula **All var** a, a'.**Some var** $b.F$, there are only two acceptable aliases clauses involving b: aliases (a, b) and aliases (a', b). The clause aliases $(\langle a, b\rangle, \langle a', b\rangle)$ is unacceptable because it implies that a and a' might both alias with b, in which case a and a' would alias.

Summary

To be efficient, extracted code must pass arrays by reference and modify them in place. Expanding the logic to include these features poses design problems because the more expressive the new logic is, the more complex (and less usable and understandable) it becomes. Restrictions on expressiveness are necessary to keep the logic managable, but too many restrictions will make the logic uninteresting.

This section discussed answers to questions that were raised in section one about the restrictions of the new logic. Here is a recap of the questions and answers. (For "procedure" below, read "all-justification, lemma justification-transformer, implication-justification, or explicitly defined procedure.")

Q.: When can a procedure modify an array parameter or an array embedded inside a parameter?

A.: All-justifications and explicitly defined procedures can modify var parameters but not val parameters. Implication-justifications and lemma justification-transformers can modify neither array parameters nor arrays embedded inside parameters.

Q.: Can a procedure have side effects?

A.: No.

Q.: How are modified values handled in the logic?

A.: Only free local variables can be modified, not the witnesses of existential hypotheses, and when they are, they become logical. The logic prohibits references to modified values by banning computational references to logical variables.

Q.: How is aliasing handled?

A.: Aliasing is not explicitly discussed in the logic. That is, there are no proof rules for assuming or proving that variables are aliased. In addition, aliasing is not allowed to affect the provability of predicates. The logic guarantees that syntactically distinct variables do not alias, except in certain well-defined ways.

Q.: How is aliasing restricted?

A.: First, free variables (and procedure parameters in particular) may not alias. Second, existential variables can only alias with universal variables, and in only $\forall\exists$ formulas. That is, a **Some var** b can only alias with an **All var** a, and only in a context like **All var** a.**Some var** $b.F$.

Chapter 5

A call-by-reference logic for arrays

This chapter describes the new call-by-reference array logic in detail. Certain justifications are redefined and various proof rules are modified. Extraction is mostly unchanged, though the semantics of extracted code changes. As discussed in chapter 4, new fields are added to proof environments and formulas, and restrictions are used in the rules to control them.

5.1 Call-by-reference justifications

The basic change to justifications is that they are no longer value-oriented: they are state-oriented. Identifiers are no longer bound to values, they are bound to ***locations***, and locations are mapped to values by ***states***. In typical call-by-reference fashion, variables are passed to functions by passing their locations; functions discover the values of locations by consulting the state, and functions change the values of variables by changing the state.

Definition 5.1 $venv$ is a ***value environment***: it maps identifiers to values. (So value environments are the environments of chapter 2.) $renv$ is a ***reference environment***: it maps identifiers to locations. st is a ***state***: it maps locations to values. Loc is the set of all locations. St is the set of all states. States and environments are partial functions.

The meanings of procedures change when states are introduced: a procedure f of type $T \rightarrow T'$ becomes a function f' of type $\mathsf{Loc} \times \mathsf{St} \rightarrow \mathsf{Loc} \times \mathsf{St}$. If $f(a)$ is b and $st(loc)$ represents a, then $f'(loc, st)$ is some (loc', st') such that $st'(loc')$ represents b. For conciseness I will sometimes simply say that f' is a function of type $T \rightarrow T'$ instead of saying that it is a function of type $\mathsf{Loc} \times \mathsf{St} \rightarrow \mathsf{Loc} \times \mathsf{St}$ that maps locations of values of type T to locations of values of type T'.

The definition of ***agreement*** that follows is used in the new definition of justification.

Definition 5.2 Given a set of locations $\mathcal{L}$, a state st agrees with another state st' modulo $\mathcal{L}$ if $dom(st') \supseteq dom(st)$ and st and st' are equal on their restrictions to $dom(st) - \mathcal{L}$. They agree modulo a set of identifiers $\mathcal{V}$ if they agree modulo $\{renv(x) \mid x \in \mathcal{V}\}$. Given $renv$, a reference environment, st' agrees with st modulo a formula F if they agree modulo the free non-logical identifiers of F.

Notation. $st' \supseteq st \pmod{\mathcal{L}}$ means that st agrees with st' modulo $\mathcal{L}$. Similar notation is used for agreement modulo a set of identifiers $\mathcal{V}$ or a formula F.

The definition of agreement of states is also used to help prove the extractor correct: it is necessary that if γ justifies F in state st, it also justifies F in any state $st' \supseteq st \pmod{F}$.

For simplicity, the definition of call-by-reference justifications does not have singleton formulas removed. In addition, function calls and other operations are omitted from formulas: the only terms that appear in formulas are identifiers. Both restrictions can be easily removed.

Definition 5.3 Let F be a formula, st be a state, and $renv$ be a reference environment. Let $venv$ be a value environment that equals $st \circ renv$ on the free val and var identifiers of F, then $\gamma, st, renv \Vdash F, venv$ means that justification γ justifies F in state st and environments $renv$ and $venv$. If e is an expression, then $e, st, renv \Vdash F, venv$ means that the meaning of e in reference environment $renv$ and state st justifies F in st, $renv$, and $venv$. The justifications of F are defined by induction on the structure of F. The cases are presented below.

- ***F is atomic.***

  $triv, st, renv \Vdash$ true, $venv$ regardless of st and $renv$. $triv, st, renv \Vdash x = y, venv$ iff the values of x and y are equal. The value of x is $venv(x)$ and the value of y is $venv(y)$. (Note that if an identifier x is val or var, then $venv(x) = st(renv(x))$.) Justifications for $x < y$, $x \leq y$, and so on are also defined similarly.[1]

- ***F is a conjunction.***

  $(\gamma_1, \gamma_2), st, renv \Vdash F_1 \mathbin{\&} F_2, venv$ iff $\gamma_1, st, renv \Vdash F_1, venv$ and $\gamma_2, st, renv \Vdash F_2, venv$.

- ***F is a disjunction.***

  $(\tau, \gamma'), st, renv \Vdash F_1 \mid F_2, venv$ iff τ is 1 or 2 and $\gamma', st, renv \Vdash F_\tau, venv$.

[1] Note that if $triv, st, renv \Vdash x = y, venv$, then $triv, st', renv \Vdash x = y, venv$ for any $st' \supseteq st \pmod{x = y}$, whether x and y are val, var, or logical.

- ***F is an implication.***

  $\gamma, st, renv \Vdash F_1 \Rightarrow F_2, venv$ iff γ is a function from justifications for F_1 to justifications for F_2. Since γ must justify F in any state $st' \supseteq st \pmod F$, γ must accept justifications for F_1 in any such st'. As discussed in chapter 4, γ must be side-effect-free, so the state returned by γ must be $\supseteq st'$.

  The full definition therefore is $\gamma, st, renv \Vdash F_1 \Rightarrow F_2, venv$ iff γ is a function of type Loc$\times$St $\rightarrow$ Loc$\times$St such that for all $st' \supseteq st \pmod F$ and all locations loc, if $st'(loc), st', renv \Vdash F_1, venv$, then $\gamma(loc, st')$ is some (loc', st'') such that $st'' \supseteq st'$, loc' is in $dom(st'') - dom(st')$, and $\gamma', st'', renv \Vdash F_2, venv$, where γ' is $st''(loc')$. In addition, for each loc_i embedded in γ', where loc_i corresponds to a var existential witness (see below), loc_i must be in $dom(st'') - dom(st')$.

- ***F is an existential.***

  $(loc, \gamma'), st, renv \Vdash$ Some $x : T.G, venv$ iff $st(loc)$ is a value v of type T and

  $$\gamma', st, renv[x \leftarrow loc] \Vdash G, venv[x \leftarrow \mathrm{v}].$$

  If $renv(x)$ is defined, then pick some new identifier y; γ' should justify $G[y/x]$ in st, $renv[y \leftarrow loc]$, and $venv[y \leftarrow \mathrm{v}]$. Existentials with more than one bound identifier are defined similarly. If F has multiple var identifiers, then the corresponding witness locations must be pairwise disjoint.

- ***F is a universal.***

Say F is All val $x_1 : T_1$, var $x_2 : T_2 . G$. (If G is an existential, then some special restrictions apply, in addition to the ones here. See the clause below on $\forall\exists$ formulas.) For γ to justify F, γ must be a function that maps values of types T_1 and T_2 to a justification for G. As discussed in chapter 4, γ must be side-effect-free and may not modify x_1, though it may modify x_2. Like an implication justification, γ must work in any state $st' \supseteq st$ (mod F).

The full definition therefore is $\gamma, st, renv \Vdash F, venv$ iff for all states $st' \supseteq st$ (mod F) and locations loc_1 and loc_2, if $loc_1 \neq loc_2$ and each $st'(loc_i)$ is a value v_i of type T_i, then $\gamma(st', loc_1, loc_2)$ is some (loc', st'') such that $st'' \supseteq st'$ (mod $\{x_2\}$) and

$$st''(loc'), st'', renv[x_1, x_2 \leftarrow loc_1, loc_2] \Vdash G, venv[x_1, x_2 \leftarrow v_1, v_2].$$

As with existentials, if x_1 or x_2 appear in the domain of $renv$, then renaming should be done.

Universals with longer lists of bound identifiers are defined similarly, with one further restriction. Say F is All var $x_1, \ldots, x_n : T . G$. No call of γ is supposed to have the x_i alias, so in addition to requiring that each loc_i be the location of a value of type T, there is the

requirement that the loc_i must be pairwise disjoint. Keep in mind that in any case, locations of var identifiers must be distinct from locations of val identifiers.

- ***F is an $\forall\exists$ formula.***

Let F be

$$\begin{array}{l} \text{All val } x_{11} : T_{11}, \ldots, x_{1m} : T_{1m}, \text{var } x_{21} : T_{21}, \ldots, x_{2n} : T_{2n} . \\ \quad \text{Some val } x_{31} : T_{31}, \ldots, x_{3p} : T_{3p}, \text{var } x_{41} : T_{41}, \ldots, x_{4q} : T_{4q} . G \\ \qquad (\text{aliases } (x_{2k_1}, x_{4l_1}), \ldots, (x_{2k_r}, x_{4l_r})). \end{array}$$

For γ to justify F, it must be an all justification, of course, but it must also obey an aliasing restriction. Let st' be a state that agrees with st modulo F, and let $loc_{11}, \ldots, loc_{1m}$ and $loc_{21}, \ldots, loc_{2n}$ be locations of values of the appropriate types. (Recall that in addition the loc_{2i} must be pairwise disjoint.) Let

$$\gamma(loc_{11}, \ldots, loc_{1m}, loc_{21}, \ldots, loc_{2n}, st')$$

be (loc, st''), where $st''(loc)$ is an existential justification, say

$$\langle loc_{31}, \ldots, loc_{3p}, loc_{41}, \ldots, loc_{4q}, \gamma' \rangle.$$

The aliasing restriction for $\forall\exists$ formulas is that the var existential witnesses may alias only with the appropriate x_{2j} or with fresh locations. That is, for $i = 1, 2, \ldots, q$, loc_{4i} must either be equal to some loc_{2j}, where (loc_{2j}, loc_{4i}) is a member of the transitive closure of the ($renv$ of the) aliases list, or loc_{4i} must be in $dom(st'') - dom(st')$.

This concludes the definition of state-oriented justifications. To be an appropriate notion of justification, it must satisfy the validity theorem.

Theorem 5.1 *(Justification validity, for call-by-reference justifications.)* Let F be a formula, st be a state, and $renv$ be a reference environment. Define $venv_0$ to be the restriction of $st \circ renv$ on the free val and var identifiers of F. Then for all $venv \supseteq venv_0$, $\models F, venv$ iff there exists a justification γ and a state $st' \supseteq st \pmod{F}$ such that $\gamma, st', renv \Vdash F, venv$.

Proof

By induction on the structure of F. Instead of proving necessity and sufficiency separately, the equivalence is proved simultaneously. (Certain cases require it.) In addition, for each kind of formula, the statement is proved for ***all*** compatible states and environments.

The only interesting cases are implications and universals, since they are functions. The argument for disjunctions is also given, in order to show the general pattern. The other easy cases are omitted.

- ***F is a disjunction.***

For sufficiency, assume that $\models F_1 \mid F_2, venv$ holds. We must find a justification γ and a state $st' \supseteq st \pmod{F}$ such that $\gamma, st', renv \Vdash F_1 \mid F_2$. Since $\models F_1 \mid F_2, venv$ holds, $\models F_k, venv$ holds for $k = 1$ or 2. By inductive hypothesis, there exists a justification γ_k and a state $st' \supseteq st \pmod{F_k}$ such that $\gamma_k, st', renv \Vdash F_k, venv$. Let γ be (k, γ_k), then $\gamma, st', renv \Vdash F_1 \mid F_2$ by the definition of justification. Since st' agrees with st modulo F_k, it also agrees modulo $F_1 \mid F_2$.

For necessity, assume that $\gamma, st, renv \Vdash F_1 \mid F_2, venv$. We must show $\models F_1 \mid F_2, venv$. By definition of justification, γ is some pair (k, γ_k), where $\gamma_k, st, renv \Vdash F_k, venv$. By inductive hypothesis, $\models F_k, venv$, so it follows that $\models F_1 \mid F_2, venv$.

- ***F is an implication.***

For sufficiency, assume that $\models F_1 \Rightarrow F_2, venv$ holds. We must find a function f that maps locations of justifications for F_1 to locations of justifications for F_2 in any state $st' \supseteq st \pmod{F}$. Say there exists a location loc such that $st'(loc), st', renv \Vdash F_1, venv$. By induction hypothesis $\models F_1, venv$, so by the definition of validity, $\models F_2, venv$. By induction hypothesis in the other direction, there exists a justification γ' and a state $st'' \supseteq st' \pmod{F_2}$ such that $\gamma', st'', renv \Vdash F_2, venv$. Define a function f by

$$f(loc, st') = \left\langle loc', st''\left[loc' \leftarrow \gamma'\right]\right\rangle$$

where loc' is a new location (one not in $dom(st')$). It is easy to verify that f justifies $F_1 \Rightarrow F_2$ in st', $renv$, and $venv$. (f might be uncomputable, but that is not important here.) Finally, $st'' \supseteq st'$ (mod F_2) $\supseteq st$ (mod F_1), so $st'' \supseteq st$ (mod $F_1 \Rightarrow F_2$).

For necessity, assume that $\gamma, st', renv \Vdash F_1, venv$, where $st' \supseteq st$ (mod F). We must show that $\models F_1 \Rightarrow F_2, venv$. Assume that $\models F_1, venv$. By induction hypothesis, there exists a justification γ_1 and a state $st'' \supseteq st'$ (mod F_1) such that $\gamma_1, st'', renv \Vdash F_1, venv$. Let loc be a new location, then by definition of justification, $\gamma(loc, st''[loc \leftarrow \gamma_1])$ is some pair $(loc', \hat{st})$, where $\hat{st}(loc'), \hat{st}, renv \Vdash F_2, venv$. By induction hypothesis in the other direction, $\models F_2, venv$.

- ***F is a universal.***

For sufficiency, first let F be All val $x_1 : T_1$; var $x_2 : T_2 . G$, and assume that $\models F, venv$. We must define a function f that maps locations of values for the x_i to justifications of G. If $st' \supseteq st$ (mod F) and each $st'(loc_i)$ is a value v_i of type T_i, then by definition of validity, $\models F, venv[x_1, x_2 \leftarrow v_1, v_2]$. By induction hypothesis, there exists a justification γ and a state $st'' \supseteq st'$ (mod G) such that

$$\gamma, st'', renv[x_1, x_2 \leftarrow loc_1, loc_2] \Vdash G, venv[x_1, x_2 \leftarrow v_1, v_2].$$

Define a function f by

$$f(loc_1, loc_2, st') = \langle loc, st''[loc \leftarrow \gamma] \rangle$$

where loc is a new location. It is easy to verify that f justifies F in st', $renv$, and $venv$.

For necessity, say that $\gamma, st', renv \Vdash F, venv$. We must show that $\models F, venv$. Let loc_1 and loc_2 be new locations, then for all values v_1 and v_2 of types T_1 and T_2 respectively,

$$\gamma\left(loc_1, loc_2, st'[loc_1, loc_2 \leftarrow v_1, v_2]\right)$$

is some (loc, st''), where $st'' \supseteq st'$ (mod $\{x_2\}$) and

$$st''(loc), st'', renv[x_1, x_2 \leftarrow loc_1, loc_2] \Vdash G, venv[x_1, x_2 \leftarrow v_1, v_2].$$

By induction hypothesis, $\models G, venv[x_1, x_2 \leftarrow v_1, v_2]$. Since G is valid for all values of x_1 and x_2, it follows that $\models F, venv$.

For other universals with more quantified identifiers, necessity and sufficiency are proved similarly. □

State-oriented and value-oriented justifications have very different structures, but they still serve the same purpose — they give a semantics to proofs. In the same way, proofs still denote justification transformers, and constructive proofs still denote computable justification transformers. As will be seen in the next section, the purpose of the extractor is unchanged: it translates proofs into more-easily executed programs, in a meaning-preserving way.

5.2 Call-by-reference proof rules

The proof rules for the call-by-reference version of the logic are given in this section. Syntactically, the main difference between the logic presented here and the one presented in chapter 2 is the use of new fields to track and control variable identifiers.

As discussed in chapter 4, the val, var, and log fields in proof environments and quantified formulas specify which identifiers are variables and which variables may be referenced and modified: val (value) identifiers are not variables, so they may be referenced but not modified, var (variable) identifiers are variables that may be referenced and modified, and log (logical) identifiers are variables that may neither be referenced nor modified.

Three new fields, aliases, aliased, and fresh, are used to describe the aliasing characteristics of a proof. The aliases field was discussed in chapter 4; the others were not. The aliases field is used in $\forall\exists$ formulas to describe the aliasing that can occur between universal and existential variables. If

$$\begin{array}{l} \text{All val } x_{11}:T_{11},\ldots,x_{1m}:T_{1m},\ \text{var } x_{21}:T_{21},\ldots,x_{2n}:T_{2n}\,. \\ \quad \text{Some val } x_{31}:T_{31},\ldots,x_{3p}:T_{3p},\ \text{var } x_{41}:T_{41},\ldots,x_{4q}:T_{4q}\,.\,G \\ \qquad (\text{aliases } (x_{2k_1},x_{4l_1}),\ldots,(x_{2k_r},x_{4l_r})) \end{array}$$

is skolem-eliminated with $y_{11},\ldots,y_{1m},y_{21},\ldots,y_{2n}$ for the universal's arguments, then y_{2k_1} can alias with x_{4l_1}, and so on. Once an *all-intro* is

done, the aliases field describes the aliasing that can occur between the introduced variables and the existential witnesses.

The aliased and fresh fields are used in proof environments. They control the identifiers that can appear as var witnesses in *some-intro* rules: in

$$\begin{array}{l} [\ldots,\text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr}] \\ \quad \mathcal{H} \gg \text{Some var } x:T\,.\,F \quad (\text{aliases } (x,z)) \quad \text{by intro } y \end{array}$$

the introduced identifier, y, must be fresh (a member of $\mathcal{V}_{fr}$), or aliased to z (related to z in the transitive reflexive closure of $\mathcal{A}$).

The fresh variables are those whose locations are newly created during a call. The aliased field lists the pairs of aliased var and log identifiers of an environment. Since the variable witnesses of an existential are only supposed to alias with the appropriate variable universals or with fresh locations, it is important to keep track of the aliased and fresh variables.

Much of the notation used in this chapter is the same as that for chapter 2. It is listed here again for easier reference.

Notation. π_{rule} is the specified proof rule. For example, $\pi_{and\text{-}intro}$ is the *and-intro* proof rule. If π_{rule} has only one subproof, then σ is that subproof. If π_{rule} has more than one subproof, then σ_1 is the first subproof, σ_2 is the second, and so on.

Notation. H is a hypothesis. $\mathcal{H}$ and $\mathcal{H}'$ are lists of hypotheses $H_1, \ldots, H_{n-1}$ and $H_{n+1}, \ldots, H_N$ respectively. J, K, L, and M are justification identifiers. $\mathcal{J}$ and $\mathcal{J}'$ are lists of justification identifiers $J_1, \ldots, J_{n-1}$ and $J_{n+1}, \ldots, J_N$ respectively; typically, J_i is a justification identifier for hypothesis H_i.

Notation. The extraction function is X; it maps a proof and a list of justification identifiers to a program. $\mathsf{X}[\pi, J_1, \ldots, J_n]$ means $\mathsf{X}[\pi, (J_1, \ldots, J_n)]$, and $\mathsf{X}[\pi, \mathcal{J}]$ means $\mathsf{X}[\pi, J_1, \ldots, J_n]$.

Some notation dealing with sets of identifiers and with sets of pairs of identifiers is new to this section.

Notation. $\mathcal{V}$ is a set of identifiers. $\mathcal{V} + x_1, \ldots, x_n$ is $\mathcal{V} \cup \{x_1, \ldots, x_m\}$, and $\mathcal{V} - x_1, \ldots, x_n$ is $\mathcal{V} - \{x_1, \ldots, x_n\}$. $\mathcal{A}$ is a set of pairs of identifiers. $\mathcal{A}^*$ is the transitive reflexive closure of $\mathcal{A}$.

The notation used here for proof environments is different from that used in chapter 2.

Notation.

> $[\mathcal{E}]$ is [val $\mathcal{V}_{val}$; var $\mathcal{V}_{var}$; log $\mathcal{V}_{log}$; aliased $\mathcal{A}$; fresh $\mathcal{V}_{fr}$].
> $[\mathcal{E}']$ is [val $\mathcal{V}_{val}$; log $\mathcal{V}_{log} + \mathcal{V}_{var}$; aliased $\mathcal{A}$].
> close ($\mathcal{E}$) means close ($\mathcal{V}_{val}$).

Type information is still included in environments but is suppressed here for readability.

In certain subproofs, computational use of outside variables (variables from the main proof) are banned altogether. This means not only that var environment identifiers may not be used, it also means that certain hypotheses like Some var $x : T . F$ may not be used in *some-elim* or *hypothesis* rules. More specifically, a hypothesis with a variable existential witness may not be used in a subproof if the value of the witness depends on the state. Because of the way justifications are defined, embedded witnesses like the x in $F \Rightarrow$ Some var $x : T . G$ *can* be used, since the location of x is independent of the state.

Definition 5.4 The witnesses of a formula that are ***independent of the state*** are found as follows:

- The independent witnesses of an existential are its val witnesses plus the independent witnesses of its body.
- The independent witnesses of a conjunction or disjunction are those of its subformulas.
- All the witnesses of a implication, negation, or universal are independent, even the var witnesses of embedded existentials. (So $F \Rightarrow G$ is state-independent, even if F and G separately are not.)

- An atomic formula has no independent witnesses.

A formula is ***state-dependent*** if any of its var witnesses are dependent on the state. It is ***state-independent*** if none of its var witnesses are dependent on the state.

Notation. $\mathcal{H}_{indep}$ is the state-independent sublist of $\mathcal{H}$. Similarly, $\mathcal{H}'_{indep}$ is the state-independent sublist of $\mathcal{H}'$.

Extracted code in this chapter is almost syntactically identical to that of chapter 2. Descriptions of extracted code are omitted except when the changes, if any, are not obvious.

Theorems

A PRL theorem is a named proof. For modularity, each theorem extracts to a justification-transforming procedure, so the bans on side effects, importing, and parameter-modifying force the following restrictions on theorems:

- They may not have state-dependent hypotheses.

- They must have empty var, log, aliased, and fresh fields.

Thus, a typical theorem has the goal [val $\mathcal{V}_{val}$] $\hat{\mathcal{H}} \gg F$ where all the hypotheses in $\hat{\mathcal{H}}$ are state-independent.

Propositional calculus rules

The ***and-intro*** proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F\ \&\ G \quad \text{by intro}$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep} \gg F$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep} \gg G.$$

In both subproofs, the members of $\mathcal{V}_{var}$ are logical, so neither proof's extracted code can modify the values of an x in $\mathcal{V}_{var}$. This guarantees that neither subproof's code can interfere with the other's. For the same reason, the state-dependent members of $\mathcal{H}$ and $\mathcal{H}'$ are dropped in the subproofs and the fresh field is made null.

The members of $\mathcal{V}_{var}$ cannot be value identifiers in the subproofs. This would enable the extracted code for the subproofs to close over their locations when building procedures, and closing over the locations of value identifiers was banned in chapter 4.

The ***and-elim*** proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ F\ \&\ G,\ \mathcal{H}' \gg F' \quad \text{by elim } n$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ F,\ G \gg F'.$$

The environment remains unchanged: splitting apart the justification for $F\ \&\ G$ does not change the status of any variable identifiers.

The *or-intro* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F_1 \mid F_2 \quad \text{by intro } k$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H} \gg F_k$$

where k is a constant. The state-dependent hypotheses are not removed so they can be available to the proof of F_k.

The *or-elim* proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ F \mid G,\ \mathcal{H}' \gg F' \quad \text{by elim } n$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ F \gg F'$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ G \gg F'.$$

The environment is not modified, and the state-dependent hypotheses are not removed. Since only one or the other branch of the proof will have its extracted code executed, the effects of the executed branch cannot affect the other.

The *impl-intro* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F \Rightarrow G \quad \text{by intro}$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep},\ F \gg G$$

where F is state-independent. The members of $\mathcal{V}_{var}$ are made logical, since they must not be available to the extracted code for the subproof. (Free variable identifiers may not be imported by procedures, and the extracted

code for the subproof forms a new procedure body.) The state-dependent hypotheses are removed from the hypotheses, since they also must not be available to the extracted code for the subproof. (Procedures may not modify state-dependent justifications.) The fresh field is made null, since a new procedure is being opened.

The *impl-elim* proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ F \Rightarrow G,\ \mathcal{H}' \gg F' \quad \text{by elim } n$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep},\ \mathcal{H}'_{indep} \gg F$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ F,\ G \gg F'.$$

The justification for F is built in an environment that cannot corrupt the environment of the main goal. In particular, the var identifiers of the main goal and the state-dependent hypotheses of $\mathcal{H}$ and $\mathcal{H}'$ cannot be used.

The *not-intro* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg \neg F \quad \text{by intro}$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep},\ F \gg \text{false}.$$

As in chapter 2, the *not-intro* rule is similar to the *impl-intro* rule.

The *not-elim* proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ \neg F,\ \mathcal{H}' \gg G \quad \text{by elim } n$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep},\ \mathcal{H}'_{indep} \gg F$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ F,\ \text{false} \gg G.$$

As in chapter 2, the *not-elim* rule is similar to the *impl-elim* rule. Here, this means that the proof of F is done in an environment that does not interfere with $\mathcal{E}$. In some sense the second subgoal is unnecessary, since *false-elim* on it (see below) would prove G directly.

The *false-intro* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg \text{false} \quad \text{by intro } F$$

- $[\mathcal{E}']\ \mathcal{H}_{indep} \gg F$
- $[\mathcal{E}']\ \mathcal{H}_{indep} \gg \neg F$.

Since the extracted code for the proofs of F and $\neg F$ are executed sequentially, the proofs are done in environments that do not interfere with the var identifiers and state-dependent hypotheses of $\mathcal{E}$. This is a bit of a moot point here, since the extracted code will never be run. The no-interference restriction thus could actually be relaxed for this rule.

The *false-elim* proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ \text{false},\ \mathcal{H}' \gg F \quad \text{by elim } n.$$

The extracted code for the proof rule, like the code for the *false-intro* rule, is never executed.

Quantifier rules

Notation. $\mathcal{X}$ is $x_1, \ldots, x_p$, $\mathcal{Y}$ is $y_1, \ldots, y_q$, $\mathcal{U}$ is $u_1, \ldots, u_p$, and $\mathcal{V}$ is $v_1, \ldots, v_q$.

The *all-intro* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg \text{All val } x_1 : T_1, \ldots, x_p : T_p; \text{var } y_1 : T_1', \ldots, y_q : T_q' .\ F$$
$$\text{(aliases } \mathcal{A}')$$
$$\text{by intro } \mathcal{U}, \mathcal{V}$$

- $[\text{val } \mathcal{V}_{val} + \mathcal{U};\ \text{var } \mathcal{V};\ \text{log } \mathcal{V}_{log} + \mathcal{V}_{var};\ \text{aliased } \mathcal{A}]$
  $\mathcal{H}_{indep} \gg F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}] \quad (\text{aliases } \mathcal{A}'[\mathcal{V}/\mathcal{Y}])$.

The u_i and v_j must be distinct new identifiers. As in the *impl-intro* rule, the extracted code for the subproof forms the body of a new procedure, so to keep the procedure from importing the members of $\mathcal{V}_{var}$, the members of $\mathcal{V}_{var}$ are made logical. The state-dependent hypotheses are removed, since they also may not be imported by the extracted procedure.

The aliases $\mathcal{A}'$ field above should be present only if F is an existential. The field may be omitted if $\mathcal{A}'$ is empty. If the field is present, then $\mathcal{A}'$ may not mention any identifier more than once. That is, the transitive closure of $\mathcal{A}'$ must equal $\mathcal{A}'$.

For simplicity, the *all-elim* rule is restricted here for use only on *non*-$\forall\exists$ formulas. The *skolem* rule is used for $\forall\exists$ formulas.[2] The *all-elim* proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ \text{All val } x_1 : T_1, \ldots, x_p : T_p; \text{var } y_1 : T'_1, \ldots, y_q : T'_q \,.\, F,$$
$$\mathcal{H}' \gg G \quad \text{by elim } n, \mathcal{U}, \mathcal{V}$$

- $[\text{val } \mathcal{V}_{val};\ \text{var } \mathcal{V}_{var} - \mathcal{V};\ \text{log } \mathcal{V}_{log} + \mathcal{V};\ \text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr} - \mathcal{V}]$
  $$\mathcal{H},\ \text{All } \ldots,\ \mathcal{H}',\ F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}] \gg G.$$

Restrictions: (1) The members of $\mathcal{V}$ must be distinct. (2) For all i, u_i must be in $\mathcal{V}_{val} \cup \mathcal{V}_{var}$, and if u_i is in $\mathcal{V}_{var}$ then u_i may not be in $\mathcal{V}$; in addition, if any u_i is a member of $\mathcal{V}_{var}$ then F cannot have embedded universals, existentials, or implications. This second restriction amounts to saying that if a var identifier is used to eliminate a val universal, then it may not also be used to eliminate a var universal, and there must also be no chance of it being imported by a procedure or returned as a val witness.

The *some-intro* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg \text{Some val } x_1 : T_1, \ldots, x_p : T_p; \text{var } y_1 : T'_1, \ldots, y_q : T'_q \,.\, F$$
$$(\text{aliases } \mathcal{A}')$$
$$\text{by intro } \mathcal{U}, \mathcal{V}$$

[2] The quicksort proof in chapter 4 used combinations of the *all-elim* and *some-intro* rules instead of the *skolem* rule because that chapter aimed for a more general discussion of the issues.

- $[\text{val } \mathcal{V}_{val};\ \text{var } \mathcal{V}_{var} - \mathcal{V};\ \text{log } \mathcal{V}_{log} + \mathcal{V};\ \text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr} - \mathcal{V}]$
  $$\mathcal{H} \gg F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}].$$

For all i and j, u_i must be in $\mathcal{V}_{val}$ and v_j must be in $\mathcal{V}_{var}$. The v_j must be distinct. If the aliases field is not present, then all the u_j must be in $\mathcal{V}_{fr}$. If the aliases field is present then for each j, either v_j is in $\mathcal{V}_{fr}$ or it must be related in $\mathcal{A}^*$ to some z_i where (z_i, y_j) is in $\mathcal{A}'$. (One or the other must hold, but not both.)

Above, the members of $\mathcal{V}$ are made logical in σ so that their values may not be used by σ.

The *some-elim* proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ \text{Some val } x_1 : T_1, \ldots, x_p : T_p; \text{var } y_1 : T'_1, \ldots, y_q : T'_q \,.\, F,$$
$$\mathcal{H}' \gg G \quad \text{by elim } n, \mathcal{U}, \mathcal{V}$$

- $[\text{val } \mathcal{V}_{val} + \mathcal{U};\ \text{var } \mathcal{V}_{var} + \mathcal{V};\ \text{log } \mathcal{V}_{log};\ \text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr} + \mathcal{V}]$
  $$\mathcal{H},\ \mathcal{H}',\ F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}] \gg G.$$

The u_i and v_j must be distinct new identifiers.

The *skolem* rule combines the *all-elim* and *some-elim* for an $\forall\exists$ formula. Let $\mathcal{X}'$ be $x'_1, \ldots, x'_r$, $\mathcal{Y}'$ be $y'_1, \ldots, y'_s$, $\mathcal{U}'$ be $u'_1, \ldots, u'_r$, and $\mathcal{V}'$ be $v'_1, \ldots, v'_s$. The *skolem* rule is

$$\begin{array}{l} [\mathcal{E}]\ \mathcal{H},\ \text{All val } x_1 : T_{11}, \ldots, x_p : T_{1p}; \text{var } y_1 : T_{21}, \ldots, y_q : T_{2q}\,. \\ \qquad \text{Some val } x'_1 : T_{31}, \ldots, x'_r : T_{3r}; \text{var } y'_1 : T_{41}, \ldots, y'_s : T_{4s}\,.\, F \\ \qquad (\text{aliases } \mathcal{A}'),\ \mathcal{H}' \gg G \quad \text{by skolem } n, \mathcal{U}, \mathcal{V}, \mathcal{U}', \mathcal{V}' \\ \bullet\ \big[\text{val } \mathcal{V}_{val} + \mathcal{U}';\ \text{var } \mathcal{V}_{var} + \mathcal{V}' - \mathcal{V};\ \text{log } \mathcal{V}_{log} + \mathcal{V}; \\ \qquad \text{aliased } \mathcal{A} \cup \mathcal{A}'\left[\mathcal{V}, \mathcal{V}'/\mathcal{Y}, \mathcal{Y}'\right];\ \text{fresh } \mathcal{V}_{fr} + \mathcal{V}'' - \mathcal{V}\big] \\ \qquad \mathcal{H},\ \text{All} \ldots,\ \mathcal{H}',\ F\left[\mathcal{U}, \mathcal{V}, \mathcal{U}', \mathcal{V}'/\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}'\right] \gg G \end{array}$$

where $\mathcal{V}''$ is the the v'_i not mentioned in $\mathcal{A}'\left[\mathcal{V}, \mathcal{V}'/\mathcal{Y}, \mathcal{Y}'\right]$. If the aliases field is not present, $\mathcal{A}'$ should be treated as the null set. For all i and j, u_i must be in $\mathcal{V}_{val}$ and v_j must be in $\mathcal{V}_{var}$. The v_j must be distinct. The x'_i and y'_j must be distinct new identifiers.

The environment changes are extensive in the *skolem* rule: the all-eliminated variable identifiers become logical and the some-introduced identifiers join the environment. In addition, the aliased identifiers can change: if $\mathcal{A}'$ is not empty, then it specifies which universal parameters alias to which existential witnesses.

Induction rules

The *int-induction* proof rule is

$$\begin{array}{l} [\mathcal{E}]\ \mathcal{H} \gg \text{All val } x : \text{int}\,.\, F \quad \text{by ind } k \\ \bullet\ [\text{val } \mathcal{V}_{val} + x;\ \text{log } \mathcal{V}_{log} + \mathcal{V}_{var};\ \text{aliased } \mathcal{A}] \\ \qquad \mathcal{H}_{indep},\ x < k,\ F[x + 1/x] \gg F \\ \bullet\ [\mathcal{E}']\ \mathcal{H}_{indep} \gg F[k/x] \\ \bullet\ [\text{val } \mathcal{V}_{val} + x;\ \text{log } \mathcal{V}_{log} + \mathcal{V}_{var};\ \text{aliased } \mathcal{A}] \\ \qquad \mathcal{H}_{indep},\ x > k,\ F[x - 1/x] \gg F \end{array}$$

where k is a constant. As in the *impl-intro* and *all-intro* rules, the members of $\mathcal{V}_{var}$ are made logical to keep them from being used by the body of the extracted procedure. The state-dependent hypotheses are removed for the same reason.

The *list-induction* proof rule is

$$\begin{array}{l} [\mathcal{E}]\ \mathcal{H} \gg \text{All val } x : \text{list}\,.\, F \quad \text{by ind } y\,.\,z \\ \bullet\ [\mathcal{E}']\ \mathcal{H}_{indep} \gg F[()/x] \\ \bullet\ [\text{val } \mathcal{V}_{val} + y + z;\ \text{log } \mathcal{V}_{log} + \mathcal{V}_{var};\ \text{aliased } \mathcal{A}] \\ \qquad \mathcal{H}_{indep},\ F[z/x],\ x = y\,.\,z \gg F \end{array}$$

where y is a new integer identifier and z is a new list identifier.

Miscellaneous rules

The *hypothesis* proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ H_n,\ \mathcal{H}' \gg H_n \quad \text{by hyp } n$$

where H_n is a state-independent formula.

The *sequence* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by seq } G$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep} \gg G$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H},\ G \gg F.$$

The justification for G is built without access to the members of $\mathcal{V}_{var}$ or the state-dependent members of $\mathcal{H}$, so they are available for use in the second subproof.

The *log-term* proof rule is a new rule. It is added here to get around the simplifying assumption made in section ?? that no formula contains terms other than identifiers. (See page ??.) Thus, the rule is simply a formal way to give names to logical intermediate terms. The *log-term* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by log-term } y, f(x_1,\ldots,x_m)$$
$$\bullet\ [\text{val } \mathcal{V}_{val};\ \text{var } \mathcal{V}_{var};\ \text{log } \mathcal{V}_{log}+y;\ \text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr}]$$
$$\mathcal{H},\ y = f(x_1,\ldots,x_m) \gg F$$

where y is a new identifier. (The rule also works with the built-in operations, so $x_1 + x_2$, $x_1 * x_2$, and so on can be used for $f(x_1,\ldots,x_m)$.) The x_i must be type-correct (array, list, or int as appropriate), but it does not matter which x_i are val, var, or log. For example, in

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by log-term } y, update(x,i,e)$$

x must be array, but not necessarily var array. The extracted code for the rule, $\mathrm{X}[\![\pi_{log\text{-}term}, \mathcal{J}]\!]$, is

$$\text{let } K = \text{trivial in } \mathrm{X}[\![\sigma, J, K]\!] \text{ ni.}$$

The *comp-term* proof rule is also new. It is used to give a name to a val or var intermediate term. Though a procedure call is shown below, built-in operations can also be used. The *comp-term* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by comp-term } y, f(x_1,\ldots,x_m)$$
$$\bullet\ [\hat{\mathcal{E}}]\ \mathcal{H},\ y = f(x_1,\ldots,x_m) \gg F$$

where y is a new identifier. The new environment $\hat{\mathcal{E}}$ depends on f and the x_i, but in any case, the following restrictions hold.

1. The collection of x_i where the i'th parameter of f is a var parameter must be distinct.

2. For all i, if the i'th parameter of f is a var parameter, then x_i must be in $\mathcal{V}_{var}$; if it is a val parameter, then x_i must be in $\mathcal{V}_{val} \cup \mathcal{V}_{var}$.

3. If the result of f is val array, then for all i, if the i'th parameter of f is a val parameter, then x_i may not be in $\mathcal{V}_{var}$.

Let $\mathcal{X}'$ be the list of the x_i such that the i'th parameter of f is a var parameter. There are three cases:

- If f returns a value of type val T, then $[\hat{\mathcal{E}}]$ is

$$[\text{val } \mathcal{V}_{val} + y;\ \text{var } \mathcal{V}_{var} - \mathcal{X}';\ \text{log } \mathcal{V}_{log} + \mathcal{X}';\ \text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr} - \mathcal{X}'].$$

- If f returns a value of type var T and does not alias its result to any parameter, then $[\hat{\mathcal{E}}]$ is

$$[\text{val } \mathcal{V}_{val};\ \text{var } \mathcal{V}_{var} + y - \mathcal{X}';\ \text{log } \mathcal{V}_{log} + \mathcal{X}';\ \text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr} + y - \mathcal{X}'].$$

- If f returns a value of type var T and aliases its result to one of its parameters, say the k'th, then $[\hat{\mathcal{E}}]$ is

$$[\text{val } \mathcal{V}_{val};\ \text{var } \mathcal{V}_{var} + y - \mathcal{X}';\ \text{log } \mathcal{V}_{log} + \mathcal{X}';\ \text{aliased } \mathcal{A} + (x_k, y);\ \text{fresh } \mathcal{V}_{fr} - \mathcal{X}'].$$

In all three cases, the extracted code for the rule, $\mathsf{X}[\![\pi_{comp\text{-}term}, \mathcal{J}]\!]$, is

$$\text{let } y = f(x_1, \ldots, x_m); K = \text{trivial in } \mathsf{X}[\![\sigma, J, K]\!] \text{ ni}.$$

As in chapter 2, the ***decision*** rule has two cases. In the first case, F is a literal and the rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by decision}.$$

The identifiers of F may be val, var, or log. In the second case, the proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F_1 \mid F_2 \quad \text{by decision}$$

where each F_i is a literal. Here, all the identifiers of the F_i must be val or var: the extracted code for this case of the rule uses the values of these identifiers, so they may not be logical identifiers.

For the explicit procedure definition rule, say that the procedure f is defined by

$$\text{proc } (x:\text{int}, x_1:T_1, \ldots, x_m:T_m):T'.$$
$$\text{if } x = 0 \rightarrow t_1 \ \square\ x < 0 \rightarrow t_2 \ \square\ x > 0 \rightarrow t_3 \text{ fi}.$$

(The aliasing information for f is not shown here because it is not relevant.)

Let z be in $\mathcal{V}_{log}$. The ***explicit-def*** proof rule has three cases. The first is

$$[\mathcal{E}]\ \mathcal{H},\ z = f(0, y_1, \ldots, y_m),\ \mathcal{H}' \gg z = t_1\,[0, y_1, \ldots, y_m / x_0, \ldots, x_m]$$
$$\text{by def } f(0, y_1, \ldots, y_m)$$

The second case is

$$[\mathcal{E}]\ \mathcal{H},\ z = f(y_0, \ldots, y_m),\ \mathcal{H}' \gg z = t_2\,[y_0, \ldots, y_m / x_0, \ldots, x_m]$$

by def $f(y_0, \ldots, y_m)$ down

• $[\mathcal{E}]\ \mathcal{H} \gg y_0 < 0.$

The third case is

$$[\mathcal{E}]\ \mathcal{H},\ z = f(y_0, \ldots, y_m),\ \mathcal{H}' \gg z = t_3\,[y_0, \ldots, y_m / x_0, \ldots, x_m]$$

by def $f(y_0, \ldots, y_m)$ up

• $[\mathcal{E}]\ \mathcal{H} \gg y_0 > 0.$

In all three cases there are no restrictions on the y_i or the identifiers that appear in the substituted t_i. Procedures that induct on lists and procedures that do not use induction have similar *explicit-def* rules.

The *theorem* proof rule is

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by thm } Th$$

where [val $\mathcal{V}$] $H_{i_1}, \ldots, H_{i_m} \gg F$ is the goal of theorem Th, for some sequence $i_1, \ldots, i_m$ and some set of identifiers $\mathcal{V} \subseteq \mathcal{V}_{val}$. Since theorems are extracted as separate procedures, and procedures may not modify state-dependent hypotheses, the $\mathcal{H}_{i_j}$ must be state-independent. For similar reasons, the *theorem* rule may not be used to prove the existential of an

$\forall\exists$ formula that has a non-empty aliases field. For example, the following is not allowed:

[ ] $\gg$ All var $x : T$. Some var $y : T . F$ (aliases (x, y))

by intro x

• [var x] $\gg$ Some var $y : T . F$ (aliases (x, y))

by thm Th.

For the extracted procedure definition rule, let f be the extracted procedure for theorem Th, which has the goal

[ ] $\gg$ All val $x_1 : T_{11}, \ldots, x_p : T_{1p}$; var $y_1 : T_{21}, \ldots, y_q : T_{2q}$.

Some $z : T''$. F (aliases $\mathcal{A}'$).

Let $\mathcal{X}$ be $x_1, \ldots, x_p$, $\mathcal{Y}$ be $y_1, \ldots, y_q$, $\mathcal{U}$ be $u_1, \ldots, u_p$, and $\mathcal{V}$ be $v_1, \ldots, v_q$. The *extr-def* proof rule is

$$[\mathcal{E}]\ \mathcal{H},\ w = f(\mathcal{U}, \mathcal{V}),\ \mathcal{H}' \gg F\,[\mathcal{U}, \mathcal{V}, w / \mathcal{X}, \mathcal{Y}, z] \quad \text{by def } w, f(\mathcal{U}, \mathcal{V})$$

where w is in $\mathcal{V}_{log}$. The u_i must be distinct, and for all i and j, u_i must be in $\mathcal{V}_{val}$ and v_j must be in $\mathcal{V}_{var}$. The v_j must be distinct.

5.3 Correctness of the extractor

Before proving the extractor correct, it is necessary to describe some properties of states and justifications. The first property is *state extensi-*

bility: justifications valid in one state are valid in extensions of the state. They are even valid if the values of free variable and logical identifiers are changed.

Notation. One or more of ***st***, ***renv***, or ***venv*** may be omitted from $\gamma, st, renv \Vdash F, venv$, or $\models F, venv$, if understood.

Theorem 5.2 *(state extensibility)*

Let γ justify F in state ***st*** and environments ***renv*** and ***venv***. If state st' agrees with ***st*** (mod F), then γ justifies F in state st' and environments ***renv*** and ***venv***.[3]

Proof

By induction on the structure of F.

- ***F is atomic.***

  Say that $\gamma, st, renv \Vdash F, venv$. By validity, $\models F, venv$, so by the definition of justification, $\gamma, st', renv \Vdash F, venv$.

- ***F is an implication or a universal.***

  Here $\gamma, st', renv \Vdash F, venv$ follows immediately from the definition of justification.

[3]Note that extensibility does not follow from the validity theorem. $\gamma, st \Vdash F$ implies $\models F$, which implies that there exist γ' and $st' \supseteq st$ (mod F) such that $\gamma', st' \Vdash F$. Validity does not imply that $\gamma = \gamma'$. It also does not let us conclude that $st' \supseteq st$ (mod F) for *any* $st' \supseteq st$ (mod F).

- ***F is a conjunction.***

  Say that $(\gamma_1, \gamma_2), st, renv \Vdash F_1 \;\&\; F_2, venv$. For $i = 1, 2$, we have by the definition of justification that $\gamma_i, st, renv \Vdash F_i, venv$, so by inductive hypothesis $\gamma_i, st', renv \Vdash F_i, venv$. Then by definition of justification, $(\gamma_1, \gamma_2), st', renv \Vdash F_1 \;\&\; F_2, venv$.

The other cases are similar. □

Extensibility is used to insure that certain justifications, valid in one state, are valid in later states. This is done by guaranteeing that the later states agree with the earlier state, modulo some locations.

The next property, ***acceptability***, restricts the aliasing done by a reference environment and state.

Definition 5.5 *(acceptability)*

Given a proof environment $\mathcal{E}$, a call-by-reference environment ***renv*** is ***acceptable*** for $\mathcal{E}$ if

1. ***renv*** does not alias value identifiers to variable or logical identifiers.
2. The variable identifiers of $\mathcal{E}$ are distinct under ***renv***.
3. The fresh identifiers of $\mathcal{E}$ are distinct under ***renv***.
4. Non-value identifiers alias under ***renv*** only if they are related in $\mathcal{A}^*$.

Given some formulas $\mathcal{H}$ and justifications Γ for them, ***renv*** and ***st*** are acceptable for $\mathcal{H}$ with Γ if they are acceptable for the free identifiers of $\mathcal{H}$

and if no state-dependent var witnesses of the members of Γ are aliased with each other or with the free identifiers of $\mathcal{H}$. They are acceptable for a proof π of $[\mathcal{E}]\ \mathcal{H} \gg F$ if they are acceptable for $\mathcal{E}$ and $\mathcal{H}$ with Γ. If *renv* is understood, I will sometimes omit it and say simply that *st* is acceptable for $\mathcal{H}$ or π.

The next property is ***avoidance***. The state-dependent witnesses of a formula's justification must avoid certain locations.

Definition 5.6 *(avoidance)*
A justification γ ***avoids*** a set of locations $\mathcal{L}$ if the set of locations of the state-dependent witnesses of γ is disjoint from $\mathcal{L}$. Given a state *st*,

- γ avoids *st* if it avoids $dom(st)$.
- γ avoids $st - \mathcal{L}$ if it avoids $dom(st) - \mathcal{L}$.
- γ avoids $st - \Gamma - \mathcal{V}$ if it avoids $st - Loc(\Gamma, st) - renv(\mathcal{V})$.

Occasionally, I will bend the notation a bit and write $st - (\Gamma + \mathcal{V})$ for $st - \Gamma - \mathcal{V}$, and $st - \Gamma - \mathcal{V}_1 - \mathcal{V}_2$ for $st - \Gamma - (\mathcal{V}_1 + \mathcal{V}_2)$, and so on.

Avoidance is needed for the correctness of the proof rules that introduce new hypotheses. For example, given a proof of $\mathcal{H} \gg F$ by seq G, it is necessary that the justification built for G avoid the identifiers of $\mathcal{E}$ and the dependent witnesses of $\mathcal{H}$. If it did not, then the code for the subproof would be run in an unacceptable state.

The final definition concerns the universal identifiers of $\forall\exists$ formulas. They and their aliases are the only non-fresh identifiers that can be used in the ***some-intro*** for the $\forall\exists$ formula.

Definition 5.7 $\mathcal{V}_{ae}$ is the set of ***aliasable*** variables, the all-introduced identifiers of an $\forall\exists$ formula. Consider a proof

$$[\ldots]\ \mathcal{H}_0 \gg \text{All val } x_1 : T_{11}, \ldots, x_p : T_{1p}; \text{var } y_1 : T_{21}, \ldots, y_q : T_{2q} .$$
$$\text{Some} \ldots . G \quad (\text{aliases } \mathcal{A})$$

by intro $\mathcal{U}, \mathcal{V}$

...

$\bullet \cdots \bullet\ \mathcal{H}_1 \gg \text{Some} \ldots . G[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}] \quad (\text{aliases } \mathcal{A}[\mathcal{V}/\mathcal{Y}])$

by intro ...

where $\mathcal{U}$ is $u_1, \ldots, u_p$ and $\mathcal{V}$ is $v_1, \ldots, v_q$. If the main node of π is the ***some-intro*** node or one of the omitted nodes, then $\mathcal{V}_{ae}$ is $\mathcal{V}$, otherwise $\mathcal{V}_{ae}$ is empty. That is, if π is on the path between an ***all-intro*** and ***some-intro*** for an $\forall\exists$ formula, then $\mathcal{V}_{ae}$ is the all-introduced variables, otherwise $\mathcal{V}_{ae}$ is empty.

The extractor correctness theorem can now be stated and proved.

Theorem 5.3 *(extractor correctness, call-by-reference)*

Let π be a proof of $[\mathcal{E}]\ \mathcal{H} \gg F$, and let J, *renv*, *venv*, and *st* be such that *renv* and *st* are acceptable for π, the members of J justify the $\mathcal{H}_i$, and the members of J avoid $st - \mathcal{V}_{ae} - \mathcal{V}_{fr}$. Assume also that $venv$ and $st \circ renv$ agree on $\mathcal{V}_{var}$ and $\mathcal{V}_{val}$. Then $\mathsf{X}[\![\pi, J]\!], st, renv \Vdash F, venv$.

Proof

By induction on the structure of π. The assumptions and conclusions must first be strengthened, and some more notation is needed.

Notation. P is $\mathsf{X}[\![\pi, J]\!]$, γ_P is the value of P in environment $renv$ and state st, and st_P is the state after evaluating P. Γ is $\{\gamma_1, \ldots, \gamma_n\}$, where each γ_i is $st(renv(J_i))$.

Notation. Agreement modulo $\mathcal{E}$ means agreement modulo $\mathcal{V}_{var} + \mathcal{V}_{log}$. Agreement modulo π means agreement modulo $\mathcal{V}_{var} + \mathcal{V}_{log} + \mathcal{L}$, where $\mathcal{L}$ is $Loc(\Gamma, st)$, the set of locations of the state-dependent witnesses of the γ_i. Agreement modulo $\mathcal{V}_{var} + \mathcal{V}_{log}$ means agreement modulo $renv(\mathcal{V}_{var} + \mathcal{V}_{log})$, and agreement modulo $\mathcal{V}_{var} + \mathcal{V}_{log} + \mathcal{L}$ means agreement modulo $renv(\mathcal{V}_{var} + \mathcal{V}_{log}) \cup \mathcal{L}$.

The strengthened conclusions are

1. $\gamma_P, st_P, renv \Vdash F, venv$.

2. st_P agrees with st modulo π.

3. st_P and $renv$ are acceptable for F.

4. γ_P avoids $st - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$, that is, γ_P avoids

$$dom(st) - Loc(\Gamma, st) - renv(\mathcal{V}_{ae} + \mathcal{V}_{fr}).$$

The extracted code below is occasionally expanded from the definitions used in the previous section to make it easier to discuss: an expression e is sometimes rewritten **let** $x = e$ **in** x **ni** to give e a name by which it can be referred to. Hoare-like assertions, in the style of [Boe84] and [Boe85] are used, though in an informal sense.

The proof now begins.

- ***π is proved by hypothesis.***

  Let π be $\mathcal{H}, H_n, \mathcal{H}' \gg H_n$ by hyp n, so the extracted code P is J_n, and st_P, the state after running P, is st. The conclusions follow immediately: J_n justifies H_n in st_P, st_P agrees with st modulo π, st_P is acceptable for H_n, and J_n avoids $st - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$. Most of these hold because st_P is st. J_n avoids $st - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$ because the *hypothesis* rule requires H_n to be state-independent: J_n can have no state-dependent witnesses to intersect with $dom(st) - Loc(\Gamma, st) - renv(\mathcal{V}_{ae} + \mathcal{V}_{fr})$.

- ***π is proved by theorem.***

  The conclusions follow by inductive hypothesis.

- ***π is proved by explicit procedure definition.***

  Trivial.

- ***π is proved by logical term.***

  Trivial.

- ***π is proved by computational term.***

  Let π be

$$[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by comp-term } y, f(x_1,\ldots,x_m)$$
$$\bullet\ [\hat{\mathcal{E}}]\ \mathcal{H},\ y = f(x_1,\ldots,x_m) \gg F.$$

Let $\mathcal{X}'$ be the list of the x_i such that the i'th parameter of f is a var parameter.

There are three cases to the rule. If f returns a value of type val T, then $[\hat{\mathcal{E}}]$ is

$$[\text{val } \mathcal{V}_{val} + y;\ \text{var } \mathcal{V}_{var} - \mathcal{X}';\ \text{log } \mathcal{V}_{log} + \mathcal{X}';$$
$$\text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr} - \mathcal{X}']$$

and P is

let $y = f(x_1,\ldots,x_m)$;
  {state $st_1 \supseteq st$ (mod $\mathcal{X}'$)}
  K = trivial;
  {state $st_2 \supseteq st_1$ and $K \Vdash y = f(x_1,\ldots,x_m)$}

  $L = \mathsf{X}[\sigma, J, K]$
  {state $st_P \supseteq st_2$ (mod σ), $L \Vdash F$,
    and L avoids $st_2 - J - (\mathcal{V}_{ae} + \mathcal{V}_{fr} - \mathcal{X}')$}
in L
ni.

The annotated code should be understood as follows. State st_1, which agrees with st (mod $\mathcal{X}'$), is the state just after binding y to the location returned by $f(x_1,\ldots,x_m)$. Similarly, st_2 is the state that results after binding K to (the location of) trivial, and it is claimed implicitly that the inductive hypotheses hold for σ under st_2.

State st_P results from binding L to the result of $\mathsf{X}[\sigma, J_{indep}, K]$. By inductive hypothesis, st_P agrees with st_2 (mod σ), K justifies F in state st_P and environments $renv$ and $venv$, L avoids $st_2 - J - (\mathcal{V}_{ae} + \mathcal{V}_{fr} - \mathcal{X}')$, and st_P and $renv$ are acceptable for F.

The final claim is that L avoids $st - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$. This is shown in parts. First, by inductive hypothesis, L avoids $st_2 - J - (\mathcal{V}_{ae} + \mathcal{V}_{fr} - \mathcal{X}')$. Second, st_2 is acceptable for L, so L avoids $\mathcal{X}'$, and thus L avoids $st_2 - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$. Finally, $st_2 \supseteq st$, so L avoids $st - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$.

If f returns a value of type var T and does not alias its result to any parameter, then $[\hat{\mathcal{E}}]$ is

$$[\text{val } \mathcal{V}_{val};\ \text{var } \mathcal{V}_{var} + y - \mathcal{X}';\ \text{log } \mathcal{V}_{log} + \mathcal{X}'\quad \text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr} + y - \mathcal{X}']$$

and P is

let $y = f(x_1, \ldots, x_m)$;

{state $st_1 \supseteq st$ (mod $\mathcal{X}'$)}

K = trivial;

{state $st_2 \supseteq st_1$ and $K \Vdash y = f(x_1, \ldots, x_m)$}

$L = \mathsf{X}[\![\sigma, J, K]\!]$

{state $st_P \supseteq st_2$ (mod σ), $L \Vdash F$,

and L avoids $st_2 - J - (\mathcal{V}_{ae} + \mathcal{V}_{fr} + y - \mathcal{X}')$}

in L

ni.

The arguments here are similar to those in the first case.

If f returns a value of type var T and can alias its result to one of its parameters, say the k'th, then $[\hat{\mathcal{E}}]$ is

$$[\text{val } \mathcal{V}_{val};\ \text{var } \mathcal{V}_{var} + y - \mathcal{X}';\ \text{log } \mathcal{V}_{log} + \mathcal{X}'\quad \text{aliased } \mathcal{A} + (x_k, y);\ \text{fresh } \mathcal{V}_{fr} - \mathcal{X}']$$

and P is

let $y = f(x_1, \ldots, x_m)$;

{state $st_1 \supseteq st$ (mod $\mathcal{X}'$) and x_k and y are aliased}

K = trivial;

{state $st_2 \supseteq st_1$ and $K \Vdash y = f(x_1, \ldots, x_m)$}

$L = \mathsf{X}[\![\sigma, J, K]\!]$

{state $st_P \supseteq st_2$ (mod σ), $L \Vdash F$,

and L avoids $st_2 - J - (\mathcal{V}_{ae} + \mathcal{V}_{fr} - \mathcal{X}')$}

in L

ni.

The arguments here are also similar to those for the first case.

- *π is proved by decision.*

The nontrivial case is $\mathcal{H} \gg F_1 \mid F_2$, where F_1 and F_2 are literal formulas. The extracted code is

if $F_1 \rightarrow$

{state st and $\models F_1$}

let $K = (1, \text{trivial})$;

{state $st' \supseteq st$ and $K \Vdash F_1 \mid F_2$}

in K

ni

$\square\ F_2 \rightarrow$

let $K = (2, \mathsf{trivial})$;

{state $st'' \supseteq st$ and $K \Vdash F_1 \mid F_2$}

in K

ni

fi

{state $st_P = st'$ or st''}

The conclusions follow immediately.

- *π is proved by sequence.*

Let π be

$[\mathcal{E}]\ \mathcal{H} \gg F$ by seq G

- $[\mathcal{E}']\ \mathcal{H}_{indep} \gg G$
- $[\mathcal{E}]\ \mathcal{H}, G \gg F$.

and let P be

let $K = \mathsf{X}[\![\sigma_1, J_{indep}]\!]$;

{state $st' \supseteq st$, $K \Vdash G$, K avoids st,

and st' is acceptable for G}

$L = \mathsf{X}[\![\sigma_2, J, K]\!]$

{state $st_P \supseteq st'$ (mod σ_2), $L \Vdash F$, L avoids $st' - J -$

$\mathcal{V}_{ae} - \mathcal{V}_{fr}$, and st_P is acceptable for F}

in L

ni

Above, K avoids $st - Loc(renv(J_{indep}), st)$ by inductive hypothesis, and $Loc(renv(J_{indep}), st)$ is empty, so K avoids st. Checking the conclusions, $st_P \supseteq st'$ (mod σ_2) $\supseteq st$, and γ_P avoids $dom(st') - J - \mathcal{V}_{ae} - \mathcal{V}_{fr} \supseteq dom(st) - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$.

- *π is proved by and-intro.*

Let π be

$[\mathcal{E}]\ \mathcal{H} \gg F_1\ \&\ F_2$ by intro

- $[\mathcal{E}']\ \mathcal{H}_{indep} \gg F_1$
- $[\mathcal{E}']\ \mathcal{H}_{indep} \gg F_2$

and let P be

let $K_1 = \mathsf{X}[\![\sigma_1, J_{indep}]\!]$;

{state $st_1 \supseteq st$, $K_1 \Vdash F_1$, K_1 avoids st,

and st_1 is acceptable for F_1}

$K_2 = \mathsf{X}[\![\sigma_2, J_{indep}]\!]$;

{state $st_2 \supseteq st_1$, $K_2 \Vdash F_2$, K_2 avoids st_1, and st_2 is acceptable for F_1 and $F_2\}$

$L = (K_1, K_2)$;

$\{$state $st_P \supseteq st_2$, $L \Vdash F_1 \,\&\, F_2$, L avoids st,

and st_P is acceptable for $F_1 \,\&\, F_2\}$

in L

ni.

Above, st_2 is acceptable for F_1 because the state-dependent witnesses of F_1 are all in $dom(st_1)$ and K_2 avoids st_1. Similarly, L avoids st because K_1 avoids $st_1 \supseteq st$ and K_2 avoids $st_2 \supseteq st_1 \supseteq st$.

The conclusions follow directly: $st_P \supseteq st_2 \supseteq st_1 \supseteq st$, and L avoids $dom(st) \supseteq dom(st) - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$.

- *π is proved by and-elim.*

Let π be

$$[\mathcal{E}]\ \mathcal{H},\ F_1 \,\&\, F_2,\ \mathcal{H}' \gg G \quad \text{by elim } n$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ F_1,\ F_2 \gg G$$

and let P be

let $(K_1, K_2) = J_n$;

$\{$state $st' \supseteq st$, $K_1 \Vdash F_1$, $K_2 \Vdash F_2$,

and st_1 is acceptable for $\sigma\}$

$L = \mathsf{X}[\![\sigma, J, J', K_1, K_2]\!]$

$\{$state $st_P \supseteq st'$ (mod σ), L avoids $st' - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$,

and st_P is acceptable for $G\}$

in L

ni.

The conclusions follow immediately.

- *π is proved by or-intro.*

Let π be

$$[\mathcal{E}]\ \mathcal{H} \gg F_1 \mid F_2 \quad \text{by intro } k$$
$$\bullet\ [\mathcal{E}]\ \mathcal{H} \gg F_k$$

and let P be

let $K = \mathsf{X}[\![\sigma, J]\!]$;

$\{$state $st' \supseteq st$ (mod σ), K avoids $st - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$,

and st' is acceptable for $F_k\}$

$L = (k, K)$

$\{$state $st_P \supseteq st'$, L avoids $st - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$,

and st_P is acceptable for $F_1 \mid F_2\}$

in L

ni.

Again, the conclusions follow immediately.

- *π is proved by or-elim.*

  Let π be

  $[\mathcal{E}]\ \mathcal{H},\ F_1 \mid F_2,\ \mathcal{H}' \gg G$ by elim n

  - $[\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ F_1 \gg G$
  - $[\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ F_2 \gg G$

and let P be

```
let (k, K) = J_n;
  {state st_1 ⊇ st, K ||- F_k, and st_1 is acceptable for σ_k}
  L =
    if k = 1 →
      let M = X[σ_1, J, J', K]
        {state st_2 ⊇ st_1 (mod σ_1), M ||- G,
            M avoids st_1 − J − V_ae − V_fr,
            and st_2 is acceptable for G}
      in M
      ni
    [] k = 2 →
      let M = X[σ_2, J, J', K]
        {state st_3 ⊇ st_1 (mod σ_2), M ||- G,
```



```
            M avoids st_1 − J − V_ae − V_fr,
            and st_3 is acceptable for G}
      in M
      ni
    fi
  {state st_P ⊇ st (mod π), L ||- G,
      L avoids st_1 − J − V_ae − V_fr,
      and st_P is acceptable for G}
in L
ni.
```

Note here that st_P is the state after executing $\mathsf{X}[\sigma_k, J, K]$, which depends on whether k is 1 or 2. (k is guaranteed to be one or the other, since J_n justifies $F_1 \mid F_2$.)

- *π is proved by impl-intro.*

  Let π be

  $[\mathcal{E}]\ \mathcal{H} \gg F \Rightarrow G$ by intro

  - $[\mathcal{E}']\ \mathcal{H}_{indep},\ F \gg G$

and let P be

$\{\text{state } st, \text{ and for each } i,\ J_i \Vdash H_i\}$

$\text{close}\,(\mathcal{V}_{val}, \mathcal{J}_{indep}) \cdot \text{proc}\,(K\!:\!\text{just})$.

$\{$state $st_1 \supseteq st$ (mod $F \Rightarrow G$), $K \Vdash F$,

  for each $J_i \in J_{indep}$, $J_i, st_1 \Vdash H_i$,

  and st_1 is acceptable for $\sigma\}$

let $L = \mathsf{X}[\![\sigma, J_{indep}, K]\!]$

  $\{$state $st_2 \supseteq st_1$, $L \Vdash G$, L avoids st_1,

    and st_2 is acceptable for $G\}$

in L

ni.

Note that the annotated code claims that L avoids st_1 even though the inductive hypothesis only implies that L avoids $st_1 - K$. This stronger claim holds because the *impl-intro* rule restricts F, the formula justified by K, to be state-independent.

- ***π is proved by impl-elim.***

  Let π be

  $[\mathcal{E}]\ \mathcal{H},\ F \Rightarrow G,\ \mathcal{H}' \gg F'$ by elim n

  - $[\mathcal{E}']\ \mathcal{H}_{indep},\ \mathcal{H}'_{indep} \gg F$
  - $[\mathcal{E}]\ \mathcal{H},\ \mathcal{H}',\ F,\ G \gg F'$

and let P be

let $K = \mathsf{X}[\![\sigma_1, J_{indep}, J'_{indep}]\!]$;

  $\{$state $st_1 \supseteq st$, $K \Vdash F$, K avoids st,

  and st_1 is acceptable for $F\}$

$L = J_n(K)$;

$\{$state $st_2 \supseteq st_1$, $L \Vdash G$, L avoids st_1,

  and st_2 is acceptable for $\sigma_2\}$

$M = \mathsf{X}[\![\sigma_2, J, J', K, L]\!]$

$\{$state $st_P \supseteq st_2$ (mod σ_2), $M \Vdash F'$,

  M avoids $st_2 - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$,

  and st_P is acceptable for $F'\}$

in M

ni.

The claim that $st_2 \supseteq st_1$ and that L avoids st_1 follow from the definition of implication justification.

- ***π is proved by not-intro.***

  Similar to *impl-intro*.

- ***π is proved by not-elim.***

  Similar to *impl-elim*.

- ***π is proved by false-intro.***

  Let π be

$$[\mathcal{E}]\ \mathcal{H} \gg \text{false} \quad \text{by intro } F$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep} \gg F$$
$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep} \gg \neg F$$

and let P be

let $K = \mathsf{X}[\![\sigma_1, \mathcal{J}_{indep}]\!]$;
 {state $st_1 \supseteq st$, $K \Vdash F$, K avoids st,
  and st_1 is acceptable for F}
 $L = \mathsf{X}[\![\sigma_2, \mathcal{J}_{indep}]\!]$;
 {state $st_2 \supseteq st_1$, $L \Vdash \neg F$, L avoids st_1,
  and st_2 is acceptable for $\neg F$}
 $M = L(K)$
 {state $st_P \supseteq st_2$, $M \Vdash$ false, M avoids st_2,
  and st_P is acceptable for false}
in M
ni.

This rule is similar to the ***impl-intro*** and ***impl-elim*** rules. The justification for $\neg F$ is built in an environment that has no computational

access to the non-stable part of st, and the justification for F is built in an environment whose stable part matches that of st. Therefore, the two justifications can be used to produce a justification for false.

Note that if we know a priori that the extractor correctness theorem holds for the logic, then we can replace the extracted code above by trivial. If correctness holds, we know that if the $\mathcal{H}$ have justifications then false has a justification. If false has a justification, then ***triv*** justifies F. Though we do not know yet that extractor correctness holds, it would be sufficient to know that the logic is sound. No separate proof of soundness is being given, so the rule is restricted here.

- ***π is proved by false-elim.***

  Trivial.

- ***π is proved by all-intro.***

  For the quantifier rules, recall that $\mathcal{X}$ is $x_1, \ldots, x_p$, $\mathcal{Y}$ is $y_1, \ldots, y_q$, $\mathcal{U}$ is $u_1, \ldots, u_p$, and $\mathcal{V}$ is $v_1, \ldots, v_q$. Let π be

$$[\mathcal{E}]\ \mathcal{H} \gg \text{All val } x_1 : T_1, \ldots, x_p : T_p; \text{var } y_1 : T_1', \ldots, y_q : T_q' \,.\, F$$
$$(\text{aliases } \mathcal{A}')$$
$$\text{by intro } \mathcal{U}, \mathcal{V}$$
$$\bullet\ [\text{val } \mathcal{V}_{val} + \mathcal{U};\ \text{var } \mathcal{V};\ \log \mathcal{V}_{log} + \mathcal{V}_{var};\ \text{aliased } \mathcal{A}]$$
$$\mathcal{H}_{indep} \gg F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}] \quad (\text{aliases } \mathcal{A}'[\mathcal{V}/\mathcal{Y}])$$

and let P be

$\{$state st, and for each $J_i \in J$, $J_i, st \Vdash H_i\}$

close $(\mathcal{V}_{val}, J_{indep}) \cdot$ proc $(\mathcal{U}, \mathcal{V})$.

$\{$state $st_1 \supseteq st$ (mod All $\ldots F$),

for each $J_i \in J_{indep}$, $J_i, st_1 \Vdash H_i$,

and st_1 is acceptable for $\sigma\}$

let $K = \mathsf{X}[\![\sigma, J_{indep}]\!]$

$\{$state $st_2 \supseteq st_1$ (mod $\mathcal{V}$), $K \Vdash F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}]$,

K avoids st_1,

and st_2 is acceptable for $F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}]\}$

in K

ni.

Note that for st_1 to be acceptable for σ, it is necessary for the members of $\mathcal{V}$ not to alias with each other or with the the members of $\mathcal{U}$. This is why $\mathcal{U}$ and $\mathcal{V}$ are restricted in the *all-elim* rule.

If F is an existential, then it is necessary to argue that a correct $\forall\exists$ justification is being built. This is done below at the end of the proof.

- ***π is proved by all-elim.***

Recall that this rule is not used to eliminate $\forall\exists$ formulas: the *skolem*

rule below is used for that purpose. Let π be

$[\mathcal{E}]\ \mathcal{H}$, All val $x_1 : T_1, \ldots, x_p : T_p$; var $y_1 : T'_1, \ldots, y_q : T'_q \,.\, F$,

$\mathcal{H}' \gg G$ by elim $n, \mathcal{U}, \mathcal{V}$

- [val $\mathcal{V}_{val}$; var $\mathcal{V}_{var} - \mathcal{V}$; log $\mathcal{V}_{log} + \mathcal{V}$;

aliased $\mathcal{A}$; fresh $\mathcal{V}_{fr} - \mathcal{V}$]

$\mathcal{H}$, All $\ldots$, $\mathcal{H}'$, $F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}] \gg G$

and let P be

let $K = J_n(\mathcal{U}, \mathcal{V})$;

$\{$state $st' \supseteq st$ (mod $\mathcal{V}$), $K \Vdash F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}]$,

K avoids st, and st' is acceptable for $\sigma\}$

$L = \mathsf{X}[\![\sigma, J, J_n, J', K]\!]$;

$\{$state $st_P \supseteq st'$ (mod σ),

L avoids $st' - J - \mathcal{V}_{ae} - \mathcal{V}_{fr}$,

and st_P is acceptable for $G\}$

in L

ni.

This rule is similar to the *impl-elim* rule: the justification for the universal was built to be valid in some state st_0. Since st agrees with st_0, the justification is valid in st.

- *π is proved by some-intro.*

Let π be

$$[\mathcal{E}]\ \mathcal{H} \gg \mathsf{Some\ val}\ x_1 : T_1, \ldots, x_p : T_p; \mathsf{var}\ y_1 : T'_1, \ldots, y_q : T'_q \,.\, F$$
$$(\mathsf{aliases}\ \mathcal{A}')$$
$$\mathsf{by\ intro}\ \mathcal{U}, \mathcal{V}$$
$$\bullet\ [\mathsf{val}\ \mathcal{V}_{val};\ \mathsf{var}\ \mathcal{V}_{var} - \mathcal{V};\ \mathsf{log}\ \mathcal{V}_{log} + \mathcal{V};$$
$$\mathsf{aliased}\ \mathcal{A};\ \mathsf{fresh}\ \mathcal{V}_{fr} - \mathcal{V}]$$
$$\mathcal{H} \gg F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}]$$

and let P be

let $K = \mathsf{X}[\![\sigma, \mathcal{J}]\!]$;

{state $st' \supseteq st$ (mod σ), K avoids $st - J - (\mathcal{V}_{fr} - \mathcal{V})$,

and st' is acceptable for $F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}]$}

$L = (\mathcal{U}, \mathcal{V}, K)$

{state $st_P \supseteq st'$, L avoids $st - J - (\mathcal{V}_{ae} + \mathcal{V}_{fr} - \mathcal{V})$,

and st_P is acceptable for Some ...}

in L

ni

Given that K avoids $st - J - (\mathcal{V}_{fr} - \mathcal{V})$, it is necessary for K to avoid $renv(\mathcal{V})$ for L to avoid $st - J - (\mathcal{V}_{ae} + \mathcal{V}_{fr} - \mathcal{V})$. K avoids

$renv(\mathcal{V})$ because the members of $\mathcal{V}$ are not computationally available in σ.

It is also necessary to argue that γ_P satisfies the appropriate aliasing restriction. This is done at the end of the proof.

- *π is proved by some-elim.*

Let π be

$$[\mathcal{E}]\ \mathcal{H},\ \mathsf{Some\ val}\ x_1 : T_1, \ldots, x_p : T_p; \mathsf{var}\ y_1 : T'_1, \ldots, y_q : T'_q \,.\, F,$$
$$\mathcal{H}' \gg G \quad \mathsf{by\ elim}\ n, \mathcal{U}, \mathcal{V}$$
$$\bullet\ [\mathsf{val}\ \mathcal{V}_{val} + \mathcal{U};\ \mathsf{var}\ \mathcal{V}_{var} + \mathcal{V};\ \mathsf{log}\ \mathcal{V}_{log};$$
$$\mathsf{aliased}\ \mathcal{A};\ \mathsf{fresh}\ \mathcal{V}_{fr} + \mathcal{V}]$$
$$\mathcal{H},\ \mathcal{H}',\ F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}] \gg G$$

and let P be

let $\langle \mathcal{U}, \mathcal{V}, K \rangle = J_n$;

{state $st' \supseteq st$, $K \Vdash F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}]$,

and st is acceptable for σ}

$L = \mathsf{X}[\![\sigma, J, J', K]\!]$

{state $st_P \supseteq st'$ (mod σ),

L avoids $st' - J - \mathcal{V}_{ae} - \mathcal{V}_{fr} - \mathcal{V}$, and st_P is acceptable for G}

in L

ni

For st' to be acceptable for σ, it is necessary that the eliminated var witnesses not alias with themselves nor with the members of $\mathcal{V}_{var}$. This holds because st is acceptable for π.

- *π is proved by skolem.*

Below, $\mathcal{X}'$ is $x'_1, \ldots, x'_r$, $\mathcal{Y}'$ is $y'_1, \ldots, y'_s$, $\mathcal{U}'$ is $u'_1, \ldots, u'_r$, and $\mathcal{V}'$ is $v'_1, \ldots, v'_s$. They are used with $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{U}$, and $\mathcal{V}$ to specify the actual and formal universal parameters and existential witnesses of the skolem formula. Let π be

$[\mathcal{E}]$ $\mathcal{H}$, All val $x_1 : T_{11}, \ldots, x_p : T_{1p}$; var $y_1 : T_{21}, \ldots, y_q : T_{2q}$.

Some val $x'_1 : T_{31}, \ldots, x'_r : T_{3r}$; var $y'_1 : T_{41}, \ldots, y'_s : T_{4s}$. F

(aliases $\mathcal{A}'$), $\mathcal{H}' \gg G$ by skolem $n, \mathcal{U}, \mathcal{V}, \mathcal{U}', \mathcal{V}'$

- $\big[$val $\mathcal{V}_{val} + \mathcal{U}'$; var $\mathcal{V}_{var} + \mathcal{V}' - \mathcal{V}$; log $\mathcal{V}_{log} + \mathcal{V}$;

aliased $\mathcal{A} \cup \mathcal{A}'[\mathcal{V}, \mathcal{V}'/\mathcal{Y}, \mathcal{Y}']$; fresh $\mathcal{V}_{fr} + \mathcal{V}''\big]$

$\mathcal{H}$, All $\ldots$, $\mathcal{H}'$, $F[\mathcal{U}, \mathcal{V}, \mathcal{U}', \mathcal{V}'/\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}'] \gg G$

where $\mathcal{V}''$ is the v'_i not mentioned in $\mathcal{A}'[\mathcal{V}, \mathcal{V}'/\mathcal{Y}, \mathcal{Y}']$.

The code below resembles the code for an *all-elim* followed by a *some-elim*, but it is annotated differently, since its correctness works differently. Let P be

let $K = J_n(\mathcal{U}, \mathcal{V})$;

{state $st_1 \supseteq st$ (mod $\mathcal{V}$), and K avoids $st - \mathcal{V}$}

$(\mathcal{U}', \mathcal{V}', L) = K$;

{state $st_2 \supseteq st_1$, L avoids st, and st_2 is acceptable for σ}

$M = \mathsf{X}[\![\sigma, J, J_n, J', K]\!]$

{state $st_P \supseteq st_2$, st_P is acceptable for G,

and M avoids $st - J - \mathcal{V}_{fr} - (\mathcal{V}_{ae} - \mathcal{V})$}

in M

ni.

The aliasing restrictions on $\forall\exists$ justifications are important here. The witnesses of Some $\ldots[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}]$ can alias with $\mathcal{V}$ but must avoid the other parts of st. Similarly, the justification for the substituted form of the matrix must avoid st, since without these restrictions st_2 would not be acceptable for σ. Let $renv'$ be $renv$ after binding $\mathcal{U}$ and $\mathcal{V}$. Because of the aliasing restriction on J_n, one of two cases holds for each v'_j in $\mathcal{V}'$: either $renv'(v'_j)$ is in $dom(st_1) - dom(st)$, or for some i, $renv'(v'_j)$ equals $renv'(v_i)$ and $\langle y_i, y'_j\rangle$ is in $\mathcal{A}^*$.

- ***π is proved by list induction.***

  Let π be

$$[\mathcal{E}]\ \mathcal{H} \gg \mathsf{All\ val}\ x : \mathsf{list}\,.\,F \quad \text{by ind } y\,.\,z$$

$$\bullet\ [\mathcal{E}']\ \mathcal{H}_{indep} \gg F\,[()/x]$$

$$\bullet\ [\mathsf{val}\ \mathcal{V}_{val} + y + z;\ \log\ \mathcal{V}_{log} + \mathcal{V}_{var};\ \mathsf{aliased}\ \mathcal{A}]$$
$$\mathcal{H}_{indep},\ F\,[z/x],\ x = y\,.\,z \gg F$$

  where y is a new integer identifier and z is a new list identifier. ($()$ is the empty list.) Let P be

$$\{\text{state } st\}$$
$$\mathsf{close}\,(\mathcal{V}_{val}, \mathcal{J}_{indep})\cdot \mathsf{proc}\,(x)\,.\,f(x, \mathcal{V}_{val}, \mathcal{J}_{indep}).$$

  Ignoring type information in its procedure header, let f be bound to

proc $(x, \mathcal{V}_{val}, \mathcal{J}_{indep})$.

{state $st_1 \supseteq st$ (mod All $\ldots F$),

and for each $J_i \in \mathcal{J}_{indep}$, $J_i, st_0 \Vdash H_i$}

if $x = () \rightarrow$

let $K = \mathsf{X}[\![\sigma_1, \mathcal{J}_{indep}]\!]$

{state $st_P \supseteq st_1$, K avoids st_1,

and st_P is acceptable for $F\,[()/x]$}

in K

ni

$\Box\ x \neq () \rightarrow$

let $y = hd(x)$; $z = tl(x)$; $K = f(z, \mathcal{V}_{val}, \mathcal{J}_{indep})$;

{state $st_2 \supseteq st_1$, $K \Vdash F\,[z/x]$, K avoids st_1,

and st_2 is acceptable for σ_2}

$L = \mathsf{X}[\![\sigma_2, \mathcal{J}_{indep}, K]\!]$

{state $st_P \supseteq st_2$, (st_1, st_P) is stable over $dom(st_1)$,

$L \Vdash F$, L avoids st_1,

and st_P is acceptable for F}

in L

ni

fi.

  The procedure f built by this rule is recursive, so it is especially important that f not modify its parameters: modifying a value in one call would make it unavailable for other calls.

- ***π is proved by integer induction.***

  Similar to list induction.

- ***Aliasing restrictions.***

  It is still necessary to argue that existential justifications obey the aliasing restrictions. The discussion centers on the *some-intro* rule, since that is the only place in the logic where such justifications are built. The *some-intro* rule is:

$$[\mathcal{E}]\ \mathcal{H} \gg \text{Some val } x_1 : T_1, \ldots, x_p : T_p; \text{var } y_1 : T'_1, \ldots, y_q : T'_q \,.\, F$$
$$(\text{aliases } \mathcal{A}')$$
$$\text{by intro } \mathcal{U}, \mathcal{V}$$
$$\bullet\ [\text{val } \mathcal{V}_{val};\ \text{var } \mathcal{V}_{var} - \mathcal{V};\ \text{log } \mathcal{V}_{log} + \mathcal{V};$$
$$\text{aliased } \mathcal{A};\ \text{fresh } \mathcal{V}_{fr} - \mathcal{V}]$$
$$\mathcal{H} \gg F[\mathcal{U}, \mathcal{V}/\mathcal{X}, \mathcal{Y}]\,.$$

There are two cases, depending on whether or not the introduced existential is the existential of an $\forall\exists$ formula. I will concentrate on the case where it is, since this case basically subsumes the other.

If the introduced existential is part of an $\forall\exists$ formula, then for the aliasing restriction to be satisfied, the var identifiers mentioned by aliases $\mathcal{A}'$ should be able to alias to the appropriate universal variables and the var identifiers *not* mentioned by $\mathcal{A}'$ must be fresh. The first condition, that variables that are allowed to alias can, follows from the acceptability of *renv*. Arguing that the other variables, the ones not mentioned by $\mathcal{A}'$, are fresh takes some work. It is sufficient to show that the members of $\mathcal{V}_{fr}$ are fresh, since the variables not mentioned by $\mathcal{A}'$ must be members of $\mathcal{V}_{fr}$.

The extracted code P that produces γ_P is only part of the procedure built to justify the $\forall\exists$ formula. Let π_0 be the parent of π that does the *all-intro* for the $\forall\exists$ formula; the extracted code for π_0 is the procedure that contains P. Let st_0 be the state at the time this procedure started execution. For the members of $\mathcal{V}_{fr}$ to be fresh, they must avoid $dom(st_0)$. That is, they must be bound to locations in $dom(st) - dom(st_0)$. There are three ways variables can enter the fresh field and become members of $\mathcal{V}_{fr}$: through the *comp-term* rule, through the *skolem* rule, and through the *some-elim* rule. Moreover, these rules must be applied on the path from π_0 to π, since π_0, with its *all-intro* rule, nulls out the fresh field. Variables marked fresh by the *comp-term* and *skolem* rules avoid $dom(st_0)$ by inductive hypothesis. Variables marked fresh by the *some-elim* rule also avoid $dom(st_0)$, but for more subtle reasons.

Let F' be an existential eliminated by a *some-elim* rule on the path from π_0 to π. It is necessary to show that the witnesses of F' avoid $dom(st_0)$. The *all-intro* rule of π_0 removes all state-dependent formulas from the hypotheses, so F' must have been added along the path from π_0 to π. In the simplest case, F' was added by a *sequence* or *impl-elim* rule: the witnesses of F' avoid $dom(st_0)$ by inductive hypothesis.

In general, F' must have entered the hypothesis list by being one of the formulas produced by the k'th iterated elimination of some formula G', where G' itself was not gotten by elimination.[4] By inspecting the rules, it is easy to verify that if F' was obtained from G' by doing nothing but *and-elims*, *or-elims*, or *some-elims*, then G' must have entered the hypothesis list via a *sequence* or *impl-elim* rule. In such a case, the witnesses of F' avoid $dom(st_0)$ by inductive hypothesis. On the other hand, if an *all-elim*, *impl-elim*, or *skolem* was done, then the witnesses of F avoid $dom(st_0)$ by the aliasing restrictions on all-justifications, ∀∃-justifications, or implication-justifications respectively. □

[4] For these purposes, a *skolem* counts as an elimination, and a *false-elim* counts as an *impl-elim*.

Chapter 6

The array inferencer

This chapter tries to bridge the gap between the logic presented in chapter 2, which does not allow array updating in place, and the logic of chapter 5, which does. The *array inferencing* algorithm presented here takes a proof of the form given in chapter 2 and inspects it to find list identifiers that can be implemented as arrays. Users can have a list implemented as an array by consistently using the list in array-like ways (with *select* or *update*, for example, not car or cdr). In addition, a list to be implemented as an array must meet the various restrictions on arrays, detailed in chapter 5. The algorithm presented correctly implements arrays for the binary search and quicksort proofs presented in earlier chapters.

6.1 Introduction

The previous chapter presented a new logic to handle in-place updating of arrays. Most of the effort spent in building the new logic was spent in controlling the complications caused by having call-by-reference parameters: aliasing, side effects, imported variables, and so on. Reasoning about the arrays themselves was not very hard. In fact, arrays can be viewed as just a particular way to implement lists: call-by-reference complications aside, reasoning about arrays is basically the same as reasoning about lists. The primitive array-like functions can be explicitly defined, and the *explicit-def* rules can be used to help prove their usual properties. For example, *select* and *update* can be defined by

$select$: proc (x : list, i : int) : int .
  if $i < 0 \rightarrow 0$
  $[]\ i = 0 \rightarrow hd(x)$
  $[]\ i > 0 \rightarrow select(tl(x), i-1)$
  fi

$update$: proc (x : list, i, e : int) : list .
  if $i < 0 \rightarrow x$
  $[]\ i = 0 \rightarrow e \,.\, tl(x)$

  $[]\ i > 0 \rightarrow update(tl(x), i-1, e)$
  fi.

(In chapter 2, primitive recursion was allowed on the first parameter of a procedure, not the second. This is easy to circumvent: for example, define $select'(i, x)$ in the allowed way and then define $select(x, i) = select'(i, x)$.)

From the definitions above it is possible to prove, for example,

$$\gg select(update(x, i, e), i) = e$$

and

$$\gg i \neq j \Rightarrow select(update(x, i, e), j) = select(x, j).$$

The code for these definitions can be generated directly using car and cdr on lists, but such code takes linear time, and *update* takes linear space. If arrays are used, then calls to the primitive array-using routines can be used, and the resulting code takes constant time and space. It is possible to combine these different pieces of code into one implementation, which does runtime datatype checking:

$update$: proc (x : list or array, i, e : int) : list or array .
  if x is a list $\rightarrow update\_naive(x, i, e)$
  $[]$ x is an array $\rightarrow update\_optimal(x, i, e)$
  fi.

(Here, *update_naive* is of type list $\times$ int $\times$ int $\rightarrow$ list and *update_optimal* is of type array $\times$ int $\times$ int $\rightarrow$ array.)

Using runtime datatype checks has two disadvantages. First, runtime datatype checks are relatively expensive. Second, either some compile-time checks must be done to ensure that the array x given to *update_optimal* is not used after it is modified, or *update_optimal* must make a copy of x, which defeats the purpose of making it an array.

Runtime datatype checks can be avoided if identifiers are implemented consistently, that is, if each identifier always has a list value or always has an array value. Consider

$$\ldots \mathsf{let}\ y = \mathit{update}(x,i,e)\ \mathsf{in} \ldots \mathsf{ni} \ldots.$$

If x is implemented as a list, then the call of *update* can be implemented with a call to the list-using version of *update*:

$$\ldots \mathsf{let}\ y = \mathit{update\_naive}(x,i,e)\ \mathsf{in} \ldots \mathsf{ni} \ldots.$$

If x is implemented as an array, then the array-using version of *update* can be called:

$$\ldots \mathsf{let}\ y = \mathit{update\_optimal}(x,i,e)\ \mathsf{in} \ldots \mathsf{ni} \ldots.$$

The implementation of x affects the implementation of y: if x is a list, then y must be a list, and if x is an array, then y must be an array.

The desired semantics for *update_optimal*(x,i,e) is that it modifies x in place and returns the address of x as its result. The binding

$$\mathsf{let}\ y \doteq \mathit{update\_optimal}(x,i,e)$$

makes x and y share storage. For efficiency, *update_optimal* will not do a runtime datatype check on x, so at runtime x *must* be an array.

Deciding whether or not a particular list identifier should be implemented using lists or arrays now becomes an optimization.[1] If x is an identifier used only in contexts like *select*(x,i) and *update*(x,i,e), then it is better implemented as an array. On the other hand, if x is used in contexts like $tl(x)$, then it is better implemented as a list.

This chapter presents an ***array inferencing*** algorithm: the algorithm takes a proof and decides which of its lists can be implemented as arrays.[2] More precisely, it takes a proof and *annotates* it: notes which list keywords can be replaced by val, var, or log array respectively, and marks proof environments with the appropriate fields (aliased, fresh, and so on).

[1] In fact, this was my original approach. The array-handling logic of chapter 5 was developed in order to prove that annotated proofs made sense.

[2] Here, "proof" generally means a proof using the logic of chapter 2, extended to include the *skolem*, *log-term*, and *comp-term* rules of chapter 5. The *log-term* and *comp-term* rules are assumed to have been generalized to a *term* rule:

$$\frac{[\mathcal{E}]\ \mathcal{H} \gg F \quad \text{by term } y, f(\mathcal{X})}{\bullet\ [\mathcal{E}, y{:}\ldots]\ \mathcal{H}, y = f(\mathcal{X}) \gg F.}$$

The inferencer will annotate each instance of the *term* rule *logical* or *computational* as necessary.

The extractor uses the annotated proof of a theorem when it generates code for the theorem. For example, the implementation datatypes attached to the proof would tell the extractor whether $select(x, i)$ should be implemented by $select\_naive(x, i)$ or $select\_optimal(x, i)$. The extractor also saves the annotated goal of each theorem. When a theorem is used as a lemma, the annotated goal gives the the datatypes used by the optimal code for the lemma. The inferencer uses that information to determine datatype information for the theorem that uses the lemma.

For example, consider the quicksort theorem. Before code is extracted for it, the inferencer marks its goal to indicate that the input and output arrays are optimally var array, and aliased. If some later theorem quicksorts an identifier x, then when the inferencer is run on that theorem, it notes that the annotated goal of quicksort implies that optimally, x should be var array.

It is not always possible to use the optimal code for a theorem for the same reasons it is not always possible to use the optimal code for a function like *update*. (For example, a val argument might have been supplied where a var argument was required.) The solution to this problem is also the same. For each theorem, two pieces of code are extracted. The optimal one is generated using the annotation provided by the inferencer, and it

uses arrays wherever the inferencer allows it to.[3] The naive piece of code uses arrays nowhere. Obviously, the inferencer should try to avoid using the naive code for lemmas, but it will if the optimal code cannot be used. The same strategy is used for implicitly and explicitly defined procedures: two pieces of code, one naive, the other optimal, are produced for each procedure.[4]

The array inferencing algorithm is broken up into two phases. Phase one marks the free and quantified identifiers of π with datatypes like val array, and phase two marks the $\forall\exists$ formulas of π with aliases fields. The markings are done with the help of information from the library about the optimal datatypes and the aliasing characteristics of the functions and lemmas used by π.

The markings are not necessarily done monotonically. For example, an identifier x might be marked, say var array at one point, but later remarked list. This happens if x is used in contradictory ways. For example, if x is used in $update(x, i, e)$, it is marked var array, but if it is later used in $tl(x)$, then this second, list-like usage overrides the original array-like use. Similarly, the aliases information attached to the $\forall\exists$ formulas can change during the course of the marking.

[3] "Preferred" is more accurate than "optimal," but "optimal" is used nonetheless.
[4] Of course, it is always possible that the two pieces of code might be identical, in which case both need not be stored.

For a rough sketch of how the inferencer works, consider the following proof π. For conciseness, type information is shown only for the first node.

$[x_1 : \mathsf{list}] \gg \mathsf{All}\ x_2, x_3 : \mathsf{list} . \mathsf{Some}\ x_4, x_5 : \mathsf{list} . F$

  by seq $\mathsf{All}\ x_6 : \mathsf{list} . \mathsf{Some}\ x_7 : \mathsf{list} . G$

• $[x_1] \gg \mathsf{All}\ x_6 . \mathsf{Some}\ x_7 . G$ by thm Th

• $[x_1]$ $\mathsf{All}\ x_6 . \mathsf{Some}\ x_7 . G$

  $\gg \mathsf{All}\ x_2, x_3 . \mathsf{Some}\ x_4, x_5 . F$ by intro

•• $[x_1, x_2, x_3]$ $\mathsf{All}\ x_6 . \mathsf{Some}\ x_7 . G$

  $\gg \mathsf{Some}\ x_4, x_5 . F$ by skolem $1, x_2, x_8$

••• $[x_1, x_2, x_3, x_8]$ $\mathsf{All}\ \ldots,\ G[x_2, x_8/x_6, x_7]$

  $\gg \mathsf{Some}\ x_4, x_5 . F$ by intro x_3, x_8

...

Analysis reveals that certain pairs of identifiers must have the same types. (For the moment, include the keyword val, var, or log with the type.) For example, x_2 and x_6 must have the same type because the *skolem* rule that x_2 is used in requires it. The *skolem* rule also forces x_7 and x_8 to have the same type. The *some-intro* rule forces x_3 and x_4 to have the same type and adds x_5 to the x_7, x_8 equivalence class. Thus we get the following partition of identifiers by their types: $\{x_1\}$, $\{x_2, x_6\}$, $\{x_3, x_4\}$, and $\{x_5, x_7, x_8\}$.

Assume the annotated goal of Th is

$$[\,] \gg \mathsf{All\ var}\ x_6 : \mathsf{array} . \mathsf{Some\ var}\ x_7 : \mathsf{array} . G.$$

This allows us to set the types of x_2 and x_5–x_8: they must all be var array. The part of π shown does not settle the types of x_1, x_3, or x_4. Note that small changes to π could change the situation. If x_1 were being used instead of x_3 in the *some-intro* rule, this would force x_1 to be list or val array: if x_1 were originally var array, it would have been marked log by the *all-intro* rule and unavailable for use in the *some-intro* rule. Similarly, if x_3 were used instead of x_8 in the *some-intro* rule, this would force x_3, x_4, x_5, and x_7 to be list or val array: var array existential witnesses must be distinct, so x_3 could not be var array.

Another complicating factor is aliasing. More will be said about this later, but it should be noted here that we need the aliasing information for Th to calculuate the aliasing information for π: x_3 and x_4 definitely alias, but we cannot know if x_2 and x_3 alias unless we know whether x_6 and x_7 can alias.

The next two sections of this chapter sketch the two phases of the inferencing algorithm. The final two sections discuss the correctness of the inferencer and techniques for improving its efficiency. The description of the inferencing algorithm is aimed toward making the algorithm understandable, not necessarily efficient. The algorithm is presented rather informally, and many simple cases are omitted, but even so, readers may find the amount of detail not to their taste. Such readers are encouraged to skim the chapter.

6.2 Marking datatypes

The first phase of the array inferencing algorithm assigns tentative datatypes (like val array) to the list identifiers of π. This includes the free and quantified identifiers of the theorem's goal and also the quantified identifiers of any added hypotheses: sequence formulas, inductive hypotheses, *impl-elim* antecedents, and so on. The assignment is done by attaching information to certain occurrences of the identifiers in the proof. If conflicting information is acquired about how to implement an identifier, the identifier's mark is changed to list.

Definition 6.1 The *implementation types* (itypes) are list, array, or int. The *storage types* (stypes) are val, var, or log. In addition, non-val, non-var, and non-log are used as storage types during part of the annotation, A *datatype* is an (*stype*, *itype*) pair.

Definition 6.2 T is an implementation type and τ is a storage type. Datatypes are written val array, not (val, array), (for example). For lists

and integers, val should be understood for the storage type if it is not explicitly mentioned.

The purpose of this phase of the algorithm is to assign datatypes to the list identifiers of π. Any particular identifier x : list can occur many times in π. For example, consider

$$[x:\text{list}]\ x = (),\ \text{All}\ x:\text{list}\,.\,F \gg \text{All}\ x:\text{list}\,.\,F \quad \text{by}\ \ldots. \tag{6.1}$$

Here, x occurs at least four times. To simplify the description of the algorithm, some of these occurrences can be removed by renaming:

$$[x:\text{list}]\ x = (),\ \text{All}\ y:\text{list}\,.\,F[y/x] \gg \text{All}\ z:\text{list}\,.\,F[z/x] \quad \text{by}\ \ldots.$$

Yet renaming must not be done indiscriminately:

$$\begin{array}{l} \text{All}\ x:\text{list}\,.\,F \gg F_1 \mid F_2 \quad \text{by intro 1} \\ \bullet\ \text{All}\ x:\text{list}\,.\,F \gg F_1 \end{array} \tag{6.2}$$

should not be renamed

$$\begin{array}{l} \text{All}\ x:\text{list}\,.\,F \gg F_1 \mid F_2 \quad \text{by intro 1} \\ \bullet\ \text{All}\ y:\text{list}\,.\,F[y/x] \gg F_1. \end{array}$$

The difference between (6.1) and (6.2) is that in (6.2) the two occurrences of All x : list . F are both justified by one expression, but in (6.1) they are justified by different expressions.

Definition 6.3 *Computationally similar* pairs of occurrences of formulas and subformulas are listed in table 6.1. Pairs of occurrences not listed are *computationally dissimilar*. Table 6.2 lists some of the computationally dissimilar occurrences of formulas. The tables only relate occurrences between the goal of a proof and its immediate subproofs. The notion must be iterated to extend to pairs of occurrences in more widely separated parts of a proof. For example, consider

$\mathcal{H} \gg F$ by seq G_1

• $\mathcal{H} \gg G_1$

• $\mathcal{H}, G_1 \gg F$ by seq G_2

•• $\mathcal{H}, G_1 \gg G_2$

•• $\mathcal{H}, G_1, G_2 \gg F$.

The three occurrences of F shown are all computationally similar, as are the five occurrences of G_1 and the three occurrences of G_2. If G_1 occurs in $\mathcal{H}$, then G_1 occurs ten times, and the ten occurrences break down into two equivalence classes of five each: one class includes the five occurrences shown explicitly above, and the other class includes the five occurrences in the $\mathcal{H}$. Similarly, if G_1 and G_2 are syntactically identical, the occurrences of G_1 as G_2 are dissimilar from the other occurrences of G_1.

Table 6.1: Computationally similar occurrences of formulas.

Occurrence of F in $\mathcal{G}$	Occurrence of F in $\mathcal{G}_i$
In non-eliminated hyp	In corresponding hyp
In *and*, *or*, *some* eliminated hyp	In corresponding new hyp(s)
In *impl-elim* hyp	In corresponding consequent
In *and*, *or*, *impl*, *all*, or *not-introduced* conclusion	In a corresponding subformula
In *false-intro* formula G	In conclusion G of $\mathcal{G}_1$
In *skolem* and *all-elim* hypotheses	In corresponding $\forall\exists$ or All formula
In a conclusion by ind	In corresponding conclusion of the inductive case
In a sequence formula G	In conclusion G of $\mathcal{G}_1$, and in the new hyp G of $\mathcal{G}_2$

Occurrence of F	Occurrence of F
In conclusion of $\mathcal{G}_1$ of *impl-elim* (consequent of eliminated formula)	In corresponding hyp of $\mathcal{G}_2$
In conclusion of $\mathcal{G}_1$ of *not-elim* (body of eliminated formula)	In corresponding hyp of $\mathcal{G}_2$

Notation: proof π, goal $\mathcal{G}$, subproofs $\sigma_1, \ldots, \sigma_k$, subgoals $\mathcal{G}_1, \ldots, \mathcal{G}_k$.

Table 6.2: Some computationally dissimilar occurrences of formulas.

Occurrence of F in $\mathcal{G}$	Occurrence of F in $\mathcal{G}_i$
In *impl-elim* hyp	In corresponding antecedent hyp of $\mathcal{G}_2$
In *all-elim* or *skolem* hyp	In corresponding substituted body or matrix
In *not-elim* hyp	In corresponding body
In *some-intro* conclusion	In conclusion of $\mathcal{G}_1$
In *false-intro* formula G	In conclusion $\neg G$ of $\mathcal{G}_2$
In a conclusion by ind	In conclusion of base case; in inductive hypotheses
hyp $y = f(\mathcal{X})$ in term rule	In new $y = f(\mathcal{X})$ hyp of $\mathcal{G}_1$

Notation: proof π, goal $\mathcal{G}$, subproofs $\sigma_1, \ldots, \sigma_k$, subgoals $\mathcal{G}_1, \ldots, \mathcal{G}_k$.

For simplicity, assume that all the free and bound identifiers of π have been uniquely renamed so that no two computationally dissimilar quantified formulas bind the same identifier. Assume also that the renaming relieves us from having to give new identifiers in the *all-intro* and *some-elim* rules. Assume furthermore that the logic has been extended to accept α-equality in some rules, like

$$\mathsf{All}\ x{:}T\,.\,F \gg \mathsf{All}\ y{:}T\,.\,F\,[y/x] \quad \text{by hyp 1.}$$

Computationally similar occurrences of formulas must have the same datatype information. For example, all the similar occurrences of the formula All x:list.F must have the same datatype for x. Instead of attaching

datatype information to every occurrence of an identifier, the inferencing algorithm attaches information to a distinguished occurrence of the identifier. Similarly, when aliases information is added to the proof in the next section, it is attached to a distinguished occurrence of each $\forall\exists$ formula.

Definition 6.4 Given a particular occurrence of a formula in a proof, the *first* occurrence that corresponds to the given occurrence is the first computationally similar occurrence found in a preorder walk of the proof. Similarly, the first occurrence of an identifier is the first occurrence found during a preorder walk. Note that if the proof were not uniquely renamed, the notion of first occurrence of an identifier would be more complicated.

Definition 6.5 An identifier is *marked* with a piece of information if the information is attached to the first occurrence of the identifier. Similarly, a formula is marked if the information is attached to its first occurrence. To *mark* a formula or identifier is to do this attachment; to *re-mark* is to replace the attached information. Marking an already-marked formula or identifier is the same as re-marking it. A formula or identifier may be marked with different kinds of information; re-marking with a particular kind of information replaces only that particular kind of information, not any others.

Notation. Double slash separates lines of a proof when displayed in-line. For example,

$$\mathcal{H} \gg F \quad \text{by seq } G$$
$$\bullet\ \mathcal{H} \gg F \ldots$$
$$\bullet\ \mathcal{H},\, G \gg F \ldots$$

becomes $\mathcal{H} \gg F$ by seq G // $\bullet\ \mathcal{H} \gg F \ldots$ // $\bullet\ \mathcal{H},\, G \gg F \ldots$.

Definition 6.6 *SameDtype* and *SameItype* are equivalence relations on identifiers. *SameDtype*(x, y) holds if x and y must have identical datatypes. *SameItype*(x, y) holds if x and y must have identical implementation types. *SameDtype*$(x_1, \ldots, x_m)$ means that $x_1, \ldots, x_m$ are equivalent under *SameDtype*. (Similar for *SameItype*.) Note that identifiers equivalent under *SameDtype* must be equivalent under *SameItype*.

These two equivalence relations are built up during the first part of the phase by inspecting π for rule instances that connect the datatypes of various identifiers. For example, say that f is a procedure whose optimal datatype is $\text{array}^2 \rightarrow \text{array}$ (so its naive datatype is $\text{list}^2 \rightarrow \text{list}$). If $\mathcal{H} \gg F$ by term $z, f(x, y)$// ... appears in π, then *SameItype*(x, y, z) holds: either x, y, and z are all arrays, or they are all lists.

Notation. If F is a formula then F^* denotes the α-equivalent formula in which each bound identifier x of F is replaced by x^*. (The appropriate

free occurrences of x are replaced also.) This notation is used in certain proof rules and procedure definitions below where occurrences of formulas are α-renamed.

Figure 6.1 lists some utility routines used to help build the *SameItype* and *SameDtype* relations.

- *MakeSameDtype*(X) makes the x_i equivalent under *SameItype* and *SameDtype*.
- *MakeSameItype*(x) makes the x_i equivalent under *SameItype*.
- *MakeSameItypeTerm*$(x_0, f(X))$ looks up the annotation of the procedure or built-in operation f and makes all the x_i in array positions equivalent under *SameItype*.
- *MakeSameDtypeFormula*(F, F^*) makes the corresponding quantified identifiers of F and F^* equal under *SameDtype* and *SameItype*. (It also makes sure that F and F^* will have identical aliases information. This is used in the next section.)
- *MakeSameDtypes*(x) finds the identifiers in the *SameItype* equivalence class of x and changes their datatypes to match x's.

Figure 6.2 lists some utility routines used to mark identifiers with implementation types and storage types.

```
MakeSameDtype(x1, ..., xm) :
    Add (x1, x2), (x2, x3), ..., (xm-1, xm) to SameDtype and SameItype;

MakeSameItype(x1, ..., xm) :
    Add (x1, x2), (x2, x3), ..., (xm-1, xm) to SameItype;

MakeSameItypeTerm(x0, f(X)) :
    Find the annotated header of f,
            say proc (x1 : τ1 T1, ..., xm : τm Tm) : τ0 T0
    X' ← the list of xi such that Ti is array
    MakeSameItype(X');

MakeSameDtypeFormula(F, F*) :
    X ← the bound identifiers of F;
    X* ← the corresponding bound identifiers of F*;
    MakeSameDtype(xi, xi*), for i = 1, ..., m;
    for each ∀∃ subformula G of F
        Add (G, G*) to SameAliases;

MakeSameDtypes(x) :
    τT ← datatype marked for x;
    for y in the SameDtype equivalence class of x
        Mark the datatype of y with τT;
```

Figure 6.1: *SameItype* and *SameDtype* utility routines.


```
MarkItype(x, T) :
    if x has no attached itype
        Attach T to x;
        MakeSameDtypes(x)
    else if x has a T' ≠ T attached
        MakeList(x);

MarkStype(x, τ) :
    if x has no attached stype
        Attach τ to x
    else
        let τ' be the storage type attached to x;
        if {τ, τ'} = { non-τ0, non-τ1 }, some τ0 ≠ τ1
            Mark the storage type of x with τ2
                    where {τ0, τ1, τ2} = {val, var, log};
            MakeSameDtypes(x)
        else if {τ, τ'} = {τ0, non-τ1}, some τ0 ≠ τ1
            Mark the storage type of x with τ0;
            MakeSameDtypes(x)
        else if τ ≠ τ'
            MakeList(x);

MarkDtype(x, τ, T) :
    MarkItype(x, T);
    if x is not marked val list
        MarkStype(x, τ);

MakeList(x) :
    V ← Equivalence class of x under SameItype;
    Mark x as val list, for all x in V
```

Figure 6.2: Datatype utility routines.

- *MarkItype*(x, T) attaches T to x. If x already has an implementation type T' not equal to T, then *MarkItype* calls *MakeList*(x) to change x and its *SameItype*-equivalents to val list.

- *MarkStype*(x, τ) works similarly: it attaches τ to x, but if x already has a storage type τ' attached and τ and τ' conflict, then x is made val list. It can also happen that τ and τ' can be combined without conflicting. For example, if τ and τ' are non-val and non-log respectively, then x's storage type becomes var.

- *MarkDtype*(x, τ, T) simply calls *MarkItype*(x, T) and *MarkStype*(x, τ).

- *MakeList*(x) re-marks the datatype of x to var list. It also re-marks the members of x's *SameItype* equivalence class.

The *BuildSameTypes* routine, shown in figures 6.3 and 6.4, constructs the *SameItype* and *SameDtype* relations. This is used in the next part of the phase, when datatypes are actually assigned to identifiers: if a datatype is assigned to an identifier, it is assigned to all its *SameDtype* equivalents also. (And similar for *SameItype*.)

Additions are made to the relations for each computation that entails a connection between the datatypes or implementation types of some identifiers. For example, for a goal $\mathcal{G}$ proved by thm Th, the identifiers marked array in the goal of Th appear in $\mathcal{G}$. The identifiers are made equivalent

BuildSameTypes(π):
  for each first occurrence of a formula y *reln* $f(\mathcal{X})$
    MakeSameItypeTerm$(y, f(\mathcal{X}))$;
  for each first occurrence of a formula x *reln* y
    MakeSameItype(x, y);
  for each $[\mathcal{E}]\ \mathcal{H} \gg F$ by thm Th
    let $[\mathcal{E}']\ \mathcal{H}^{*\prime} \gg F^*$ be the annotated goal of Th;
    $\mathcal{X} \leftarrow$ free identifiers marked array in $\mathcal{E}'$;
    $\mathcal{Y}^* \leftarrow$ quantified identifiers of $\mathcal{H}^{*\prime}$, F^* marked array;
        (So $\mathcal{Y}$ is the list of corresponding un-renamed identifiers)
    MakeSameItype$(\mathcal{X}, \mathcal{Y})$;
  for each $\mathcal{H}, u_0 = f(\mathcal{U}), \mathcal{H}' \gg F^*[u_0, \mathcal{U}/x_0, \mathcal{X}]$ by def $u_0, f(\mathcal{U})$,
      where f is extracted from a theorem with matrix F
    let $\tau_1\, T_1 \times \cdots \tau_m\, T_m \rightarrow \tau_0\, T_0$ be the annotated type of f;
    $\mathcal{Y} \leftarrow$ quantified array identifiers of F;
    MakeSameItype$(\{u_i \mid T_i = \text{array}\}, \mathcal{Y}^*)$;
  for each $\mathcal{H}, F, \mathcal{H}' \gg F'$ by elim $n, \mathcal{U} // \bullet\ \mathcal{H}, F^*, \mathcal{H}', G^*[\mathcal{U}/\mathcal{X}] \gg F'$
      where F is All $x_1 : T_1, \ldots, x_m : T_m \,.\, G$
    if G contains an existential, universal, or implication
      MakeSameDtype(x_i, u_i), all $i = 1, \ldots, m$
    else
      MakeSameItype(x_i, u_i), all $i = 1, \ldots, m$;
    MakeSameDtypeFormula(F, F^*);
    MakeSameDtypeFormula$(G, G^*[\mathcal{U}/\mathcal{X}])$;

Figure 6.3: The *BuildSameTypes* routine (part 1).

for each $\mathcal{H} \gg$ Some $x_1:T_1,\ldots,x_m:T_m$ by intro $\mathcal{U}$
  MakeSameDtype(x_i,u_i), all $i=1,\ldots,m$;
for each $\mathcal{H},F,\mathcal{H}' \gg F'$ by skolem $n,\mathcal{U},\mathcal{U}'$//
    • $\mathcal{H},F^*,\mathcal{H}',G^*\left[\mathcal{U},\mathcal{U}'/\mathcal{X},\mathcal{X}'\right] \gg F'$
  where F is All $x_1:T_1,\ldots,x_m:T_m$. Some $x_1':T_1',\ldots,x_r':T_r'$. G
  MakeSameDtype(x_i,u_i), all $i=1,\ldots,m$;
  MakeSameDtype(x_i',u_i'), all $i=1,\ldots,r$;
  MakeSameDtypeFormula(F,F^*);
  MakeSameDtypeFormula$(G,G^*\left[\mathcal{U},\mathcal{U}'/\mathcal{X},\mathcal{X}'\right])$;
for each $\mathcal{H} \gg$ All x:list . F by ind $y.z$//• $\mathcal{H} \gg F^*[()/x]$ //
    • $\mathcal{H},F^{**}[z/x], x=y.z \gg F$
  MakeSameDtypeFormula$(F,F^*[()/x])$;
  MakeSameDtypeFormula$(F,F^{**}[z/x])$;
(Integer induction is treated similarly)
for each $\mathcal{H},F\Rightarrow G,\mathcal{H}' \gg F'$ by elim n//• $\mathcal{H},\mathcal{H}' \gg F^*$//
    • $\mathcal{H},\mathcal{H}',F^*,G \gg F'$
  MakeSameDtypeFormula(F,F^*);
for each $\mathcal{H},F,\mathcal{H}' \gg F^*$ by hyp n
  MakeSameDtypeFormula(F,F^*);
for each $\mathcal{H} \gg$ false by intro F//• $\mathcal{H} \gg F$//• $\mathcal{H} \gg \neg F^*$
  MakeSameDtypeFormula(F,F^*);
for each $\mathcal{H},\neg F,\mathcal{H}' \gg G$ by elim n//• $\mathcal{H},\mathcal{H}' \gg F^*$//
    • $\mathcal{H},\mathcal{H}',F^*$, false $\gg G$
  MakeSameDtypeFormula(F,F^*);

Figure 6.4: The *BuildSameTypes* routine (part 2).

under *SameItype*: either they are all implemented as arrays (if optimal code for *Th* is used), or they are all lists (if the naive code is used). The *def-extr* rule is similar: the identifiers that appear as arguments for array parameters are equal under *SameItype*.

Additions to *SameItype* and *SameDtype* are also made for the quantifier rules. In *skolem* and *some-intro*, the datatype of each eliminated and introduced identifier must match the datatype of its corresponding universal or existential identifier; the two identifiers are made equal under *SameDtype*. The *all-elim* rule is similar but can induce a weaker connection: the implementation types of an identifier and its corresponding universal parameter must match, but the storage types do not always have to match. (The definition of the *all-elim* rule gives specifics.) For example, take

$$[\mathcal{E}]\ \text{All } x_1:T_1,\ldots,x_n:T_n \,.\, F \gg G \quad \text{by elim } 1,y_1,\ldots,y_n.$$

If the storage types do not have to match, then each pair (x_i,y_i) are made equal under *SameItype*, else they are made equal under *SameDtype*.

In addition, *BuildSameTypes* looks for formulas like x *reln* y and x_0 *reln* $f(\mathcal{X})$. In x *reln* y, x and y must have the same implementation type. In x_0 *reln* $f(\mathcal{X})$, the x_i that correspond to array arguments or array results of f must all have the same implementation type.

Once *SameItype* and *SameDtype* are built, implementation and storage types can be assigned to the free and bound identifiers of π. Figures 6.5

and 6.6 show the *GenDtypes* routine, which does the first part of this assignment. The assignments break down into various cases:

- Every identifier whose first occurrence is quantified is marked non-log, since quantified identifiers may not be logical. Similarly, identifiers used in quantifier rules are marked non-log. Free identifiers that appear in the top-level goal of π are marked val, since they may not be var or log. Top-level existential witnesses that would be state-dependent if they were var are marked val for the same reason.

- Information from the library is used to mark appropriate identifiers in *theorem* and *def-extr* rules. For example, if a goal $\mathcal{G}$ is proved by thm *Th*, then the datatypes for the annotated goal of *Th* are used to mark corresponding identifiers in $\mathcal{G}$.

- Many rules make var identifiers log in certain subproofs. (The *sequence* rule is an example.) An identifier used computationally in such a subproof must therefore be non-var.

Figures 6.7 through 6.9 contain the compatibility checking routines (*CC*, for short), which make up the next part of the datatype assignment. *CC* inspects the *term*, *theorem*, and *def-extr* rules of π and decides which of them can use optimal code for the referenced functions and lemmas and which of them must use the naive code. *CC* also decides which *term* rules

GenDtypes(π):
for each first occurrence of a quantified identifier x in π
MarkStype(x, non-log);
for each first occurrence of a formula y *reln* $f(\mathcal{X})$
let $\tau_1 T_1 \times \cdots \tau_m T_m \rightarrow \tau_0 T_0$ be the annotated type of f;
MarkItype(x_i, T_i), all $i = 0, \ldots, m$;
for each $\mathcal{H}, F, \mathcal{H}' \gg G$ by elim $n, \mathcal{U}$
where F is All $x_1 : T_1, \ldots, x_m : T_m . G$
MarkStype(x_i, non-log), all $i = 1, \ldots, m$;
MarkStype(u_i, non-log), all $i = 1, \ldots, m$;
(Similar for the all-intro, some-intro, some-elim, and skolem rules)
for each $[\mathcal{E}]\ \mathcal{H} \gg F$ by thm *Th*
let $[\mathcal{E}']\ \mathcal{H}^{*\prime} \gg F^*$ be the annotated goal of *Th*;
for each $x : \tau\, T$ in $\mathcal{E}'$
MarkDtype(x, τ, T);
for each quantified identifier $x^* : \tau\, T$ in $\mathcal{H}^{*\prime}$ and F^*
MarkDtype(x, τ, T);
for each $\mathcal{H}, u_0 = f(\mathcal{U}), \mathcal{H}' \gg F^*[u_0, \mathcal{U}/x_0, \mathcal{X}]$ by def $u_0, f(\mathcal{U})$
where f is an extracted procedure
let All $x_1 : \tau_1 T_1, \ldots, x_m : \tau_m T_m$. Some $x_0 : \tau_0 T_0 . F$
be the annotated goal of f's theorem;
if $\tau_0 T_0$ = var array
MarkStype(u_i, var), all $i = 0, \ldots, m$, where τ_i = var;
MarkStype(u_i, non-log), all $i = 0, \ldots, m$, where τ_i = val;
else
MarkStype(u_i, τ_i), all $i = 0, \ldots, m$;
let $y_1 : \tau_1' T_1', \ldots, y_r : \tau_r' T_r'$ be the quantified identifiers of F;
MarkDtype(y_i^*, τ_i', T_i'), all $i = 0, \ldots, r$;

Figure 6.5: The *GenDtypes* routine (part 1).

```
for each H >> All x:list . F by ind y . z// ...
    MakeList(x);
    MakeList(z);
for each free identifier x of the top-level goal
    MarkStype(x, val);
for each hypothesis H_i of the top-level goal
    for each existentially quantified identifier x of H_i
            that would be state-dependent if it were var
        MarkStype(x, val);
for each π' = [E] H >> F & G by intro// ...
    for each H in H and each subproof σ_i of π'
        if H is used in σ_i or its descendents
            for each existentially quantified identifier x of H_i
                    that would be state-dependent if it were var
                MarkStype(x, val);
        for each free identifier x in E
            if x is used computationally in σ_i or its descendents,
                    but not just in term rules
                MarkStype(x, val)
            else if x is used in σ_i or its descendents only in term rules
                MarkStype(x, non-var);
(Similar for all other rules that drop var environment
        identifiers or state-dependent hypotheses)
```

Figure 6.6: The *GenDtypes* routine (part 2).


```
CC(π) : (* CheckCompatibility *)
    let R ← the set of all instances of term, def-ext, theorem,
            and all-elim rules of π;
    while R ≠ ∅
        remove a rule instance R from R;
        case kind of rule
            term → CC_term(R, R);
            def-ext → CC_def-ext(R, R);
            ...

CC_theorem(R, R) :
    let R be [E] H >> F by thm Th;
    let the annotated goal of Th be [E'] H*' >> F*;
    if some x : r' T' in E' is not marked r' T'
          or some quantified identifier x* : r' T' in H*' or F*
            corresponds to an x not marked r' T'
        Mark the rule naive;
        AddRules(x, R)
    else
        Mark the rule optimal;

AddRules(x, R) :
    MakeList(x);
    for each y in the SameItype-equivalence class of x
        Add to R all term, theorem, def-ext, and all-elim rule
                instances not already in R but in which y occurs;
    (* The relevant occurrences are in the rules themselves, for
            the term, def-ext, and all-elim rules, and in the proof
            environment of the theorem goal, for the theorem rule *)
```

Figure 6.7: Compatibility checking (part 1).

$CC\_term(R, \mathcal{R})$:
let R be $\mathcal{H} \gg F$ by term $x_0, f(\mathcal{X})$;
let $\tau_0\, T_0, \ldots \tau_m\, T_m$ be the datatypes of the x_i;
let $\tau_1'\, T_1' \times \cdots \tau_m'\, T_m' \rightarrow \tau_0'\, T_0'$ be the annotated datatype of f;
if $T_i = T_i'$, all $i = 0, \ldots, m$
if ($\tau_0'\, T_0'$ = var array but some τ_i, τ_i' are not strongly compatible)
or ($\tau_0'\, T_0' \neq$ var array but some τ_i, τ_i' are not weakly compatible)
or τ_0, τ_0' are not strongly compatible
Mark the rule *logical*;
$MarkStype(x_i, \text{log})$;
if x_0 is now marked val list
Re-mark the rule *naive* and *computational*;
$AddRules(x_i, \mathcal{X})$
else
Mark the rule *optimal* and *computational*;
$MarkStype(x_i, \tau_i')$, all $i = 0, \ldots, m$
except where (τ_i = var and τ_i' = val)
else (* *there exists some* $T_i \neq T_i'$ *)
Mark the rule *naive* and *computational*;

Figure 6.8: Compatibility checking (part 2).

$CC\_def\text{-}ext(R, \mathcal{R})$:
let R be $\mathcal{H}, u_0 = f(\mathcal{U}), \mathcal{H}' \gg F^*[u_0, \mathcal{U}/x_0, \mathcal{X}]$ by def $u_0, f(\mathcal{U})$;
let $\tau_0\, T_0 \times \cdots \tau_m'\, T_m'$ be the annotated types of the x_i;
let $\tau_1'\, T_1' \times \cdots \tau_m'\, T_m' \rightarrow \tau_0'\, T_0'$ be the annotated type of f;
if $T_i = T_i'$ all $i = 0, \ldots, m$
if some τ_i, τ_i' are not strongly compatible
Mark the rule *naive*;
$AddRules(x_i, \mathcal{R})$
else
Mark the rule *optimal*;
$MarkStype(x_i, \tau_i')$, all $i = 0, \ldots, m$
except where τ_i = var and τ_i' = val
else
Mark the rule *naive*;

$CC\_all\text{-}elim(R, \mathcal{R})$:
let R be $\mathcal{H}, F, \mathcal{H}' \gg F'$ by elim $u, \mathcal{U}$ // ...
where F is All $x_1 : \tau_1\, T_1, \ldots, x_m : \tau_m\, T_m \,.\, G$;
let $\tau_1'\, T_1', \ldots \tau_m'\, T_m'$ be the datatypes of the u_i;
if $T_i = T_i'$, all $i = 1, \ldots, m$
if (G contains an existential, universal, or implication
but some τ_i, τ_i' are not strongly compatible)
or (G contains no such subformula
but some τ_i, τ_i' are not weakly compatible)
$AddRules(u_i, \mathcal{R})$
else
$MarkStype(u_i, \tau_i')$, all $i = 1, \ldots, m$
except where τ_i = var and τ_i' = val;

Figure 6.9: Compatibility checking (part 3).

are computational and which are logical. It marks rules ***naive/optimal*** and ***logical/computational*** as appropriate. In addition, *CC* verifies that identifiers used in ***all-elim*** and ***skolem*** rules have the appropriate storage types.[5]

Let $[\mathcal{E}]\ \mathcal{H} \gg F$ by thm *Th* be a theorem rule instance, where the annotated goal of *Th* is $[\mathcal{E}']\ \mathcal{H}^{*\prime} \gg F^{*}$. For *CC* to mark the rule instance ***optimal***, the datatypes of the free identifiers of $\mathcal{E}'$ must match their datatypes in $\mathcal{E}$, and the datatypes of the quantified identifiers of $\mathcal{H}^{*\prime}$ and F^{*} must match the corresponding quantified identifiers in $\mathcal{H}$ and F.

Definition 6.7 Datatypes τ and τ' are ***strongly compatible*** if

$$\{\tau, \tau'\} \notin \{\{\mathsf{val}, \mathsf{non\text{-}val}\}, \{\mathsf{var}, \mathsf{non\text{-}var}\}, \{\mathsf{log}, \mathsf{non\text{-}log}\}, \{\mathsf{val}, \mathsf{var}\}, \{\mathsf{val}, \mathsf{log}\}, \{\mathsf{log}, \mathsf{var}\}\}.$$

They are ***weakly compatible*** if they are strongly compatible or if $\tau = \mathsf{var}$ and $\tau' = \mathsf{val}$.

For a term rule instance $\mathcal{H} \gg F$ by term $x_0, f(\mathcal{X})$ // $\mathcal{H}, x_0 = f(\mathcal{X}) \gg F$ to be computational and optimal, the implementation types of the x_i must match the optimal implementation types listed in the annotated header of f, and the storage types of the x_i must be compatible. (See *CC_term* for details.) The ***term*** rule is marked ***logical*** if the implementation types match

[5] It is important to keep in mind that neither *CC* nor *GenDtypes* pay any attention to aliasing. Rules marked *optimal* here can be re-marked *naive* by the aliasing checks in the next section. Similarly, an identifier marked array can be re-marked list.

but the storage types are incompatible. If the implementation types do not match, the rule is marked ***naive***.

The ***def-extr*** rule is treated like the ***term*** rule. Let

$$\mathcal{H}, u_0 = f(\mathcal{U}), \mathcal{H}' \gg F^{*}[u_0, \mathcal{U}/x_0, \mathcal{X}] \quad \text{by def } u_0, f(\mathcal{U})$$

be a ***def-extr*** rule instance, where f was extracted from a theorem with annotated goal

$$[\,] \gg \mathsf{All}\ x_1 : \tau_1\, T_1, \ldots, x_m : \tau_m\, T_m\,.\ \mathsf{Some}\ x_0 : \tau_0\, T_0\,.\ F.$$

For *CC* to mark the rule instance ***optimal***, the implementation types of the u_i must match the T_i and the storage types of the u_i must be compatible. If the rule cannot be marked ***optimal***, it is marked ***naive***, and all u_i marked array are re-marked list.

The ***all-elim*** rule is not marked ***optimal*** or ***naive***; it is simply checked to make sure that the storage types of the various identifiers are compatible. If the body of the eliminated universal contains an embedded existential, universal, or implication, then the storage type τ of an identifier being used to eliminate with must be strongly compatible with the storage type τ' of the eliminated parameter. If the body of the universal contains none of the above subformulas, then τ and τ' only need to be weakly compatible.

Once the compatibility of the various rules has been checked, default datatypes can be assigned to any leftover identifiers. Figure 6.10 shows the

```
DefaultDtypes:
    for each first occurrence of a list identifier x
                not marked with an implementation type
        MarkDtype(x, val list);
    for each first occurrence of an identifier x marked non-log
        MarkStype(x, val);
```

Figure 6.10: Default datatypes.

DefaultDtypes routine, which does this assignment. Identifiers of type list that have no marked datatype are marked val list. From the structure of the algorithms, such identifiers must be free in the top-level environment of π and are not used computationally or in formulas, so there is no harm in implementing them as lists. Array identifiers marked non-log are quantified identifiers not used in rules connected to the library: they might be used in the quantifier or *skolem* rules, for example, but not in the *term*, *theorem*, or *def-extr* rules.[6] Such identifiers are re-marked val.

Once default datatypes are assigned, every list identifier in π has a datatype. Routine CL, shown in figures 6.11 through 6.13, checks π for computational uses of logical identifiers. The check is done by recursively walking the proof tree looking for uses of var identifiers that would be log at the places they are used. CL takes three parameters: σ, the current

[6] The canonical proof of || All x : list . (F & G) >> All x : list . F is an example.


```
CL(σ, V_var, V_log):     (* Check use of logical identifiers *)
    case rule of σ
        and-intro → CL_and-intro(σ, V_var, V_log);
        and-elim → CL_and-elim(σ, V_var, V_log);
        ...;

CL_and-intro(σ, V_var, V_log):
    CL(σ1, ∅, V_var + V_log);
    CL(σ2, ∅, V_var + V_log);

CL_and-elim(σ, V_var, V_log):
    CL(σ1, V_var, V_log);

CL_seq(σ, V_var, V_log):
    CL(σ1, ∅, V_var + V_log);
    CL(σ2, V_var, V_log);

CL_all-intro(σ, V_var, V_log):
    let σ be H >> All x1 : τ1 T1, ..., xm : τm Tm . F by intro // ...;
    CL(σ1, {xi | τi Ti = var array}, V_var + V_log);

CL_all-elim(σ, V_var, V_log):
    let σ be H, F, H' >> G by elim n, U // ...;
            where F is All x1 : τ1 T1, ..., xm : τm Tm . G
    if some ui is in V_log
        MakeNaive(ui);
    CL(σ1, V_var − U_var, V_log + U_var);
                where U_var is {xi | τi Ti = var array}
```

Figure 6.11: Checking the use of logical identifiers (part 1).

CL_some-elim$(\sigma, \mathcal{V}_{var}, \mathcal{V}_{log})$:
  let σ be $\mathcal{H}, F, \mathcal{H}' \gg G$ by elim n // ...;
    where F is Some $x_1 : \tau_1\, T_1, \ldots, x_m : \tau_m\, T_m \,.\, G$
  $CL(\sigma_1, \mathcal{V}_{var} + \{x_i \mid \tau_i T_i = \text{var array}, \mathcal{V}_{log}\})$;

CL_some-intro$(\sigma, \mathcal{V}_{var}, \mathcal{V}_{log})$:
  let σ be $\mathcal{H} \gg$ Some $x_1 : \tau_1\, T_1, \ldots, x_m : \tau_m\, T_m \,.\, G$ by intro $\mathcal{U}$ // ...
  if some u_i is in $\mathcal{V}_{log}$
    MakeNaive(u_i);
  $CL(\sigma_1, \mathcal{V}_{var} - \mathcal{U}_{var}, \mathcal{V}_{log} + \mathcal{U}_{var})$
    where $\mathcal{U}_{var}$ is $\{u_i \mid u_i$ is marked var array$\}$

CL_skolem$(\sigma, \mathcal{V}_{var}, \mathcal{V}_{log})$:
  (Similar to CL_all-elim and CL_some-elim);

CL_term$(\sigma, \mathcal{V}_{var}, \mathcal{V}_{log})$:
  let σ be $\mathcal{H} \gg F$ by term $u_0, f(\mathcal{U})$ // ...
  if the rule is marked *logical*
    $CL(\sigma_1, \mathcal{V}_{var}, \mathcal{V}_{log})$;
  else if the rule is marked *naive (computational)*
    $CL(\sigma_1, \mathcal{V}_{var}, \mathcal{V}_{log})$
  else if some u_i marked var array is in $\mathcal{V}_{log}$
    MakeNaive(u_i);
    $CL(\sigma_1, \mathcal{V}_{var}, \mathcal{V}_{log})$
  else
    $CL(\sigma_1, \mathcal{V}_{var} - \mathcal{U}_{var}, \mathcal{V}_{log} + \mathcal{U}_{var})$
      where $\mathcal{U}_{var}$ is $\{u_i \mid i$'th parameter of f is marked var array$\}$;

Figure 6.12: Checking the use of logical identifiers (part 2).

CL_def-ext$(\sigma, \mathcal{V}_{var}, \mathcal{V}_{log})$:
  let σ be $\mathcal{H}, u_0 = f(\mathcal{U}), \mathcal{H}' \gg F^* [u_0, \mathcal{U}/x_0, \mathcal{X}]$ by def $u_0, f(\mathcal{U})$
  if the rule is marked *naive*
    $CL(\sigma_1, \mathcal{V}_{var}, \mathcal{V}_{log})$
  else if some u_i marked var array is in $\mathcal{V}_{log}$
    MakeNaive(u_i);
    $CL(\sigma_1, \mathcal{V}_{var} - \mathcal{U}_{var}, \mathcal{V}_{log} + \mathcal{U}_{var})$
      where $\mathcal{U}_{var}$ is $\{u_i \mid i$'th parameter of f is marked var array$\}$

MakeNaive(x) :
  MakeList(x);
  $\mathcal{X} \leftarrow$ *Sameltype*-equivalence class of x;
  for each identifier u in $\mathcal{X}$
    for each occurrence of u in a *term* rule
      by term $u_0, f(\mathcal{U})$ marked *optimal computational*
      Mark the *term* rule *naive*;
    for each occurrence of u in a *def-ext* rule
      by def $u_0, f(\mathcal{U})$ marked *optimal*
      Mark the *def-ext* rule *naive*;
    for each occurrence of u in the environment
      of an *theorem* rule by thm *Th* marked *optimal*,
      where u appears in the environment of *Th*
      Mark the *theorem* rule *naive*;

Figure 6.13: Checking the use of logical identifiers (part 3).

subproof of π, and $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$, the sets of variable and logical identifiers for the environment of σ. These sets can be supersets of their counterparts in the final version of the annotation, since identifiers in $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ may later be re-marked list. As will be seen in the proof of correctness for the algorithm, it is not a problem for these identifiers to be re-marked: any identifier that fails the tests here, with the too-large $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ sets, would have to be re-marked list eventually anyway.

The *CL* routine builds $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ during the walk of π by using the environment restrictions from the rules of the logic. At the top level, $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ are empty because the top-level goal of a theorem may have only free val identifiers. (The *GenDtypes* routine makes all of the top-level identifiers val for this reason.) In the general case, σ is the current subproof of π (or subsubproof, etc). The rule of σ determines how $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ change for the different subproofs of π. For example, in the ***and-intro*** rule both subgoals have no var identifiers: the var identifiers of their parent become log. Accordingly, the *CL_and-intro* routine calls *CL* on each σ_i with $\emptyset$ and $\mathcal{V}_{var} + \mathcal{V}_{log}$. (In the figures, the subproofs of σ are denoted σ_1, σ_2, and so on.)

The *CL* routines for certain rules need to check for the computational use of logical identifiers. For example, if a ***term*** rule instance marked ***optimal computational*** contains the term $f(\mathcal{X})$, then *CL_term* makes sure that

none of the x_i are in $\mathcal{V}_{log}$. (The *CL* routines for the quantifier rules, the ***def-extr*** rule, and the ***theorem*** rule are similar.) If one of the x_i is in $\mathcal{V}_{log}$, then the rule is re-marked ***naive computational***, and the offending x_i is re-marked list. The *MakeNaive* routine, shown in figure 6.13, re-marks x_i. Since re-marking x_i also changes all the identifiers *SameItype*-equivalent to it, *MakeNaive* also looks for ***term***, ***theorem***, and ***def-extr*** rule instances that use identifiers *SameItype*-equivalent to x_i. It re-marks these rules ***naive*** or ***naive computational***.

The first phase of the inferencing algorithm ends with every list identifier of π marked with a datatype. The datatype for each identifier x is valid in the sense that it consistent with how x is used computationally — in the quantifier, ***term***, ***theorem***, and ***def-extr*** rules. The datatype is also consistent with the restrictions on logical identifiers. The next section adds aliases fields to the $\forall\exists$ formulas of π and checks for illegal aliasing of existential witnesses.

6.3 Adding and checking aliasing information

Phase two of the inferencing algorithm marks the $\forall\exists$ formulas of π with aliasing information. The first occurrence of each computationally distinct $\forall\exists$ formula is marked with an aliases field. (For simplicity, even $\forall\exists$ formulas without array datatypes are marked, with aliases $\emptyset$.) Phase two of the algorithm also verifies that var array identifiers are not aliased when they should not be: existential witnesses may only be aliased to universal parameters in $\forall\exists$ formulas; other existential witnesses must be fresh.

For an example, consider the following proof. The annotated datatypes are shown in place of the various list keywords.

[var x_1] $\gg$ All var x_2, x_3 . Some var x_4, x_5 . F

  by seq All var x_6 . Some var x_7 . G

• [var x_1] $\gg$ All var x_6 . Some var x_7 . G by thm Th

• [var x_1] All var x_6 . Some var x_7 . G

  $\gg$ All var x_2, x_3 . Some var x_4, x_5 . G by intro

•• [var x_2, x_3; log x_1] All var x_6 . Some var x_7 . G

  $\gg$ Some var x_4, x_5 . G by skolem $1, x_2, x_8$

••• [var x_3, x_8; log x_1, x_2] All ..., $G[x_2, x_8/x_6, x_7]$

  $\gg$ Some var x_4, x_5 . G

  by term x_9, $update\_optimal(x_3, 1, 0)$

•••• [var x_8, x_9; log x_1, x_2, x_3] All ..., $G[\ldots]$,

  $x_9 = update\_optimal(x_3, 1, 0)$

  $\gg$ Some var x_4, x_5 . G by intro x_9, x_8

....

The proof as shown certainly aliases x_3 to x_4, but does it alias x_2 to x_5? It depends on the aliasing behavior of the proof of Th: if aliases (x_6, x_7) is attached to the goal of Th, then x_2 and x_8 may be aliased, so x_2 and x_5 may also be. Note that if x_1 or x_3 are substituted for x_8 in the *skolem* rule, changes to the datatypes would have to be made. If x_1 were used in place of x_8, it would have to be re-marked list, since an $\forall\exists$ formula may not return a var array unless it is a parameter. This would force x_5 to be re-marked list also. Similarly, if x_3 were used for x_8, the witness instances for x_4 and x_5 would not be distinct, so x_3 would have to be re-marked list, and this would force x_4 and x_5 to be re-marked list.

This example shows how aliasing information is calculated: the structure of the proof is used along with information from the annotated members of the library. In addition, the flow of the proof is important: Th affects the aliasing of x_2 and x_5 because the justification-transformer for Th is executed before the extracted code that calculates the value of x_5.

Figure 6.14 contains the *MarkAliases* routine. It is used to attach an aliases field to an $\forall\exists$ formula F. (For efficiency, *MarkAliases* attaches the information to the first occurrence of F, not every occurrence of F.) If F already has aliasing information attached to it, the new information is united with it. The union is checked to make sure that it obeys the restriction that it may not mention any identifier more than once. If the new

```
MarkAliases(F, A):
  Â ← aliases field attached to the first occurrence of F   (∅ if none);
  if Â + A mentions some identifier x: array more than once
    MakeList(x);
    Mark the first occurrence of F with aliases ∅
  else if F has an x: array quantified identifier
    Mark the first occurrence of F with aliases Â + A
  else
    Mark the first occurrence of F with aliases ∅;
  ℱ ← equivalence class of F under SameAliases,
        minus the first occurrence of F;
  for each first occurrences of an F′ in ℱ
      Mark the first occurrence of F′ with the same
            aliases information as F;
```

Figure 6.14: The *MarkAliases* routine.

aliases information is illegal, then the array datatypes of F are changed to list and aliases $\emptyset$ is attached instead. Since formulas *SameAliases*-equivalent to F are supposed to have the same aliases information as F, *MarkAliases* also attaches the new information to them. (*SameAliases* was built during phase one of the inferencing algorithm: whenever *MakeSameDtypeFormula* was called with G and G^*, (G, G^*) was added to *SameAliases*.)

The actual addition and checking of aliasing is done by walking π in *execution* order: with only a few exceptions, proof nodes are visited in the

order that their extracted code will be executed.[7] This ordering enables aliasing information from the library to flow into the proof from ***comp-term***, ***def-extr***, and ***theorem*** rules.

Figure 6.15 lists some of the routines used to add and check the aliasing information. Each of the ACA routines takes a subproof σ of π, an aliased $\mathcal{A}$ field, and a fresh $\mathcal{V}_{fr}$ field. It walks σ and returns a pair containing the aliases and fresh information effective after σ.[8] This new information is designed to be suitable for passing on to the next sibling of σ (presumably by an unfinished recursive call of ACA). The figure also includes ACA routines for some simple rules like ***and-intro*** and ***and-elim***. The routines for most of the simple cases are omitted.

Figures 6.16 and 6.17 list the ACA routines for the quantifier rules, ***all-intro***, ***all-elim***, ***some-intro***, ***some-elim***, and ***skolem***. The interesting cases are ***all-elim***, ***some-intro***, and ***skolem***.

- *ACA_**all-elim*** checks to make sure that the var identifiers used to eliminate with are distinct.

[7] The exceptions include rules like *or-elim*, where only one of the subgoals has its code executed, and *list-ind*, where the aliasing done by the inductive case depends on the aliasing done by the inductive hypothesis. For the *or-elim* rule in particular, the aliasing information produced by the two subproofs is simply unioned; for the *list-ind* rule, the aliasing produced by the base case is iterated through the inductive case until the union of the aliasing information stabilizes.

[8] The routine actually returns supersets of aliases and fresh. As will be seen in the proof of correctness for the inferencing algorithm, this turns out not to be harmful.

```
ACA(σ, A, V_fr):
    (* Add and check aliasing — returns A and V_fr
        after executing the code for σ *)
  case rule of σ
    and-intro → return ACA_and-intro(σ, A, V_fr);
    and-elim → return ACA_and-elim(σ, A, V_fr);
    ...;

ACA_and-intro(σ, A, V_fr):
  ACA(σ1, A, ∅);
  ACA(σ2, A, ∅);
  return (A, V_fr);

ACA_and-elim(σ, A, V_fr):
  return ACA(σ1, A, V_fr);

ACA_or-intro(σ, A, V_fr):
  return ACA(σ1, A, V_fr);

ACA_or-elim(σ, A, V_fr):
  (A', V'_fr) ← ACA(σ1, A, V_fr);
  return ACA(σ2, A', V'_fr);

ACA_seq(σ, A, V_fr):
  (A', V'_fr) ← ACA(σ1, A, V_fr);
  return ACA(σ2, A', V'_fr);
```

Figure 6.15: Adding and checking aliasing (part 1).


```
ACA_all-intro(σ, A, V_fr):
  ACA(σ1, A, ∅);
  return (A, V_fr);

ACA_all-elim(σ, A, V_fr):
  let σ be H, F, H' ≫ F' by elim n, U // ...;
        where F is All x1 : τ1 T1, ..., xm : τm Tm . G;
  for each u_p = u_q
        where p ≠ q, T_p = T_q = array, and (τ_p or τ_q is var)
     MakeNaive(u_p);
  return ACA(σ1, A, V_fr − U_var)
        where U_var is {u_i | τ_i T_i = var array};

ACA_some-intro(σ, A, V_fr):
  let σ be H ≫ F by intro U' // ...
        where F is Some y1 : τ'1 T'1, ..., yr : τ'r T'r . F;
  for each u'_p = u'_q
        where p ≠ q and τ'_p T'_p = τ'_q T'_q = var array
     MakeNaive(u'_p);
  if F was part of an ∀∃ conclusion G
    let G be All x1 : τ1 T1, ..., xm : τm Tm . F;
    for each u'_i : var array not a member of V_fr and not related
          by Closure(A) + U to any x_j : var array
       MakeNaive(u'_i);
    A' ← {{x_i, x'_j} | x_i, u'_j : var array and {x_i, u'_j} ∈ Closure(A) + U'};
    MarkAliases(G, A')
  else
    for each u'_i : var array not in V_fr
       MakeNaive(u'_i);
  return ACA(σ1, A, V_fr − U');
```

Figure 6.16: Adding and checking aliasing (part 2).

ACA_some-elim$(\sigma, \mathcal{A}, \mathcal{V}_{fr})$:
  let σ be $\mathcal{H}, F, \mathcal{H}' \gg F'$ by elim n // ...
    where F is Some $x_1 : \tau_1\, T_1, \ldots, x_m : \tau_m\, T_m \,.\, G$;
  return $ACA(\sigma_1, \mathcal{A}, \mathcal{V}_{fr} + \{x_i \mid x_i : \text{var array}\})$;

ACA_skolem$(\sigma, \mathcal{A}, \mathcal{V}_{fr})$:
  let σ be $\mathcal{H}, F, \mathcal{H}' \gg F'$ by skolem $n, \mathcal{U}, \mathcal{U}'$ // ...
  let $\mathcal{X}$, $\mathcal{X}'$ be the outermost universal and existential identifiers of F;
  for each $u_p = u_q$
    where $p \neq q$, $T_p = T_q$ = array, and (τ_p or τ_q is var)
   $MakeNaive(u_p)$;
  $\mathcal{A}' \leftarrow$ aliases field attached to first occurrence of F ($\emptyset$ if none);
  $\mathcal{V}'_{fr} \leftarrow \{u'_j \mid u'_j : \text{var array and for no } i \text{ is } \{x_i, x'_j\} \in \mathcal{A}'\}$;
  return $ACA(\sigma_1, \mathcal{A} + \mathcal{A}'\left[\mathcal{U}, \mathcal{U}'/\mathcal{X}, \mathcal{X}'\right], \mathcal{V}_{fr} + \mathcal{V}'_{fr} - \mathcal{U}_{var})$
    where $\mathcal{U}_{var}$ is $\{u_i \mid i\text{'th univ. param. is marked var array}\}$;

Figure 6.17: Adding and checking aliasing (part 3).

- *ACA_some-intro* checks to see if its introduced existential was part of an $\forall\exists$ formula being proved. If so, it calculates an aliases field for the formula and marks the formula with it. In any case, it makes sure that the var array witnesses are distinct.

- *ACA_skolem* works in reverse of *ACA_some-intro*: it looks up the aliases field for its $\forall\exists$ formula and uses it to update the current list of aliased identifiers. It also uses the aliases field to find the existential witnesses that should be made fresh in the subproof.

Figure 6.18 contains the *ACA* routines for the ***theorem*** and ***def-extr*** rules. *ACA_theorem* copies aliases information from the referenced theorem into the proof π. *ACA_def-extr* is similar: if the rule is marked ***optimal***, then *ACA_def-extr* copies in aliases information for the matrix of the $\forall\exists$ formula from which the referenced function was extracted. If nondistinct var arrays are passed to the function, the offending identifiers are changed to lists and the ***def-extr*** rule is re-marked ***naive***.

Figure 6.19 contains the *ACA* routines for the ***term*** and ***list-ind*** rules. If *ACA_term* is given a rule marked ***optimal computational*** then it checks the var array arguments to make sure they are distinct and then uses the library information on the given function to update its information on the currently aliased and fresh identifiers. If the rule fails the test, then its var array identifiers are re-marked val list and the rule is re-marked ***naive***. *ACA_list-ind*

```
ACA_theorem(σ, A, V_fr):
    let σ be [E] H ≫ F by thm Th;
    if the rule is marked optimal
        let [E'] H*' ≫ F* be the annotated goal of Th;
        for each ∀∃ subformula G* of H*' and F*
            let A* be the aliases field attached to G*;
            MarkAliases(G, A);
    return (A, V_fr);

ACA_def-extr(σ, A, V_fr):
    let σ be H ≫ F*[u_0, U/x_0, X] by def u_0, f(U);
    if the rule is marked naive
        return (A, V_fr)
    else (* the rule is marked optimal *)
        let f be extracted from a theorem with annotated goal
                All x_1 : τ_1 T_1, ..., x_m : τ_m T_m . Some x_0 : τ_0 T_0 . F;
        if some u_p = u_q
                where p ≠ q, T_p = T_q = array, and (τ_p or τ_q = var)
            MakeNaive(u_p);
            return (A, V_fr)
        else
            for each ∀∃ subformula G of F
                let A be the aliases information for G;
                MarkAliases(G*[u_0, U/x_0, X], A*);
            return (A, V_fr − {u_i | 1 ≤ i ≤ m and u_i : var array});
```

Figure 6.18: Adding and checking aliasing (part 4).


```
ACA_term(σ, A, V_fr):
    let σ be H ≫ F by term u_0, f(U) // ...;
    if the rule is marked naive computational
        return ACA(σ_1, A, V_fr)
    else if the rule is marked optimal computational
        let τ_0 T_0, ..., τ_m T_m be the datatypes of the u_i;
        let τ'_0 T'_0 × ··· τ'_r T'_r → τ'_0 T'_0 be the annotated type of f;
        if some u_p = u_q
                where p ≠ q, T_p = T_q = array, and (τ'_p or τ'_q = var)
            MakeNaive(u_p);
            return ACA(σ_1, A, V_fr)
        else
            let A' be the aliases field from the annotated header of f;
                    (∅ if none);
            return ACA(σ_1, A + A'[u_0, U/f(U), U], V_fr + V'_fr − U_var)
                    where V'_fr is {u_0} if A' ≠ ∅ and ∅ otherwise
                    and U_var is {u_i | 1 ≤ i ≤ m and τ'_i T'_i = var array};

ACA_list-ind(σ, A, V_fr):
    let σ be H ≫ All x:list . F by ind y . z // ...;
    fix an ordering of the ∀∃ subformulas of F;
    ACA(σ_1, A, ∅);    (* explore the base case *)
    A'_1, ..., A'_k ← the aliases field for the ∀∃ subformulas of F;
            (∅ for any unmarked ∀∃ subformulas);
    repeat
        A_1, ..., A_k ← A'_1, ..., A'_k;
        ACA(σ_2, A, ∅);    (* explore the inductive case *)
        A'_1, ..., A'_k ← the aliases field for the ∀∃ subformulas of F;
                (∅ for any unmarked ∀∃ subformulas);
    until Closure(A_i) = Closure(A'_i), all i = 1, ..., k
    return (A, V_fr);
```

Figure 6.19: Adding and checking aliasing (part 5).

is different from all the other *ACA* routines (except for *ACA_int-ind*, which is not shown, since it is similar). In *ACA_list-ind*, *ACA* is called first on the base case and then repeatedly on the inductive case. Exploring the base case provides a first approximation to the amount of aliasing done by the body of the inductive universal. The inductive case is explored repeatedly until the aliases information stabilizes. It is easy to see that the process must reach a fixed point: for any particular $\forall\exists$ subformula of the body, its aliases information can only grow monotonically so far. If for some reason the $\forall\exists$ subformula has its var identifiers re-marked list, then its aliasing information becomes the empty set from then on. In either case, the aliasing information eventually stabilizes.

6.4 Correctness of the inferencer

The array inferencing algorithm does not in itself take a proof π and generate a proof π' of the form used in chapter 5. The annotated version of π, however, does map naturally to such a π'. The goal of this section is to show how to build π' and to show that π' is indeed a legal proof, one that meets all the restrictions on arrays.[9]

[9] In this section, "proof" generally means a possibly illegal proof, using the logic of chapter 5. Strictly speaking, the notions of *legal* and *possibly illegal* proofs should be defined: a legal proof is just a proof, and a possibly illegal proof is a proof-like object that might not meet all the restrictions on datatypes and aliasing.

The *BuildPf* routine defined in this section constructs π'. *BuildPf* takes two arguments: the annotated proof π and a proof environment

$$[\mathcal{E}'] = [\text{val } \mathcal{V}'_{val};\ \text{var } \mathcal{V}'_{var};\ \text{aliased } \mathcal{A}';\ \text{fresh } \mathcal{V}'_{fr}].$$

$\mathcal{E}'$ becomes the environment of π'. The following properties of π' are given to it by *BuildPf*.

- Identifiers of type list in π marked array become identifiers of type array in π'. The basic storage types of identifiers is also taken from the annotation, but the structure of π' is used to build its individual goals' proof environments.

- References to already-defined functions and lemmas are modified to take into account the optimal datatypes of the functions and theorems. For example, a rule in π like

  [list $x, \ldots$] ...by term y, *update*(x, i, e)

  • [list $x, y, \ldots$] ...

  becomes

  [var array $x, \ldots$] ... by comp-term y, *update_optimal*(x, i, e)

  • [var array y; log array $x, \ldots$] ...

  if the rule is marked *optimal computational*.

- Aliasing information is added to π' using the annotation of π.

For simplicity in defining *BuildPf*, π' retains the renaming of identifiers used in the annotation of π. This renaming is easily undone, so it is mostly ignored below.[10] Nonetheless, correctness of *BuildPf* must be understood to be modulo this renaming.

The definition of *BuildPf* and its correctness proof are given simultaneously and recursively. To assist the proof of correctness, another requirement is imposed on *BuildPf*. Let $\mathcal{A}$ and $\mathcal{V}_{fr}$ be the arguments to the last call of *ACA* for π, and let $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ be the arguments passed to *CL* for π. Let $\mathcal{V}$ be the set of all identifiers whose first occurrence in π is marked var array. The requirement is that $\mathcal{E}'$ must equal the restriction of $\mathcal{E}$, in the following sense: $\mathcal{V}'_{var} = \mathcal{V}_{var} \cap \mathcal{V}$, $\mathcal{V}'_{log} = \mathcal{V}_{log} \cap \mathcal{V}$, and $Closure(\mathcal{A}') = Closure(\mathcal{A}) \cap (\mathcal{V} \times \mathcal{V})$. This restriction must also hold for each subproof (subsubproof, etc.) of π.

For the top-level call of *BuildPf*, this requirement is easy to verify. Define the top-level environment $\mathcal{E}_0 = $ [val $\mathcal{V}'_{val}$], where $\mathcal{V}_{val}$ is the set of free identifiers of the goal of π. (So $\mathcal{V}'_{var}$, $\mathcal{V}'_{log}$, $\mathcal{A}'$, and $\mathcal{V}'_{fr}$ are all $\emptyset$.) For

[10] Formulas related by renaming in this way are related by *SameDtype* and *SameAliases*, so their datatypes and aliasing must match. This guarantees that any occurrence of an F obtained by un-renaming from a F^* will have the same datatypes and aliasing as the occurrences of F from which F^* was obtained.

the top-level calls of *CL* and *ACA*, we have $\mathcal{V}_{var}$, $\mathcal{V}_{log}$, $\mathcal{A}$, and $\mathcal{V}_{fr}$, are $\emptyset$, so the requirement is satisfied.

The definition and correctness of the rest of $BuildPf(\sigma, \mathcal{E}')$ depend on the structure of the proof σ passed to it. The local parameter σ denotes the current part of π being processed: σ is π or one its subproofs, subsubproofs, and so on. The subproofs of σ are $\sigma_1, \sigma_2, \ldots$; σ' is the proof returned by $BuildPf(\sigma, \mathcal{E}')$; $\sigma'_1, \sigma'_2, \ldots$ are the subproofs of σ'.

- *σ is proved by a propositional calculus rule: and-intro, and-elim, or-intro, or-elim, impl-intro, impl-elim, not-intro, not-elim, false-intro, or false-elim.*

  Building $\mathcal{G}'$, the goal of σ', is straightforward. For the environment [val $\mathcal{V}_{val}$; $\mathcal{E}'$], let $\mathcal{V}'_{val}$ be the free identifiers of the environment of π, minus the identifiers of $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$. To build the hypotheses and conclusion of $\mathcal{G}'$, do the following:

  1. Copy the hypotheses and conclusion of $\mathcal{G}$, the goal of σ, to $\mathcal{G}'$.
  2. For each quantified identifier x:list in $\mathcal{G}'$, find the datatype marked for x and replace the list keyword with the datatype.

3. If on the path from π to σ there are rules that cause state-dependent hypotheses to be removed, then remove the appropriate hypotheses from $\mathcal{G}'$.

4. For each occurrence of a formula in $\mathcal{G}'$ involving a function, like $y = \mathit{update}(x, i, e)$, find the datatypes of the identifiers involved and modify the function appropriately. For example, if x and y are both marked list in π, change the formula to be $y = \mathit{update\_naive}(x, i, e)$, and if both x and y are marked array, change the formula to $y = \mathit{update\_optimal}(x, i, e)$.

5. For each $\forall\exists$ subformula G that occurs in $\mathcal{G}'$, look up its aliases field in π and copy it to the corresponding occurrence of G in $\mathcal{G}'$. If necessary, restrict the aliases information to the set of var array universal parameters and existential witnesses.

This process for building $\mathcal{G}'$ is the same regardless of the rule used in σ. The rule field of σ' does depend on the rule. Here it is simply copied over from σ, except that if it is an *elim* rule (or a *skolem* or *hyp* rule, in the general case), the hypothesis number may have to be adjusted to take care of any hypothesis removed in step three.

The internal type-consistency of $\mathcal{G}'$ must be verified.[11] For example, if the formula $y = \mathit{update\_optimal}(x, i, e)$ occurs in $\mathcal{H}'$ or F', then x and y must have the same implementation type. This is taken care of by the *Sameltype* relation: the pair (x, y) (or its equivalent renaming) is added to *Sameltype* by *BuildSameTypes* because x and y appear in a formula u_0 *reln* $f(\mathcal{U})$. This also causes *GenDtypes* to mark x and y array, since the optimal datatype of *update* is var array $\times$ int$^2 \rightarrow$ var array. If either x or y is re-marked list, the other is also: *MakeList* is the only routine used to make an identifier list, and it changes all the identifier's *Sameltype* equivalents.

The subproofs of σ' are built by calling *BuildPf* on the subproofs of σ. The environment field $\mathcal{E}_i''$ for a subproof σ_i' depends on the rule: for *and-elim*, for example, $\mathcal{E}_1''$ is $\mathcal{E}'$, since the *and-elim* rule does not change the environment. For an *and-intro*, $[\mathcal{E}_i']$ is [log $\mathcal{V}_{var} + \mathcal{V}_{log}$; aliased $\mathcal{A}$], since the *and-intro* rule removes var and fresh identifiers from the environment.

It is also necessary to argue that the recursive calls to *BuildPf* satisfy the requirement for proof environments. (That is, the fields of the subproof environments should equal the restriction of the cor-

[11] This internal type-consistency must be checked regardless of the rule used by σ, and the discussion here is appropriate for the general case.

responding arguments of CL and ACA to the set $\mathcal{V}$ of all var identifiers of π.) None of these rules modify the aliases field, so that part of the restriction is satisified trivially. The rules that modify the fresh field simply reset it to empty, but the ACA routines have corresponding empty arguments, so this part is also satisfied. For the var and log fields, consider the *and-intro* rule. (The other rules are similar.) For the *and-intro* rule, CL is called on the subproofs of π with the arguments $\emptyset$ and $\mathcal{V}_{var} + \mathcal{V}_{log}$. Since $\emptyset = \emptyset \cap \mathcal{V}$ and $(\mathcal{V}'_{var} \cup \mathcal{V}'_{log}) \cap \mathcal{V} = (\mathcal{V}'_{var} \cap \mathcal{V}) \cup (\mathcal{V}'_{log} \cap \mathcal{V})$, the requirement is satisfied.

- *σ is proved by all-intro.*

The goal and rule of σ' are built using the same process as for the propositional calculus rules. The correctness of $\mathcal{E}''_1$ also works similarly.

The *all-intro* rule has two restrictions not encountered by the propositional calculus rules. First, the introduced universal identifiers must be new for σ'. This follows because they are new for σ. Second, if the introduced universal F is an $\forall\exists$ formula then the aliases field (possibly $\emptyset$) attached to it must not mention any identifier more than once. This is taken care of by the *MarkAliases* routine. It is the only way that an $\forall\exists$ formula can become marked with an aliases

field, and it always checks to make sure that it never marks a formula with an aliases field that mentions an identifier more than once. (This aliases field restriction applies generally to any $\forall\exists$ formula embedded in the hypothesis or conclusion, and the argument here also applies generally.)

- *σ is proved by all-elim.*

Let $\mathcal{G}'$ be

$$[\text{val } \mathcal{V}_{val};\ \mathcal{E}']\ \mathcal{H},\ F,\ \mathcal{H}' \gg G \quad \text{by elim } n, \mathcal{U}$$

where F is All $x_1 : \tau_1 T_1, \ldots, x_m : \tau_m T_m . F'$.

There are two restrictions to be met. First, the set of u_i where $\tau_i T_i =$ var array must be distinct. This follows from the definition of *ACA_all-elim*. The second restriction comes in parts: (1) no u_i may be in $\mathcal{V}_{log}$, (2) if $\tau_i T_i =$ var array then u_i must be in $\mathcal{V}'_{var}$, and (3) if $\tau_i T_i =$ val array and u_i is in $\mathcal{V}'_{var}$, then u_i must not equal any u_j where $\tau_j T_j =$ var array. In addition, in this third case G must not have any embedded universals, existentials, or implications. This is taken care of partly by *GenDtypes* (which markes all the u_i non-log) but mostly by *CC_all-elim* and *ACA_all-elim*. The ACA routine makes sure that no $u_i = u_j$, where $i \neq j$ and $\{\tau_i T_i, \tau_j T_j\}$ is {var array} or {val array, var array}. The CC routine checks for the other conditions.

- *σ is proved by some-intro.*

  Let $\mathcal{G}'$ be

$$[\mathsf{val}\ \mathcal{V}_{val};\ \mathcal{E}'|\ \mathcal{H} \gg F \quad \text{by intro } \mathcal{U}$$

where F is Some $x_1 : \tau_1\, T_1, \ldots, x_m : \tau_m\, T_m$. G. First, for τ_i = var, u_i must be in $\mathcal{V}'_{var}$. This is guaranteed by *BuildSameTypes*: each (x_i, u_i) pair is added to *SameDtype*, so they must have the same basic storage type. *CL_some-intro* helps by making sure that no u_i is in $\mathcal{V}_{log}$. Second, if F is the existential of an $\forall\exists$ formula G, then for each i where τ_i = var, either u_i is in $\mathcal{V}_{fr}$ or it is aliased to some universal parameter of G (but not both). In addition, if it is aliased, then the aliasing must agree with the aliases field attached to G. The checks for this are done by *ACA_some-intro*. If u_i were not in $\mathcal{V}_{fr}$ and did not alias with a universal var array parameter, it would have been re-marked list. If it does alias, then *ACA_some-intro* must have added that fact to the aliases field for G.

- *σ is proved by some-elim.*

  Let $\mathcal{G}'$ be

$$[\mathsf{val}\ \mathcal{V}_{val};\ \mathcal{E}'|\ \mathcal{H},\ F,\ \mathcal{H}' \gg G \quad \text{by elim } n, \mathcal{U}$$

where G is Some $x_1 : \tau_1\, T_1, \ldots, x_m : \tau_m\, T_m$. F'. If F has any var array witnesses it is necessary to argue that when $\mathcal{G}'$ was built, F was not

omitted. (This problem comes up generally but is most obvious in this rule.) *GenDtypes* finds all hypotheses to be omitted and marks the appropriate quantified identifiers non-var, so if F has any var witnesses, it could not have been omitted when building $\mathcal{G}'$.

- *σ is proved by skolem.*

  Similar to *all-elim*.

- *σ is proved by list-ind.*

  Let the conclusion of $\mathcal{G}'$ be All x : list . F. For this rule, it is important that each $\forall\exists$ formula G embedded in F be consistently marked with an aliases field: the occurrences of G in the base case and inductive hypothesis are computationally distinct from the occurrence in $\mathcal{G}'$ but must still have the same aliases field as G. This is taken care of *BuildSameTypes* and *MarkAliases*: *BuildSameTypes* equates the various versions of G under *SameAliases*, so whenever aliasing information is added to one version of G, it is added to all the versions.

It should be noted that the aliases information for F is an upper bound for the actual aliasing behavior. For example, if aliases $((x_1, y_1), (x_2, y_2))$ is attached to G, it may be that G only aliases y_1 to x_1 if the induction identifier x is nil and only aliases y_2 to x_2 if x is not nil, so that G never aliases both y_i to x_i simultaneously.

- *σ is proved by int-ind.*

  Similar to *list-ind*.

- *σ is proved by hyp or sequence.*

  Similar to the propositional calculus rules.

- *σ is proved by expl-def or decision.*

  Trivial.

- *σ is proved by term.*

  Although the goal $\mathcal{G}'$ of σ is built in the usual way, the rule is not. Let the rule of σ be by term $u_0, f(\mathcal{U})$. There are three cases, depending on how the rule is marked in the annotation.

  1. The rule is marked *logical.* The rule of σ' is

     by log-term $u_0, f\_optimal(\mathcal{U})$.

  2. The rule is marked *naive computational*. The rule of σ' is

     by comp-term $u_0, f\_naive(\mathcal{U})$.

  3. The rule is marked *optimal computational*. The rule of σ' is

     by comp-term $u_0, f\_optimal(\mathcal{U})$.

The logical and naive computation term rules pose no special problem. For optimal computational term rules, two special restrictions must be met. Let $\tau_1 T_1, \ldots, \tau_m T_m$ be the local types of the u_i, and let $\tau_1' T_1' \times \cdots \tau_r' T_r' \rightarrow \tau_0' T_0'$ be the type of *f_optimal*. The first restriction is that the set of u_i such that $\tau_i' =$ var must be distinct. This is handled by *ACA_comp-term*, in a way similar to that for *all-elim*. The second restriction has to do with the compatibility of the τ_i and τ_i': if τ_0' is var array, then the τ_i, τ_i' must be strongly compatible, else they only need to be weakly compatible. This is checked for by *CC_term*.

- *σ is proved by theorem.*

  There are two cases for building the rule of σ'. Let the rule of σ be by thm *Th*. If the rule of σ is marked *naive*, then the rule of σ' should be by thm *Th_naive*. If the rule of σ is marked *optimal*, then the rule of σ' by thm *Th_optimal*.

  If the rule is marked *optimal*, then the types of the free and bound identifiers of the goal of *Th_optimal* must exactly match the types of the corresponding identifiers in $\mathcal{G}'$. This is ensured by *GenDtypes*: the types of the identifiers, as they occur in σ, are marked in the annotation with the types from the optimal code for *Th*. If any of the identifiers were re-marked list, then *CC_term* or *MakeNaive* would have re-marked the rule *naive*.

- ***σ is proved by def-extr.***

  Similar to *theorem*. □

6.5 Efficiency of the inferencing algorithm

This section is devoted to a discussion of the running time of the inferencing algorithm. The aim is to show that it takes time nearly-linear in the area of a proof to annotate it.

Definition 6.8 The *complexity* of a term, formula, or goal is roughly equal to the size of its parse tree. The complexity of a term is the number of occurrences of constants, identifiers, procedure names, etc., that it contains. The complexity of a formula is the sum of the complexities of its terms plus the number of connectors and bound identifiers it has, plus the number of occurrences of true and false in it. The complexity of a goal is the number of identifiers in its environment plus the complexities of its hypotheses and conclusion.

Notation. Let A and S be the area and size respectively of π. Let Ack^{-1} be the inverse of the Ackermann function. Define $A' = A \cdot Ack^{-1}(A)$ and $S' = S \cdot Ack^{-1}(S)$.

Theorem 6.1 *Running time of the array inferencing algorithm.*

The inferencing algorithm takes $O(A')$ time to annotate π.

Proof

Some assumptions are made on π: its top-level goal has $O(S)$ complexity, each formula in π has $O(1)$ complexity, and the sum of the complexities of all lemma goals of π (the goals of theorems used as lemmas by π) is $O(S)$. Note that it follows that the sum of the lengths of all the proof environments of π is $O(A)$.

The description of the inferencing algorithm was designed to make it easier to explain, not necessarily efficient. To get the $O(A')$ running time, it is necessary to modify different parts of the algorithm.

- *Building SameItype and SameDtype.*

  The routines to mark datatypes must be sped up. This can be done by distinguishing one member from each ***SameItype***, ***SameDtype***, and ***SameAliases*** equivalence class and using it as the representative for the class.[12] For example, ***MarkItype*** becomes

  > *MarkItype*(x, T) :
  >
  > let y be the representative of the
  >
  > *SameItype* equivalence class for x;

[12] Assume that a path-compressing disjoint set union algorithm is being used to represent equivalence classes so that in general, n union and find-representative commands take $O(n \cdot Ack^{-1}(n))$ time [AHU,Tar].

if y has no attached itype

  Attach T to first occurrence of y

else if first occurrence of y has a $T' \neq T$ attached

  MakeList(y).

Note that if two identifiers are equivalent under *SameItype* but not under *SameDtype*, separate storage types must be attached to each identifier if their *SameItype*-representative is not marked list. If their *SameItype*-representative is marked list, then the two identifiers both implicitly have the same storage type, val.

Once modified, the routines *MarkItype*, *MarkStype*, *MarkDtype*, and *MakeList* all take time proportional to the time it takes to find the equivalence class representative for their input identifier. (The *MakeSameDtypes* routine vanishes.) The *BuildSameTypes* routine does $O(S)$ markings, so it takes $O(S')$ time in total.

- *Generating datatypes.*

The only part of *GenDtypes* that should be modified is the part that looks for uses of free and quantified identifiers that would be dropped from the goal if they were var identifiers. Instead of looking through each subproof's descendents, it is more efficient to simply traverse the whole proof π exactly once, gathering and using information as we go that indicates which identifiers have to be re-marked if they are used.

With this change, *GenDtypes* takes $O(A + S')$ time: $O(A)$ to walk the tree and look at each hypothesis, and $O(S')$ time to execute the *MarkItype* and *MarkStype* calls.

- *Compatibility checks.*

To speed up the *CC* routines, the *AddRules* routine must be modified. Instead of adding rules back to $\mathcal{R}$, have *AddRules* simply build a list $\mathcal{X}$ of all x with which it was called. After running all the compatibility checks on π, make one more pass through π looking for *term*, *theorem*, and *def-extr* rules. If the rule uses an identifier equivalent under *SameItype* to some x in $\mathcal{X}$, re-mark the rule *naive* or *naive computational* as appropriate.

Aside from the *AddRules* processing, *CC* takes $O(S + A) = O(A)$ time: $O(S)$ to walk the tree and $O(A)$ to check all theorem hypotheses, term arguments, and so on. The *AddRules* processing takes a total of $O(S + S') = O(S')$ time: $O(S)$ to build up $\mathcal{X}$, and $O(S')$ time to check the various rule identifiers for membership in the *SameItype* equivalence class of $\mathcal{X}$.

- *Checks for logical identifiers.*

The *MakeNaive* routine should be modified like the *AddRules* routine above: save the set $\mathcal{X}$ of arguments x, and at the end make one more pass through π looking for rules to re-mark.

The total running time for CL breaks into two parts: the time used by ***MakeNaive*** and the time used by the rest of CL. As modified, ***MakeNaive*** is like ***AddRules***: the $O(S)$ calls require $O(S')$ time. The rest of CL is dominated by the time it takes to manipulate $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$. Assume that $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ are being kept as global variables instead of as local parameters, and assume they are implemented as disjoint sets. A rule like ***and-intro*** causes $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ to be unioned. There are $O(S)$ such rules, and each union takes unit time, so the total cost for such rules is $O(S)$. A rule like ***all-elim*** causes $O(1)$ find-representative operations to be done on $\mathcal{V}_{log}$. The total cost for the $O(S)$ finds is $O(S')$. A rule like ***term*** causes identifiers to be deleted from $\mathcal{V}_{Var}$ and added to $\mathcal{V}_{log}$.[13] The total cost for the $O(S)$ deletions is $O(S)$. Finally, some rules, like ***sequence***, have more than one subproof. The values of $\mathcal{V}_{var}$ and $\mathcal{V}_{log}$ must be saved and restored between calls. The effort involved in these copying operations, summed over all of π, is bounded by the sum of the sizes of the local environments, $O(A)$, so that is the running time for these copying operations. The total running time for CL, therefore, is $O(A + S')$.

[13]The tree form of set union does not normally support deletion of elements, but here it can be arranged by adding members to $\mathcal{V}_{var}$ one at a time, so that the tree for $\mathcal{V}_{var}$ has height one. Deletion thus takes constant time.

- ***Adding and checking aliasing.***

  The dominating factor here is the amount of time spent checking for aliased identifiers in rules like ***all-elim***. Assume the aliasing information is represented by equivalence classes of aliased identifier occurrences. The sizes of all the equivalence classes, summed over all of π, is bounded by the sum of the sizes of the proof environments of π, which is $O(A)$, and $O(S) \leq O(A)$ aliasing checks are made during the traversal of π, so the total work involved is $\leq O(A')$.

- ***The grand total.***

  To recap, ***BuildSameTypes*** takes $O(S')$ time, ***GenDtypes*** takes $O(S')$ time, CC takes $O(A)$ time, CL takes $O(A+S')$ time, and ACA takes $O(A')$ time. The grand total time taken by the annotation of π therefore is $O(A')$. $\square$.

Chapter 7

Conclusions and future problems

There is many a rich stone laid up in the bowels of the earth,
many a fair pearl laid up in the bosom of the sea
— *Joseph Hall*

This chapter contains a review of the previous chapters and gives my thoughts on future lines of work in λ PRL and λ PRL-like systems. I believe that PRL has an excellent future ahead of it, and I am sure that people will continue to work on it and improve it.

7.1 Review

This thesis has been concerned with improving the efficiency of code extracted from constructive proofs. Three techniques were used, the first being code optimization. As observed by Bates, code optimizations greatly reduce the size of extracted code and can also reduce the running time.

To ease the burden on the code optimizer, one useful optimization was built directly into the extractor. To avoid most of the β-reductions used to optimize code in [Bat], extraction was modified: instead of mapping a proof to an equivalent justification-transforming function, extraction in this thesis maps a proof and a list of justification identifiers for its hypotheses to an expression whose value justifies the conclusion. This kind of extraction produces unoptimized code that is much shorter than the corresponding code for θ PRL (it is linear in the size of the proof, not in the area of the proof), so the code optimizer has an easier job.

Another important code optimization is the downward movement of recursive function calls. Because of the way extraction from inductive proofs is defined, it can happen that these recursive calls are done too early, so that the justification for the inductive hypothesis is built even when the inductive hypothesis is not used. Moving the recursive calls down into the code delays their execution to when they are needed. This optimization enables proof schemes like course-of-values induction to generate code with the right running time.

Code optimizations are important, but they are not a cure-all for all that ails extracted code. Some problems come from the form of the logic, and I will say more about this in the next section. Other problems come from the form of the semantics, from the definitions of justification and extraction. The other two techniques for improving extracted code both modified these notions.

The first used singleton formulas. Here, the observation was that many formulas have very uninteresting justifications, justifications that are trivial (in the base case) or are various combinations of uninteresting justifications (in the inductive case). The justifications for these "singleton" formulas are unnecessary because they are never relevant to the problem of calculating witness values for existentials or tag fields for disjunctions. One way to deal with these formulas would have been to combine their justifications into one big equivalence class, which would have become the new trivial justification, *triv*. This would still have left *triv*-valued identifiers in the code, so instead justifications were redefined so that such formulas need no actual justification at all: they are simply true or false. When used with the appropriate proof style, this optimization eliminates enormous pieces of code, as illustrated by the code for quicksort.

The final technique for improving the code called for radical changes to the semantics. To implement operations like array updating, which involve in-place destructive operations on data, call-by-reference parameters were added. This entailed adding a state to the semantics, which led to various syntactic and semantic design issues.

The design of the logic was guided by two main principles: the lack of assignment statements and the use of $\forall\exists$ formulas for side-effect-free functional computations. Since there is no assignment statement in the logic, users explicitly give different names to new values built during the computation, even if they happen to be stored at the same locations as some of the old values. Thus, the logic emphasizes the notion that logically different values truly *are* different, an advantage compared to logics that focus on identifiers as names of locations.

The use of $\forall\exists$ formulas as function-like objects was intended to help users understand how to change one value into another: the proof node

$$\textsf{All var } x : T\,.\,\textsf{Some var } y : T'\,.\,F \quad \textsf{by skolem } x', y'$$

takes the $\forall\exists$ justification, a function of type $T \to (T' \times \mathsf{J})$, and runs it on the input x' to get the output y'. The ban on side-effects ensures that the justification does not destroy the values of any variables except possibly x'. The ban on importing variables ensures that the justification does not produce a result that depends on values that might later be destroyed. The restriction on aliasing ensures that y' either aliases with x' or is stored at a freshly-generated location, not some arbitrary location.

To avoid some of the mental overhead involved in using the new logic, a proof annotator was presented. The annotator takes a proof, finds list-of-integer values that can be implemented as arrays-of-integers, and marks the proof accordingly. This enables users to get efficient code, code that uses arrays, simply by writing proofs that use lists in restricted ways. The array inferencing process therefore allows users to think of using arrays as an optimization, if they wish.

Extracted code that is optimized using the techniques described in this thesis is quite good if reasonable care is taken when building proofs. The combination of code optimizations and singleton formulas produces code that runs in the right time bounds for the primitive data structures used, and the introduction of call-by-reference parameters allows for a wider range of primitive data structures. Not enough large proofs have been done in λPRL to be able to say unequivocally that these optimizations suffice, but given the sample proofs of quicksort and binary search done here, I am confident that other proofs will also generate good code. It would be a useful exercise to verify my belief that the logic is expressive enough to handle the algorithms in Gries's *The Science of Programming* [Gri].

7.2 New problems

In this section I would like to discuss ideas for various lines of PRL-related work, some related to work done here, and others related to more general issues.

I am generally pleased with the code optimizations presented in chapter 3. They form a basic set of operations that do much to improve code. Extracted code is somewhat unusual in that it uses functions of high order quite naturally. Future workers should look for more sophisticated optimizations that take advantage of this property. Techniques like currying and uncurrying, partial parameterization, and development of normal forms for high order routines should be useful here.

An entirely new class of optimizations to work on is proof-guided optimization. As Goad observes [Goa], a program can be more thoroughly optimized when you have its proof of correctness. In addition, the proofs themselves can be optimized. For example, say that the proof of a lemma does a case split on $x = 0 \mid x \neq 0$. It may happen that the lemma is used in an environment where x is always zero. If this can be determined, and if the code for the lemma were being produced in-line, then the case split could be avoided and the subproof for the non-zero case could be thrown away.

One way this situation might be recognized is through simple inspection of the proof: **All** x:int might be eliminated with zero. The more interesting possibility is that the extractor may be able to *infer* that x is zero. It might seem that this situation is unlikely: after all, users are usually interested in efficiency, so wouldn't they simply hand optimize their own proofs? This is true for small collections of theorems, but as larger libraries of theorems are built up, I believe it will become less true.[1] Naturally, there are questions as to how much extractor effort should be spent trying to optimize proofs in this way, and exactly what kinds of optimizations should be done.

The other optimizations presented, those that dealt with arrays, should be extended. The logic of chapter 5 should be useful for other structures besides arrays: stacks, graphs, modifable S-expressions, and so on, but it is necessary to test this conjecture by trying these constructs out. The philosophy behind the design of the logic may be instructive for those who want to try other approaches: the functional, side-effect-free orientation of the logic is clean, but it may be too restrictive for some people. A different logic, one that deals with aliasing and side effects in some other way, would

[1]The general question is: given a proof π of a theorem Th with goal $|\mathcal{E}|\ \mathcal{H} \gg F$, and given a use of Th, say

$$|\mathcal{E},\mathcal{E}'|\ \mathcal{H},\mathcal{H}' \gg F \quad \text{by thm } Th,$$

what kinds of optimizations can be done on the extracted code for π, given that $\mathcal{H}'$ are true?

be very useful: the strengths and weaknesses of both logics could then be determined.

Related to the array logic is the array inferencing algorithm. Though I am pleased with the logic, I would like to see the inferencing algorithm improved: it is not as elegant as I would like. Some members of the PRL project are looking at the possibility of using attribute grammars to define the logic. Perhaps this approach will simplify the inferencing problem.

Another line of work that should be investigated is the utility of this work for other PRL systems, particularly νPRL. Code optimizations obviously are applicable, but what about the call-by-reference version of the logic? Can similar concepts be built into a PRL system with more general types? The analogue of the array inferencing algorithm would be a more general implementation-type/storage-type inferencer. This is infeasible in general, since for the νPRL logic, type membership is undecidable, but it should be practical for subsets of the logic. Harper has worked on the problem of finding parts of νPRL for which type inference is decidable [Har]. Perhaps such work can be extended to inferencing of implementation and storage types.

The programming part of PRL should be generally augmented. Currently, only very simple programs can be explicitly defined. This has the advantage that the proof rules for these programs are also very simple, but nonetheless it would be better to augment the programming logic in order to make more programs definable. Aliasing should be explicitly available in the language. The work of Boehm, of Cartwright and Oppen, and of Gries and Levin would be useful here [Boe85,CaO,GrL].

Ideally, I would like to see a system in which verified programs can be approached from two sides: from constructive proofs of their specifications, and from verification of explicitly defined programs. Designing a system that this does well will require some work: the logic for the programming language will have to be carefully defined so that it is complete enough for practical purposes, and the programming language will have to include a rich type structure.[2] Verification has its drawbacks [DML], but the strong semantic underpinning of PRL, combined with proof building routines and application-specific decision procedures [NeO] should make for a rich verification system [BoM,Pra80].

Another possibility is exciting. The implementations of λPRL and νPRL already support the definition and use of *tactics*, proof-building programs that combine the primitive proof rules with proof searching, lemmas, and other tactics. The tactic mechanism is known in the metatheory to be correct, but this correctness cannot be discussed in the logic itself. If the programming logic could be extended to include tactics, tactics could be

[2]For this purpose, some members of the PRL project are already trying to extend an ML-like language to include more types.

proved correct. In addition, tactics could do more complicated proof transformations that are now forbidden because they are not definable using the available hard-wired primitive tactics. It would then be possible for users to define their own decision procedures, prove them correct, and add them to their personal libraries. Currently, new decision procedures can only be added as new primitive rules, and only by the implementors. Making the programming language also the metalanguage is therefore be an important new step. Knoblock is working on adding this to νPRL [Kno].

The form of the logic should also be investigated. The refinement-style sequent proofs used in PRL have pleasant theoretical properties but present unpleasant practical problems. The denotational meaning of sequents as justification transformers is very clean, but proof trees of sequents are enormous and impossible to write down on paper.[3] In programming language terms, it's as if we had a programming language that required us to list the symbol table at each point of the program.

There is another problem with refinement-style sequent proofs: they ban communication between the subproofs of a node. For example, in an *and-intro* proof, the subproofs of the conjuncts are completely separate and cannot pass information from one to the other. This restriction does not cause any problems in the call-by-value version of the logic, but the logic

[3]Consider the quicksort proof of chapter 4.

of chapter 5 was strongly affected by this. In the case of the ***and-intro*** rule, the subproofs have no access to the variables of the main proof:

$$[\text{var } \mathcal{V}_{var};\ \text{log } \mathcal{V}_{log}; \ldots]\ \mathcal{H} \gg F_1 \ \&\ F_2 \quad \text{by intro}$$

- $[\text{log } \mathcal{V}_{var} + \mathcal{V}_{log}; \ldots]\ \mathcal{H} \gg F_1$
- $[\text{log } \mathcal{V}_{var} + \mathcal{V}_{log}; \ldots]\ \mathcal{H} \gg F_2$.

In a more flexible system, the members of $\mathcal{V}_{var}$ could be split into those used by σ_1 and those used by σ_2. This could not be done here because the pure-refinement design philosophy forced the partitioning to be a function of the goal and its rule application; the proofs of the subgoals could not enter into it.[4]

Similar problems affected Harper [Har]. He suggested that "refinement" be a notion applied to problem statements, not to proof constructions. I agree. Refinement is excellent for decomposing problems, but it adds a high cost to the development of proofs because the disconnectedness of subproofs forces unpleasant restrictions on the logic, as in the ***and-intro*** rule above. Perhaps it is time to resurrect natural deduction proofs [COD, Pra65] and to reserve refinement for the breakdown of theorems into lemmas.

[4]The partitioning of $\mathcal{V}_{var}$ could have been made a parameter of the rule, but this would have been ugly.

It is also important to consider possible non-traditional proof styles. Constable notes that though many programs are like constructive proofs of their specifications [Con82], some are not: concurrent programs, for example. An interesting question is "Can PRL be extended to include other kinds of programming styles?" For example, can PRL-based semantics be augmented to include programs with exception handling? Can object-oriented programming be subsumed? How can approaches to modules in higher languages [MaQ] be incorporated?

There is the danger that all these possible changes could divorce the programming language from the logic. This absolutely must not happen. Reasoning about programs is important but should not supplant reasoning about specifications.

The strength of PRL comes from its strong connection between reasoning about specifications and writing programs. As such, it represents an attempt to redefine the problem of programming by moving it up to a new level. PRL, and especially λ PRL, is only a first step toward this new level. By building on the current PRL systems, future systems will provide even better solutions to the problem of writing correct programs.

Appendix

The appendix presents the extracted code that results if one avoids extracting code for singleton formulas. (See chapter 5.) Recall that S is a singleton formula and N is a non-singleton formula. For any given proof rule, P is its extracted code. For brevity, rules that never generate extracted code are omitted, and various mirror-image cases are also omitted. Below, the phrase "P is unchanged" means that P has the same form when singletons are being removed as it does when singletons are not being removed.

Propositional calculus rules

And-intro

$\mathcal{H} \gg N_1 \,\&\, N_2$ by intro
- $\mathcal{H} \gg N_1$
- $\mathcal{H} \gg N_2$

P is unchanged.

$\mathcal{H} \gg N \,\&\, S$ by intro
- $\mathcal{H} \gg N$
- $\mathcal{H} \gg S$

P is $X[\sigma_1, \mathcal{J}]$.

And-elim

$\mathcal{H},\ N_1 \,\&\, N_2,\ \mathcal{H}' \gg N'$ by elim n
- $\mathcal{H},\ \mathcal{H}',\ N_1,\ N_2 \gg N'$

P is unchanged.

$\mathcal{H},\ N \,\&\, S,\ \mathcal{H}' \gg N'$ by elim n

- $\mathcal{H},\ \mathcal{H}',\ N,\ S \gg N'$

P is $\mathsf{X}[\sigma, J, J', J_n, \text{trivial}]$.

Or-intro

$\mathcal{H} \gg N_1 \mid N_2$ by intro k

- $\mathcal{H} \gg N_k$

P is unchanged.

$\mathcal{H} \gg N \mid S$ by intro 1

- $\mathcal{H} \gg N$

P is unchanged.

$\mathcal{H} \gg N \mid S$ by intro 2

- $\mathcal{H} \gg S$

P is $(2, \text{trivial})$.

Or-elim

In this first case, at least one of F_1, F_2 must be non-singleton.

$\mathcal{H},\ F_1 \mid F_2,\ \mathcal{H}' \gg N$ by elim n

- $\mathcal{H},\ \mathcal{H}',\ F_1 \gg N$
- $\mathcal{H},\ \mathcal{H}',\ F_2 \gg N$

P is unchanged.

$\mathcal{H},\ S_1 \mid S_2,\ \mathcal{H}' \gg N$ by elim n

- $\mathcal{H},\ \mathcal{H}',\ S_1 \gg N$
- $\mathcal{H},\ \mathcal{H}',\ S_2 \gg N$

P is

if $J_n = 1 \rightarrow \mathsf{X}[\sigma_1, J, J', K]$

$[]$ $J_n = 2 \rightarrow \mathsf{X}[\sigma_2, J, J', K]$

fi

Impl-intro

$[\mathcal{E}]\ \mathcal{H} \gg N \Rightarrow N'$ by intro

- $[\mathcal{E}]\ \mathcal{H}, N \gg N'$

P is unchanged.

$[\mathcal{E}]\ \mathcal{H} \gg S \Rightarrow N$ by intro

- $[\mathcal{E}]\ \mathcal{H}, S \gg N$

P is $\mathsf{X}[\![\sigma, J, \text{trivial}]\!]$.

Impl-elim

$\mathcal{H},\ N_1 \Rightarrow N_2,\ \mathcal{H}' \gg N'$ by elim n

- $\mathcal{H},\ \mathcal{H}' \gg N_1$
- $\mathcal{H},\ \mathcal{H}',\ N_1,\ N_2 \gg N'$

P is unchanged.

$\mathcal{H},\ S \Rightarrow N,\ \mathcal{H}' \gg N'$ by elim n

- $\mathcal{H},\ \mathcal{H}' \gg S$
- $\mathcal{H},\ \mathcal{H}',\ S,\ N \gg N'$

P is $\mathsf{X}[\![\sigma_2, J, J', \text{trivial}, J_n]\!]$.

Quantifier rules

All-intro

$[\mathcal{E}]\ \mathcal{H} \gg \text{All } x_1 : T_1, \ldots, x_m : T_m \,.\, N$ by intro $\mathcal{Y}$

- $[\mathcal{E}, y_1 : T_1, \ldots, y_m : T_m]\ \mathcal{H} \gg N[\mathcal{Y}/\mathcal{X}]$

P is unchanged.

All-elim

$\mathcal{H},\ \text{All } x_1 : T_1, \ldots, x_m : T_m \,.\, N,\ \mathcal{H}' \gg N'$ by elim $n, \mathcal{T}$

- $\mathcal{H},\ \text{All } x_1 : T_1, \ldots, x_m : T_m \,.\, N,\ \mathcal{H}',\ N[\mathcal{T}/\mathcal{X}] \gg N'$

P is unchanged.

Some-intro

$\mathcal{H} \gg \text{Some } x_1 : T_1, \ldots, x_m : T_m \,.\, N$ by intro $\mathcal{T}$

- $\mathcal{H} \gg N[\mathcal{T}/\mathcal{X}]$

P is unchanged.

$\mathcal{H} \gg \text{Some } x_1 : T_1, \ldots, x_m : T_m \,.\, S$ by intro $\mathcal{T}$

- $\mathcal{H} \gg S\,[\mathcal{T}/\mathcal{X}]$

P is $(\mathcal{T})$.

Some-elim

$[\mathcal{E}]\ \mathcal{H},\ \text{Some } x_1 : T_1, \ldots, x_m : T_m \,.\, N,\ \mathcal{H}' \gg N'$ by elim $n, \mathcal{Y}$

- $[\mathcal{E},\ y_1 : T_1, \ldots, y_m : T_m]\ \mathcal{H},\ \mathcal{H}',\ N\,[\mathcal{Y}/\mathcal{X}] \gg N'$

P is unchanged.

$[\mathcal{E}]\ \mathcal{H},\ \text{Some } x_1 : T_1, \ldots, x_m : T_m \,.\, S,\ \mathcal{H}' \gg N$ by elim $n, \mathcal{Y}$

- $[\mathcal{E},\ y_1 : T_1, \ldots, y_m : T_m]\ \mathcal{H},\ \mathcal{H}',\ S\,[\mathcal{Y}/\mathcal{X}] \gg N$

P is let $(\mathcal{Y}) = J_n$ in $\mathsf{X}[\![\sigma, J, J', \text{trivial}]\!]$ ni.

Induction rules

Int-ind

$[\mathcal{E}]\ \mathcal{H} \gg \text{All } x : \text{int} \,.\, N$ by ind k

- $[\mathcal{E},\ x : \text{int}]\ \mathcal{H},\ x < k,\ N\,[x+1/x] \gg N$
- $[\mathcal{E}]\ \mathcal{H} \gg N\,[k/x]$
- $[\mathcal{E},\ x : \text{int}]\ \mathcal{H},\ x > k,\ N\,[x-1/x] \gg N$

P is unchanged.

List-ind

$[\mathcal{E}]\ \mathcal{H} \gg \text{All } x : \text{list} \,.\, N$ by ind $y \,.\, z$

- $[\mathcal{E}]\ \mathcal{H} \gg N\,[()/x]$
- $[\mathcal{E}, y : \text{int}, z : \text{list}]\ \mathcal{H},\ N\,[z/x],\ x = y \,.\, z \gg N\,[y \,.\, z/x]$

P is unchanged.

Miscellaneous rules

Decision

$\mathcal{H} \gg S_1 \mid S_2$ by decision

P is unchanged.

Hypothesis

$\mathcal{H},\ N,\ \mathcal{H}' \gg N$ by hyp n

P is unchanged.

Sequence

$\mathcal{H} \gg N$ by seq N'

- $\mathcal{H} \gg N'$
- $\mathcal{H}, N' \gg N$

P is unchanged.

$\mathcal{H} \gg N$ by seq S

- $\mathcal{H} \gg S$
- $\mathcal{H}, S \gg N$

P is $\mathsf{X}[\![\sigma_2, J, \text{trivial}]\!]$.

Theorem

$[\mathcal{E}]\ \mathcal{H} \gg N$ by thm Th

P is unchanged.

Extr-def

$\mathcal{H} \gg N[T, f(T)/X, y]$ by def $f(T)$

where N is the matrix of the goal of the theorem from which f was extracted. P is unchanged.

Bibliography

[AHU] Alfred V. Aho, John E. Hopcroft, and Jeffry D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[All] John Allen. *Anatomy of LISP*. McGraw-Hill, 1978.

[Bat] Joseph L. Bates. *A Logic for Correct Program Development*. Ph.D. Thesis, Cornell University, 1979. Also available as Department of Computer Science Technical Report TR 81-455, March 1981.

[BaC81] Joseph L. Bates and Robert L. Constable. Definition of μPRL. Department of Computer Science Technical Report TR 81-492 Cornell University, October 1981.

[BaC85] Joseph L. Bates and Robert L. Constable. Proofs as Programs. *ACM Trans. on Prog. Lang. and Sys.*, (7)1: 113–136, January 1985.

[BeM] John Lane Bell and Moshé Machover. *A Course in Mathematical Logic*. North Holland, 1977.

[Bee] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[Bis67] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, 1967.

[Bis70] Errett Bishop. Mathematics as a Numerical Language. In *Intuitionism and Proof Theory*, J. Myhill, et al., editors, pages 53–71. North Holland, 1970.

[BDD] H. Boehm, A. Demers, and J. E. Donahue. An Informal Description of Russell. Department of Computer Science Technical Report, TR 80-430. Cornell University, October 1980.

[Boe84] Hans-Juergen Boehm. *A Logic for the Russell Programming Language*. Ph.D. Thesis, Cornell University, 1984. Also available as Department of Computer Science Technical Report TR 84-593, February 1984.

[Boe85] Hans-Juergen Boehm. *Side Effects and Aliasing Can Have Simple Axiomatic Descriptions*. *ACM Trans. on Prog. Lang. and Sys.* (7)4: 636–655, October 1985.

[BoM] R. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[CaO] R. Cartwright and D. Oppen. The logic of aliasing. *Acta Inf.*, 15: 365–384, 1981

[Con71] Robert L. Constable. Constructive Mathematics and Automatic Programs Writers. In *Proc. of IFIP Congress*, pages 229–233. Ljubljana, 1971.

[Con82] Robert L. Constable. Programs As Proofs. Department of Computer Science Technical Report TR 82-532. Cornell University, November 1982.

[Con85] Robert L. Constable. The Semantics of Evidence. Department of Computer Science Technical Report TR 85-684. Cornell University, May 1985.

[Con86] R. L. Constable, S. A. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[CoB] Robert L. Constable and Joseph L. Bates. The Nearly Ultimate PEARL. Department of Computer Science Technical Report TR 83-551. Cornell University, January 1984.

[CKB] Robert L. Constable, Todd Knoblock, and Joseph L. Bates. Writing Programs that Construct Proofs. *Journal of Automated Reasoning* 1: 285–326, 1985. Also available as Department of Computer Science Technical Report TR 84-645. Cornell University, October 1984.

[CoJ] Robert L. Constable and Scott D. Johnson. A PL/CV Precis. *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, 1979, pages 7–20.

[CJE] Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*. Lecture Notes in Computer Science, vol. 135, Springer-Verlag, 1982.

[COD] Robert L. Constable, and Michael J. O'Donnell. *A Programming Logic*. Winthrop, Cambridge MA, 1978.

[Cros] J. N. Crossley, C. J. Ash, C. J. Brickhill, J. C. Stillwell, and N. H. Williams. *What is Mathematical Logic?* Oxford University Press, 1972.

[deB] Jaco de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, 1980.

[DML] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social Processes and Proofs of Theorems and Programs. *Comm. of the ACM* (22)5: 271–280, May 1979.

[Dij] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DoD] J. E. Donahue and A. J. Demers. Revised Report on Russell. Department of Computer Science Technical Report, TR 79-389. Cornell University, September 1979.

[Dum] Michael Dummett. *Elements of Intuitionism*. Oxford Logic Series, Clarendon Press, 1977.

[End] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[ErP] Yu. L. Ershov and E. A. Palyutin. *Mathematical Logic*. Translated from the Russian by Vladimir Shokurov. Mir Publishers, Moscow, 1984. Available through Imported Publications, Inc., 320 West Ohio Street, Chicago IL, 60610.

[Gen] Gerhard Gentzen. Investigations Into Logical Deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, North Holland, Amsterdam, 1969.

[Goa] Christopher Alan Goad. *Computational Uses of the Manipulation of Formal Proofs*. Ph.D Thesis, Stanford University, 1980.

[Gor] Michael J. C. Gordon. *The Denotational Description of Computer Languages: An Introduction*. Springer-Verlag, 1979.

[GMW] Michael Gordon, Arthur Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Lecture Notes in Computer Science, vol. 78, Springer-Verlag, 1979.

[Gri] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[GrL] David Gries and Gary Levin. Assignment and Procedure Call Proof Rules. *ACM Trans. on Prog. Lang. and Sys.*, (2)4: 564–579, October 1980.

[Har] Robert W. Harper. *Aspects of the Implementation of Type Theory*. Ph.D. Thesis, Cornell University, 1985. Also available as Department of Computer Science Technical Report TR 85-675, April 1985.

[Hec] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North Holland, 1977.

[How] Douglas J. Howe. Forthcoming Ph.D. Thesis, Cornell University.

[ILL] Shigeru Igarashi, R. L. London, and D. C. Luckham. Automatic Program Verification I: A Logical Basis and its Implementation. *Acta Informatica* (4)2: 145–182, 1975.

[Joh] Scott D. Johnson. *A Computer System for Checking Proofs*. UMI Press, New York, 1983. Also available as Department of Computer Science Technical Report TR 80-444. Cornell University, January 1981.

[Ken] Ken Kennedy. *A Survey of Data Flow Analysis Techniques*. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54, Prentice-Hall, 1981.

[Kle45] Stephen C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, (10)4: 109–124, December 1945.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*. van Nostrand, Princeton, 1952.

[Kle67] Stephen C. Kleene. *Mathematical Logic*. John Wiley & Sons, 1967.

[Kno] Todd B. Knoblock *Formal Metamathematics and Reflection in Constructive Type Theory*. Forthcoming Ph.D. Thesis, Cornell University.

[Koz] Dexter C. Kozen. *Complexity of Finitely Presented Algebras*. Ph.D. Thesis, Cornell University, 1977.

[MaY] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Elsevier North Holland, 1978.

[MaQ] David MacQueen. Using Dependent Types to Express Modular Structure. *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, 1986, pages 277–286.

[ML73] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H. E. Rose and J. C. Shepherdson, editors, pages 73–118. North Holland, 1973.

[ML79] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North Holland, 1982. Also in C. A. R. Hoare and J. C. Sheperdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice-Hall, 1985.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes, *BIBLIOPOLIS*, Napoli, 1984.

[Nel] Greg Nelson. *Techniques for Program Verification*. Ph.D. Thesis, Stanford University, 1980.

[NeO] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. on Prog. Lang. and Sys.*, 1(1): 245-257, October 1979.

[Pau] Lawrence Paulson. Tactics and Tacticals in Cambridge LCF. Technical Report No. 39, Computer Laboratory, University of Cambridge, July 1983.

[Pra80] V. R. Pratt. On Specifying Verifiers. *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, 1980, pages 106–115.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-theoretical Study*. Almquist and Wiskell, Uppsala, Sweden, 1965.

[Prl] The PRL Staff. (S. A. Allen, J. L. Bates, H. M. Bromley, R. L. Constable, J. F. Cremer, R. W. Harper, T. B. Knoblock, N. P. Mendler, and J. T. Sasaki.) *PRL: Proof Refinement Logic — Programmer's Manual (Lambda PRL, VAX Version)*. Computer Science Department, Cornell University, 1984.

[Rog] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[Sch] Joseph R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[Sta] Ryan D. Stansifer. *Representing Constructive Theories in High-Level Programming Languages*. Ph.D. Thesis, Cornell University, 1985. Also available as Department of Computer Science Technical Report TR 85-664, March 1985.

[Sto] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Tar] Robert Endre Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[TeR] Tim Teitelbaum and Tom Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Comm. of the ACM* (24)9: 563–573, September 1981.

[Tro] A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Lecture Notes in Mathematics vol. 344, Springer-Verlag, 1973.