

Specification and Proof of Higher-Order Programs

Johannes Kanig

November 15, 2010

Contents

1. Introduction	1
1.1. Generalities	1
1.2. Methods to Obtain more Reliable Programs	2
1.2.1. Testing	2
1.2.2. Formal Methods	4
1.2.3. Language Techniques	6
1.3. Two Techniques Presented in more Detail	8
1.3.1. Hoare Logic in more Detail	10
1.3.2. The ML Type System and Extensions	20
1.4. Overview of the Document, Contributions and Related Work	30
1.4.1. An Overview of the Document	30
1.4.2. Design Choices	32
1.4.3. Related Work	34
2. The Specification Language	39
2.1. The Programming Language W	39
2.1.1. Syntax	39
2.1.2. Semantics	45
2.1.3. Typing	50
2.1.4. Properties	54
2.1.5. A Generalization of the Results	61
2.2. The Logic L	62
2.2.1. Syntax	63
2.2.2. Typing	64
2.2.3. Semantics	66
2.2.4. Annotating Programs	69
3. A Weakest Precondition Calculus	75
3.1. The wp Predicate Transformer	75
3.2. Soundness of the wp Calculus	82
3.3. Completeness	90
3.4. Extensions	102
3.4.1. Logical Symbols in Programs	102
3.4.2. Read-Write Effects	103
3.4.3. Algebraic Data Types and Pattern Matching	104
4. A Language without Aliasing	107
4.1. Excluding Aliasing of Regions	108
4.2. Singleton Regions	115

5. Implementation and Case Studies	125
5.1. The Who Tool	125
5.2. Translation from L to L_0	128
5.3. Translation from Higher-Order Logic to First-Order Logic	133
5.3.1. Motivation	133
5.3.2. An Overview of the Encoding	135
5.3.3. The Source Language	136
5.3.4. Elimination of Quantifiers	138
5.3.5. Elimination of λ -Abstractions	138
5.3.6. The Target Language: FOL	140
5.3.7. The Encoding	142
5.3.8. Optimizations of the Encoding	142
5.3.9. An Example	145
5.3.10. Justifying the Encoding	146
5.4. Case Studies	149
5.4.1. Introductory Examples	150
5.4.2. Memoization Functions	153
5.4.3. The Array Module	156
5.4.4. The List Module	161
5.4.5. Koda and Ruskey’s Algorithm	163
5.4.6. A Challenge for the Who Tool	168
6. Conclusion and Outlook	171
6.1. A Summary of the Contributions of this Thesis	171
6.2. Using Who to Verify OCaml Programs	172
A. Résumé en Français	177
A.1. Introduction	177
A.1.1. La logique de Hoare	177
A.1.2. L’ordre supérieur	178
A.1.3. Le langage ML	178
A.1.4. Les systèmes à effets	178
A.1.5. Cette thèse	179
A.2. Le langage de programmation W et la logique L	179
A.3. Le calcul de la plus faible précondition	180
A.4. Les restrictions d’alias	181
A.4.1. L’exclusion d’aliasing de régions	182
A.4.2. Les régions singletons	183
A.5. L’outil et des exemples	184
A.5.1. Traduction vers la logique du premier ordre	184
A.5.2. Études de cas	184
A.6. Conclusion	185
Bibliography	187
Index	199

1. Introduction

Educators, generals, dieticians, psychologists, and parents program. Armies, students, and some societies are programmed. An assault on large problems employs a succession of programs, most of which spring into existence en route. These programs are rife with issues that appear to be particular to the problem at hand. [...] For all its power, the computer is a harsh taskmaster. Its programs must be correct, and what we wish to say must be said accurately in every detail. As in every other symbolic activity, we become convinced of program truth through argument.

Alan J. Perlis, in the foreword for *Structure and Interpretation of Computer Programs*, (Abelson and Sussman, 1996).

Computers and software have become ubiquitous in our every-day life. The huge majority of office work is done using computers, communication is done using email or a software-controlled mobile phone, LEGO sells programmable robots. These applications are *non-critical*, so when there is a problem with the hardware or the software, one will usually try again, fix the problem when there is time or simply live with it. However, computers have been increasingly used in environments where wrong behavior can have serious consequences: military and civil airplanes, cars and subway trains are now partly controlled by software; software makes medical devices work; huge amounts of money are traded on financial markets by computer programs every second. These applications are *critical*, a single fault in the system can trigger loss of money, or human lives, or both.

1.1. Generalities

A *computer program*, or simply *program*, is a list of instructions intended to be executed by a computer. It is also intended to be human-readable and human-writable. The text of a program is also called the *source code* or simpler *code*. Computer programs are written in a *programming language*; such a language is defined by its *syntax*, which describes the form of valid programs, and its *semantics*, which describes the meaning of valid programs. Programs in a human-readable language are in general not directly understandable by a computer, so they have to be *compiled* (translated) to *machine code*, a simpler programming language that is easy to interpret by the computer but very difficult to understand for humans, because of the absence of structure and the presence of many operational details. This translation process is done by a *compiler*, another program.

1. Introduction

A computer program generally accepts a problem description as an *input*, and gives an answer as an *output*. All but the simplest programs are divided into *functions* or *procedures*, smaller parameterized units of instructions. Instructions manipulate *data*, which is stored in the *memory* of the computer. *Data structures* are a way of representing and structuring data in a program. A *module* is a group of functions or procedures that deal with the same problem domain and/or data structures. An *algorithm* is the *idea* of a program, independently of a particular programming language or computer. An algorithm is intended to solve a *problem*, *i.e.*, to give a correct answer to a question given in a certain way. There may be several algorithms which solve the same problem. An algorithm can also be formulated in English. A concrete program which has been written with an algorithm in mind is called an *implementation*.

A *specification* is a text that describes the expected behavior of a program or algorithm. Often, a specification describes conditions on the input (a *precondition*) of the program under which it is supposed to function. It also describes properties of the output of program (a *postcondition*) that are guaranteed if the program is correct. As an example, a program to compute the square root of a real number should expect this number to be positive, and should return a number which actually is (or is reasonably close to) the square root of the input. A specification can be either *informal*, *i.e.*, given in English or with the help of diagrams, or *formal*, in which case it will probably be expressed using *logical formulas*. The informal specification of a program computing the square root has just been given. The formal one can be expressed as follows. If the input value, call it x , satisfies the condition

$$x \geq 0,$$

then the output value, call it r for *result*, has to satisfy the condition

$$r = \sqrt{x}.$$

An error in a computer program which causes it to not produce the intended result is called a *bug*. The activity of searching and correcting bugs is called *debugging*. A program or algorithm is *correct* when under all circumstances the specification is verified. This definition also makes it clear that a program or algorithm can never be correct by itself, it can only be correct *with respect to a specification*. Another important distinction is between a *safety specification* which basically only states that the program does not crash and a *functional specification*, which includes properties about the intended result.

1.2. Methods to Obtain more Reliable Programs

For decades now, high reliability has been an important research focus. Many methods, to either find bugs in existing software or avoid writing faulty software in the first place, have been devised. We only present a few of them.

1.2.1. Testing

The first such method is *testing* (Kaner et al., 1993; Myers and Sandler, 2004). In its simplest form, testing consists in choosing a set of input values and determining the

expected output for each of it, then running the program to be tested on each input and comparing the actual output of the program with the expected output. Testing can take place at the program or function level. We briefly describe a few notions in the area of testing.

Black box testing takes place when the program or function is considered to be an opaque unit, *i.e.*, one does not take into account its internal implementation. Black box testing can test for general correctness.

White box testing takes place when the selection of the test input depends on the program code to be tested. This can be useful to test a particular aspect of an implementation or algorithm, which may not be relevant for other implementations of the same algorithm, or different algorithms solving the same problem.

Coverage A test may not execute all instructions of a program. The *coverage* is the proportion of code executed during a test. In general, testing aims to maximize coverage.

Unit tests test a single function or module of the program, instead of the entire program.

Integration tests test the interaction between two modules.

Regression tests test the program against known problems in prior versions, so that an unintended reintroduction of these problems by future modifications of the program can be easily detected.

Contracts (Meyer, 2000) are test conditions, similar to pre- and postconditions, that usually appear at the beginning and at the end of a function. During the development of the program, they can be executed each time the function is called, to test the validity of input of the function and the correctness of its output. These test conditions are usually removed for efficiency reasons once the program is used in a production environment. Originally introduced by the language Eiffel (Meyer, 1992), this concept has been reused by others, for example in the executable specification language JML (Leavens et al., 2009).

Test driven development describes a process of software development in which the test is written *before* writing the program or function to be tested. This process is meant to encourage proper analysis of the problem before starting to write the program. Test-driven development has first been proposed by the eXtreme Programming movement (Beck and Andres, 2004).

Testing is intuitive and relatively cheap to set up. This is certainly the reason why testing is nowadays extensively applied to software development in all parts of the industry. Much research has been dedicated to automatically generating test input. In this setting, the distinction between black box and white box testing becomes even more important. Formal or informal specifications have been used to guide this automated generation.

Probably the biggest drawback of testing is that it gives relatively weak guarantees about the correctness of the tested program. It can only verify the well-behavior of

1. Introduction

the program for the chosen pairs of input and output values, but other situations may occur when the program is actually used. The reason is that even in very simple cases, the input domain of a program is infinite, but only a finite (and usually small) amount of input values can be tested. This fact implies that testing alone is not sufficient for critical systems, where one wants to guarantee the absence of bugs.

1.2.2. Formal Methods

Given the inadequacy of testing to guarantee the absence of bugs, we have to turn to other methods. The field of *formal methods* (Monin, 2002) tries to give better guarantees; it is an extremely vast field, but one can say that formal methods try to apply advances in mathematics, logics and theoretical computer science to the correctness of programs. The notion of formal method is related to the notion of *static analysis*. Static analysis describes processes that reason about computer programs *without executing them*. Testing does not belong to static analysis because it necessarily has to run the program to observe its output. Static analysis is most successful when the programming language has a clear semantics, *i.e.*, the meaning of every statement of the language is clearly defined, so that one does not need to execute it in order to understand its behavior.

Model checking and bounded model checking. Model checking (Clarke et al., 2000) can be seen as exhaustive testing. Historically, model checking has first been applied to hardware components, and it has been applied to software later. Model checking represents the component to be checked by an abstract *model* consisting of *states* that may or may not verify the properties to be checked. Each instruction corresponds to a state change. The resulting diagram can now be exhaustively explored (in the case of hardware components), starting from the initial state, and the property to be checked can be verified in each reachable state. The variant *bounded model checking* does only test if all states reachable in n steps are valid, for a given n .

Exhaustive model checking is again impossible for even very simple computer programs, because the number of states is infinite. So only bounded model checking is directly applicable, but it can give only limited guarantees.

Abstract interpretation. Another way to deal with the infinite number of states is to only consider a finite number of *abstract* states. Each possible state of the program is mapped to an abstract state. Each instruction, which was modeled as a transition between concrete states in model checking, now becomes a transition between abstract states. The mapping from concrete to abstract states is also called *abstraction*. The properties to be verified, initially formulated for concrete states, are reformulated to deal with abstract states instead. The transition diagram now becomes finite again and can be exhaustively explored, similar to the way model checking works.

This method of abstracting the search space is called *abstract interpretation* (Cousot and Cousot, 1979), and it has been used to verify properties of a large number of industrial-sized programs. The verifier *Astree* (Blanchet et al., 2003) and the commercial tool *The Mathworks Polyspace*, among others, are based on abstract interpretation.

Because there are many more concrete states than there are abstract states, multiple concrete states may be mapped to the same abstract state. In particular, valid states and invalid concrete states may be mapped to the same abstract state. For the abstraction to be correct, it must be an *overapproximation* of the concrete states, *i.e.*, every abstract state that contains at least one invalid state has to be marked invalid as well. Therefore, the analysis on the abstract diagram may fail although the initial program is correct, for example when all accessible concrete states are valid, but the abstraction rendered invalid states accessible. In such cases, abstract interpretation will report *false positives*, spurious errors that are present only due to the choice of the abstraction. Different abstraction methods may result in more or less false positives.

Hoare logic. When the other methods fail, we need a more powerful technique to prove the correctness of our programs. Probably the most powerful, but also the most expensive way of reasoning about programs is called Hoare logic, sometimes also Floyd-Hoare logic (Floyd, 1967; Hoare, 1969). The basic principle is to push the idea of specifications to its extreme and specify each *instruction* of a program with a precondition and a postcondition:

$$\{P\} C \{Q\},$$

where C is an instruction or a *command*, and P and Q are logical formulas expressing the pre- and postcondition, respectively. The meaning of this expression is: in any state where P is true, one can execute instruction C and finish in a state where Q is true. These so-called *Hoare triples* can be combined under certain conditions. As an example, when one has established that the postcondition of an instruction C_1 is equal to the precondition of an instruction C_2 , one can certainly first execute C_1 , then C_2 and obtain a state validating the postcondition of C_2 . In Hoare logic, this reasoning is written as an *inference rule*:

$$\text{SEQ} \frac{\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\}}{\{P\} C_1; C_2 \{R\}}$$

where $C_1; C_2$ precisely means “execute C_1 , then C_2 ”. An inference rule describes a reasoning step. The hypotheses are written above the bar, and the conclusion below. Similar rules have been developed for most constructs found in many programming languages. The aim is to obtain, by successive application of these rules, a triple $\{P\} C \{Q\}$ where C is the entire program to be considered, and P and Q correspond to the specification of the program. Once this has been done, one can conclude that the program is correct with respect to its specification. Many sets of rules of Hoare logic have been proved to be *complete*, *i.e.*, in principle any correct program can be proved to be correct.

The huge drawback of Hoare logic is that for programs of even moderate size it is simply too much work to consider each and every instruction and establish a Hoare triple for it. There have been many efforts to improve this situation. One way is to propagate specifications automatically, and to require only certain key places to contain specification annotations, such as function definitions and loops. For loops, these special annotations are called *loop invariants*. The application of the rules can then be

1. Introduction

discovered automatically. The most popular variant of this is called the *weakest precondition calculus* (Dijkstra, 1975). Other methods try to discover specifications or loop invariants automatically. Nevertheless, Hoare logic and related methods remain by far the most heavyweight solutions. On the other hand, to prove functional specifications of complex programs they may be the only alternative.

Refinement. Refinement is a popular variant of Hoare logic where first a very high-level specification of a program or module is given, a specification that does not need to care about implementation details. In this initial phase of the process, abstract, or mathematical instructions can replace actual instructions of the programming language. This initial specification is then subject to one or several *refinement* steps; high-level and abstract parts of the program and the specification are replaced by more and more concrete instructions and specifications, until a fully executable program with a concrete specification is obtained. Refinement rules help to assure that each refinement step is correct. The correctness of the initial program and specification can be verified for example using Hoare logic. Compared to using only Hoare logic, a user of refinement can concentrate on the overall-structure of the program, and its high-level meaning, instead of being overwhelmed by implementation details early in the development process. Refinement proceeds top-down, while Hoare logic is a bottom-up approach. The technique of refinement has been enjoying a bit more use in the industry than approaches based only on Hoare logic, in particular the B method (Abrial, 1996). It has been used to verify, for example, the safety of the code of metro line 14 in Paris (Behm et al., 1999).

1.2.3. Language Techniques

We have mentioned several programming language independent techniques¹ to test or guarantee a certain program behavior. There is another class of techniques that aims to rule out certain well-defined classes of errors, but says nothing about other errors. The two most prominent classes of errors are bugs due to memory access and typing errors. Techniques to avoid these errors are usually directly built into the programming language. We briefly present the problems and proposed techniques.

Memory management. In so-called low-level languages such as C (Kernighan and Ritchie, 1978), the programmer has complete control over the memory used by the program. If the program needs more storage space, the programmer has to *allocate* (ask the system to provide) more memory; this memory needs to be *freed* after use, to be able to reuse it later for some other purpose. The programmer can read from and write to unallocated memory, and can leave memory uninitialized (it may contain any data that was previously stored at the same place). This liberty is sometimes needed to communicate with external devices, or can be used to write more efficient programs, but it is also a source of bugs. Typical programming errors in languages such as C are failure to initialize data (and get random results), failure to free unused memory (this results in *memory leaks*, a situation where a long-running program uses up more and

¹Note that the *principles* of these techniques are language independent, but to apply them to a particular program, they have to be adapted to the programming language that has been used.

more memory of the system, while it actually uses only a moderate amount of space) and access to unallocated memory (this can result in undefined behavior or program crashes).

There have been many approaches to eliminate this kind of faulty behavior related to memory management. Most modern languages have a *garbage collector*, a mechanism that automatically deals with allocation and deallocation of memory as needed. This is true for most *object oriented* languages with the notable exception of C++ (Stroustrup, 1991), all so-called functional languages such as the ML family (Leroy et al., 2008; Milner et al., 1997) and Haskell (Peyton Jones et al., 2003), and all dynamic languages such as Python, Perl and Ruby. Usually, at the same time the language denies direct access to the memory. As a consequence, this mechanism makes memory leaks and access to unallocated parts of the memory simply impossible. A garbage collector on its own does not rule out problems related to the initialization of data.

Another technique is the obligation to initialize all allocated memory. Despite the simplicity and usefulness of this restriction (of course it rules out problems with uninitialized memory, and the dreaded *NullPointerException* in Java), it has not seen as much success in programming languages as garbage collectors. It is present in dynamic languages and in functional languages, but mostly not in object oriented languages.

Type systems. Another widespread technique to avoid common programming errors is a *type system*. A type system is a technique to avoid operations that do not make sense, such as adding the boolean `true` to a number, as in

$$5 + \text{true}$$

If executed, such an operation can have undefined behavior or can even crash.

The idea of a type system is to attribute types to objects such as `5` and `true`. In particular, `5` will be of integer type, often written `int`, and `true` of boolean type, often written `bool`. One also associates a *signature* to each operation describing its input and output types; for example `+` takes two integers and returns an integer. Looking at the types, it becomes clear that `true` is not of the right type to be an argument to `+`. The expression `5 + true` contains a *type error*.

The field of type systems is very vast. A possible distinction is between *static* and *dynamic* types. In a dynamic type system, the program itself, while running, detects that a type error would occur and stops the execution of the program. The idea is that such an abortion is usually preferred to continuing with undefined behavior. On the other hand, static type systems are capable of detecting type errors at *compile time*, *i.e.*, during the translation of the program to machine code. A program that contains a type error does not even get executed. Languages with static type systems usually exploit typing information to compile programs more efficiently. A drawback of static types is that the analysis can be imprecise, *i.e.*, a program that cannot exhibit a type error at runtime may be rejected anyway.

Another distinction is the obtained *type safety*. The language C has a static type system, but it also gives the possibility to change the type of objects with an operation called *cast*. Casts basically disable the type system for a part of the program. Unfortunately, the type system of C is so restrictive that casts are often the only way to

1. Introduction

achieve a certain behavior. Due to frequent casts and manual memory management, the fact that a C program is well-typed does not give many guarantees about its run-time behavior.

On the other hand, languages such as Haskell and ML give strong guarantees about well-typed programs. Their type systems, in connection with their garbage collected memory management, can guarantee that a well-typed program does not access unallocated memory nor read uninitialized memory. It also cannot contain memory leaks (in the sense described above). They also avoid type errors: the types of objects never change, and expressions such as $5 + \text{true}$ can never occur, even during program execution. Certain types of runtime errors can still occur when the type system is not strong enough to detect them, e.g., division by zero. Already in this setting, one can regard typing as a way to *prove* the absence of certain errors.

Languages such as Coq (The Coq Development Team, 2008), Epigram (McBride and McKinna, 2004) and Agda (Coquand and Coquand, 1999) take this idea even further and have so expressive type systems that even functional correctness can be expressed within the system. As an example, while usual type systems such as the ML type system have a type `list` for list-like structures, in Coq one can define the type describing *sorted* lists. As a consequence, a sorting function for lists would be characterized by stating that it returns not only a list, but a sorted list. Its functional correctness has been expressed using types. Types of this more powerful form are called *dependent types* and there is very active research on how to use dependent types to prove programs. A nice aspect of this way of specifying correctness is that there is no additional mechanism as Hoare logic to be applied. Everything is already in the type system. However, these systems put an additional burden on the programmer, who now has to prove that, for example, the sorting function has indeed the expected type. In the end, a user of a system with dependent types has to prove similar properties as a user of Hoare logic.

The dependent type systems of the last three languages are so powerful that they open up new applications; they can be used to prove mathematical theorems. This becomes possible through the remarkable *Curry-Howard isomorphism* (Curry, 1958; Howard, 1980), whose slogan is that “proofs are programs and theorems are types.” The main idea is that types in dependent type systems have the same structure as logical formulas in logical systems that belong to the family of *type theory*. This means that dependent types can be used to express theorems. To prove such a theorem, expressed by a type, the user must supply a program that can be given that type. In this sense, programming and proving really are the same activity.

1.3. Two Techniques Presented in more Detail

In this section, we want to introduce more thoroughly two aspects of program analysis that are particularly relevant to this work, the ML type system and Hoare logic.

1.3. Two Techniques Presented in more Detail

x, y, \dots Variables
 $E ::= x \mid E + E \mid \dots$
 $B ::= \text{True} \mid \text{False} \mid E = E \mid E \leq E \mid \dots$
 $C ::= \text{skip} \mid x := E \mid C; C \mid \text{while } B \text{ do } C \text{ done}$
 $\circ ::= \wedge \mid \vee \mid \Rightarrow$
 $P, Q ::= B \mid P \circ Q \mid \neg P \mid \forall x. P$

$\{ P \} C \{ Q \}$

SKIP $\frac{}{\{ P \} \text{skip} \{ P \}}$ ASSIGN $\frac{}{\{ P[x \mapsto E] \} x := E \{ P \}}$
 SEQ $\frac{\{ P \} C_1 \{ Q \} \quad \{ Q \} C_2 \{ R \}}{\{ P \} C_1; C_2 \{ R \}}$
 WHILE $\frac{\{ B \wedge P \} C \{ P \}}{\{ P \} \text{while } B \text{ do } C \text{ done} \{ \neg B \wedge P \}}$
 CONSEQUENCE $\frac{P \Rightarrow P' \quad \{ P' \} C \{ Q \} \quad Q \Rightarrow Q'}{\{ P \} C \{ Q' \}}$

Figure 1.1: Hoare Logic, as defined by Hoare (1969).

1.3.1. Hoare Logic in more Detail

The Definitions

C.A.R. Hoare, in his seminal 1969 paper (Hoare, 1969), introduces a very simple programming language commonly called WHILE, and rules to reason about it. Fig. 1.1 describes the language and the rules. Programs in this language manipulate a fixed set of mutable integer variables. *Expressions* E can be formed using variables and arithmetic operators. There are also *boolean expressions* B , returning a truth value. Finally, *commands* C consist of the no-op command `skip`, assignment $x := E$ which modifies the current value of x to the result of the expression E , sequencing using `;` and while loops which test against a boolean expression.

Along with this programming language, Hoare defines rules to establish *Hoare triples* of the form $\{ P \} C \{ Q \}$, where P and Q are logical formulas. Formulas are either boolean expressions, combined formulas using logical connectors or quantifications over variables. It should be noted that expressions and boolean expressions are shared between programs and logical formulas. Of course, the propositional values `True` and `False` are also available.

The intended meaning of the triple $\{ P \} C \{ Q \}$ is that in any state where P is true, if one executes C , one reaches a state where Q is true. This implies that the formulas P and Q can access the state; they do so by directly using the variables that are manipulated by the program. All the rules, with maybe the exception of the `ASSIGN` rule, are very intuitive: `SKIP` states that `skip` indeed does nothing (each formula that is true before is true afterwards), `SEQ` states that one can chain two commands if the postcondition of the first is equal to the precondition of the second. `CONSEQUENCE` states that one can strengthen the precondition and weaken the postcondition. `WHILE` states that if a formula P is true before the loop, and if the loop body C preserves P when the condition B is true, then P is true when exiting the loop, as well as the negation of B . This rule is interesting because it introduces the notion of *invariant*; P is a formula whose validity must be left *unchanged* by the loop body C .

The assignment rule `ASSIGN`² seems to be backwards: it states that if P is true *after* assigning E to x , the formula obtained from P by substituting E for x is true *before* executing the command. However, looking at an example shows that this is indeed a good way to formulate what assignment does. The Hoare triple

$$\{ 0 = 0 \} x := 0 \{ x = 0 \}$$

is an instance of the axiom and is indeed intuitively correct. The precondition $0 = 0$ is trivially true and can be replaced by `True` using the `CONSEQUENCE` rule.

Notation. This is a good moment to make some comments about the notations used throughout the document. A defined language is always written in capital letters, such as `WHILE`. Parts of the concrete syntax, in particular keywords of the language, are written in sans serif, such as the keyword `while`. Finally, names of inference rules are written in small caps, such as `WHILE`.

²`ASSIGN` is actually an *axiom*, as it has no premises.

1.3. Two Techniques Presented in more Detail

We use *metavariables* to denote different elements of the syntax, such as programs, or formulas, or boolean tests in the case of the WHILE language. Metavariables are either atomic, such as the metavariables x, y, \dots representing program variables, or they can correspond to a syntactic category. To describe the structure of such a category, we use the following notation:

$$V ::= S_1 \mid S_2 \mid \dots \mid S_n$$

V is the metavariable, and the S_i are the different possibilities to build elements in this syntactic category. The metavariable V and any other previously defined metavariable can appear in one or more of the S_i , in this case the category has recursive structure. We can use more than one metavariable to define a syntactic category. As an example consider the definition of formulas in WHILE:

$$P, Q ::= B \mid P \circ Q \mid \neg P \mid \forall x.P$$

Here we say that the metavariables P and Q stand for formulas, and these formulas can be constructed in four different ways; we use both P and Q to express the recursive nature of the structure of formulas. For formulas, we will sometimes introduce the notation $P(x)$ to denote a formula that may contain the variable x . In this context, we may then write $P(E)$ to denote the same formula, with E substituted for x everywhere.

Inference rules are always presented like this:

$$\text{NAME} \frac{H_1 \quad \dots \quad H_n}{C}$$

NAME is the name of the inference rule, H_1 to H_n are the hypotheses of the rule, and C is the conclusion of the rule. An *instance* or *application* of an inference rule can be obtained by replacing all free metavariables of a rule by concrete instances. Inference rules are always attached to some predicate or *judgment*, such as the Hoare triple $\{ P \} C \{ Q \}$. The conclusion of every inference rule belonging to this judgment is of this form. In general, the hypotheses H_i can contain judgments of this form as well. The inference rules of a judgment thus describe a way to build *proof trees* to obtain instances of this judgment. Rules whose hypotheses do not contain this judgment are called *leaves*, because the proof tree ends at this point. Rules without hypotheses are called *axioms*.

Another issue is the one of variable bindings. Most people agree that for formulas such as

$$\forall x.P(x),$$

the variable name x chosen here should not be of any importance. The formula

$$\forall y.P(y)$$

is an equally good way of expressing the same property. Also, when we write

$$x = 0 \Rightarrow \forall x.P(x) \tag{1.1}$$

the x on the left is distinct from the one on the right; in particular, as we just mentioned, we can replace the x on the right by y to make this distinction clear. Language

1. Introduction

constructs that introduce a variable name, such as \forall , are called *binders*. Variables that refer to a variable introduced by a binder (such as the rightmost x in 1.1) are called *bound*, while the other variables (such as the leftmost x in 1.1) are called *free*.

It follows from the discussion that bound variables are always distinct from all free variables in the context and from each other. So, (1.1) is actually an abbreviation for

$$x = 0 \Rightarrow \forall x'. P(x')$$

because bound variables are always different from free variables.

All this is very clear intuitively, but it becomes complicated when writing it down (very) formally, or when implementing variable bindings, in particular when one uses variable names such as x and y as binders. Indeed, one proposed solution to this problem is to remove variable names from binders and to replace bound variables by some means to *point* to the corresponding binder. These pointers could for example be so-called *de-Bruijn indices* (de Bruijn, 1972), integers that count the number of binders between the variable and the corresponding binder. In a pure de-Bruijn approach, free variables are also replaced by integers, and the context has to clarify which integer stands for which variable. In another, increasingly popular approach called the *locally nameless representation* (Pollack, 1994; Aydemir et al., 2008), only bound variables are replaced with integers, while free variables are represented by the usual variable names. Of course, the context still has to state what a variable name stands for.

As variable binding is not the primary issue of our discussion, we will stick with the less formal *Barendregt convention* (Barendregt, 1984), stating that bound variables are always different from free variables in any given context, using variable names everywhere. We believe that this practice is actually quite close to the locally nameless approach, without its formal rigor.

Let us close the discussion about notation and go on with the discussion about Hoare logic.

Partial correctness. It is important to note that, even when one has proved that $\{ P \} C \{ Q \}$ holds, it can be the case that executing C in a state validating P does *not* result in a state validating Q . Namely, when C does *not* terminate, it does not halt in such a state. In fact, the definition of Hoare triples only guarantees *partial correctness*, *i.e.*, it does not exclude non-termination. So the more precise interpretation of a Hoare triple $\{ P \} C \{ Q \}$ is the following: If C is executed in a state where P is true, *and if C terminates*, it does so in a state where Q is true. In a partial correctness setting, if C does not terminate, one can prove $\{ P \} C \{ False \}$. Hoare logics that guarantee *total correctness* can be formulated, and in this case the Hoare triple guarantees termination as well. In the WHILE languages, the only rule that would have to be changed is the WHILE rule, as *while* is the only construct that can be a source for non-termination. A common way to achieve this is to introduce, in addition to the *invariant*, a *variant*. A variant is an integer expression that must be non-negative and decreases at each iteration of the loop. An additional hypothesis of such a modified WHILE rule could look like this:

$$\{ z = E \wedge z > 0 \} C \{ z' = E \wedge z' \geq 0 \wedge z' < z \}$$

Auxiliary variables. Hoare logic in the presented form requires the use of *auxiliary variables*, variables that only appear in the logic, but not in programs, and whose purpose is to stand for the value of a program variable at a given time. For example, in Hoare logic one cannot directly express that the command $x := x + 1$ increases the program variable by one; instead, one has to say the equivalent of “if x is equal to some z before, x is equal to $z + 1$ after executing the command”. In Hoare triple notation, we write:

$$\{ x = z \} x := x + 1 \{ x = z + 1 \}$$

Here, z stands for the value of x before executing the command.

Auxiliary variables can quickly become overwhelming in program proofs, and a user may easily lose track which auxiliary variable stands for which program variable at which point in time. A possible solution has been proposed in Why (Filliâtre, 2003) with the use of *labels*. A label L is simply a name given to a program point. We write $L(x)$ to say “the value of x at program point L .” Now, the previous Hoare triple can be rewritten as follows:

$$\{ \text{True} \} L : x := x + 1 \{ x = L(x) + 1 \}$$

where L is the program point just before the assignment.

Predicate Transformers

When justifying the rules of Hoare logic, they are usually read from top to bottom, as the upper part represents the hypotheses and the lower part represents the conclusion. When trying to prove a program using Hoare logic, however, one usually does the opposite: Starting from a triple $\{ P \} C \{ Q \}$ that one would like to obtain, one tries to apply the different rules bottom to top, in order to build a *proof tree*. Leaves of the tree are either applications of one of the axioms SKIP or ASSIGN, or logical formulas to prove such as in the CONSEQUENCE rule.

After doing a few proofs in Hoare logic, one finds that the applications of SEQ and ASSIGN are completely mechanical. Sure enough one can come up with a function f that takes as input a command C and a postcondition Q , and returns a precondition P such that $\{ P \} C \{ Q \}$. And indeed, for skip, the sequence and assignment this is easy:

$$\begin{aligned} f(\text{skip}, Q) &= Q \\ f(x := E, Q) &= Q[x \mapsto E] \\ f(C_1; C_2, Q) &= f(C_1, f(C_2, Q)) \end{aligned}$$

This is just a different formulation of the inference rules, and it is easy to see that we have

$$\{ f(C, Q) \} C \{ Q \}.$$

For the WHILE rule as currently stated, this does not work, because the postcondition in the conclusion is not atomic, and neither is the precondition. We therefore need to reformulate this rule. A possibility is

$$\text{WHILE2} \frac{B \wedge I \Rightarrow P \quad \{ P \} C \{ I \} \quad \neg B \wedge I \Rightarrow Q}{\{ I \} \text{ while } B \text{ do } C \text{ done } \{ Q \}}$$

1. Introduction

This formulation can actually be derived from WHILE by the CONSEQUENCE rule. But now there is a difficulty to carry over this rule to our function f ; the problem is that in order to call f on the body C , one needs the invariant I . There is no simple solution to this problem, so for now we assume that the invariant is given along with the while loop.

Let us try to find the corresponding case for the function f by looking at an example. Let n be a positive integer; we would like to prove the following Hoare triple:

$$\begin{array}{l} \{ x = n \wedge y = 0 \wedge 0 \leq n \} \\ \text{while } x > 0 \text{ do } \{I\} x := x - 1; y := y + 1 \text{ done} \\ \{x = 0 \wedge y = n\} \end{array}$$

where the loop invariant I is

$$I \Leftrightarrow x + y = n \wedge 0 \leq x \wedge 0 \leq y$$

In the following, let us call P the precondition of the triple, Q its postcondition, and C the loop body. It is clear that we have to prove three different facts:

- The loop invariant I must initially hold. This can be reformulated by stating that the precondition implies the loop invariant:

$$x = n \wedge y = 0 \wedge 0 \leq n \Rightarrow x + y = n \wedge 0 \leq x \wedge 0 \leq y$$

This can be easily proved.

- The loop invariant I is preserved by the loop body C , if the loop test was true. This can be expressed by the following Hoare triple:

$$\{ x > 0 \wedge I \} C \{ I \} \tag{1.2}$$

Instead of searching for a proof tree, let us compute $f(C, I)$:

$$\begin{aligned} f(C, I) &\Leftrightarrow f(x := x - 1; y := y + 1, I) \\ &\Leftrightarrow f(x := x - 1, x + y + 1 = n \wedge 0 \leq x \wedge 0 \leq y + 1) \\ &\Leftrightarrow x - 1 + y + 1 = n \wedge 0 \leq x - 1 \wedge 0 \leq y + 1 \\ &\Leftrightarrow x + y = n \wedge 0 \leq x - 1 \wedge 0 \leq y + 1 \end{aligned}$$

The properties of f guarantee that we have $\{ f(C, I) \} C \{ I \}$, and if $x > 0 \wedge I$ implies $f(C, I)$ — which is clearly the case — we can conclude by CONSEQUENCE that (1.2) holds.

In essence, we had to prove $x > 0 \wedge I \Rightarrow f(C, I)$, and this for *any* possible situation. Translated to logic, this means that we have to *quantify* over the variables in the formula:

$$\forall x \forall y \forall n. x > 0 \wedge I \Rightarrow f(C, I)$$

- The invariant, when the loop condition is false, has to imply the postcondition Q . Again, this must be true for *any* situation in which the loop stops, so we need to quantify:

$$\forall x \forall y \forall n. \neg x > 0 \wedge I \Rightarrow x = 0 \wedge y = n$$

To summarize, a preliminary formulation of f for while loops is the following:

$$f(\text{while } B \text{ do } \{I\} C \text{ done}, Q) = I \wedge (\forall \bar{v}. B \wedge I \Rightarrow f(C, I)) \wedge (\forall \bar{v}. \neg B \wedge I \Rightarrow Q) \quad (1.3)$$

where \bar{v} is the list of involved variables.³

An important optimization can be applied to this transformation. We recall that in a Hoare triple, the precondition refers to the state before the execution of the instruction, and the postcondition to the state after. The function f can be seen as *transforming* a logical formula (or predicate) about the final state into one about the initial state; such functions are called *predicate transformers*. This discussion makes clear that the result of f is a formula concerning the initial state, and this is why we didn't quantify any variables in the leftmost occurrence of I in (1.3). The variables in this occurrence of I directly refer to the initial state. Now, if a variable does not change during the execution of the loop, e.g., n in our example, *all occurrences* of n should be equal, independently of the number of loop iterations. Therefore, we can restrict the quantification to variables that are actually modified by the loop body. The final formulation of the case for while loops is now the following:

$$f(\text{while } B \text{ do } \{I\} C \text{ done}, Q) = I \wedge (\forall \bar{\omega}. B \wedge I \Rightarrow f(C, I)) \wedge (\forall \bar{\omega}. \neg B \wedge I \Rightarrow Q)$$

where $\bar{\omega}$ denotes all variable names that are modified by the command C . The list ω of modified variables is also called *effect* of C , and this will be a central notion in this work.

Dijkstra (1975) proposed this style of reasoning: instead of using Hoare logic directly, *i.e.*, annotating every subexpression of the program, annotate only while loops and let the rest be done automatically. Using the function f , one obtains a formula which has to be proved to establish the correctness of the program. This formula corresponds roughly to the conjunction of all the leaf formulas in the Hoare logic proof tree. In his paper, Dijkstra calls the predicate transformer wp instead of f ; wp stands for *weakest precondition*, because $wp(C, Q)$ is indeed the weakest formula P such that $\{P\} C \{Q\}$. This can be proved easily. Dijkstra actually gives a slightly different definition of wp , partly because his language is a bit different from the WHILE language but also because he is interested in total correctness.

Structural rules. There is a difference between the CONSEQUENCE rule and the other rules of the system: CONSEQUENCE is not tied to a particular program construct; instead, it can be applied everywhere. Such rules are called *structural rules*. On the one hand, the presence of structural rules gives a lot of liberty over the way proofs are done, on the other hand they are a hindrance when one wants to automate parts of the process. The presence of structural rules also makes proofs *about* Hoare logic more difficult: indeed, such proofs will often proceed by a case analysis either on the structure of a program or the structure of a derivation tree in Hoare logic. Without structural rules, both would be essentially equivalent: the structure of the program would dictate the structure of the derivation and vice-versa. Contrarily, in the presence of structural rules, these can be found everywhere in the proof tree. The wp formulation

³Here, and in the remainder of the document, we use a bar, such as \bar{v} , to indicate a list of objects.

1. Introduction

can also be seen as a variant of Hoare logic without structural rules, and thus easier to automate and to reason about. As evidence that this is indeed the case, observe that our formulation of wp has no equivalent of the CONSEQUENCE rule; therefore, in proofs, there is one case less to be considered. Moreover, the structure of the program and the structure of the recursive calls of the wp function are identical.

Limitations of Traditional Hoare Logic

We now describe two particular aspects of programs that are more difficult to deal with in Hoare logic.

Aliasing. In the WHILE language, the set of program variables is fixed in advance. This has two consequences which make life easier for Hoare logic. The first one is that one cannot create any new variable; we will come back to this problem later. The second one is that one always knows when two variables are different: simply when they have different names. To illustrate this, let us look again at the ASSIGN rule:

$$\{ P[x \mapsto E] \} x := E \{ P \}$$

or even better, a simple instance of it:

$$\{ 0 = 0 \wedge y = 1 \} x := 0 \{ x = 0 \wedge y = 1 \}$$

How can we know that the assignment of x does not modify y ? The answer is that this must be guaranteed by the language; in our case, the semantics of WHILE (which we have not given) guarantee that modifying a variable can never modify another. If this is not the case, Hoare logic as presented breaks down.

To see why, imagine that we add function definitions to our language, with the following syntax:

$$\text{function } F(x_1, \dots, x_n) = C$$

where the x_1, \dots, x_n are the arguments of F which can be used just as program variables inside F . Also, they are passed *by reference*, which means that F can modify its arguments, and these modifications are visible once F has returned. A function call has the following syntax:

$$C ::= \dots | F(x_1, \dots, x_n)$$

Now it seems natural to postulate the following rule:

$$\text{CALL} \frac{\text{function } F(x_1, \dots, x_n) = C \quad \{ P \} C \{ Q \}}{\{ P[x_i \mapsto y_i] \} F(y_1, \dots, y_n) \{ Q[x_i \mapsto y_i] \}}$$

In English: when calling a function whose body is C , and for which we have proved $\{ P \} C \{ Q \}$, we can prove the same triple for the function call, but replacing the formal parameters x_i by the actual parameters y_i . But adding this rule is already incorrect and the reason is that, in our modified language, the assignment rule is no

longer valid. To see this, let us return to our previous example, but now let us put the code into a function call:

```
function  $F(x, y) = x := 0$ 
```

This function sets its first argument to zero and ignores its second argument. We can easily prove the Hoare triple

$$\{ 0 = 0 \wedge y = 1 \} x := 0 \{ x = 0 \wedge y = 1 \}$$

for the body of the function, using the ASSIGN axiom. But if we call F with the same actual parameter a for both formal parameters, we arrive at a contradiction:

$$\{ 0 = 0 \wedge a = 1 \} F(a, a) \{ a = 0 \wedge a = 1 \}$$

while the precondition can be easily satisfied (just assume that $a = 1$ before calling F), the postcondition is clearly false: a cannot be simultaneously equal to 0 and to 1.

The reason for this faulty reasoning is a breach of contract. When proving the Hoare triple for the body of the function, we have assumed that x and y are different (by application of the ASSIGN rule), but this turned out to be false when F was actually called. The variables x and y have become *aliased*.

This is the *aliasing problem* and there have been uncountable attempts to solve it. Our example of aliasing may seem contrived, but similar situations arise when reasoning about more complex structures such as arrays, mutable lists, and trees. There are roughly three possibilities to solve this problem:

- One can disallow aliasing entirely by restricting the language, in particular function calls. For example, a common syntactic restriction of early papers in the Hoare logic community was that function calls were not allowed to mention the same variable twice (as in $F(a, a)$). These approaches may potentially add some analysis to recover from common situations. More systematic proposals, but still based on the idea of disallowing aliasing, are the proposal called *Syntactic interference* (Reynolds, 1978) and the Why language (Filliâtre, 2003).
- One can disallow using mutable program variables directly in the logic; instead, the logic provides access and update functions (`acc` and `upd`) and some means to talk about the state. In our example of the assignment of x that may or may not leave y unchanged, the following Hoare triple is now correct:

$$\begin{aligned} & \{ acc(x, upd(x, 0, s)) = 0 \wedge acc(y, upd(x, 0, s)) = 1 \} \\ & x := 0 \\ & \{ acc(x, s) = 0 \wedge acc(y, s) = 1 \} \end{aligned}$$

As before, the first equation of the precondition reduces to $0 = 0$ and is trivial, but the second equation can only be simplified if one knows if x is equal to y or not. In particular, if $x = y$, the equation reduces to $0 = 1$ and cannot be proved; the Hoare triple is still correct. The functions `acc` and `upd` actually form a simple *memory model*, a way to reason about the memory layout of a program.

The Pioneers of this approach are Cartwright and Oppen (1981). A problem of this approach is that all assignments must now be justified using the `acc` and

1. Introduction

`upd` functions, and a large amount of inequalities between variables is needed. The reason is that when a single variable is modified, the entire state of the program is considered to be modified. Many extensions have been proposed to obtain inequalities between variables for free, for example when two variables are of different type. Examples of these extensions are the papers of Burstall (1972) and Bornat (2000). Tools such as Caduceus (Filliâtre and Marché, 2007) are based on these techniques. Another line of work uses analyses such as the one by Tofte and Talpin (1997) to cut the memory into small separate pieces instead of one huge block. Variables that live in different regions are automatically different. An example of an application of this technique is (Hubert and Marché, 2007).

One can also observe that this approach corresponds to the standard Hoare logic approach with only a single mutable variable s , the state. The actual mutable variables of the program, such as x and y above, are simply considered as entries in the state, and they never change (they always point to the same entry). The *contents* of the entry can change, using the *upd* function. An assignment operation such as $x := 0$ becomes *syntactic sugar* (an abbreviation intended to increase readability of a program) for an assignment to s :

$$s := \text{upd}(x, 0, s)$$

Using this viewpoint, the rules of Hoare logic do not need to be changed at all.

- Another, radically different approach is *separation logic*. Here, the state is built into the semantics of the logic, but does not appear explicitly in formulas. A formula of the form $x \mapsto v$ states that the program variable x does contain the value v ; the formula's *support* is the memory location of x . The *separating conjunction* $*$ can be used to connect formulas with disjoint supports: the formula

$$x \mapsto 0 \wedge y \mapsto 1$$

states that x and y currently have certain values, but the formula

$$x \mapsto 0 * y \mapsto 1$$

additionally states that x and y are not in the same memory location; the formula is false if x and y are in fact the same location. In our running example, the Hoare triple

$$\{ x \mapsto n * y \mapsto 1 \} x := 0 \{ x \mapsto 0 * y \mapsto 1 \}$$

is correct for any n and can become the specification of the function F . For a function call $F(a, a)$, the precondition of F becomes

$$a \mapsto n * a \mapsto 1$$

which is always false, because the left and right hand side of the star share the memory location of a in their support. The Hoare triple itself remains correct even after substitution.

Separation logic was first introduced by Reynolds (2002), and it has attracted considerable attention since its discovery (Nanevski et al., 2006). We call this

approach radical because it means a departure from the well-understood and well-supported tools that first-order logic and standard higher-order logic have been for decades.

Functions as values. Hoare logic was initially introduced using the WHILE language, but it has been subsequently extended and refined to languages with procedures and functions, recursive procedures, local variables, arrays and so on. Apt (1981) gives a detailed overview of the first ten years of Hoare logic. With the rise of the influential Algol language (Backus et al., 1960), it also became a challenge for the Hoare logic community to apply their techniques to this language. However, Algol is relatively rich and contains procedures and functions, nested procedures and *higher-order functions*, *i.e.*, procedures that can take other procedures as parameters. The problem is the following: if F is a function parameter of some other function, what can possibly be said about a call to F ? In other words, what does a Hoare triple for calls to function parameters look like:

$$\{ ? \} F(x) \{ ? \}$$

A possibility is to *inline* higher-order function calls; if

$$\text{function } G(F) = C$$

is a higher-order function definition, instead of reasoning about C separately, and then about calls to G , say $G(F')$, one can simply reason about $C[F \mapsto F']$. This approach, expressed by a rule which is usually called the COPY rule, is simple and works, but it is very inelegant. Most importantly, it is not modular: if C is a big and complex piece of code, and if it is called several times in the main program, one has to reason several times about (almost) the same code. Modularity, the ability to reason separately about separate components of a program, is considered to be one of the key properties to have for any proof system.

Clarke (1979) was able to prove that a sound and complete Hoare logic for a programming language with the following features:

1. procedures as parameters (higher-order functions)
2. recursion
3. static scope
4. global variables in procedure bodies
5. nested procedure declarations

could *not* be defined using a first-order annotation language. This negative result came as a surprise to many. So it was clear that either the features of the programming language had to be restricted or the logic used in annotations had to be enriched. Damm and Josko (1983) gave a Hoare logic for a subset of Algol containing higher-order features. They circumvent Clarke's impossibility result by moving to a higher-order logic. Their rule for function application is the following: If one has

$$\{ p \} F \{ q \}$$

1. Introduction

for a function F , one can derive

$$\{ p \ x \} F(x) \{ q \ x \}$$

for an application of this function. The symbols p and q are variables that represent one-argument predicates; the notation $p \ x$ denotes application of the predicate p to the argument x . To express this particular Hoare triple concerning F , they need higher-order logic. This work is limited to a certain very restricted form of preconditions, which have to describe exactly the state before the function call. Also, they *only* allow procedure parameters, no ordinary parameters.

More recently, Honda and others were able to design a program logic for a language with higher-order functions and closer to actual programming languages, first only with global state (*i.e.*, without aliasing), later including aliasing (Honda et al., 2005; Berger et al., 2007). Again, they extend the logic used in annotations to avoid the impossibility result of Clarke. Their logic is first-order, but it contains Hoare triples *in the logic*, so that one can write, for example:

$$\{ \forall x. \{ P(x) \} F(x) \{ Q(x) \} \wedge P(z) \} F(z) \{ Q(z) \}.$$

The Hoare triple inside the precondition states exactly what is needed to justify the call to F and the postcondition.

Régis-Gianas and Pottier (2008) established a Hoare-like system for a *purely functional* higher-order language. Their approach is very elegant, but eludes the more difficult part of dealing with side effects. Here is how it works. They allow themselves to use the names of program functions in the logic; however, one cannot *execute* these functions in the logic, because that could easily lead to incorrect derivations (because of the possible non-termination of program functions). Instead, in the logic, it is possible to access the pre- and postconditions of this function by operators **pre** and **post**. This makes it possible to write the following (trivial) rule for function calls:

$$\{ \text{pre } f \ x \} f \ x \{ \text{post } f \ x \}$$

For any *concrete* function f , the formula $\text{pre } f \ x$ is equivalent to the actual precondition of f , specialized for x . The advantage of this approach is that one does not need to *know* the specification of f to prove something about calls to f . Instead, relations between specifications of functions can be written in a generic way.

1.3.2. The ML Type System and Extensions

A particularity of ML programs, shared by other strongly typed languages such as Haskell, is summarized by the slogan “well-typed programs can not go wrong”. This means that a well-typed program in these languages cannot exhibit certain forms of runtime behavior such as invalid memory accesses, access to uninitialized data and type errors. How can this be achieved? Let us look at the definition of ML. We will be formal, but not go into too much detail.

Syntax. ML does not distinguish between expressions and instructions. As a consequence, the syntax is remarkably short. ML expressions e can either be variables x , constants c (such as integers, booleans, but also functions such as $+$ can be constants), an application of a function to its argument, written $e e'$, an anonymous function in one argument, written $\lambda x.e$ and finally a way to introduce names for intermediate results, written $\text{let } x = e_1 \text{ in } e_2$. In total, we obtain the following syntactic categories:

$$\begin{aligned} c &::= n \mid \text{true} \mid \text{false} \mid + \mid \dots \\ e &::= x \mid c \mid e e \mid \lambda x.e \mid \text{let } x = e \text{ in } e \end{aligned}$$

Application in ML is left associative; the application of a two-argument function to its arguments can be written without parenthesis. As an example, the addition of two numbers can be written in two ways: $(+ m) n$ or $+ m n$. In this particular case we also allow the infix syntax $m + n$.

Semantics. In the theory of ML, the syntax is accompanied of its *semantics*, the definition of the meaning of ML programs. It can be expressed in many different ways, a common way (Wright and Felleisen, 1994) is to define a so-called small step semantics, a relation $e \rightarrow e'$ that describes that e will reduce (evaluate) to e' . There is also a notion of *values*. In ML, variables, constants and anonymous functions are values; values have the property that they do not reduce anymore. When we have $e \rightarrow v$, then v is the *result* of the evaluation of e .

To define the relation \rightarrow , we proceed by three steps. We first define a *top-level reduction relation* \rightarrow which is defined as follows:

$$\begin{aligned} (\lambda x.e) v &\rightarrow e[x \mapsto v] \\ \text{let } x = v \text{ in } e &\rightarrow e[x \mapsto v] \\ c v &\rightarrow \delta(c, v) && \text{if } \delta(c, v) \text{ is defined} \end{aligned}$$

This relation \rightarrow describes a reduction step at the top of an ML term. An anonymous function $\lambda x.e$, when applied to a value v , reduces to its body e with the variable x replaced by the argument v . This substitution is written $e[x \mapsto v]$. The reduction of a let-expression happens in the same way. When a constant c is applied to a value v , we check if c is defined for this argument, and return the result of this application; the check and the result are both realized using a function δ that “knows” all defined constants.

The next step is the definition of a relation \longrightarrow , describing a *single reduction step*. We first define a *reduction context* E as follows:

$$E := [] \mid \text{let } x = E \text{ in } e \mid E e \mid v E$$

A reduction context is either empty, or a let-expression with a context at the left hand side, or an application with a context on the left, or finally an application with a value on the left and a context on the right. One can also say that a reduction context is an ML expression with a *hole* (the empty context), but the hole can only appear in certain places of the expression. We write $E[e]$ for an expression that has been obtained by

1. Introduction

replacing the hole in a context E with the expression E . We now can define the relation \longrightarrow :

$$e \rightarrow e' \quad \text{implies} \quad E[e] \longrightarrow E[e']$$

To summarize, the relation \longrightarrow describes a single reduction step “anywhere” in a term, where the precise meaning of “anywhere” is described by the definition of reduction contexts. The definition we have given corresponds to a *strict* evaluation — this also corresponds to the evaluation order in ML — where arguments are evaluated before being passed to functions and let-bound expressions are evaluated before continuing the reduction. Other choices are possible and have been explored in the literature.

Finally, the relation \twoheadrightarrow is the reflexive and transitive closure of \longrightarrow . This means that for any expression e we have $e \twoheadrightarrow e$, and for any three expressions $e_1 \twoheadrightarrow e_2$ and $e_2 \longrightarrow e_3$, we also have $e_1 \twoheadrightarrow e_3$.

The syntax of ML permits ill-typed expressions such as the addition of a boolean and an integer. What does the semantics say about those? It says that these expressions do not continue to evaluate; there is no reduction rule for nonsensical expressions such as $1 + \text{true}$. We say that such expressions are *stuck*.

With types, we can forbid such expressions. We begin by a simplified vision of types called *simple types*. In this setting, types can either be constant types such as `int` or `bool`, and *function types* or *arrow types*, written with an arrow \rightarrow . The type `int \rightarrow int`, for example, describes the type of functions having an integer argument and returning an integer. The structure of types is summarized by the syntactic category τ :

$$\begin{aligned} \iota &::= \text{int} \mid \text{bool} \mid \dots \\ \tau &::= \iota \mid \tau \rightarrow \tau \end{aligned}$$

Now, the following typing rules define a relation $\Gamma \vdash e : \tau$ that describes that an expression e is of type τ under the *environment* Γ . An environment is simply a list of bindings from variables to types. The typing rules also define how this environment can be enriched.

$$\begin{array}{c} \text{S-CONST} \frac{\text{Typeof}(c) = \tau}{\Gamma \vdash c : \tau} \quad \text{S-VAR} \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \text{S-APP} \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \\ \\ \text{S-ABS} \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x. e : \tau' \rightarrow \tau} \quad \text{S-LET} \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \end{array}$$

Let us briefly describe these rules.

- A function *Typeof* defines the types of all constants;
- Variables are typed using the information stored in the environment;
- On an application, the left term must be of function type, and the right term must be of the corresponding argument type;
- When e is of type τ under the assumption that x is of type τ' , then we can build the function $\lambda x. e$, and this expression is of function type $\tau' \rightarrow \tau$.

1.3. Two Techniques Presented in more Detail

- The `let`-form permits to give names to results of sub-expressions; the name necessarily has the same type as the expression which it abbreviates.

Now, using the definition for $\Gamma \vdash e : \tau$ and the definition for \rightarrow , one can prove two important theorems.

Theorem 1.1 (Preservation). *For any well-typed expression e , i.e., an expression for which we can prove $\Gamma \vdash e : \tau$, and any e' such that $e \rightarrow e'$, e' is also well-typed and of the same type, i.e., $\Gamma \vdash e' : \tau$.*

Theorem 1.2 (Progress). *For any well-typed expression e , either e is a value, or there exists an expression e' such that $e \rightarrow e'$.*

The first theorem, preservation, states that during the evaluation of the program, the type of the expression is preserved. This property is also called *subject reduction*. The second theorem, progress, states that, if the program has not finished evaluating yet, one can still continue. Together, both theorems imply the fact that any well-typed expression in ML evaluates to a value of the same type. However, more subtle runtime errors such as division by zero are not considered by these theorems and can still happen. Wright and Felleisen (1994) seem to have been the first to prove the type safety of ML in this form.

The system of simple types we have presented is very restrictive. Every variable must have a fixed, concrete type. In practice, this means that functions that operate, for example, on lists, must be duplicated for each type the list may contain: lists of integers, lists of booleans and so on. There are many more cases where this restriction to concrete types is a problem. For this reason, ML actually has a more powerful *polymorphic* type system. To present it, we only need a few modifications to the definitions we have given. The first modification is that we allow *type variables* α as types:

$$\tau ::= \alpha \mid \dots$$

Next, our environment Γ maps not from variables to types, but to *type schemes*. A type scheme is a type with a quantifier prefix:

$$\sigma ::= \forall \bar{\alpha}. \tau$$

We use the bar to describe lists of objects (here: type variables). If an environment Γ says that x has the type scheme $\forall \bar{\alpha}. \tau$, this means that x can be used with any type that can be obtained by substituting the $\bar{\alpha}$ by a list of (more) concrete types. As an example, consider the type scheme $\forall \alpha. \alpha \rightarrow \alpha$. This type describes all functions that take an argument of any type, and return a result of the same type. The identity function $\lambda x. x$ is a good candidate for such a type scheme. A type scheme does not need to quantify over variables; in this case we say it is *monomorphic*.

The typing rules for constants, variables and `let`-bindings need a little bit of modification as well:

$$\begin{array}{c} \text{ML-CONST} \frac{\text{Typeof}(c) = \sigma \quad \tau \leq \sigma}{\Gamma \vdash c : \tau} \qquad \text{ML-VAR} \frac{\Gamma(x) = \sigma \quad \tau \leq \sigma}{\Gamma \vdash x : \tau} \\ \\ \text{ML-LET} \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \text{Gen}(\Gamma, \tau') \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \end{array}$$

1. Introduction

The rules ML-APP and ML-ABS are identical to their simply typed counterparts. We write $\tau \leq \sigma$ when τ is an *instance* of σ , *i.e.*, when τ can be obtained from σ by substituting the quantified type variables in σ by types. In the rules, we see that variables and constants can have polymorphic type, and that the typing rules can choose an instance of the type scheme to be the type of the expression. Note that abstractions using λ still introduce monomorphic variables. The only way to *introduce* polymorphic variables is using `let`.

Not all variables in a type (such as τ' in the ML-LET rule) can be generalized. The function *Gen* determines the type variables $\bar{\alpha}$ that *can* be generalized in τ' and returns the type scheme $\forall \bar{\alpha}. \tau'$. The environment is needed to determine which variables are generalizable, so we write $Gen(\Gamma, \tau')$ to underline this dependency. This function checks if a type variable in τ' also appears in the environment Γ . If it does, it can *not* be generalized; if not, it can be generalized.

In this variant of ML, one can write *polymorphic* functions, that work for several concrete types. But polymorphic types are not just a tool to avoid code duplication; they also express generic properties about the objects of the programming language, without knowing the concrete instantiations that will appear later in the program. For example, using the type scheme $\forall \alpha. \alpha \rightarrow \alpha$, we are able to express that a function of this type has always identical argument and return types, even though we do not know yet what these types will turn out to be. It is worth noting that in a restricted setting, polymorphic types can say much more about the corresponding functions than one might expect (Reynolds, 1983).

The theorems of Preservation and Progress are of course correct also in the polymorphic setting of ML.

Type inference. An important aspect of ML, and in fact the source of some particularities of its formulation, is type inference. The reader may have noticed that ML terms do not contain any types; how does one know which type to associate to the variable of a λ -abstraction? Finding this out is the job of *type inference*, and one of the most important properties of ML is that type inference is decidable and complete. There is an algorithm, called *W* in the literature (Damas and Milner, 1982) that, for any ML program, finds a typing derivation if one exists. This is a valuable property, and ML is among the most powerful type systems that still enjoy this property. To illustrate this, consider a modification of ML where the variables at λ -abstractions can also be polymorphic; the resulting system is called *System F* (Girard, 1972). This seems like a minor modification, but it came as a surprise to many that type inference in System F is undecidable (Wells, 1998).

Recursion. A looping construct such as `while` in the WHILE language, or another means to express iteration, is essential to obtain *Turing completeness* for a programming language, *i.e.*, the property that in principle every *computable* function can be expressed in this language. There are (at least) two ways this can be achieved in ML. The first is by introducing a *fixed-point combinator*, written *Y*, as a new expression:

$$e ::= \dots \mid Y$$

We also add a new reduction rule for the relation \rightarrow :

$$Y v \rightarrow v (\lambda x. Y v x)$$

To understand what this does imagine that v is a two-argument function:

$$v = \lambda f. \lambda y. E[f v']$$

The argument f of v is used for recursive calls; we have fixed the body of v to some expression whose first step is precisely a call to f using some value v' . Then for some value v_0 ,

$$\begin{aligned} Y v v_0 &\rightarrow (\lambda f. \lambda y. E[f v']) (\lambda x. Y v x) v_0 \\ &\rightarrow (\lambda y. E[(\lambda x. Y v x) v']) v_0 \\ &\rightarrow E[(\lambda x. Y v x) v'] [y \mapsto v_0] \\ &\rightarrow E[Y v v'] [y \mapsto v_0] \end{aligned}$$

The idea is that in v , every occurrence of f is bound to be replaced with $Y v$, in which again every occurrence of f is replaced by $Y v$ and so on. Y computes the *fixed point* of v , hence the name of fixed point combinator. One cannot replace Y by an actual *expression* in ML, but one can still give it a type scheme:

$$Y : \forall \alpha \beta. ((\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

Another, equivalent way is to allow anonymous function to be recursive. We simply need a second variable name that stands for recursive calls. To differentiate between usual λ -abstractions, we use the keyword `rec` instead of λ :

$$e ::= \dots \mid \text{rec } f(x). e$$

The variable f can be used for recursive function calls, and x is the name of the function argument. The reduction rules for \rightarrow must be changed again:

$$\text{rec } f(x). e v \rightarrow v[f \mapsto \text{rec } f(x). e, x \mapsto v]$$

Here, in addition to the substitution of v for x , we also replace the name f by the recursive anonymous function `rec $f(x). e$` . Now, one can choose to either supply these recursive anonymous functions in addition to usual λ -abstractions or replacing the latter by the former. After all, a non-recursive function is a recursive one that does not contain any recursive calls.

Algebraic data types and pattern matching. One of the most important concepts of ML-like programming languages, second only to strong typing and first-class functions, is the concept of *algebraic data types* and *pattern matching*. The idea of algebraic data types is that a user can define potentially polymorphic type constants along with *constructors*, *i.e.*, function constants which return objects of that type. A simple example is the option type:

1. Introduction

```
type option  $\alpha$  =  
  | None  
  | Some of  $\alpha$ 
```

This definition introduces the unary type constant *option* along with the constructor **None** of type scheme $\forall\alpha. \text{option } \alpha$ and the constructor **Some** of type scheme $\forall\alpha. \alpha \rightarrow \text{option } \alpha$. The definition of this algebraic data type also guarantees that *every* value of type *option* τ is either of the form **None** or of the form **Some** v' where v' is a value of type τ . The option type is a safe possibility to represent a value that may be present or not.

Another example is the list type:

```
type list  $\alpha$  =  
  | Nil  
  | Cons of  $\alpha * \text{list } \alpha$ 
```

Similarly, this defines a type constant *list* along with the constructor *Nil* of type scheme $\forall\alpha. \text{list } \alpha$, representing the empty list, and the constructor *Cons* of type $\forall\alpha. \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha$ (note that definitions of algebraic data types can mention the type to be defined, *i.e.*, they can be recursive), the operator which prepends an element to an already existing list. Note the separator $*$ in the definition of the algebraic data type, which separates the argument types of the constructor *Cons*. For a list of the form *Cons* x xs , we call x the *head*, and xs the *tail* of the list.

This way of defining a type and its constructors is already a nice help for programmers, compared to other, more low-level programming languages such as C, where defining such types along with constructors is tedious and error-prone. But ML also offers *pattern matching*, *i.e.*, a syntactic convenience to simplify the analysis of values that belong to an algebraic data type.

Imagine a program p that is supposed to execute an expression e_1 when a list l is empty, and execute some expression e_2 when the list is not empty. In e_2 , we want to use the head and the tail of l for computations. Let us first look at a program that does *not* use pattern matching. In this case, we assume that the following functions are provided:

$$\begin{aligned} \text{is_nil} &: \text{list } \alpha \rightarrow \text{bool} \\ \text{head} &: \text{list } \alpha \rightarrow \alpha \\ \text{tail} &: \text{list } \alpha \rightarrow \text{list } \alpha \end{aligned}$$

The function *is_nil* decides whether a list is empty (equal to *Nil*) or not. The functions *head* and *tail*, unsurprisingly, return the head, respectively the tail, of a non-empty list. If called on the empty list, they fail. In our idealized language, this failure can for example be modeled by non-termination; in a real language, some kind of runtime error would occur. We can now return to our program p and implement it as follows⁴:

⁴We assume here the existence of an if-then-else-construct. For the sake of completeness, let us describe the necessary steps to add such a construct to the language we defined in this section. We first need to extend the expressions accordingly:

$$e ::= \dots \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3.$$

if *is_nil* *l* then e_1 else e_2

In e_2 we can use the functions *head* and *tail* to obtain the desired information.

Now this approach works fine, but it has several drawbacks. First of all, the functions *head* and *tail* can fail when called on the empty list, for example in e_1 . Another problem arises when the algebraic data type has more than two constructors; we then need functions like *is_nil* for each constructor, and accessor functions similar to *head* for each of the arguments of each constructor. All of these functions fail when called with a value that does not correspond to the right constructor. It also becomes a burden for the writer of these functions to invent names for them, as well as for the programmer to remember them. Finally, the programmer might simply forget to check for certain cases (such as the empty list in the previous example), in particular when the number of constructors is high or is subject to change.

Pattern matching avoids these problems. Let us rewrite our example:

```
match l with
| Nil →  $e_1$ 
| Cons (x, xs) →  $e_2$ 
```

The *match* keyword introduces a pattern matching construct. It is followed by an expression whose type is an algebraic data type (here *list*) and the keyword *with*. It is followed by a list of *branches*, separated by the symbol *|*. Each branch consists of a *pattern* to the left of the arrow and an expression to the right. A pattern is a full application of a constructor to variables. The idea is that the list *l* is compared in turn with each of the patterns. If one pattern matches, the corresponding expression is executed, otherwise the next pattern is tried. When a pattern matches, its variables are bound to the corresponding components of the matched object. For example, in the pattern *Cons(x, xs)*, which matches if *l* is not the empty list, *x* is bound to the head of the list, while *xs* is bound to the tail of the list. The expression e_2 now can use *x* and *xs* instead of *head l* and *tail l*.

Pattern matching greatly simplifies programs dealing with algebraic data types. It removes the need for accessor functions, that are not only tedious to define and to use, but also error-prone. The code becomes clearer because of the clear separation of the different cases. Finally, the compiler can check for completeness of the pattern matching, *i.e.*, that the programmer has dealt with all possible cases.

We also need to adapt the top level reduction relation \rightarrow :

$$\begin{aligned} \text{if true then } e_1 \text{ else } e_2 &\rightarrow e_1 \\ \text{if false then } e_1 \text{ else } e_2 &\rightarrow e_2 \end{aligned}$$

as well as the reduction contexts:

$$E ::= \dots \mid \text{if } E \text{ then } e_1 \text{ else } e_2.$$

Finally, we have to extend the typing relation:

$$\text{S-IF} \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

1. Introduction

References. Another language feature of ML that we have not presented yet is the feature of mutable variables or *references*. At first glance, it is easy to add them, using a new type constructor `ref` τ and three new function constants, with their type schemes:

$$\begin{aligned} \tau & ::= \dots \mid \text{ref } \tau \\ \text{ref} & : \quad \forall \alpha. \alpha \rightarrow \text{ref } \alpha \\ ! & : \quad \forall \alpha. \text{ref } \alpha \rightarrow \alpha \\ := & : \quad \forall \alpha. \text{ref } \alpha \rightarrow \alpha \rightarrow \text{unit} \end{aligned}$$

To capture the semantics of these three functions, we need to modify the definition of \rightarrow significantly; expressions e are now evaluated with respect to a *store* s containing the values of all created references. The semantics is now of the form $s, e \rightarrow s', e'$. The old rules of the semantics do not modify the store so that we have $s = s'$, but the rules concerning references may modify it: `ref` adds a new entry to the store, `!` reads it, and `:=` modifies the store.

Damas (1985) was the first to give an account in this style, but his proof of type safety (Theorems 1.1 and 1.2) was incorrect, and that is because this modified language is not type safe! Without going into too much detail, the reason is that the generalization of type variables (written $Gen(\Gamma, \tau)$ in our notation) at `let`-bindings does generalize too much. As a consequence a reference can be written using one type and read assuming another, which can lead to a crash. Tofte (1990) discovered the error in the proof and published a version where less generalization takes place; others, for example Talpin and Jouvelot (1994), presented different systems to obtain the same restrictions on generalization. All those systems had the drawback to be relatively complex for a type checker in a programming language. Wright (1995), after having analyzed a huge amount of real ML programs, came to the conclusion that the vast majority of `let`-bound expressions are values; generalization of type variables in values does not pose a problem, so his proposal was to cut the ML-LET typing rule in two, according to whether e_1 is a value or not:

$$\text{ML-LETPOLY} \frac{\Gamma \vdash v : \tau' \quad \Gamma, x : Gen(\Gamma, \tau') \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = v \text{ in } e_2 : \tau}$$

$$\text{ML-LETMONO} \frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}$$

When e_1 is a value, we can generalize as before, but when e_1 is an expression (potentially containing side effects), we do not generalize at all. Wright showed that this modification to the type system, while it reduces the set of typable programs, has little negative impact in practice. It got adopted by most ML type checkers. In the same time, Harper (1994) proposed a variant of the typing rules where, in addition to the environment Γ , a *store typing* Σ can be used to type *store locations* l . These locations are the runtime equivalent of references and are added to the syntax. The store typing Σ is a mapping from locations to types such that the store contains at that location a value of that type. This variant increases the size of the definitions, but makes the proofs much simpler.

Effects and regions. Parallel to the integration of references into ML, it became obvious that typing can do more than just exclude invalid programs; it can also be used to discover additional properties of a program. A good example where typing helps to obtain additional information are *effect systems*. In an effect system, each expression is not only assigned a type, but also an *effect*; usually, an effect is a set of some atomic observable side effects. Such side effects can include assignments to mutable variables or raised exceptions, but also calls to certain functions or access to external resources such as files and peripherals. Now the idea is that some basic expressions, such as writing a mutable variable or opening a file, are defined to have a certain effect, and combining expressions combines their effects. Consider the two following typing rules:

$$\text{E-ASSIGN} \frac{\Gamma \vdash v : \tau \quad \Gamma \vdash x : \text{ref } \tau}{\Gamma \vdash x := v : \text{unit}, \{x\}} \quad \text{E-LET} \frac{\Gamma \vdash e_1 : \tau', \varphi_1 \quad \Gamma, x : \tau' \vdash e_2 : \tau, \varphi_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau, \varphi_1 \cup \varphi_2}$$

The first rule states that writing a variable x produces an effect $\{x\}$. The second rule concerns **let**-bindings and states that if e_1 has effect φ_1 and e_2 has effect φ_2 , the overall effect of the expression is the union $\varphi_1 \cup \varphi_2$ of both effects. The rule E-LET deliberately omits issues of type generalization.

In languages where functions can be values (such as ML), one has to distinguish between the *immediate* effect of an expression whose result is a function, and the *latent* effect of the function when it is called. The immediate effect is of the kind we have just seen; in the typing rules, it is stated after the type of the expression. The information about the latent effect must be put in the *type* of the function. Now, instead of a simple function type $\tau \rightarrow \tau'$, we deal with types of the form $\tau \rightarrow^\varphi \tau'$; functions of this type take an argument of type τ , return a result of type τ' and have the effect φ during execution. The usual rule for the application in such systems is thus the following:

$$\text{E-APP} \frac{\Gamma \vdash e_1 : \tau' \rightarrow^\varphi \tau, \varphi_1 \quad \Gamma \vdash e_2 : \tau', \varphi_2}{\Gamma \vdash e_1 e_2 : \tau, \varphi_1 \cup \varphi_2 \cup \varphi}$$

As can be read in the typing rule, the overall effect of an application is the union of the immediate effects of both expressions *and* the latent effect of the called function.

In the previous paragraphs we have used the names of mutable variables to form effects. This is possible and has been done, for example, in the Why language (Filliâtre, 2003), but this approach has severe limitations. Consider the following expression, in which x is assumed to be a mutable variable:

$$\text{let } y = x \text{ in } y := 0$$

We are clearly modifying the memory location attached to x , but via the program variable y . Should the effect of this expression contain x or y , or both? A consequence of this example is that systems that use program variable names to express effects must impose restrictions on the programming language to maintain precision. In Why, for example, renaming mutable variables using **let** is forbidden.

A more flexible, but more involved approach is to use the type system to obtain additional information. We dissociate the program variable and the memory location by giving a name to the location; this name is called a *region*. We now put this

1. Introduction

information in the type of references; instead of simply stating the type contained by the reference, we also state its location. The type $\text{ref}_r \tau$ describes the reference that points to the location r which contains a value of type τ . The assignment rule now does not use the variable name; instead it uses the region name to describe the effect of the assignment:

$$\text{R-ASSIGN} \frac{\Gamma \vdash v : \tau \quad \Gamma \vdash x : \text{ref}_r \tau}{\Gamma \vdash x := v : \text{unit}, \{r\}}$$

As the region information is contained in the type, we now can freely rename mutable variables. For example, in the example above, x and y are both of the same reference type, thus we know *by typing* that they point to the same location. There is no ambiguity anymore about the effect of the expression.

Regions which correspond exactly to a memory location are called *singleton regions*. It can become quite involved to maintain this property, so many type systems use *group regions* that group several memory locations, either exclusively or in addition to singleton regions. Group regions are less precise, but more flexible than singleton regions.

Lucassen and Gifford (1988) were the first to present an effect system containing effects and regions. Talpin and Jouvelot (1994) designed a correct and complete type inference algorithm for this system. Tofte and Talpin (1997) used the same basic ideas to propose a region-based alternative to garbage collection. Nielson et al. (1999) give a good introduction to type and effect systems in general and advanced issues such as type inference and subtyping.

1.4. Overview of the Document, Contributions and Related Work

In this section, we give an overview of the contributions of this document, justify central design choices and compare our work to the work of similar systems in the literature.

1.4.1. An Overview of the Document

The goal of this thesis is to obtain a practical system to prove properties of programs written in an ML-like language. This means that our system does have to support side effects and higher-order functions, and combinations of these features. “Practical” means that it should be possible to automatically prove simple properties of such programs.

In a nutshell, to achieve this goal, this thesis defines an ML-like language with an effect analysis, and a specification language (a variant of higher-order logic) that can express properties of the state. A wp calculus is introduced, as a means to obtain proof obligations from annotated programs. As these obligations are in higher-order logic, we also show a way to translate these formulas to first-order logic, which is the logic used by almost all automated provers. We now give a more detailed overview of each part.

In Chapter 2 we define a higher-order ML-like language, called W , with side-effects (references), equipped with an effect calculus. W has a type system similar to the one

of ML, but instead of a typing relation $\Gamma \vdash e : \tau$, there are actually two typing relations: one for values, of the form

$$\Gamma; \Sigma \vdash_v v : \tau,$$

and one for expressions that may have a side effect, of the form

$$\Gamma; \Sigma \vdash e : \tau, \varphi,$$

where φ is an *effect*, an expression that approximates the effect that the expression e may have on a store s when reducing to a value.⁵ Additionally, we improve this *effect analysis* by introducing *regions*; a region is a portion of the store. Regions also appear in types: each reference type is annotated by a region, so when a reference is modified, this modification effect can be associated to a certain region. To increase flexibility, the system also contains region and effect polymorphism, two mechanisms that are very similar to ML type polymorphism. It is effect polymorphism that guarantees the modularity of the effect analysis for higher-order functions. Chapter 2 contains a type soundness proof for this system.

The language W is a minor contribution of this thesis; The basic idea of this system is already present in [Lucassen and Gifford \(1988\)](#), but we also integrate refinements by other authors, for example the lighter syntax which is much closer to ML, by [Talpin and Jouvelot \(1994\)](#), the `letregion` keyword introduced by [Tofte and Talpin \(1997\)](#) in a different setting and finally the `region` keyword introduced by [Calcagno et al. \(2002\)](#) which greatly simplifies the soundness proof.

Chapter 2 also defines a specification language, called L , adapted to the programming language and its type system. This specification language is new. The two main ingredients of this specification language are higher-order logic and the presence of state types, of the form $\langle \varphi \rangle$, where φ is an effect expression. An object of state type represents a portion of the store and permits to express properties that depend on the state. Both ingredients together enable us to reason about higher-order functions with side effects. We refine the definition of W to include specifications written using formulas of L .

The central contribution of this thesis is the definition of a weakest precondition calculus for programs in W , given in Chapter 3. This calculus, written $\text{wp}(e, q)$, takes an expression e and a formula that specifies properties of the return value of e and final state, and returns a formula p in L , depending on the initial state. The formula p guarantees that e executes correctly and that the return value and the final state verify q . The formula p is in general a conjunction of *proof obligations*, that have to be proved in order to guarantee the correctness of the program. Thanks to the properties of the effect analysis and the logic L , the formulation of this calculus is relatively simple and resembles first-order formulations. We prove the soundness of this calculus, *i.e.*, the fact that if p is true, then e indeed executes correctly. We also prove a completeness result, that states that any “correct” program can be proved correct in this calculus. The wp calculus takes its general form from [Filliâtre \(2003\)](#), and a few ideas, in particular the reflection of effectful functions as pairs in the logic, are taken from [Régis-Gianas and Pottier \(2008\)](#).

⁵We cannot, at this point, explain the meaning of the symbol Σ in the typing relation.

1. Introduction

In Chapter 4, we show two modifications of W . The first one rules out aliasing of regions by excluding certain instantiations of region and effect polymorphism. We show that this leads to a simplification of the proof obligations, with only moderate impact on expressiveness. This system is a generalization of the system of Hubert and Marché (2007) to a setting with higher-order functions. A second modification of W , in addition to the first, rules out *aliasing* between regions entirely, limiting the system to so-called *singleton regions*. We again show that this implies important simplifications in the produced proof obligations, but this time the expressiveness of the system is greatly reduced; programs with shared mutable data structures are not well-typed anymore in this modified system. This more restricted system turns out to be an adaptation of the Why system (Filliâtre, 2003) to regions.

In Chapter 5, we consider practical issues that have been eluded in the previous sections. The wp calculus returns a number of proof obligations in L , a custom, non-standard logic. We show that formulas in L can easily be translated to a more standard higher-order logic. But to use state-of-the-art automated provers, which almost exclusively expect first-order formulas, there is more work to do. Therefore, we introduce an encoding from higher-order formulas to first-order formulas. Parts of the translation are well-known, for example from Meng and Paulson (2008) and Pottier and Gauthier (2006). However, we achieve two desirable properties: the first one is that formulas that are already in first-order form are left unchanged, which greatly improves efficiency of automated provers, in particular for built-in symbols like arithmetic operators. The second one is that a proof of equivalence between a concrete pair of input and output formulas can essentially be obtained by *evaluation*, and therefore is trivial.

In Chapter 5, we also briefly detail the prototype implementation called *Who*, which includes an implementation of W and L , the wp calculus and the translation from L to higher-order logic. The translation from higher-order logic to first-order logic is implemented in a separate tool called *Pangoline*, which has been developed in collaboration with Yann Régis-Gianas.

We close by giving many examples of programs that have been proved correct using *Who* and *Pangoline* (and the *Why* system to call different automated provers). The most complex example is Koda and Ruskey’s algorithm to enumerate certain colorings of forests.

1.4.2. Design Choices

The choice of the programming language. One could ask why we are interested in *ML* and not other, more widespread programming languages such as *C* or *Java*. A first answer is that while we indeed focus on *ML*, the techniques proposed in this document can also apply in other programming languages, even the treatment of higher-order functions. In *C*, one can manipulate pointers to functions and thus effectively write higher-order programs. In *Java* (and most other object-oriented languages), the concept of an object itself is already higher-order, as an object always contains *methods* (functions that are related to a particular object), except in the most trivial cases. Common idioms such as *callback functions* are simply higher-order mechanisms.

If these languages provide higher-order features, then why not use such a language as the basis of our work? The main reason is that the type system of *ML* gives us

much more guarantees and does in fact reduce the number of proof obligations one has to prove. In proof systems for C, one often has to maintain additional properties, in particular about pointers. They should always point to an allocated region, one needs to specify the size of the corresponding memory block, and so on. In practice, this encumbers the specifications *and* the proofs of a particular program. The choice of ML as programming language seems to be a natural step towards program verification. If the objective is to write a program and prove it correct, as opposed to prove correct an existing program, then ML or similar languages seem to be a better choice than “unsafe” languages such as C. It is therefore surprising that there are so few proof systems for such languages. This thesis is also an attempt to change this situation.

Finally, ML is close to be the smallest system in which higher-order features and effects are both present and can interact in interesting ways. Dealing with larger languages such as C and Java would also mean dealing with additional language construct whose treatment is mostly orthogonal to effects and higher-order functions. So, to obtain a simple formalization, the natural choice is again ML.

Capabilities and why they are absent from this thesis. A dual notion to effects, so-called *capabilities*, has been introduced by Smith et al. (2000). Effects describe what changes during the evaluation of an expression. Capabilities describe what an expression *has the right* to change. In an effect system, an effect can always occur, and the typing relation *registers* this effect. In a system with capabilities, an expression *needs* the corresponding capabilities to execute an effectful statement. Instead of a typing relation of the form

$$\Gamma \vdash e : \tau, \varphi,$$

where the effect φ is a *result* of the derivation, we would have a relation

$$\Gamma, \varphi \vdash e : \tau,$$

where the capabilities φ are *granted* to e . In itself this modification of the point of view does not increase the expressiveness of the system. But capabilities, much more than effects, express the standpoint that a side effect needs a *resource*, which can be *consumed*. Therefore, since their introduction, capabilities have been used *linearly*, *i.e.*, they must be used once and exactly once. This means, for example, that the function `!`, that reads the contents of a reference, must not only require a capability on that reference, but must also *return* another one; otherwise, no one else could read from this reference again. On the one hand, such systems require a lot of bookkeeping. On the other hand, linear capabilities enable a number of improvements over more traditional effect systems. For example, deallocation can be expressed much more conveniently than in other systems: the function `free`, that deallocates a reference, requires a capability on that reference, but it does not return one. This reference now becomes inaccessible. Capabilities also enable *strong update*, *i.e.*, assignment that can change the type of a reference. As capabilities are linear, only a single portion of the code can possess each capability at each time. Thus, changing the type of a reference can not affect other parts of the system.

Systems with capabilities belong to the family of *substructural systems*, because of their usage of linearity. Another variant exists, where objects are used in an *affine* way,

1. Introduction

i.e., they can be used *at most once*. This discipline is a bit less strict than the linear one.

As has been explained, capabilities have first been introduced by Smith et al. (2000), and then improved by Walker et al. (2000). Fähndrich and DeLine (2002) presented a capability-based system with *group regions* (regions containing more than one reference) and *singleton regions* (regions containing a single reference), as well as two mechanisms, *adoption* and *focus* that permit to pass from one to the other. This improvement permits aliasing and precise tracking of references. Charguéraud and Pottier (2008) present a very precise translation of an ML-like language with side effects to a pure language without side effects, using capabilities with adoption and focus.

Capabilities are a useful tool, and it is legitimate to ask why the W language, introduced in Chapter 2, does not feature them. The answer is two-fold. First, there is a concern of simplicity. Capabilities *are* slightly more complex than effects, because of their linear nature. Adding this additional complexity to an already complex system, containing annotations and a wp calculus, is something that has to be thought about twice. Second, capabilities have two main strengths with respect to effects that are the upside of this added complexity: simpler reasoning about allocation and deallocation, and better tracking of aliasing. We argue that, in our particular setting, these two advantages are not worth the complexity.

ML-like languages are traditionally equipped with a garbage collector, so deallocation is never explicitly requested by the programmer, but taken care of automatically by the language. Allocation is also implicit in ML: a programmer never explicitly allocates memory.

Aliasing, however, can appear in ML programs. Being able to reason about aliasing is primarily useful when reasoning about mutable data structures with sharing. Our system is capable of reasoning about sharing, but capabilities are certainly superior in that aspect. However, we believe that this kind of programming is not representative for ML programs. We believe that sharing data structures can be encapsulated in particular modules, and be given an external specification that does not rely on sharing. The rest of the (sharing-free) code can then be reasoned about using techniques that do not need to be able to support sharing.

1.4.3. Related Work

Other techniques, and combinations of techniques, have been proposed to prove properties of effectful higher-order programs.

Techniques dealing with first-order programs. Systems for verification of first-order programs are well-established now and are increasingly applied even in industrial applications. Among the most prominent systems are the Why platform (Filliâtre and Marché, 2007) and the Spec# platform (Barnett et al., 2004b). Usually, these systems propose one or several input languages with annotations (C and Java in the case of the Why platform) and translate programs in this language to proof obligations in first-order logic, often via an intermediate language (for example the Why language). These proof obligations are then sent to automated or interactive provers. When all proof

obligations are proved, either manually or automatically, the program is considered to be correct with respect to its specification.

Compared to *Who*, the implementation of our weakest precondition calculus, these systems are not capable of reasoning about higher-order functions, the main reason being that this programming style is not an often-used idiom in these languages. However, they deal very well with the usual features of first-order programs, for example arrays, use SMT solvers to discharge proof obligations and strive for the best possible automation. The objective of *Who* is to keep the same degree of automation while adding support for effectful higher-order functions.

Techniques dealing with purely functional programs. Proof assistants can be used to implement and prove programs. One strong point of many proof assistants is that one can use the same language for programs and specifications, and sometimes even for proofs. The Coq proof assistant (The Coq Development Team, 2008) is such a system. It has been applied to prove many complex programs. The plain Coq system has been used to prove several implementations of compilers, from proof-of-concept (Chlipala, 2010) to realistic (Leroy, 2009). Sozeau (2007) has extended Coq with a syntax that simplifies the manipulation of dependent types and has applied his extension to the correctness proof of an implementation of finger trees. The appeal of that approach is that the only limit of expressiveness is the one of Coq. One can also do meta-reasoning about programs inside Coq and, at the end of the development, extract a certified implementation in OCaml or Haskell. The confidence in programs proved in this way is very high, because in the end one obtains a Coq proof term that corresponds to the correctness of the program. This term has been checked by Coq’s kernel. The drawback of this approach is that, without extensions such as Ynot (see the next paragraph), it is impossible to implement (and reason about) effectful computations. Another drawback of this approach is lack of automation. Indeed, in systems like Coq, all proofs are in principle done “by hand”, using a tactic language that permits to manipulate the goal to be proved and the context (lemmas to apply, equations to rewrite). A number of automated tactics exist, that deal for example with linear arithmetic, or propositional tautologies. However, these tactics usually combine badly, and the resulting sum is weaker than state-of-the-art automated provers.

The Pangolin system, the implementation of the theoretical system of Régis-Gianas and Pottier (2008), can also be used to reason about purely functional programs. Together with the *Why* tool, it is one of the starting points of our work, and our purely functional fragment is basically the Pangolin system, although we removed algebraic data types for clarity of presentation. In Pangolin, one can write functional higher-order programs and annotate them using higher-order logic. Pangolin and our system share the advantages and drawbacks in the sense that both are relatively simple systems from a theoretical point of view and provide good automation via automated provers to the user, but use slightly less expressive logics than systems with dependent types, and do not provide a machine-checkable proof term. Note that the name “Pangolin” (without “e”) stands for the tool we just described, while the name “Pangoline” (with “e”) stands for the tool that encodes formulas in higher-order logic to formulas in first-order logic, and which is part of the contributions of this thesis.

1. Introduction

Charguéraud (2010) presents a way to compute a *characteristic formula* from a purely functional function definition. This formula summarizes exactly the given function and can then be used, e.g., in Coq, to prove derived properties about the function. Charguéraud has implemented this system and has proved correct many purely functional algorithms. In particular, the size of the specifications and proofs is close to the size of the programs to be proved. However, it is not yet clear how this work can be extended to support side effects.

Systems for higher-order programs with side effects. Berger, Honda and Yoshida (Honda et al., 2005; Berger et al., 2007) present a logic for imperative higher-order programs. They present their calculus using rules in Hoare logic style, while we propose a **wp**-calculus. The main advantage of a **wp** calculus is that it can be easily implemented, while a system in Hoare logic style, possibly with many structural rules, can be difficult to implement. A **wp** calculus basically is a Hoare logic together with a *strategy*, that decides at which moment the rules should be applied. The presentation of a system in Hoare logic style can be slightly more intuitive compared to a **wp** calculus. On the one hand, the systems they propose are slightly lighter, because they do not include any effect analysis or similar mechanisms. On the other hand, we believe that the absence of such an analysis would render proofs in this system relatively cumbersome. In particular, one needs to describe in annotations what does *not* change; also, the absence of any mechanism similar to effect polymorphism seems to make it impossible to reason modularly about higher-order functions.

One particularity of the systems of Berger, Honda and Yoshida, that is at the source of the ability to reason about higher-order functions, is the Hoare triple

$$\{ P \} e :_m \{ Q \}$$

that is available in the logic directly. In this Hoare triple, the variable m is a binding variable and stands for the result of the evaluation of e . Another ingredient of their logic is the application operator \bullet : if f is an effectful function of type $\tau \rightarrow \tau'$, then, in the logic, the expression $f \bullet x$ represents the return value of the call to f using x as argument. This expression does not state anything about the pre- or postcondition of f , nor its effects. It should be noted that \bullet is useful only when the precondition of f holds.

The Hoare triple can be simulated in our logic, and is indeed part of the syntax of the input language of our implementation **Who**, as we explain in Section 5.1. The \bullet operator cannot be represented as-is, but the variant $f \bullet x = v$ where v is any formula, can be expressed using Hoare triples, in their system and ours:

$$f \bullet x = v \equiv \{ True \} f x :_r \{ r = v \}$$

Fixing the precondition to **True** is not a serious restriction, as the operator is only useful in this case anyway.

Berger, Honda and Yoshida have not presented an implementation of their Hoare logic, therefore it is not clear how such a system could be realized. From our point of view, there are two main difficulties to overcome. First, the system description by a set of Hoare logic rules is quite far from an algorithm; second, in the case where

aliasing is allowed, they generate proof obligations in a non-standard logic, and they do not explain how to obtain formulas in a standard logic to be able to discharge proof obligations using available interactive or automated provers.

The Ynot System (Nanevski et al., 2008; Chlipala et al., 2009) is maybe the most advanced proposal for the proof of higher-order programs with side effects. It is an extension to the Coq proof assistant, capable of reasoning about imperative higher-order programs, including effectful functions as arguments, using a monad in which effectful computations can take place. Ynot belongs to the proponents of *programming with dependent types*, whose thesis is that properties of objects, for example the preconditions on function arguments, should be declared directly in the types instead of being stated apart in the precondition. In practice, programming using Ynot means programming with dependent types, while being able to use side effects; programming and proving go hand in hand, partly using the same language.⁶ To deal with aliasing, Ynot proposes the use of separation logic formulas.

A central ingredient of Ynot, and the underlying theory called Hoare Type Theory (Nanevski et al., 2006), is the Hoare triple type of the form

$$\tau \rightarrow \{P\}\tau'\{Q\},$$

which describes functions of argument type τ and return type τ' , whose precondition is P and the postcondition is Q . This type is simply defined in Coq, so arbitrary Coq types can be used in place of τ and τ' , and arbitrary Coq formulas can be used in place of P and Q . In connection with a higher-order logic, this enables reasoning about higher-order functions. A mechanism similar to our effect polymorphism can be achieved by abstracting over the heap that is modified by a function. This enables reasoning about the effects of a function in argument.

An impressive collection of programs has been proved correct using Ynot, including a number of container libraries (Nanevski et al., 2008), web services (Wisnesky et al., 2009) and a database system (Malecha et al., 2010).

Ynot is inevitably tied to the Coq system. While Coq is a very flexible language, it is also relatively difficult to learn. In addition, to use Ynot, one has to master many different techniques: programming with dependent types, using the tactics language of Coq, and separation logic, any of which are considered to be difficult to master. Also, being tied to the Coq system means that all proofs have to be done in this system. While a set of custom tactics provides a limited form of automation (Chlipala et al., 2009), one sometimes would like to use another system, or even automated provers, to discharge an obligation. This is not possible using Ynot, and the fact that formulas are in separation logic hinders even more the use of external tools.

Maingaud et al. (2010), based on the Paf! proof assistant (Baro and Manoury, 2003; Baro, 2003), have developed a system to prove properties of effectful higher-order programs. The basic idea is to transform the semantic reduction rules of the considered programming language into reasoning rules of the system. This amounts to *symbolically evaluate* the program under consideration. Formulas of the form

$$[e].q$$

⁶In practice, one also needs to use the *tactic language* of Coq, which in most cases is more convenient to write proofs than the actual programming language of Coq.

1. Introduction

state that after program e is executed, q is true. This approach is very intuitive and simple, and can deal with aliasing. Similarly to the system of Berger, Honda and Yoshida, the absence of any effect analysis or similar mechanism will require to prove framing properties that delimit the effect of an expression. To our knowledge, an implementation for this system does not exist.

The work by Borgström et al. (2010), to our knowledge, is one of the few works with an implementation where a substructural type system is used for program verification of higher-order programs with effects. At the basis of their system is *Fine*, a purely functional language with dependent types and affine types. This language permits to express many properties relative to sharing and even mutation, thanks to the presence of affine types. They also present an effectful programming language *FX* and a translation from *FX* to *Fine*. They are capable of producing proof obligations from *Fine* programs and discharge them using SMT solvers. Comparing to our work, their system seems to be more convenient for programs with sharing thanks to affine types. On the other hand, they have not yet proved any higher-order programs, and only conjecture that they could.

Encodings of higher-order logic. Hurd (2003), to our knowledge, was one of the first to encode higher-order formulas in HOL (Gordon, 2000) into first-order logic with the goal to use automated provers to prove them. He does so by simply encoding λ -abstractions by the combinators S , K , I and C . The higher-order unary application is encoded by a binary application symbol $@$. He does not give any proof or formalization of his translation.

Meng and Paulson (2008) continue the work of Hurd in a similar setting, taking formulas from Isabelle (Nipkow et al., 2002) instead of HOL. They have compared several different encodings of higher-order features, in particular λ -abstractions, either using combinators, or λ -lifting. Again, they do not give any formalization of their translation. They have compared these different choices and have found that it makes little difference to choose one or the other, in terms of efficiency.

Both the work of Hurd and Meng/Paulson translate formulas in a typed higher-order logic to an untyped first-order logic. To be correct, such a translation must include information about the types of the terms. Meng and Paulson developed several different encodings of type information of varying verbosity.

Compared to the work of Meng and Paulson (2008) and Hurd (2003), we introduce two improvements. By giving definitions to the function symbols introduced by the encoding, we first obtain an ad-hoc justification of the equivalence of the original formula and its encoding, simply by evaluation. Second, the definitions can also lead to simplifications in the formulas. As a result, our encoding does not add any overhead to formulas that are *already* in first-order form.

2. The Specification Language

In this chapter, we define the language of our discussion. In Section 2.1, we present a programming language called W , which is very similar to ML, but contains a type and effect system with regions. In W , the type system allows us to know the effect of an expression precisely. Along with W , we define the specification language L in Section 2.2. This second language is designed to be used to specify W programs. We give examples throughout the chapter.

2.1. The Programming Language W

We start by introducing the syntax, semantics and type system of W , along with a type soundness proof.

2.1.1. Syntax

We now present the syntax of W . As types and effects are part of the syntax, we introduce them first.

Regions and effects. One central point of W are *regions* and *effects*. A region ρ is a set of mutable memory cells. Every reference in a W program belongs to a region. Regions come in two flavors: *concrete* regions or region *constants* r and region *variables* ϱ :

$$\begin{array}{ll} \varrho & \text{Region Variable} \\ r & \text{Region Constant} \\ \rho ::= r \mid \varrho \end{array}$$

The difference between a region constant and a region variable will become clear later. Region expressions are always atomic (being either a constant or a variable), there is no way to somehow combine regions.

An *effect* φ is simply a set of regions. In the syntax, such a set is written as a list, sometimes with curly braces: $\{\varrho, r_1\}$ is an effect expression, and ϱr_1 is the same effect expression, written more concisely. There are also effect *variables* ε that stand for an entire effect. Effects can be joined together to form larger effects. However, instead of the usual set notation to describe the union of two effects, such as $\{\varrho, r_1\} \cup \varepsilon$, we simply list effect variables inside the curly braces: $\{\varrho, r_1, \varepsilon\}$, or in the shorter form $\varrho r_1 \varepsilon$. We still use the notation $\varphi_1 \cup \varphi_2$ to describe the union of two arbitrary effects in the metatheory.

$$\varphi ::= (\rho \mid \varepsilon)^*$$

2. The Specification Language

Types. Just as ML, W is a typed programming language. Besides the usual base types such as `bool`, and `int`, and the type `unit` whose only value is the constant `void`, a type τ in W can also be a type $\text{ref}_\rho \tau$ of mutable references of type τ . As every reference must be in a certain region, this is also indicated in the type. The type $\tau_1 \rightarrow^\varphi \tau_2$ describes one-argument functions whose input type is τ_1 and whose output type is τ_2 . Additionally, these “arrow types” are annotated with the effect φ of the function. Functions with different effects have different types. There are also type constructors ι , which have a certain arity n associated to them. If ι is a type constructor of arity n , then $\iota(\tau_1, \dots, \tau_n)$ is a valid type in W . The base types `bool`, `unit` and `int` can be seen as nullary type constructors. Finally, just as there are region and effect variables, there are also *type variables* α .

$$\begin{aligned} \iota &::= \text{bool} \mid \text{unit} \mid \text{int} \mid \dots \\ \tau &::= \alpha \mid \tau \rightarrow^\varphi \tau \mid \iota \bar{\tau} \mid \text{ref}_\rho \tau \end{aligned}$$

Metavariables for generalization and instantiation. Types, regions and effects can contain variables. These variables can be instantiated again by types, regions and effects. This mechanism is common to all three constructs. Therefore, we use the metavariable χ to stand for any kind of type, region or effect variable, and the metavariable κ to stand for any kind of type, region or effect. When we describe substitutions, such as in the notation $\tau[\chi \mapsto \kappa]$, we assume χ and κ to be of compatible type. So the expression $\tau[\chi \mapsto \kappa]$ describes either a substitution of a region for a region variable, a substitution of an effect for an effect variable, or finally a substitution of a type for a type variable.

$$\begin{aligned} \chi &::= \alpha \mid \varrho \mid \varepsilon \\ \kappa &::= \tau \mid \rho \mid \varphi \end{aligned}$$

Values and expressions. We now can describe the programming language W itself. As the languages of the ML family, it does not make a distinction between statements and expressions; there are only the latter. Every programming language needs *constants* c :

$$c ::= \text{void} \mid n \mid \text{true} \mid \text{false} \mid \text{ref} \mid := \mid ! \mid :=_{r,l,\tau} \dots$$

In W we have the constant `void`, the only value of the type `unit`, as well as integer constants n and the boolean constants `true` and `false`. More interestingly, the three functions `ref`, `!` and `:=` for reference creation, reading and writing are also constants. The *family* of constants $:=_{r,l,\tau}$ is a technical necessity and describes a partial application of `:=` to a location l in region r of type τ . Right now we cannot go into more detail, but the meaning of this family of constants will become clear when we explain the semantics of W .

While W does not distinguish commands and expressions, it *does* distinguish *values* v from other *expressions* e . A value is an expression that cannot be evaluated any more.

Constants c and program variables x are of course values. Constants and variables can be instantiated with types, regions and effects:

$$v ::= c [\bar{\kappa}] \mid x [\bar{\kappa}] \mid \dots$$

Functions are also values and can be built using the syntax

$$v ::= \dots \mid \text{rec } f (x : \tau). e \mid \dots$$

This creates a recursive function with one argument x of type τ and function body e . In W , all functions are recursive, and they may use the first variable name in the `rec` binding for recursive calls. Of course, functions that do not contain recursive calls are accepted as well. The name f is *not* available outside of the function body. We will see later how to give a name to such a function to be able to use it with the usual function call syntax. Functions are values: as long as they are not applied, they cannot be evaluated any more. The last kind of values are *memory locations* l , whose meaning we will explain in the Section 2.1.2.

$$l \quad \text{Location}$$

$$v ::= c [\bar{\kappa}] \mid x [\bar{\kappa}] \mid l \mid \text{rec } f (x : \tau). e$$

Values can be used in several ways to form *expressions*, that are the central syntactic notion of W . Every value can be an expression as well. An application

$$v_1 v_2$$

of a function value v_1 to another value v_2 is an expression. The branching construct

$$\text{if } v \text{ then } e \text{ else } e$$

is an expression. Using an expression of the form

$$\text{let } x [\bar{\chi}] = e_1 \text{ in } e_2$$

users can give the name x to the *result* of the expression e_1 and use the name x in e_2 . The type, effect and region variables $\bar{\chi}$ can be used inside e_1 . The construct

$$\text{letregion } \varrho \text{ in } e$$

permits to create a new (empty) region. The meaning of the `region` construct

$$\text{region } r \text{ in } e$$

will be explained in Section 2.1.2 about semantics.

$$e ::= v \mid v v \mid \text{let } x [\bar{\chi}] = e \text{ in } e \mid \text{if } v \text{ then } e \text{ else } e \mid \text{letregion } \varrho \text{ in } e \mid \text{region } r \text{ in } e$$

2. The Specification Language

ϱ	Region Variable
r	Region Constant
ε	Effect Variable
$\rho ::= r \mid \varrho$	
$\varphi ::= (\rho \mid \varepsilon)^*$	
$\iota ::= \text{bool} \mid \text{unit} \mid \text{int}$	
$\tau ::= \alpha \mid \tau \rightarrow^\varphi \tau \mid \iota \bar{\tau} \mid \text{ref}_\rho \tau$	
$\chi ::= \alpha \mid \varrho \mid \varepsilon$	
$\kappa ::= \tau \mid \rho \mid \varphi$	
l	Location
$c ::= \text{void} \mid n \mid \text{true} \mid \text{ref} \mid := \mid ! \mid :=_{r,l,\tau}$	
$v ::= c \ [\bar{\kappa}] \mid x \ [\bar{\kappa}] \mid l \mid \text{rec } f \ (x : \tau). \ e$	
$e ::= v \mid v \ v \mid \text{let } x \ [\bar{\chi}] = e \text{ in } e \mid \text{if } v \ \text{then } e \ \text{else } e \mid \text{letregion } \varrho \ \text{in } e \mid \text{region } r \ \text{in } e$	
$\Gamma ::= \emptyset \mid \Gamma, \chi \mid \Gamma, x : \forall \bar{\chi}. \tau$	

Figure 2.1: The syntax of language W.

A-normal form and syntactic sugar. The syntax of W is summarized in Fig. 2.1. The reader may be surprised that one cannot write

$$e_1 \ e_2$$

for an application of a function returned by the expression e_1 to the expression e_2 , or why the boolean condition in an if-branch can only be a value instead of any expression e . The reason is that for the sake of simplicity of presentation of the later sections, W is presented in the so-called *A-normal form* (Flanagan et al., 1993), in which every intermediate result of computation must be named using `let`. The application $e_1 \ e_2$ must be rewritten as follows:

```
let f = e1 in
let x = e2 in
f x
```

This seems very restrictive at first, as even nested function calls such as $f \ (g \ x)$ must be rewritten into `let z = g x in f z`. However, as is shown in the paper by Flanagan et al., this is not a serious restriction, because every program in a language with full expressions can be translated to A-normal form. For this reason, we will allow ourselves to use arbitrarily nested expressions in examples.

Additionally, we introduce syntactic sugar for common constructs. First, we accept an underscore `_` at binding positions instead of a variable name, when we do not want to give a name to this variable. This concerns `let`-bindings and recursive functions. We use the pair of parentheses `()` in two ways: first as a synonym for the value `void` of type

unit, and second, similarly to the underscore, as a way to indicate a function argument of unit type for which we do not want to invent a name.

The semicolon ; indicates chaining of expressions while ignoring the return value of the first one, and we propose syntactic sugar for function definitions:

$$\begin{aligned} e_1; e_2 &:= \text{let } _ = e_1 \text{ in } e_2 \\ \text{let } f [\bar{x}] (x : \tau) = e_1 \text{ in } e_2 &:= \text{let } f [\bar{x}] = \text{rec } _ (x : \tau). e_1 \text{ in } e_2 \\ \text{let rec } f [\bar{x}] (x : \tau) = e_1 \text{ in } e_2 &:= \text{let } f [\bar{x}] = \text{rec } f (x : \tau). e_1 \text{ in } e_2 \end{aligned}$$

In addition, we introduce syntactic sugar for the usual for loop:

for $i = e_1$ to e_2 do e_3 done

stands for the following program:

```
let a = e1 in
let b = e2 in
let f i = e3 in
let rec aux k =
  if k > b then void else (f k; aux (k + 1))
in
aux a
```

Occasionally, we will omit type annotations for function arguments, when the context allows it; for example, in the above expansion of the for loop, the variables i and k are clearly of integer type.

While we have not introduced function constants for operations on integers such as addition and multiplication, we still use such functions in our examples. For example, we use the well-known functions $+$, $-$, \times and \div , which are of type $\text{int} \rightarrow^\emptyset \text{int} \rightarrow^\emptyset \text{int}$. The usual comparison functions $<$, \leq , \geq , $>$ are also occasionally used, and are of type $\text{int} \rightarrow^\emptyset \text{int} \rightarrow^\emptyset \text{bool}$. We allow infix notation for all these functions.

Another issue is the one of equality tests; most programming languages supply a binary function ($=$) to test for equality between two values. Formally, we do not introduce a generic equality test, that works for all types. Instead, we use specific equality tests such as

$$=_{\text{int}}: \text{int} \rightarrow^\emptyset \text{int} \rightarrow^\emptyset \text{bool}.$$

for the equality test between integers. In example programs, we still use the infix symbol $=$ for clarity.

Top-level bindings. In most programming languages, a program is a list of top-level declarations and definitions. However, we only have presented expressions in W. It is easy to remedy to this situation. Take the three introduction forms `let`, `letregion` and `region` and define three top-level forms without the `in` part. Now a list of such top-level forms, for example

```
letregion  $\varrho$ 
let  $f (x : \tau) = e$ 
```

is transformed into a single expression using the corresponding expression forms *with*

2. The Specification Language

the in part, terminating with, for example, the `void` expression. In our example, we obtain:

```
letregion  $\varrho$  in
let  $f(x : \tau) = e$  in
void
```

We use top-level bindings in our examples.

We observe that the main differences between `W` and `ML` are explicit type, effect and region generalization and instantiation, the two constructs `letregion` and `region` and finally the presentation of `W` in A-normal form.

Examples. Let us give a few example programs to see what programs in `W` look like. We start with a simple factorial function:

```
let rec fact ( $n : \text{int}$ ) = if  $n = 1$  then 1 else  $n \times f(n - 1)$ 
```

and its imperative counterpart, which uses a for loop instead of a recursive function:

```
let factwhile ( $n : \text{int}$ ) =
  letregion  $\varrho$  in
  let  $x = \text{ref} [\text{int}, \varrho] 1$  in
  for  $i = 1$  to  $n$  do
     $x := [\text{int}, \varrho] ![\text{int}, \varrho] x \times i$ 
  done;
  ! $x$ 
```

Note that the type and region polymorphic functions `ref`, `:=` and `!` have to be given the corresponding type and region instantiations. We will often omit these instantiations when the context permits it; in the case of `:=` and `!`, both the type instantiation and the region instantiation can be easily derived. We will always give the region instantiation for `ref`, however.

Maybe the simplest higher-order function is the one that takes a function argument and executes it:

```
let exec [ $\varepsilon$ ] ( $f : \text{unit} \rightarrow^\varepsilon \text{unit}$ ) =  $f ()$ 
```

The function definition itself is not very interesting; we will show how this function can be typed in Section 2.1.3.

Yet another interesting function is the so-called Landin's Knot (Landin, 1966), which achieves recursion without using recursive functions. It does so by stocking the function in a reference; recursive calls are replaced by calls to the function stored in the reference. Here is the factorial function, implemented using this method:

```
letregion  $\varrho$ 
let circfact =
  let  $id\ n = n$  in
  let  $x = \text{ref} [\varrho] id$  in
  let  $f\ n = \text{if } n = 0 \text{ then } 1 \text{ else } n \times (!x)\ (n - 1)$  in
   $x := f$ ;
  ! $x$ 
```

The *circfact* function returns the contents of the reference x , *i.e.*, a function that computes the factorial of its argument.

2.1.2. Semantics

We now move on to the semantics of the language. The semantics expresses what each language construct actually does, and under what conditions a program executes normally or gets stuck.

One can imagine using Hoare logic to define the semantics of a programming language; there have been attempts to do just that (Hoare, 1974). But in our setting, this would have several drawbacks. First, such rules would possibly be relatively complex and not particularly illuminating. Second, more importantly, as we want to prove that the Hoare logic we are going to define is correct, we need some other definition of the semantics of the language. We therefore settle for the now-standard approach of Wright and Felleisen (1994), called *small step semantics*. We define the semantics of W by a syntactic relation \rightarrow that describes under what condition an expression can reduce to another. The intention is that this relation is reasonably close to what actually happens on a computer, and reasonably easy to formulate and to understand. The rules of Hoare logic will be proved correct with respect to this semantics.

We do not define \rightarrow directly; instead, we first define a reduction relation \rightarrow describes a reduction step at the top of an expression. Then we define a relation \longrightarrow , that describes a single such step at certain positions in an expression. The desired relation \rightarrow is then simply the *transitive closure* of \longrightarrow : it describes a sequence of zero or more steps of the relation \longrightarrow .

We now define more formally a few notions that are necessary to express the semantics.

Definition 2.1. A *dynamic region* R is a finite mapping from locations l to values v . We write $dom(R)$ to denote its domain. We write $R[l \mapsto v]$ to add the binding $l \mapsto v$ to R , or update an existing binding of l in R . We write \emptyset for the empty region.

Region R

Definition 2.2. A *store* s is a finite mapping from region names r to dynamic regions R . We write $dom(s)$ to denote its domain. We write $s[r \mapsto R]$ to add the binding $r \mapsto R$ to s , or update an existing binding of r in s .

Store s

Definition 2.3. We assume the existence of a function δ with the signature

 δ

$$Store \times Const \times Value \rightarrow Store \times Value,$$

which knows how to interpret constant symbols when they are applied to an argument. This function may be undefined, in particular for ill-typed applications. Throughout our development, we assume a number of hypotheses about δ . We will give a partial definition of δ . We also assume that δ verifies a number of properties, and we will prove these properties for our partial definition.

To be able to formulate the semantics of W, we need a notion of substitution of values for variables. However, in W, values can be *polymorphic*, and variables can carry instantiations. The consequence is that the notion of substitution is slightly more

2. The Specification Language

$s, e \rightarrow s', e'$	
(β)	$s, (\text{rec } f (x : \tau'). e) v \rightarrow s, e[x \mapsto v, f \mapsto \text{rec } \dots]$
(let)	$s, \text{let } x [\bar{\chi}] = v \text{ in } e \rightarrow s, e[x \mapsto \Lambda \bar{\chi}.v]$
(δ)	$s, c [\bar{\kappa}] v \rightarrow \delta(s, c[\bar{\kappa}], v)$
(iftrue)	$s, \text{if true then } e_1 \text{ else } e_2 \rightarrow s, e_1$
(iffalse)	$s, \text{if false then } e_1 \text{ else } e_2 \rightarrow s, e_2$
(letreg)	$s, \text{letregion } \varrho \text{ in } e \rightarrow s[r \mapsto \emptyset], \text{region } r \text{ in } e[\varrho \mapsto r] \quad r \notin \text{dom}(s)$
(region)	$s, \text{region } r \text{ in } v \rightarrow s, v$

Figure 2.2: The reduction relation of W.

involved than in the standard literature. We therefore define a *polymorphic substitution of values*, written $[x \mapsto \Lambda \bar{\chi}.v]$. Its informal meaning is the following: replace every occurrence of x , possibly applied to some instantiations $\bar{\kappa}$, by v , in which the parameters $\bar{\chi}$ have been replaced by κ .

$[x \mapsto \Lambda \bar{\chi}.v]$

Definition 2.4. We define the polymorphic substitution as descending in every sub-term, but on variables, it is defined as follows:

$$\begin{aligned} (x [\bar{\kappa}]) [x \mapsto \Lambda \bar{\chi}.v] &= v[\bar{\chi} \mapsto \bar{\kappa}] \\ (y [\bar{\kappa}]) [x \mapsto \Lambda \bar{\chi}.v] &= y [\bar{\kappa}] \quad x \neq y \end{aligned}$$

If the list $\bar{\chi}$ is empty, we simply write $[x \mapsto v]$. For the simultaneous substitution of multiple values for multiple variables, we write $[x \mapsto \Lambda \bar{\chi}.v, y \mapsto \Lambda \bar{\chi}'.v']$.

Now we have provided everything to define the reduction relation of our language. It is summarized in Fig. 2.2. A state of our semantics is described by a pair s, e , where s is a store and e is an expression. The pair s, e is also called a *configuration*. The relation $s, e \rightarrow s', e'$ describes how e may reduce to e' , and provoke modifications in s to obtain s' , *at the top-level* of the expression, *i.e.*, the relation inspects only the top-most structure of e . An application of a recursive function to an argument v is called a *β -reduction*: we replace every occurrence of x in the body by the argument, and every recursive occurrence f by the anonymous function. The reduction of a *let*-expression is similar, but the value v is potentially polymorphic, so we need to use polymorphic substitution. When we encounter a constant applied to a value, we use the function δ to evaluate this expression and return the potentially modified state. Conditional expressions are evaluated in the usual way. Finally, a *letregion*-construct creates a new region with a fresh name in the store and reduces to a *region*-construct with the same body. A *region*-expression can only reduce if its body is a value, in which case it simply returns the value.

The reduction semantics makes use of the function δ , which interprets constants.

δ

Definition 2.5. We assume δ to contain at least the following mappings:

$$\begin{aligned} \delta(s, ! [\tau r], l) &= s, s(r)(l) && r \in \text{dom}(s), l \in \text{dom}(s(r)) \\ \delta(s, := [\tau r], l) &= s, :=_{r,l,\tau} \\ \delta(s, :=_{r,l,\tau}, v) &= s[r \mapsto s(r)[l \mapsto v]], \text{void} && r \in \text{dom}(s), l \in \text{dom}(s(r)) \\ \delta(s, \text{ref}[\tau r], v) &= s[r \mapsto s(r)[l \mapsto v]], l && r \in \text{dom}(s), l \notin \text{dom}(s(r)) \end{aligned}$$

So the function $!$, with a certain type and region instantiation, applied to a memory location, looks up the region in the store, and the memory location in the region, and returns the contained value. The function ref creates a new memory location in an existing region. The intention of the function constant $:=$ is that $:= [\tau r] l v$ overwrites the current value of l in region r by v . However, δ needs two steps to achieve this, passing by the constant $:=_{r,l,t}$. We now see the meaning of this constant; it represents the partial application of $:=$ to a memory location, and its instantiation. This formality is necessary because we only allow a single argument to constants.

The relation \rightarrow is undefined for tuples s, e for which none of the reduction rules applies. This includes application of constants c to arguments for which δ is not defined, the attempt to apply non-functional values, or an if-branch whose test is neither **true** nor **false**.

To be able to describe reduction deeply inside a term, we use *evaluation contexts*:

$$E ::= \square \mid \text{let } x [\bar{x}] = E \text{ in } e \mid \text{region } r \text{ in } E$$

An evaluation context describes where a reduction can take place *inside* a term. Hence, an evaluation context can be seen as a term with a hole. However, only a very limited choice is available for building evaluation contexts. Such a context can either be empty, noted \square , which means that the reduction takes place at top-level. It can also be a let-expression with the hole on the left side. Finally, an evaluation context can be a region-expression.

The actual one-step reduction \rightarrow is now simply the closure w.r.t. contexts of the top-level one-step reduction:

$$s, e \rightarrow s', e'$$

$$\text{CONTEXT} \frac{s, e \rightarrow s', e'}{s, E[e] \rightarrow s', E[e']}$$

It describes how a term can be reduced to another using a single rewriting step of the relation \rightarrow , but only at certain points of an expression.

Finally, we define the reduction relation \twoheadrightarrow as the reflexive and transitive closure of the relation \rightarrow :

$$s, e \twoheadrightarrow s', e'$$

$$\text{REFL} \frac{}{s, e \twoheadrightarrow s, e} \quad \text{STEP} \frac{s, e \twoheadrightarrow s', e' \quad s', e' \rightarrow s'', e''}{s, e \twoheadrightarrow s'', e''}$$

Stated otherwise, if $s, e \twoheadrightarrow s', e'$, then s, e reduces in a finite number of steps of the relation \rightarrow , possibly no step at all, to s', e' .

A special situation occurs when for a given configuration s, e , there is no s', e' such that $s, e \rightarrow s', e'$. We distinguish two cases. Either e is actually a value v (no rule of \rightarrow or \twoheadrightarrow applies to values); we then say that the evaluation has terminated and v is the *result* of the evaluation. On the other hand, if e is not a value, we say that e is *stuck*. Being stuck is the way our semantics models undesirable errors such as typing errors (applying, for example, the addition function to boolean values or trying to apply an object which is not a function, say an integer, to some arguments) and memory errors (accessing a memory location that does not exist).

2. The Specification Language

Surface language. A few elements of the syntax are not actually part of the language, at least not the language a programmer might use. Users cannot refer to memory locations in their programs, they cannot use region constants, and they cannot use the `region` construct. These constructs are only there to simplify the reasoning about the semantics of the language. This will become clearer in Section 2.1.3 concerning typing and Section 2.1.4 containing the type soundness proof.

So far we have seen four different letters for the concept of regions: R, r, ϱ and ρ . Conceptually, they are all the same thing, but technically we have to distinguish between: the actual mappings R from memory locations to values; the names r for regions, that can appear in a running program; and the region variables ϱ which stand for a region name r and can appear in programs written by a user or in running programs, but are always bound by a `letregion` or `let` construct. Finally, the metavariable ρ can be used to designate one of ϱ and r . Apart from Section 2.1.2, we mostly use the metavariable ρ to deal with the most general case. A user of W only ever sees region variables ϱ .

The relation between memory locations l and program variables is somehow similar; one the one hand, the surface language contains only variables x , but in a running program, memory locations l may also appear.

As a bibliographic note, the `letregion` construct has appeared, among others, in the work of Tofte and Talpin (1997). We learned about the `region` construct in the paper of Calcagno et al. (2002).

Examples. Fig. 2.3 presents an example of an evaluation, namely the application of the `factwhile` function to the argument 2. The example shows the effects of the `letregion` construct as well as the functions that manipulate references. The symbol \emptyset represents either the empty store (in the first two steps) or an empty region (in the third configuration).

A few properties. Before we go on, we prove a few technical properties about the reduction relation. As we have stated, the `region` construct is not part of the surface language, but it can be created at certain places during the evaluation of an expression. Some of the type soundness theorems (Section 2.1.4) are not correct if we do not restrict the occurrences of `region` to the ones that can actually occur in a reduction. We introduce the notion of purity.

pure

Definition 2.6. Let a value v be *pure* if v does not contain the `region` keyword at all. We call an expression e pure if it only contains pure values. A store s is pure if all stored values are pure. For example, the expression `region r in 5` is pure, even though it contains `region`, because every value in this expression is pure.

We now assume the function δ to preserve purity of expressions. This is an assumption; we cannot prove it in general because we did not define δ . However, we need to prove that our assumption is coherent with Definition 2.5, in which we fixed some of the mappings present in δ . Therefore, our hypothesis is followed by a partial “proof”, that only considers these fixed mappings. We use this scheme for all hypotheses concerning δ .

$\emptyset, \text{ factwhile } 2 \longrightarrow \emptyset, \text{ letregion } \varrho \text{ in}$ $\text{let } x = \text{ref } [\text{int}, \varrho] \text{ 1 in}$ $\text{for } i = 1 \text{ to } 2 \text{ do}$ $x := !x \times i$ done; $!x$ $\longrightarrow [r \mapsto [l \mapsto 1]],$	$\text{region } r \text{ in}$ $\text{let } x = \text{ref } [\text{int}, r] \text{ 1 in}$ $\text{for } i = 1 \text{ to } 2 \text{ do}$ $x := !x \times i$ done; $!x$ $\longrightarrow [r \mapsto \emptyset],$
$\text{region } r \text{ in}$ $\text{let } x = l \text{ in}$ $\text{for } i = 1 \text{ to } 2 \text{ do}$ $x := !x \times i$ done; $!x$ $\longrightarrow [r \mapsto [l \mapsto 1]],$	$\text{region } r \text{ in}$ $\text{let } f \ i = l := !l \times i \text{ in}$ $\text{let rec aux } k =$ $\text{if } k > 2 \text{ then void}$ $\text{else } (f \ k; \text{aux } (k + 1))$ in $\text{aux } 1; !l$ $\longrightarrow [r \mapsto [l \mapsto 1]],$
$\text{region } r \text{ in}$ $\text{aux } 1; !l$ $\longrightarrow [r \mapsto [l \mapsto 1]],$	$\text{region } r \text{ in}$ $\text{if } 1 > 2 \text{ then void}$ $\text{else } (f \ k; \text{aux } (k + 1));$ $!l$ $\longrightarrow [r \mapsto [l \mapsto 1]],$
$\text{region } r \text{ in}$ $l := !l \times 1; \text{aux } 2; !l$ $\longrightarrow [r \mapsto [l \mapsto 1]],$	$\text{region } r \text{ in}$ $\text{aux } 2; !l$ $\longrightarrow [r \mapsto [l \mapsto 1]],$
$\text{region } r \text{ in}$ $l := !l \times 2; \text{aux } 3; !l$ $\longrightarrow [r \mapsto [l \mapsto 1]],$	$\text{region } r \text{ in}$ $\text{if } 3 > 2 \text{ then void}$ $\text{else } (f \ k; \text{aux } (k + 1));$ $!l$ $\longrightarrow [r \mapsto [l \mapsto 2]],$
$\text{region } r \text{ in } !l$ $\text{region } r \text{ in } !l$ $\longrightarrow [r \mapsto [l \mapsto 2]],$	$\text{region } r \text{ in } 2$ $\longrightarrow [r \mapsto [l \mapsto 2]],$

Figure 2.3: An example of an evaluation. All region instantiations concern the region r created in the second step. Variables in **bold**, contrary to program variables, represent anonymous functions previously introduced using **let**. For example, **f** represents the closure $\text{rec } _ (i : \text{int}). l := !l \times i$. In the step from configuration four to configuration five, we have desugared the **for** loop and applied the reduction rule *let* three times.

2. The Specification Language

Hypothesis 2.7. *If s is pure, and the value v is pure, and*

$$s', v' = \delta(s, c[\overline{\kappa}], v),$$

then s' and v' are pure.

Proof for the constants of Definition 2.5. This hypothesis can easily be checked for the mappings of Definition 2.5. \square

Lemma 2.8. *If e and s are pure, and if for s' and e' one of the conditions*

1. $s, e \rightarrow s', e'$
2. $s, e \longrightarrow s', e'$
3. $s, e \twoheadrightarrow s', e'$

is true, then s' and e' are pure as well.

Proof. 1. For the relation \rightarrow , we simply check for each rule that purity is preserved. Hypothesis 2.7 guarantees this for the rule (δ) . The rules `iftrue`, `iffalse` and `region` do not change the store, and the resulting expression is a subexpression of the initial one, so `region` is not produced. The rules (β) and `let` substitute values for variables, but all values have been present before and cannot contain `region`. Finally, the rule `letregion` produces a `region` construct, but it is not contained in a value.

2. For the relation \longrightarrow , we proceed by induction on the form of the reduction context E . If E is empty, the claim follows from the one about \rightarrow . In the other two cases, it is clear that any subexpression of the result of the reduction has either been present before (and thus is pure) or has been obtained by reduction using \rightarrow , and is pure as well, as we already have proved the claim for \rightarrow . The resulting state s' is pure because it necessarily results from a reduction with \rightarrow .

3. The claim for \twoheadrightarrow is trivial; just proceed by induction over the length of the chain of reductions using \longrightarrow . \square

2.1.3. Typing

One of the great advantages of ML programs compared to other languages such as C or C++ is that ML programs can never get stuck. How is this possible, given that we have seen that there exist expressions that can get stuck in our language? The answer is well-typedness: not every well-formed expression is considered a valid program; in addition to the syntactic restrictions, a *typing relation* ensures that no typing errors occur and that all memory locations have been used correctly. In this section, we present our typing relation, which also rules out wrong behavior with respect to regions.

An expression or a value must always be well-typed with respect to an *environment* Γ , which basically gives types to all variables in the expression. In our case, programs do not only contain variables, but also memory locations. Memory locations should

be of type $\text{ref}_r \tau$, but the typing environment gives no information which region r and which type τ is appropriate. As a consequence, we need an additional environment that associates to each memory location a type and a region.

Definition 2.9. A store typing Σ is a mapping from locations l to a pair of a region name r and a type τ : $Location \rightarrow Region \times Type$.

Store Typing Σ

Finally, our programs also contain constants. A predefined function $Typeof()$ gives the type schemes of all constants; we only partially define this function here for the constants we are interested in:

 $Typeof(c)$

$$\begin{aligned}
Typeof(ref) &:= \forall \alpha \rho. \alpha \rightarrow^\rho \text{ref}_\rho \alpha \\
Typeof(:=) &:= \forall \alpha \rho. \text{ref}_\rho \alpha \rightarrow^\emptyset \alpha \rightarrow^\rho \text{unit} \\
Typeof(:=_{r,l,\tau}) &:= \tau \rightarrow^r \text{unit} \\
Typeof(!) &:= \forall \alpha \rho. \text{ref}_\rho \alpha \rightarrow^\rho \alpha \\
Typeof(void) &:= \text{unit} \\
Typeof(n) &:= \text{int} \\
Typeof(true) &:= \text{bool} \\
Typeof(false) &:= \text{bool}
\end{aligned}$$

For example, the function $:=$ takes as an argument a memory location in any region ρ , of any type α , a value of type α and, according to the semantics defined earlier, updates the memory location with this value. The word “any” in the previous sentence is translated by the generalization of the type and the region of the reference. Of course, since the region which contains the location is affected by this operation, the region appears in the effect of $:=$ on the second arrow. Similar considerations are valid for the other functions manipulating references. Notice that we do not distinguish between reading and writing a region, so $!$ also affects the region of its location argument. The let construct permits to define polymorphic functions in W itself. Such polymorphic constants and variables have to be used with an *instantiation*, which specializes its type. This is the purpose of the syntax $c [\bar{\kappa}]$ and $x [\bar{\kappa}]$ for variables. Of course, for monomorphic variables and constants, no instantiation needs to be given.

We now define the typing relations for values and programs; both are mutually recursive. The typing relation for values is of the form $\Gamma; \Sigma \vdash_v v : \tau$, which means that in the environment Γ , given the store typing Σ , the value v has type τ . The typing relation for expressions is of the form $\Gamma; \Sigma \vdash e : \tau, \varphi$ and states that under Γ and Σ , e has type τ and effect φ . The relation for values is defined in Fig. 2.4. Since we are working with store typings, the case for memory locations is easy: l is well-typed of type $\text{ref}_r \tau$ if Σ contains a mapping $l \mapsto r, \tau$. Variables with an instantiation $\bar{\kappa}$ are well-typed if they are contained in the environment and if their instantiation corresponds to their type scheme. Constants are typed in a similar way. Recursive functions are of function type $\tau' \rightarrow^\varphi \tau$ when, assuming that their argument is of type τ' and assuming that the recursive call is already of type $\tau' \rightarrow^\varphi \tau$, the body can be typed to be of type τ and effect φ .

The typing relation for expressions is defined in Fig. 2.5. Every well-typed value is a well-typed expression with no effect (rule VALUE). The effect of any expression can be

2. The Specification Language

$$\boxed{\Gamma; \Sigma \vdash_v v : \tau}$$

$$\begin{array}{c} \text{PVAR} \frac{\Gamma(x) = \forall \bar{\chi}. \tau}{\Gamma; \Sigma \vdash_v x [\bar{\kappa}] : \tau[\bar{\chi} \mapsto \bar{\kappa}]} \qquad \text{PCONST} \frac{\text{Typeof}(c) = \forall \bar{\chi}. \tau}{\Gamma; \Sigma \vdash_v c [\bar{\kappa}] : \tau[\bar{\chi} \mapsto \bar{\kappa}]} \\ \\ \text{REC} \frac{\Gamma, f : \tau' \rightarrow^\varphi \tau, x : \tau'; \Sigma \vdash e : \tau, \varphi}{\Gamma; \Sigma \vdash_v \text{rec } f(x : \tau'). e : \tau' \rightarrow^\varphi \tau} \qquad \text{LOC} \frac{\Sigma(l) = r, \tau}{\Gamma; \Sigma \vdash_v l : \text{ref}_r \tau} \end{array}$$

Figure 2.4: The typing rules for program values.

$$\boxed{\Gamma; \Sigma \vdash e : \tau, \varphi}$$

$$\begin{array}{c} \text{VALUE} \frac{\Gamma; \Sigma \vdash_v v : \tau}{\Gamma; \Sigma \vdash v : \tau, \emptyset} \qquad \text{APP} \frac{\Gamma; \Sigma \vdash_v v : \tau' \rightarrow^\varphi \tau \quad \Gamma; \Sigma \vdash_v v' : \tau'}{\Gamma; \Sigma \vdash v v' : \tau, \varphi} \\ \\ \text{LETPOLY} \frac{\Gamma, \bar{\chi}; \Sigma \vdash_v v : \tau' \quad \Gamma, x : \forall \bar{\chi}. \tau'; \Sigma \vdash e_2 : \tau, \varphi}{\Gamma; \Sigma \vdash \text{let } x [\bar{\chi}] = v \text{ in } e_2 : \tau, \varphi} \\ \\ \text{LET} \frac{\Gamma; \Sigma \vdash e_1 : \tau', \varphi_1 \quad \Gamma, x : \tau'; \Sigma \vdash e_2 : \tau, \varphi_2}{\Gamma; \Sigma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau, \varphi_1 \cup \varphi_2} \\ \\ \text{IF} \frac{\Gamma; \Sigma \vdash_v v : \text{bool} \quad \Gamma; \Sigma \vdash e_1 : \tau, \varphi \quad \Gamma; \Sigma \vdash e_2 : \tau, \varphi}{\Gamma; \Sigma \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 : \tau, \varphi} \\ \\ \text{LETREG} \frac{\Gamma, \varrho; \Sigma \vdash e : \tau, \varphi \quad \varrho \notin \tau}{\Gamma; \Sigma \vdash \text{letregion } \varrho \text{ in } e : \tau, \varphi \setminus \varrho} \qquad \text{REGION} \frac{\Gamma; \Sigma \vdash e : \tau, \varphi \quad r \notin \tau}{\Gamma; \Sigma \vdash \text{region } r \text{ in } e : \tau, \varphi \setminus r} \\ \\ \text{SUB} \frac{\Gamma; \Sigma \vdash e : \tau, \varphi \quad \varphi \subseteq \varphi'}{\Gamma; \Sigma \vdash e : \tau, \varphi'} \end{array}$$

Figure 2.5: The typing rules for program expressions.

increased using the rule SUB. The `letregion` construct introduces a new region variable in the typing environment, and removes it from the effect of the overall expression, given that the region does not occur in the type, which is expressed by $\varrho \notin \tau$. The `region` construct is typed similarly, with the difference that the region constant r is not introduced into the context. If a value v is of function type and if v' is of the same type as the argument of the function, then $v v'$ is of the return type of the function. The overall effect of this expression is the latent effect of the function (rule APP). If-then-else expressions are well-typed if the condition is indeed of type `bool` and if the two branches are of the same type and same effect. The overall effect is the union of the effects of the two branches. A `let`-expression introduces a variable-binding into the environment; this binding can only be polymorphic when the inner term e_1 is actually a value v (rule LETPOLY); in this case, the effect of the `let`-expression is the effect of the outer expression e_2 . Otherwise, if e_1 is not a value, the introduced variable binding must be monomorphic (rule LET).

Examples. Recall the following simple function (see page 44) that takes a function argument f and applies it to the `unit` value:

```
let exec [ε] (f : unit →ε unit) = f () in ...
```

We show its typing derivation in our system, where we have assumed that the expression following the `in`, represented by the three dots, is of type τ and effect φ :

$$\text{LETPOLY} \frac{\text{REC} \frac{\text{APP} \frac{\Gamma, \varepsilon, f : \text{unit} \rightarrow^{\varepsilon} \text{unit}; \Sigma \vdash_v f : \text{unit} \rightarrow^{\varepsilon} \text{unit}}{\Gamma, \varepsilon, f : \text{unit} \rightarrow^{\varepsilon} \text{unit}; \Sigma \vdash f () : \text{unit}, \varepsilon}}{\Gamma, \varepsilon; \Sigma \vdash_v \text{rec } _ (f : \text{unit} \rightarrow^{\varepsilon} \text{unit}). f () : (\text{unit} \rightarrow^{\varepsilon} \text{unit}) \rightarrow^{\varepsilon} \text{unit}}}{\Gamma, \text{exec} : \forall \varepsilon. (\text{unit} \rightarrow^{\varepsilon} \text{unit}) \rightarrow^{\varepsilon} \text{unit}; \Sigma \vdash \dots : \tau, \varphi}}{\Gamma; \Sigma \vdash \text{let exec } [\varepsilon] = \text{rec } _ (f : \text{unit} \rightarrow^{\varepsilon} \text{unit}). f () \text{ in } \dots : \tau, \varphi}$$

The particularity of this derivation is that `exec` is *effect polymorphic*; its type

$$\text{exec} : \forall \varepsilon. (\text{unit} \rightarrow^{\varepsilon} \text{unit}) \rightarrow^{\varepsilon} \text{unit}$$

describes that *whatever the effect of f* , the `exec` function has the same effect. This generalization of the type of `exec` is achieved using the LETPOLY rule.

Let us show some other subtleties of the typing relation using an example. We recall the factorial function defined using Landin's knot (see page 44):

```
letregion ρ
let circfact () =
  let id n = n in
  let x = ref [ρ] id in
  let f n = if n = 0 then 1 else n × (!x) (n - 1) in
  x := f;
  !x
```

This function can be typed as follows. First, we set Γ_0 to be the environment inside the `letregion` construct: $\Gamma_0 = \varrho$. Next, we observe that we can obtain the judgment

$$\Gamma_0, n : \text{int}; \Sigma \vdash n : \text{int}, \{\varrho\}$$

2. The Specification Language

with the following derivation (setting $\Gamma_n = \Gamma_0, n : \text{int}$):

$$\text{SUB} \frac{\text{VALUE} \frac{\text{VAR} \frac{\Gamma_n(n) = \text{int}}{\Gamma_n; \Sigma \vdash_v n : \text{int}}}{\Gamma_n; \Sigma \vdash n : \text{int}, \emptyset}}{\Gamma_n; \Sigma \vdash n : \text{int}, \{\varrho\}}}$$

This proves that the identity function id can be typed with the effectful type $\text{int} \rightarrow^\varrho \text{int}$. We go on and give the type $\text{ref}_\varrho (\text{int} \rightarrow^\varrho \text{int})$ to the reference x . Let us set

$$\Gamma_x = \Gamma_0, id : \text{int} \rightarrow^\varrho \text{int}, x : \text{ref}_\varrho (\text{int} \rightarrow^\varrho \text{int}).$$

Setting again Γ_n to $\Gamma_x, n : \text{int}$, we can prove

$$\text{IF} \frac{\text{SUB} \frac{\text{VALUE} \frac{\Gamma_n; \Sigma \vdash_v 1 : \text{int}}{\Gamma_n; \Sigma \vdash 1 : \text{int}, \emptyset}}{\Gamma_n; \Sigma \vdash 1 : \text{int}, \{\varrho\}} \quad \text{APP} \frac{\Gamma_n; \Sigma \vdash !x : \text{int} \rightarrow^\varrho \text{int}, \{\varrho\}}{\Gamma_n; \Sigma \vdash n \times (!x)(n-1) : \text{int}, \{\varrho\}}}{\Gamma_n; \Sigma \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \times (!x)(n-1) : \text{int}, \{\varrho\}}$$

where we have omitted the typing derivations for the boolean condition $n = 0$ and the integer expressions n and $n - 1$. This derivation now can be used to prove that f can be given the type $\text{int} \rightarrow^\varrho \text{int}$, just as id . As a consequence, we can assign x to f and return the new contents of x , *i.e.*, f .

The crucial part of this derivation is that id and f can be given the same type, although f has an effect on ϱ while id is pure. However, this purity can be abandoned for typing purposes.

2.1.4. Properties

To prove that a program in our language cannot be stuck, we first prove that a closed well-typed program is either a value or can do a reduction step using \longrightarrow . This property is called *progress*. But this is not sufficient, because maybe our program will be stuck after that one step, or after n steps. So the second part of the proof is *preservation*, also called *subject reduction*, which states that all the reduction relations \rightarrow , \longrightarrow and \twoheadrightarrow preserve the property of well-typedness.

First, we assure ourselves that there are no unexpected constants of certain types:

Hypothesis 2.10. *We assume that true and false are the only constants c such that $\text{Typeof}(c) = \text{bool}$. Furthermore, we assume that there is no constant c such that $\text{Typeof}(c) = \text{ref}_\varrho \tau$.*

Next, we assume that δ does not get stuck on well-typed values.

Hypothesis 2.11. *$\delta(s, c[\bar{\kappa}]) v$ is always defined if $c[\bar{\kappa}] v$ is well-typed in the empty environment.*

Proof for the constants of Definition 2.5. One can easily convince oneself that the well-typed applications of ref , $!$ and $:=$ are exactly the ones for which δ is defined. \square

Proposition 2.12 (Canonical values). *In the empty environment $\Gamma = \emptyset$,*

1. *a value of type $\tau \rightarrow^\varphi \tau'$ is either a constant or of the form $\text{rec } f (x : \tau). e$,*
2. *a value of type bool is a constant,*
3. *a value of type $\text{ref}_\rho \tau$ is a memory location.*

Proof. Simply by looking at the typing rules, and observing that, in the empty environment, the rule PVAR can never be applied. \square

Additionally, we have to assume that the only constants of type bool are the values true and false , and that there are no other constants of that type.

Hypothesis 2.13 (Constants of type bool). *If a constant c is of type bool , then either $c = \text{true}$ or $c = \text{false}$.*

Before we can prove progress, we have to introduce a notion of compatibility between stores and store typings.

Definition 2.14. A store s is *compatible* with a store typing, written $\Sigma \vdash s$, when for each region $r \in \text{dom}(s)$ and each location $l \in \text{dom}(s(r))$, the store typing Σ contains a mapping $(l \mapsto r, \tau)$ for some τ , and we have $\emptyset; \Sigma \vdash_v s(r)(l) : \tau$. Compatibility of Σ with s also includes *preciseness*, i.e., the domain of Σ and the set of locations in s (the union of the domains of all regions in s) are equal. $\Sigma \vdash s$

Given these requirements on the typing of values and the constant interpretation function δ , proving the “progress”-property is relatively easy.

Theorem 2.15 (Progress). *For every expression e such that $\emptyset; \Sigma \vdash e : \tau, \varphi$, and every state s such that $\Sigma \vdash s$, either e is a value or there is a state s' and an expression e' such that $s, e \longrightarrow s', e'$.*

Proof. The proof is by induction of the structure of the typing derivation of e , and we do a case analysis on the last rule that is applied.

Case VALUE In this case we are done because then e is actually a value.

Case SUB In this case, we can use the induction hypothesis, because e is unchanged.

Case APP In this case, $e = v v'$ and both values are well-typed in the empty environment. We can now apply Proposition 2.12 and we obtain that v is either of the form $\text{rec } f (x : \tau). e_0$ or a constant. In the former case, e admits a reduction, because of the rule β , and in the other case e is an application of a constant c , and we know that all (even partial) well-typed applications of constants admit a reduction using the δ rule.

Cases LET and LETPOLY In this case $e = \text{let } x [\bar{x}] = e_1 \text{ in } e_2$. If e_1 is a value, this expression admits a reduction using let , and otherwise we can conclude by the induction hypothesis and the CONTEXT rule.

2. The Specification Language

Case IF Now we use again Proposition 2.12 to obtain that the condition is a constant. We also know that `true` and `false` are the only constants of type `bool`, by Hypothesis 2.13. Therefore, one of the reduction rules `iftrue` and `iffalse` must apply.

Case LETREG An expression of the form $e = \text{letregion } \varrho \text{ in } e'$ can always be reduced; it is sufficient to choose a fresh region name.

Case REGION Then e is of the form $e = \text{region } r \text{ in } e'$. There are two subcases here. If e' is a value, then e reduces to e' . If e' is not a value, then we can apply the induction hypothesis using the `CONTEXT` rule again.

□

Next we would like to prove subject reduction: when a reduction takes place, the well-typedness of an expression is preserved. Actually, it is even better: reduction also keeps the *type* of an expression. So, our subject reduction theorem for an expression e could look like this:

$$\Gamma; \Sigma \vdash e : \tau, \varphi \quad \text{and} \quad s, e \rightarrow s', e' \quad \text{imply} \quad \Gamma; \Sigma \vdash e' : \tau, \varphi$$

but this is actually not quite right, for two reasons. The first reason is that the reduction might decrease the effect of an expression. Think of the reduction of a lookup $! [r\tau] l$; This expression has an effect on the region ρ , but it reduces to an effect-free value $s(r)(l)$. Fortunately, the `SUB` rule helps out; using it, we can increase at will the effect of an expression. In the example, we can simply pretend that $s(r)(l)$ also has an effect on r . The other reason is that a reduction can create new memory locations. Memory locations are typed with the help of the store typing Σ , so when a new location is created, we must update Σ to reflect this. To capture this update of Σ we define:

$$\Sigma \subseteq \Sigma'$$

Definition 2.16. A store typing Σ is *subsumed* by another one Σ' , written $\Sigma \subseteq \Sigma'$, if every mapping of Σ also exists in Σ' .

And now we can formulate the actual subject reduction theorem (see Theorem 2.23 on page 61):

For every e such as $\Gamma; \Sigma \vdash e : \tau, \varphi$ and $\Sigma \vdash s$ and $s, e \rightarrow s', e'$, we have $\Gamma; \Sigma' \vdash e' : \tau, \varphi$ for some $\Sigma' \supseteq \Sigma$.

A first step is to prove that δ preserves well-typedness. Again, we cannot prove, but must assume this property. We prove the property for the mappings fixed in Definition 2.5.

Hypothesis 2.17 (δ preserves well-typedness). *For each defined mapping $(s, c[\bar{\kappa}], v \mapsto s', v')$ in δ , if $\Gamma; \Sigma \vdash c[\bar{\kappa}] v : \tau, \varphi$ and $\Sigma \vdash s$, then there is some $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash s'$ and $\Gamma; \Sigma' \vdash v' : \tau, \varphi$.*

Proof for the constants of Definition 2.5. By definition, this is true for `!`, because the fact that $\Sigma \vdash s$ precisely states that $v' = s(r)(l)$, the result of evaluating `! l`, is of the type specified in Σ . So here it is sufficient to choose $\Sigma' = \Sigma$. We also need to increase the effect of v' using `SUB`.

For $:= [\tau r]l$, the claim is obvious.

For $:=_{r,l,\tau} v$, we need to rely on the well-typedness of its argument v , which we obtain by inversion of the typing rules. Thus we know that the type of v w.r.t. to Γ and Σ is the same as is specified by $\Sigma(l)$. As for the return type, the application of $:=$ and **void** are both of type **unit**.

In the case of *ref*, we have to extend Σ to accommodate the newly created reference cell. If the expression in question is *ref* $[\tau r] v$, and if the newly created location is called l , then define $\Sigma' = \Sigma[l \mapsto (r, \tau)]$. It is now clear that $\Sigma' \vdash s'$ by construction, and, as $v' = l$, we obtain well-typedness of the result value.

In all cases, we actually obtain $\Gamma; \Sigma' \vdash_v v' : \tau$, but by application of **VALUE** and **SUB**, we can obtain the desired judgment $\Gamma; \Sigma' \vdash v' : \tau, \varphi$ for any effect φ . \square

Lemma 2.18. *Type, effect and region substitution preserve well-typedness:*

1. For a pure value v , from $\Gamma, \chi, \Gamma'; \Sigma \vdash_v v : \tau$ follows $\Gamma, \Gamma' \phi; \Sigma \vdash_v v \phi : \tau \phi$,
2. For a pure expression e without **region** constructs, from $\Gamma, \chi, \Gamma'; \Sigma \vdash e : \tau, \varphi$ follows $\Gamma, \Gamma' \phi; \Sigma \vdash e \phi : \tau \phi, \varphi \phi$

where ϕ is a substitution of the form $[\chi \mapsto \kappa]$.

Proof. We can proceed by simultaneous induction of the typing derivations, because pure values do only contain expressions without **region** constructs, and pure expressions only contain pure values. The actual proof is very easy, because we have evacuated the only problematic case, the one of the **region** construct, for which the claim is actually false. In all cases, we can deconstruct the typing derivation, apply the induction hypothesis and put things back together again. \square

Substitution is an important part of the reduction relation; we prove that substitution preserves the type of an expression separately.

Lemma 2.19 (Substitution Lemma). *Suppose we have a pure, well-typed value v , such that $\Gamma, \bar{\chi}; \Sigma \vdash_v v : \tau$. For the two typing relations, we have:*

1. $\Gamma, x : \forall \bar{\chi}. \tau, \Gamma'; \Sigma \vdash_v v' : \tau'$ implies $\Gamma, \Gamma'; \Sigma \vdash_v v'[x \mapsto \Lambda \bar{\chi}. v] : \tau'$.
2. $\Gamma, x : \forall \bar{\chi}. \tau, \Gamma'; \Sigma \vdash e : \tau', \varphi$ implies $\Gamma, \Gamma'; \Sigma \vdash e[x \mapsto \Lambda \bar{\chi}. v] : \tau', \varphi$.

Proof. We have to prove these properties by mutual induction over the typing derivations. We start by proving the first claim. We thus assume that $\Gamma, \bar{\chi}; \Sigma \vdash_v v : \tau$ and $\Gamma, x : \forall \bar{\chi}. \tau; \Sigma \vdash_v v' : \tau'$, and proceed by a case analysis of the last typing rule of the latter derivation. We denote by ϕ the substitution $[x \mapsto \Lambda \bar{\chi}. v]$ and we set $\Gamma_x = \Gamma, x : \forall \bar{\chi}. \tau, \Gamma'$. We also set $\Gamma_\phi = \Gamma, \Gamma'$.

Case VAR We have $v' = y[\bar{\kappa}]$ and the following typing derivation:

$$\text{VAR} \frac{\Gamma_x(y) = \forall \bar{\chi}. \tau_y}{\Gamma_x; \Sigma \vdash_v y[\bar{\kappa}] : \tau_y[\bar{\chi} \mapsto \bar{\kappa}]}$$

Let us first assume that $x \neq y$. In this case, v' is untouched by the substitution, and the typing derivation remains unchanged when switching from Γ_x to Γ_ϕ , as

2. The Specification Language

the removal of the binding for x does not change anything when looking up the binding for y . Therefore, we have the desired claim.

Let now $x = y$. We set $\theta = [\bar{\chi} \mapsto \bar{\kappa}]$. In this case, $v'\phi = v\theta$. We know that $v\theta$ verifies $\Gamma_\phi; \Sigma \vdash_v v\theta : \tau\theta$ because of Lemma 2.18 and the fact that v is pure. Now we only have to prove that $\tau\theta = \tau'$ and we are done. But this is obvious, as in Γ_x we have $x : \forall \bar{\chi}. \tau$ and we know that under Γ_x , the term $x [\bar{\kappa}]$ has type τ' .

We have shown that whether $x = y$ or not, we obtain the claim in both cases.

Cases LOC and CONST Both memory locations and constants are not affected by substitutions, and their typing is not affected by the typing environment. Therefore, the lemma is trivially true.

Case ABS We have $v' = \text{rec } f (y : \tau_y). e$. We obtain the following derivation:

$$\text{ABS} \frac{\Gamma_x, f : \tau_y \rightarrow^\varphi \tau_1, y : \tau_y; \Sigma \vdash e : \tau_1, \varphi}{\Gamma_x; \Sigma \vdash_v \text{rec } f (y : \tau_y). e : \tau_y \rightarrow^\varphi \tau_1}$$

and we assume the lemma to be true for the hypothesis of the typing rule. We therefore obtain the substituted typing derivation

$$\Gamma_\phi, f : \tau_y \rightarrow^\varphi \tau_1, y : \tau_y; \Sigma \vdash e\phi : \tau_1, \varphi$$

we can now simply reconstruct the function definition using the ABS rule, and we are done.

We have shown the first claim, provided the second claim is true for smaller expressions. The missing part of the proof, showing that the second claim is true, provided that the first claim is true for values, is simple; we just have to decompose the typing derivation, apply the induction hypothesis and put the different parts together again. We do not show this mechanical development here. \square

Proposition 2.20 (Values have empty effect). *For any value v such that $\Gamma; \Sigma \vdash v : \tau, \varphi$, we can also prove $\Gamma; \Sigma \vdash_v v : \tau$ and $\Gamma; \Sigma \vdash v : \tau, \varphi'$, for any effect φ' .*

Proof. If we have $\Gamma; \Sigma \vdash_v v : \tau$, we can prove $\Gamma; \Sigma \vdash v : \tau, \emptyset$ by an application of the VALUE rule and $\Gamma; \Sigma \vdash v : \tau, \varphi'$ by an application of SUB, because $\emptyset \subseteq \varphi'$ is always true. So we have proved that the first claim implies the second. Now let us prove the first claim, by induction of the length of the typing derivation of $\Gamma; \Sigma \vdash v : \tau, \varphi$ and by case analysis of the last rule applied. The only two cases to consider are the rules VALUE and SUB, the other typing rules do not apply to values. For the case of SUB, we simply apply the induction hypothesis and we are done. For the case of VALUE, the claim is stated in the hypothesis of the typing rule. \square

Basically, what we have proved is that the only way a value can be typed as an expression is by using the VALUE rule and an arbitrary number of applications of SUB.

We can now proceed and prove the subject reduction property for each of the reduction relations \rightarrow , \longrightarrow and \rightarrow .

Lemma 2.21 (Top Level Subject Reduction). *For every pure e such as $\Gamma; \Sigma \vdash e : \tau, \varphi$ and $\Sigma \vdash s$ and $s, e \rightarrow s', e'$, we have $\Gamma; \Sigma' \vdash e' : \tau, \varphi$ and $\Sigma' \vdash s'$ for some $\Sigma' \supseteq \Sigma$.*

Proof. By case analysis over the last rule of the typing derivation of e . We must consider every rule except VALUE.

Case APP If APP was the last typing rule to be applied, we are considering one of the reductions (β) or (δ) . If the rule (δ) was applied, we can conclude by Hypothesis 2.17. If the reduction (β) was applied, we obtain the following derivation tree:

$$\text{APP} \frac{\text{REC} \frac{\Gamma'; \Sigma \vdash e_1 : \tau, \varphi}{\Gamma; \Sigma \vdash_v \text{rec } f(x : \tau'). e_1 : \tau' \rightarrow^\varphi \tau} \quad \Gamma; \Sigma \vdash_v v : \tau'}{\Gamma; \Sigma \vdash (\text{rec } f(x : \tau'). e_1) v : \tau, \varphi}$$

where $\Gamma' = \Gamma, f : \tau' \rightarrow^\varphi \tau, x : \tau'$. Looking at the (β) rule, we see that

$$e' = e_1[x \mapsto v, f \mapsto \text{rec } f(x : \tau'). e_1].$$

We can set $\Sigma' = \Sigma$, because s is unchanged, so we are left to prove:

$$\Gamma; \Sigma \vdash e_1[x \mapsto v, f \mapsto \text{rec } f(x : \tau'). e_1] : \tau, \varphi.$$

Using the two hypotheses of the derivation together with a double application of Lemma 2.19 we obtain the desired conclusion. Both substituted values are pure by hypothesis.

Case LET In this case, the only possible reduction rule is (let), and as we are in the monomorphic case, there is no generalization. The term e must be of the form $\text{let } x = v \text{ in } e_1$, and we can deduce the following derivation tree:

$$\text{LET} \frac{\Gamma; \Sigma \vdash v : \tau', \varphi_1 \quad \Gamma, x : \tau'; \Sigma \vdash e_1 : \tau, \varphi_2}{\Gamma; \Sigma \vdash \text{let } x = v \text{ in } e_1 : \tau, \varphi_1 \cup \varphi_2}$$

We can derive that $e' = e_1[x \mapsto v]$ and we have to prove that e' can be typed with type τ and effect φ . We can conclude by Lemma 2.19, because we can also derive a judgment of the form $\Gamma; \Sigma \vdash_v v : \tau'$ for v , and v is pure. We can again leave Σ unchanged, because s did not change.

Case LETPOLY This case is identical, but now we have to fully use the Substitution Lemma in its polymorphic case.

Case IF The two possible reduction rules are iftrue and iffals; we will focus on the former, the proof for the latter being symmetric. By inversion, we obtain the following typing derivation for $e = \text{if } v \text{ then } e_1 \text{ else } e_2$:

$$\text{IF} \frac{\Gamma; \Sigma \vdash_v v : \text{bool} \quad \Gamma; \Sigma \vdash e_1 : \tau_1, \varphi \quad \Gamma; \Sigma \vdash e_2 : \tau_1, \varphi}{\Gamma; \Sigma \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 : \tau_1, \varphi}$$

The reduction of iftrue leaves us with $e' = e_1$, and the induction hypothesis applied to the second hypothesis of the typing derivation allows to obtain the required statement for e_1 , setting $\Sigma' = \Sigma$.

2. The Specification Language

Case SUB Here we cannot tell what reduction rule has been applied. But we know that we have the following typing derivation:

$$\text{SUB} \frac{\Gamma; \Sigma \vdash e : \tau, \varphi' \quad \varphi' \subseteq \varphi}{\Gamma; \Sigma \vdash e : \tau, \varphi}$$

We can now apply the induction hypothesis to the shorter typing derivation of $\Gamma; \Sigma \vdash e : \tau, \varphi'$, and obtain $\Gamma; \Sigma' \vdash e : \tau, \varphi'$ for some Σ' . Now we simply apply the rule SUB again and we are done. This last application is correct since $\varphi' \subseteq \varphi$.

Case REGION The only possible reduction rule is region, so we know that

$$e = \text{region } r \text{ in } v.$$

We have the following typing derivation:

$$\text{REGION} \frac{\Gamma; \Sigma \vdash v : \tau, \varphi}{\Gamma; \Sigma \vdash \text{region } r \text{ in } v : \tau, \varphi' \setminus r}$$

The result of the reduction is v . By Proposition 2.20, we can also prove

$$\Gamma; \Sigma \vdash v : \tau, \varphi' \setminus r,$$

and as s is unchanged, we are done.

Case LETREG The only possible reduction rule is letreg, and we obtain

$$e = \text{letregion } \varrho \text{ in } e.$$

We have the following typing derivation:

$$\text{LETREG} \frac{\Gamma, \varrho; \Sigma \vdash e : \tau, \varphi \quad \varrho \notin \tau}{\Gamma; \Sigma \vdash \text{letregion } \varrho \text{ in } e : \tau, \varphi \setminus \varrho}$$

The result of the reduction is $\text{region } r \text{ in } e[\varrho \mapsto r]$, where r is the region name chosen during the execution of the reduction. Setting $\theta = [\varrho \mapsto r]$, and using Lemma 2.18, we can prove that $\Gamma; \Sigma \vdash e\theta : \tau\theta, \varphi\theta$. As $\varrho \notin \tau$, we have $\tau\theta = \tau$. We also have $\varphi \setminus \varrho = \varphi\theta \setminus r$, so we can conclude using the REGION rule. As the newly created region does not contain any memory locations yet, we do not need to change Σ .

□

We have completed the most difficult part. We can now continue to prove the same property about the relation \longrightarrow .

Lemma 2.22 (One Step Subject Reduction). *For every pure e such as $\Gamma; \Sigma \vdash e : \tau, \varphi$ and $\Sigma \vdash s$ and $s, e \longrightarrow s', e'$, we have $\Gamma; \Sigma' \vdash e' : \tau, \varphi$ and $\Sigma' \vdash s'$ for some $\Sigma' \supseteq \Sigma$.*

Proof. We prove this lemma by induction over the form of reduction contexts. If the context is empty, *i.e.*, of the form \square , the lemma states the same property as Lemma 2.21. There are two cases remaining.

Case region We know that $e = \text{region } r \text{ in } E[e_1]$ and that we have $s, e_1 \rightarrow s', e'_1$. We can derive $s, E[e_1] \rightarrow s', E[e'_1]$ as well. Using the induction hypothesis, we obtain a derivation for $\Gamma; \Sigma' \vdash E[e'_1] : \tau, \varphi'$ for some Σ' for which $\Sigma' \vdash s'$. Now we simply apply the REGION rule to obtain the desired statement for the overall expression $e = \text{region } r \text{ in } E[e'_1]$.

Case let The term we are considering is of the form $e = \text{let } x [\bar{\chi}] = E[e_1] \text{ in } e_2$, and we have $s, e_1 \rightarrow s', e'_1$. We can derive $s, E[e_1] \rightarrow s', E[e'_1]$ as well. Therefore, $E[e_1]$ is not a value, and the applied typing rule must have been LET, with the list $\bar{\chi}$ being empty. We also know, by induction hypothesis, that s' and $E[e'_1]$ are well-typed w.r.t. to Γ and some Σ' . We can conclude by applying the LET rule once again, obtaining that $\Gamma; \Sigma' \vdash \text{let } x = E[e'_1] \text{ in } e_2 : \tau, \varphi$.

□

Theorem 2.23 (Subject Reduction). *For every pure e such as $\Gamma; \Sigma \vdash e : \tau, \varphi$ and $\Sigma \vdash s$ and $s, e \rightarrow s', e'$, we have $\Gamma; \Sigma' \vdash e' : \tau, \varphi$ for some $\Sigma' \supseteq \Sigma$.*

Proof. We prove this theorem by induction over the length n of the reduction chain described by $s, e \rightarrow \dots \rightarrow s', e'$. If n is zero, this chain is empty and $s = s'$ as well as $e = e'$; the theorem is trivial. In the induction step, we assume the property for a chain of length n , ending with s'', e'' and have to prove it for a chain of length $n + 1$, ending with s', e' . By the induction hypothesis, the property is true for s'', e'' , and as we have $s'', e'' \rightarrow s', e'$, we can use Lemma 2.22 for the last step. □

Together with the Progress property of Theorem 2.15, we now can establish the safety property of our language. We have underlined that **region** is not part of the surface language, and this property is now a hypothesis of the main result.

Corollary. *For any well-typed expression e without **region** such that $\emptyset; \emptyset \vdash e : \tau, \varphi$, starting in the empty store \emptyset , the configuration \emptyset, e either reduces forever or reduces to a configuration s', v with $\emptyset; \Sigma \vdash_v v : \tau$ for some Σ' .*

2.1.5. A Generalization of the Results

By looking closely at the results of the previous section, it is clear that we have used only a few properties of effects. Indeed, the precise form of effects is largely irrelevant. We only need the union \cup of effects and the removal \setminus of a region from an effect:

$$\varphi_1 \cup \varphi_2 \quad \text{and} \quad \varphi \setminus \rho$$

as well as the tests of presence and inclusion of effects:

$$\rho \in \varphi \quad \text{and} \quad \varphi \subseteq \varphi'$$

To be able to prove Lemma 2.18, we need the effect union to be stable under substitution:

$$(\varphi_1 \cup \varphi_2)\phi = \varphi_1\phi \cup \varphi_2\phi.$$

2. The Specification Language

Stability of the the region removal operation \setminus is also required, but we can restrict the substitution ϕ to particular cases, because of the precise places where \setminus is employed in the typing rules. In the case of the region being a region variable ϱ , we are in the case of the `letregion` construct, and therefore we can assume that ϱ is fresh, *i.e.*, that ϕ does not contain ϱ , neither in the domain nor the image.

$$(\varphi \setminus \varrho)\phi = \varphi\phi \setminus \varrho\phi = \varphi\phi \setminus \varrho \quad \text{if } \varrho \notin \phi$$

In the case where ρ is a region constant r we are in the case of the `region` construct; we do not need stability in this case, because of purity requirement in Lemma 2.18. All these properties also suffice to prove Lemma 2.19.

To prove subject reduction, we also need the following property:

$$\varphi \setminus \varrho = \varphi[\varrho \mapsto r] \setminus r$$

It is needed in the case concerning `letregion` in Lemma 2.21.

In Chapter 3 and Chapter 4, we will use this more general formulation of type soundness to enable interesting variants of the presented system, just by modifying the representation of effects.

Nielson et al. (1999) have noticed this genericity of effects long ago. Leroy and Pessaux (2000) have developed an effect system for exceptions, that is very similar to ours, but with an efficient type inference algorithm. Marino and Millstein (2009) have proposed a generic effect system, along with a mechanized soundness proof.

2.2. The Logic L

The previous section has shown that programs in our language cannot get stuck — they cannot crash. Looking at it from the right angle, *typing* a program (establishing its well-typedness) actually means *proving* something about a program. According to Theorem 2.23, someone who establishes the well-typedness of a program has *proved* that this program will not crash.

However, as we have seen in the introduction, a program can exhibit undesirable behavior other than a simple crash; for example, an incorrect implementation of an algorithm may not be very useful. Non-termination is another undesirable behavior. Similarly to the well-typedness, one would like to *prove* that the program terminates, and *prove* that the program is correct, *i.e.*, that it computes the expected value.

Unfortunately, proving the correctness of programs is as hard as proving mathematical theorems. This is not surprising, as many algorithms are *based on* mathematical theorems. There are countless examples, but a very simple one is the second program of *The Art of Computer Programming* (Knuth, 1997), which computes the first n primes. To minimize the number of division tests, this program uses an optimization that can be justified by Bertrand's Theorem, a non-trivial result of the theory of prime numbers. To prove the program correct, one has to prove Bertrand's Theorem (Théry, 2002).

This also means that program correctness is *undecidable*: there can not be an algorithm which takes a program and its specification as an input and checks that the program indeed verifies the specification.

In this section, we describe our own language of specifications. It is a formal logic language. As we have stated, a specification language should be able to express all aspects of the programming language. As our language is a higher-order language with side effects, the two main aspects of our logic are the presence of higher-order functions and state. We call this logic the language L.

2.2.1. Syntax

State types. The logic L is aimed at the verification of programs written in W. Therefore, there should be some possibility to speak about the state, as programs in W may have side effects. This possibility is provided by *state types*. A state type is written $\langle\varphi\rangle$, where φ is an effect expression. An Object of this type, a *state object*, can be seen as a portion of the store whose domain is described by the effect expression φ . Such a state object can be restricted in its domain, or merged with another state object, or queried at a certain region and memory location.

Logic types. The types in the logic, denoted by the symbol σ , are very similar to the types in the programming language. As before, we have type variables α , type constructors ι and reference types $\text{ref}_\rho \sigma$. There is no type for effectful functions; instead, there is the type $\sigma \rightarrow \sigma'$ for *pure* functions of argument type σ and return type σ' . These functions always terminate as well. We have already discussed state types, written $\langle\varphi\rangle$. There are also two new type constructors: the first one is **prop**, it represents the type of propositions. The second one is \times , a binary type constructor that represents pairs. We write $\sigma_1 \times \sigma_2$ instead of $\times(\sigma_1, \sigma_2)$. The metavariable χ is used in the logic as well, but we replace the metavariable κ by the metavariable \varkappa , which stands for effects, regions or *logic* types.

$$\begin{aligned} \iota &::= \dots \mid \text{prop} \mid \times \\ \sigma &::= \alpha \mid \sigma \rightarrow \sigma \mid \iota \bar{\sigma} \mid \langle\varphi\rangle \mid \text{ref}_\rho \sigma \\ \varkappa &::= \sigma \mid \rho \mid \varphi \end{aligned}$$

Constants. Even the syntax of the logic is not that different from programs. All program constants are also part of the constants in L. Additionally, we have two constants **True** and **False** of type **prop**, the usual logical connectives, the constant **mkpair** to construct pairs and the constants **fst** and **snd** to access the left and right component of a pair, respectively. Finally, there are three other constants **combine**, **restrict** and **get** which will be explained later.

$$\begin{aligned} c &::= \dots \mid \text{True} \mid \text{False} \mid \Rightarrow \mid \wedge \mid \vee \\ &\quad \mid \text{combine} \mid \in \mid \text{restrict} \mid \text{get} \mid \text{mkpair} \mid \text{fst} \mid \text{snd} \end{aligned}$$

Terms. The actual language is a standard higher-order language, as for example defined in (Andrews, 1986). As in the programming language, a term t can be a constant c or a variable x , both with possible type, effect and region instantiations. There is

2. The Specification Language

$$\begin{aligned}
\iota & ::= \dots \mid \text{prop} \mid \times \\
\sigma & ::= \alpha \mid \sigma \rightarrow \sigma \mid \iota \bar{\sigma} \mid \langle \varphi \rangle \mid \text{ref}_\rho \sigma \\
c & ::= \dots \mid \text{True} \mid \text{False} \mid \Rightarrow \mid \wedge \mid \vee \\
& \quad \mid \text{combine} \mid \in \mid \text{restrict} \mid \text{get} \mid \text{mkpair} \mid \text{fst} \mid \text{snd} \\
t, p, q & ::= c [\bar{z}] \mid x [\bar{z}] \mid l \mid \vartheta \mid t t \mid t = t \mid \lambda(x : \sigma).t \\
& \quad \mid \text{let } x [\bar{\chi}] = t \text{ in } t \mid \forall x : \sigma.t \mid \forall \bar{\chi}.t \mid \rho \in t \mid \varrho = \varrho \\
\Delta & ::= \emptyset \mid \Delta, x : \forall \bar{\chi}.\sigma \mid \Delta, \bar{\chi}
\end{aligned}$$

Figure 2.6: The syntax of L.

the pure application $t t'$ and the equality between two terms $t = t'$. In the logic, one can build anonymous functions using a λ -abstraction $\lambda(x : \sigma).t$. It is important to note that these functions are non-recursive, terminating and cannot have any side effects. Just as in W, we have a polymorphic `let` construct. For the sake of the later formal development, we also need to refer to memory locations l in L, just as we do in W. There is the usual quantification $\forall x : \sigma.t$ and the introduction of type, effect and region variables $\forall \bar{\chi}.t$. The expression $\rho \in t$ is a predicate that decides if a certain region is in the domain of a value of state type. Finally, we can check if two regions are equal using the test $\varrho = \varrho$.

$$\begin{aligned}
t, p, q & ::= c [\bar{z}] \mid x [\bar{z}] \mid l \mid \vartheta \mid t t \mid t = t \mid \lambda(x : \sigma).t \\
& \quad \mid \text{let } x [\bar{\chi}] = t \text{ in } t \mid \forall x : \sigma.t \mid \forall \bar{\chi}.t \mid \varrho \in t \mid \varrho = \varrho
\end{aligned}$$

The special term ϑ is not available to the user and is only there for the purpose of the correctness proof. It stands for the *current state*; what that means becomes clear in section Section 2.2.4.

A summary of the syntax of the logic appears in Fig. 2.6. The logical connectors \Rightarrow , \wedge and so on will be used in infix form, *i.e.*, $t \Rightarrow t'$ instead of $\Rightarrow t t'$. Instead of `mkpair $t t'$` we will write (t, t') . If type, region and effect instantiations are not important or easy to derive from the context, we omit them.

2.2.2. Typing

Just as our programming language, logical formulas are typed to avoid writing meaningless formulas. The judgment $\Delta; \Sigma \vdash_l t : \sigma$ expresses that in the typing environment Δ , and under the store typing Σ , the formula t has type σ . This judgment is defined in Fig. 2.7. It is straightforward, but it contains a few unusual parts. First, just as in programs, we have type, effect and region substitutions (rules L-CONST and L-VAR). To type constants, we use the function *LogicTypeof* instead of the counterpart *Typeof* for programs. Second, as logical terms are always pure, we can generalize every `let`-binding (L-LET rule). Application, abstraction, quantification and equality are typed in the usual way. Memory locations l are typed using the store typing Σ (rule L-LOC). It should also be noticed that albeit the syntax and typing derivation of the logic are

$$\boxed{\Delta; \Sigma \vdash_l t : \sigma}$$

$$\begin{array}{c}
\text{L-VAR} \frac{\Delta(x) = \forall \bar{\chi}. \sigma}{\Delta; \Sigma \vdash_l x [\bar{z}] : \sigma[\bar{\chi} \mapsto \bar{z}]} \qquad \text{L-CONST} \frac{\text{LogicTypeof}(c) = \forall \bar{\chi}. \sigma}{\Delta; \Sigma \vdash_l c [\bar{z}] : \sigma[\bar{\chi} \mapsto \bar{z}]} \\
\text{L-LOC} \frac{\Sigma(l) = r, \tau}{\Delta; \Sigma \vdash_l l : \text{ref}_r [\tau]} \qquad \text{L-STORE} \frac{\text{regions}(\Sigma) = \varphi}{\Delta; \Sigma \vdash_l \vartheta : \langle \varphi \rangle} \\
\text{L-APP} \frac{\Delta; \Sigma \vdash_l t : \sigma' \rightarrow \sigma \quad \Delta; \Sigma \vdash_l t' : \sigma'}{\Delta; \Sigma \vdash_l t t' : \sigma} \qquad \text{L-ABS} \frac{\Delta, x : \sigma'; \Sigma \vdash_l t : \sigma}{\Delta; \Sigma \vdash_l \lambda(x : \sigma'). t : \sigma' \rightarrow \sigma} \\
\text{L-FORALL} \frac{\Delta, x : \sigma'; \Sigma \vdash_l t : \text{prop}}{\Delta; \Sigma \vdash_l \forall x : \sigma'. t : \text{prop}} \qquad \text{L-LET} \frac{\Delta, \bar{\chi}; \Sigma \vdash_l t' : \sigma' \quad \Delta, x : \forall \bar{\chi}. \sigma'; \Sigma \vdash_l t : \sigma}{\Delta; \Sigma \vdash_l \text{let } x [\bar{\chi}] = t' \text{ in } t : \sigma} \\
\text{L-EQ} \frac{\Delta; \Sigma \vdash_l t_1, t_2 : \sigma}{\Delta; \Sigma \vdash_l t_1 = t_2 : \text{prop}} \qquad \text{L-FORALLTYPE} \frac{\Delta, \bar{\chi}; \Sigma \vdash_l t : \text{prop}}{\Delta; \Sigma \vdash_l \forall \bar{\chi}. t : \text{prop}} \\
\text{L-INDOM} \frac{\Delta, \varrho; \Sigma \vdash_l t : \langle \varphi \rangle}{\Delta; \Sigma \vdash_l \varrho \in t : \text{prop}} \qquad \text{L-REGEQ} \frac{}{\Delta, \varrho_1, \varrho_2; \Sigma \vdash_l \varrho_1 = \varrho_2 : \text{prop}}
\end{array}$$

Figure 2.7: The typing rules for L.

quite similar to the one of programs, and despite the presence of memory locations and reference types in the syntax, logical terms can not have any side effects, a fact that is also reflected in the form of the typing derivation. An appropriate translation from program values to logical formulas (presented in this chapter) will take care of this property. The general idea is that in the logic, references can only be used to *read*, and only with respect to an explicitly given portion of the store at a certain point in time. Finally, the special term ϑ is of state type, whose domain is given by the store typing Σ (rule L-STORE).

There is a correspondence between the types τ of programs and the types σ of formulas: every program type τ has a corresponding logical type $[\sigma]$. This correspondence is defined as follows.

Definition 2.24. The logical reflection of a program type τ , written $[\tau]$, is defined as follows: $\boxed{[\tau]}$

$$\begin{aligned}
[\alpha] &= \alpha \\
[\iota \bar{\tau}] &= \iota \overline{[\bar{\tau}]} \\
[\text{ref}_\rho \tau] &= \text{ref}_\rho [\tau] \\
[\tau_1 \rightarrow^\varphi \tau_2] &= ([\tau_1] \rightarrow \langle \varphi \rangle \rightarrow \text{prop}) \times ([\tau_1] \rightarrow \langle \varphi \rangle \rightarrow \langle \varphi \rangle \rightarrow [\tau_2] \rightarrow \text{prop})
\end{aligned}$$

This basically means that type variables and type constructors can be taken over as-is to the logic. But effectful functions do not exist in L, so we need to do something.

2. The Specification Language

The idea (Régis-Gianas and Pottier, 2008) is to represent an effectful function in the logic by its specification. A specification is a precondition depending on the argument and a state and a postcondition depending on the *initial state* and the *final state*, the argument and the return value. Therefore, an effectful function type is translated as a tuple whose components have exactly the expected types. Note that the state types have as domain precisely the effect of the body of the function.

LogicTypeof

Definition 2.25. We assume the existence of a function *LogicTypeof*, which associates a logic type to each constant. We also assume that the domain of the *Typeof* function is included in the one of *LogicTypeof*. We finally assume that it is defined such that for every constant in the domain of *Typeof*, we have

$$\text{LogicTypeof}(c) = [\text{Typeof}(c)].$$

This means, for example, that even though *ref*, *!* and *:=* are constants in the logic, one cannot use them to produce side effects, as in **L** their types are tuples with pre- and postconditions and not function types.

Additionally, we assume that *LogicTypeof* is defined as follows for the previously introduced constants:

$$\begin{aligned} \text{True, False} & : \text{prop} \\ \wedge, \vee, \Rightarrow & : \text{prop} \rightarrow \text{prop} \rightarrow \text{prop} \\ \text{mkpair} & : \forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha \times \beta \\ \text{fst, pre} & : \forall \alpha \beta. \alpha \times \beta \rightarrow \alpha \\ \text{snd, post} & : \forall \alpha \beta. \alpha \times \beta \rightarrow \beta \end{aligned}$$

The following constants are associated with state and are typed as follows:

$$\begin{aligned} \text{combine} & : \forall \varepsilon_1 \varepsilon_2 \varepsilon_3. \langle \varepsilon_1 \varepsilon_2 \rangle \rightarrow \langle \varepsilon_2 \varepsilon_3 \rangle \rightarrow \langle \varepsilon_1 \varepsilon_2 \varepsilon_3 \rangle \\ \text{restrict} & : \forall \varepsilon_1 \varepsilon_2. \langle \varepsilon_1 \varepsilon_2 \rangle \rightarrow \langle \varepsilon_1 \rangle \\ \text{get} & : \forall \alpha \varrho \varepsilon. \langle \varrho \varepsilon \rangle \rightarrow \text{ref}_\varrho \alpha \rightarrow \alpha \\ \text{set} & : \forall \alpha \varrho \varepsilon. \langle \varrho \varepsilon \rangle \rightarrow \text{ref}_\varrho \alpha \rightarrow \alpha \rightarrow \langle \varrho \varepsilon \rangle \\ \in & : \forall \alpha \varrho \varepsilon. \langle \varrho \varepsilon \rangle \rightarrow \text{ref}_\varrho \alpha \rightarrow \text{prop} \end{aligned}$$

Their meaning is explained in Section 2.2.3.

When writing terms of **L**, we will mostly omit type, effect and region instantiations. In the particular case of the functions concerning state types, we sometimes explicit such instantiations. We therefore introduce the following notations:

$$\begin{aligned} \text{get}_\rho & = \text{get} [\sigma \rho \varphi] && \text{where } \sigma \text{ and } \varphi \text{ can be derived from the context} \\ \text{set}_\rho & = \text{set} [\sigma \rho \varphi] && \text{where } \sigma \text{ and } \varphi \text{ can be derived from the context} \\ \text{restrict}_\varphi & = \text{restrict} [\varphi \varphi'] && \text{where } \varphi' \text{ can be derived from the context} \end{aligned}$$

In other words, the subscript of *get* determines its region instantiation, while the subscript of *restrict* determines its return type.

2.2.3. Semantics

A logic is useless if we do not know what its formulas mean. To be able to say anything about our formulas, we need a *semantics* for it. Defining the semantics of a logic is a

very difficult task. We reuse here parts of the semantics of the higher-order logic defined by Andrews (1986). We only extend the semantics to cover the language elements we have added, namely region and effect polymorphism and store objects.

Andrews (1986), based on the work by Henkin (1950), defines the semantics by *frames*, a pair

$$(\{D_\sigma\}_\sigma, J)$$

where for each type σ , D_σ is the set of all semantics values of type σ . For example, $D_{\text{int}} = \{\dots, -1, 0, 1, 2, \dots\}$ and the set $D_{\sigma_1 \rightarrow \sigma_2}$ is the set of all functions from σ_1 to σ_2 . J is a mapping from the constants to their intended meaning, e.g., the conjunction \wedge is mapped to the function that is equal to True whenever both its arguments are True.

In this semantics, each construction in the logic is mapped to a similar construct of the semantic domain. For example, application of a term t_1 to another term t_2 is modeled by the application of the meaning of t_1 , which is a mathematical function, to the meaning of t_2 . These definitions make it trivial to prove the following properties:

Proposition 2.26 ((Andrews, 1986)). *The following equalities can be proved in L:*

$$\begin{aligned} (\lambda x : \sigma. t_1) t_2 &= t_1[x \mapsto t_2] \\ \text{let } x [\bar{\chi}] = t_1 \text{ in } t_2 &= t_2[x \mapsto \Lambda \bar{\chi}. t_1] \end{aligned}$$

We make the additional hypothesis that all models we consider also admit η -equivalence.

Hypothesis 2.27 (η -equivalence). *The following equality can be proved in L:*

$$(\lambda x : \sigma. f x) = f$$

We now extend this semantics to the constructs that we have added: region variables are interpreted as region names such as r , and effect variables are interpreted as sets of region names. Under this interpretation, an effect is simply a set of region names, and the type $D_{(\varphi)}$ is the set of stores with exactly the region domain described by φ . Finally, for all ϱ and σ , the set $D_{\text{ref}_\varrho \sigma}$ is the set of memory locations l .

We now have to give an meaning to the constants of the logic that concern state objects. Most of them are obvious and quite simple. The formula $\varrho \in s$ is true when the region name designated by ϱ is in the domain of the store described by s ; the function symbol \in concerning references checks if a memory location is in the domain of the region of a store. The constant `restrict` is the function that restricts the store to a certain domain of regions, but leaves the contents unchanged. The function `combine` merges two stores, such that, when there is overlap in the region domains of the two stores, the contents of the second store are kept. The domain of the result store is always the union of the initial domains. The function `set` corresponds to the update operation of a location in a store: $s[r \mapsto s(r)[l \mapsto v]]$.

The only delicate constant is `get`. It is intended to represent the lookup of a location in a certain region of a store, written $s(r)(l)$. By the typing information, we can prove that r is indeed in the domain of s , otherwise the application of `get` would be ill-typed. However, we cannot guarantee that l is indeed in the domain of $s(r)$.

However, it is easy to render the interpretation of `get` $[\sigma \varrho \varphi] s x$ total. Let us first make the assumption that all types in L are inhabited. This seems to be a strong

2. The Specification Language

requirement, but it is easy to see that it is true. First remark that all base types such as `int` and `bool` are inhabited. The set of memory locations, which is the interpretation of reference types, is also inhabited, as is any concrete state type: simply choose the store with the corresponding region domain, where all regions are empty. The only remaining type are function types $\sigma_1 \rightarrow \sigma_2$. There is always a function of this type, namely the function that throws away its argument and returns an inhabitant of σ_2 . Note that HOL (Gordon, 2000), whose logic is very similar to L, requires types to be inhabited as well (Keller and Werner, 2010).

This property now helps us to render the interpretation of `get` $[\sigma r \varphi] s l$ total. We define `get` to check whether the memory location l exists in the region r ; if it does, the value $s(r)(l)$ is returned. If it does not, some arbitrary value of type σ is returned. Such a value exists because σ is inhabited.

Using this interpretation as the semantic basis of our logic, we can prove the following properties of our functions:

- A1 : `get (set s x v) x = v`
- A2 : $x \neq y \Rightarrow \text{get (set s y v) } x = \text{get s } x$
- A3 : $\varrho_1 \neq \varrho_2 \Rightarrow \text{get}_{\varrho_1} (\text{set}_{\varrho_2} s y v) x = \text{get}_{\varrho_1} s x$
- A4 : `get (restrict s) x = get s x`
- A5 : $\varrho \notin s_2 \Rightarrow \text{get}_{\varrho} (\text{combine } s_1 s_2) x = \text{get } s_1 x$
- A6 : $\varrho \in s_2 \Rightarrow \text{get (combine } s_1 s_2) x = \text{get } s_2 x$
- A7 : `combine (restrict ϱ_1 s) (restrict ϱ_2 s) = restrict $\varrho_1 \cup \varrho_2$ s`

The first three properties simply summarize the usual properties of finite maps: writing a value and reading at the same place recovers the same value (A1); when reading at other places — either a different location in the same region (A2) or in a different region (A3), the update is irrelevant. Property (A4) expresses that `restrict` never changes the values of a map, while (A5) and (A6) rephrase that the second argument of `combine` may overwrite parts of the first one. Property (A7) states that restricting the same state twice and recombining it is equivalent to restricting it only once. In the following, we will accept these properties as axioms.

Examples. Let us show a few examples of formulas, including formulas that reason about state. The following anonymous predicate over integers states that its second argument is one greater than the first:

$$\lambda i : \text{int}. \lambda j : \text{int}. i = j + 1$$

The next formula is a higher-order predicate over a pair of functions:

$$\lambda f : (\text{int} \rightarrow \langle \varrho \rangle \rightarrow \text{prop}) \times (\text{int} \rightarrow \langle \varrho \rangle \rightarrow \langle \varrho \rangle \rightarrow \text{int} \rightarrow \text{prop}). \\ \forall x : \text{int}. \forall s : \langle \varrho \rangle. x > 0 \Rightarrow \text{fst } f x s$$

It states that the first component of f — a predicate over an integer and a state with region ϱ — holds whenever its first argument x is positive. Considering the type of f to be the image of the program type $\text{int} \rightarrow^e \text{int}$, this statement can also be interpreted as saying that the *precondition* of f holds whenever the function argument x is positive. We could have used the synonym `pre` instead of `fst`, to underline the fact that we access the precondition of f .

Finally, a predicate concerning state can look like this:

$$\lambda s : \langle \varrho \rangle. \lambda x : \text{ref}_\varrho \text{ int. } \text{get}_\varrho x s = 0$$

We use the function `get` to access the state s at the memory location denoted by x . We could have also used our notation `!!` instead of `get` to abbreviate the formula:

$$!! x s = 0$$

The region instantiation of `get` is implied by the type of x .

Properties of the typing derivation. We give here, without proof, some important properties about typing derivations for terms. First note that type, region and effect substitutions are defined in the same way as in \mathbb{W} (see Definition 2.4).

Lemma 2.28 (Substitution Lemma for Metavariables). *From*

$$\Delta, \chi, \Delta'; \Sigma \vdash_l t : \sigma$$

follows

$$\Delta, \Delta'[\chi \mapsto \varkappa]; \Sigma \vdash_l t[\chi \mapsto \varkappa] : \sigma[\chi \mapsto \varkappa]$$

Lemma 2.29 (Substitution Lemma for L). *From*

$$\Delta, x : \forall \bar{\chi}. \sigma', \Delta'; \Sigma \vdash_l t : \sigma \quad \text{and} \quad \Delta, \bar{\chi}; \Sigma \vdash_l t' : \sigma'$$

follows

$$\Delta, \Delta'; \Sigma \vdash_l t[x \mapsto \Lambda \bar{\chi}. t'] : \sigma.$$

Both lemmas are straightforward to prove.

2.2.4. Annotating Programs

Up to now, we have defined a logic \mathbb{L} and a programming language \mathbb{W} , but they do not interact. Before we define the weakest precondition calculus in Chapter 3, we integrate annotations into the programming language. A natural building block of programs are functions, so functions should be annotated with pre- and postconditions. In \mathbb{W} , functions are built using the `rec` construct. Therefore, we change the syntax of recursive functions to this one:

$$e ::= \dots \mid \text{rec } f (x : \tau'). \{p\}e\{q\} \mid \dots$$

where p, q are logical terms. The semantics of functions does not change; upon β -reduction, the annotations are simply thrown away. Typing does change slightly; we replace the rule for recursive functions by this one:

$$\text{REC} \frac{\Gamma, f : \tau' \rightarrow^\varphi \tau, x : \tau'; \Sigma \vdash e : \tau, \varphi \quad [\Gamma, x : \tau']; \Sigma \vdash_l p : \langle \varphi \rangle \rightarrow \text{prop} \quad [\Gamma, x : \tau']; \Sigma \vdash_l q : \langle \varphi \rangle \rightarrow \langle \varphi \rangle \rightarrow [\tau] \rightarrow \text{prop}}{\Gamma; \Sigma \vdash_v \text{rec } f (x : \tau'). \{p\}e\{q\} : \tau' \rightarrow^\varphi \tau}$$

We can argue that this does not change anything about the progress and subject reduction theorems. In particular, the substitution lemma still holds because we also have a substitution lemma for logical terms.

2. The Specification Language

Notation and examples. Let us look at purely functional factorial function. We can fully specify this function like this:

```
let rec fact (n : int) =  
  { λcur : ⟨⟩. n ≥ 1 }  
  if n = 1 then 1 else n * f(n - 1)  
  { λold, cur : ⟨⟩. λr : int. r = n! }
```

where $n!$ is the *mathematical* definition of the factorial. Note that our precondition (the first formula framed by curly braces) has, as expected, the type $\langle \rangle \rightarrow \mathbf{prop}$; it expresses that the argument n must be at least 1 for a call to *fact* to be meaningful. The postcondition has type $\langle \rangle \rightarrow \langle \rangle \rightarrow \mathbf{int} \rightarrow \mathbf{prop}$ and states that the return value is indeed the factorial of the argument.

It would be tedious to write all those abstractions all the time. We therefore agree upon names for state abstractions once and for all: we fix them to be *cur* for the current state in the pre- and postcondition, and to be *old* for the initial state of the function call (available only in the postcondition). Having set these names, we can leave out the abstractions and let them be implicit. For the abstraction of the return value, we choose another syntax, so that we can write the specification of *fact* in this way:

```
let rec fact (n : int) =  
  { n ≥ 1 }  
  if n = 1 then 1 else n * f(n - 1)  
  { r : r = n! }
```

which is much more readable.

Now let us move on to an example with side effects. We present here a trivial example just to introduce the notation.

```
let setzero [ρ] (n : refρ int) =  
  {}  
  n := 0  
  { getρn cur = 0 }
```

We have used the function *get* and the convention *cur* for the current state to express that n , after executing the function, is indeed equal to 0. To be able to say this more concisely, we use *!!* instead of *get*, the region parameter is implied by the reference type, and we say that if no state variable is given, the considered state is always *cur*. The postcondition can now be written as:

```
{ !! n = 0 }
```

The examples we have seen only intend to introduce the syntax and some notation. Chapter 5 contains many examples that are much more realistic and interesting than the ones in this chapter.

Functions with more than one argument. Up to now, we have only dealt with one-argument functions, because that is what the syntax contains directly. It is very easy to obtain functions with several arguments, both in the simple and the recursive case. Let us forget for a moment about specification annotations. Simple and recursive functions with several arguments can be written as follows: the two function definitions

```

let f x1 ... xn = e
let rec g x1 ... xn = e

```

stand for

```

let f = rec _ x1. ... rec _ xn. e
let g = rec g x1. ... rec _ xn. e

```

Now, the situation is slightly more complicated when specifications come into play. We would like to write specifications for multiple-argument functions like this:

```

let f x1 ... xn = { pf } e { qf }
let rec g x1 ... xn = { pg } e { qg }

```

where the pre- and postconditions may refer to the argument variables. For such a function definition, we only have a single pair of specifications, but we need n pre- and postconditions for the expansion of the syntactic sugar. The missing preconditions are easy to find: it is always possible to partially apply the first $n - 1$ arguments to obtain the function closure. We therefore can simply put `True` as precondition of the $n - 1$ outer abstractions, and p_f (or p_g) as precondition of the innermost abstraction. In this way, p_f is in the scope of all abstraction variables, and this choice does reflect the fact that p_f has to be true when all arguments of f are applied. The same argument makes clear that q_f must be the postcondition of the innermost abstraction, but we cannot simply set the other postconditions to `True` as we did for the preconditions. Otherwise, a program that partially applies f would not be able to state anything about the result; and remember that in our language, all applications to n -ary functions are transformed into n partial applications because of the transformation into A-normal form. The best solution is to state in each postcondition precisely that the partial application returns a function with the correct specification.

Let us see a very simple example. We want to define our own addition function, here in the surface syntax:

```

let plus x y =
  { }
  x + y
  { r : r = x + y }

```

Following what we said before, we can translate this program to the basic syntax as follows:

```

let plus =
  rec _ x.
    { True }
    rec _ y. { True }. x + y { r : r = x + y }
    { r : r = (λx.λcur.True, λx.λold.λcur.λr. r = x + y) }

```

The initial specification of `plus` becomes the specification of the inner anonymous function. The precondition of the outer function is `True`, while the postcondition states that its result is equal to the lifted inner function.

2. The Specification Language

Lifting values. We have seen that program types can be lifted to logical types using the $\lceil \cdot \rceil$ transformation. If we want to refer to values of our program in annotations, we need a similar operation for values of the programming language. Starting from the basic idea that effectful functions are represented by their specification, this is actually easy.

$\boxed{[v]}$

Definition 2.30. The reflection of a program value v in the logic is defined as:

$$\begin{aligned} \lceil c \bar{\kappa} \rceil &= c \lceil \bar{\kappa} \rceil \\ \lceil x \bar{\kappa} \rceil &= x \lceil \bar{\kappa} \rceil \\ \lceil l \rceil &= l \\ \lceil \text{rec } f (x : \tau). \{p\} e \{q\} \rceil &= (\lambda x : \lceil \tau \rceil. p, \lambda x : \lceil \tau \rceil. q) \end{aligned}$$

The notation $\lceil \bar{\kappa} \rceil$ indicates that a lift operation is applied to all types in this list; lifting has no impact on effects and regions.

Of course, the intention is that the two definitions for lifting of values and types are consistent:

Proposition 2.31. $\Gamma; \Sigma \vdash_v v : \tau$ implies $\lceil \Gamma \rceil; \Sigma \vdash_l \lceil v \rceil : \lceil \tau \rceil$.

Proof. By case analysis on the structure of the value. The case for constants and variables is trivial; we simply have to observe that lifting and type substitution commutes. We have

$$\lceil \tau \phi \rceil = \lceil \tau \rceil \phi$$

for any type, effect or region substitution ϕ . The case for memory locations is equally trivial. For the case of recursive functions, observe that p and q already have the expected types of the first and second component of the pair (see the REC rule), or almost; we just have to add the abstraction over the argument x of the function. \square

Proposition 2.32. *Lifting and value substitution commute:*

$$\lceil v[x \mapsto \bar{\chi}.v'] \rceil = \lceil v \rceil[x \mapsto \bar{\chi}. \lceil v' \rceil]$$

Proof. Trivial; simply unfold the definition of $\lceil \cdot \rceil$ and of the substitution. \square

Validity of a formula with respect to a state. Formulas refer to the state of a program using state objects, whose types are of the form $\langle \varphi \rangle$, as we have seen. In the next chapter, we link the semantics of \mathbb{W} to the semantics of \mathbb{L} . In particular, this requires to be able to interpret the store s in the logic. Let us remark that the domain of a store, a set of regions, can be interpreted as an effect expression.

$\boxed{[s]}$

Definition 2.33. If s is a store of domain φ , we define $\lceil s \rceil$ to be a state object of type $\langle \varphi \rangle$ such that all well-typed instances of the following axiom are true:

$$\text{get}_r s l = \lceil s(r)(l) \rceil.$$

We can argue, using the program typing rule LOC and Proposition 2.31, that $\lceil s(r)(l) \rceil$ is of the expected type, namely if l is of (program) type $\text{ref}_r \tau$ for some region r , then $\lceil s(r)(l) \rceil$ is of type $\lceil \tau \rceil$.

The store s in which a program e evolves is not directly available; it is outside of the program, and it is subject to change. In the proofs of Chapter 3, we still need to refer to the “current” store sometimes; we therefore introduce a special constant ϑ that stands for the current store; its type is determined by the store typing Σ . We now can define the validity of a formula w.r.t. a store:

Definition 2.34. Let s be a store, $\Sigma \vdash s$ and f a formula such that $\Delta; \Sigma \vdash_l f : \text{prop}$. We define $s \models f$ to be true if $f[\vartheta \mapsto \lceil s \rceil]$ is true. Said otherwise, in $s \models f$, the special term ϑ refers to the lifted store $\lceil s \rceil$.

$s \models f$

Similar to the region construct and memory locations l , the term ϑ is not available in the surface language. It appears only in the correctness proofs of Chapter 3. We will use the notation ϑ_φ to refer to the restriction of ϑ to φ .

3. A Weakest Precondition Calculus

In the previous chapter we have defined a programming language containing annotations in the form of logic formulas. Up to now we have not done anything with these annotations, other than verifying that they are well-typed. It is now time to put these annotations to use, and to ultimately prove that a given program is correct with respect to its annotations.

In this chapter, we define a function `wp`, similar to the function f of Section 1.3.1, which takes a program and a postcondition to prove and returns a valid precondition. What distinguishes a pre- and a postcondition? The state they refer to. The precondition refers to the initial state, at the beginning of the execution of the program; the postcondition refers to the state after the execution of the program. In addition, the postcondition may refer to the return value of the program. Therefore let us be more precise: we want to define a function `wp` which takes a program e of effect φ and type τ as input, and a postcondition of type $\langle\varphi\rangle \rightarrow [\tau] \rightarrow \text{prop}$, and returns a precondition, a formula of type $\langle\varphi\rangle \rightarrow \text{prop}$.

Having defined this function, we will prove its correctness with respect to the semantics of W programs that we have defined in Chapter 2. In the community of program verification, correctness usually means the combination of

- *soundness*: if the `wp` calculus computes a valid formula, the program is indeed partially correct with respect to its specification, *i.e.*, the program either does not terminate or computes a value that indeed verifies the intended postcondition;
- *completeness*: every program can be proved correct if it *is* correct, or said otherwise, if there is a precondition p that, if it is true, guarantees that a program e computes values that verify a postcondition q , then one can annotate e such that the `wp` calculus computes a formula that is implied by p (weaker than p).

3.1. The `wp` Predicate Transformer

Before delving into the details of the definition of `wp`, let us first make clear that we only consider well-typed expressions and values from now on, *i.e.*, expressions e with a derivation $\Gamma; \Sigma \vdash e : \tau, \varphi$. Therefore, it makes sense now to speak of “the type” τ and “the effect” φ of e . We rarely need to refer to the typing environment Γ or the store typing Σ , so we write $e_{\tau, \varphi}$ to refer to an expression e with type τ and effect φ . Finally, we also use the subscript notation for formulas, e.g., $p_{\langle\varphi\rangle \rightarrow \text{prop}}$ means that p is a predicate on objects of type $\langle\varphi\rangle$. Usually, programs and formulas that appear in the same context (a theorem or a proof case) are typed in the same typing environment Γ (or lifted environment $[\Gamma]$ for formulas) and store typing Σ . If we also mention a store s , we assume $\Sigma \vdash s$.

3. A Weakest Precondition Calculus

We also introduce special syntax for the function symbols concerning state:

$$\begin{aligned} s|_{\varphi} &= \text{restrict}_{\varphi} s \\ s_1 \oplus s_2 &= \text{combine } s_1 s_2 \end{aligned}$$

We have seen that **wp** should return a function of type $\langle \varphi \rangle \rightarrow \text{prop}$, taking the initial state as an argument. In practice, this would require the definition of **wp** to start with an abstraction over the state:

$$\lambda s : \langle \varphi \rangle. \dots,$$

and this for all cases of its definition. To avoid this, and to allow a simpler presentation of the **wp**-calculus, we consider a predicate transformer $\text{wp}_s(e, q)$ where s is of type $\langle \varphi \rangle$, e is of type τ and effect φ , and q is of the expected type for postconditions. The state s plays the role of the *initial* state, before executing e and replaces the initial abstraction over s . Using this modification, $\text{wp}_s(e, q)$ returns a simple formula of type **prop**, with free occurrences of the state s . To obtain a closed formula of type $\langle \varphi \rangle \rightarrow \text{prop}$, we can of course simply abstract over s :

$$\text{wp}(e, q) = \lambda s : \langle \varphi \rangle. \text{wp}_s(e, q)$$

If an expression's evaluation has terminated, it is a value. For values, the **wp**-calculus has nothing to do: the postcondition becomes the precondition. We simply have to apply the current state and the returned value to the postcondition. The value has to be lifted to the logic first.

$$\text{wp}_s(v, q) = q \ s \ [v]$$

For a function application (application of a functional value to a value), we have to prove that we have the right to do so (the precondition of the function is true) and that the postcondition of the function implies the postcondition to be established.

$$\begin{aligned} \text{wp}_s(v_{\tau' \rightarrow \varphi \tau} v', q) = \\ \text{pre } [v] \ [v'] \ s \ \wedge \ \forall s' : \langle \varphi \rangle. \ \forall x : [\tau]. \ \text{post } [v] \ [v'] \ s \ s' \ x \Rightarrow q \ s' \ x \end{aligned}$$

Let us explain. In the case where v is a function variable, say f , to establish the condition q of a function call $f \ v'$ (for example), we first have to prove the precondition of f on its argument and the current state, $\text{pre } f \ [v'] \ s$ and then prove that for any possible final state s' , and any return value x , which both have to satisfy the postcondition of f , we also have the condition q , applied to s' and x . Recall that in the logic, an effectful function is represented by the pair of its pre- and postcondition; to emphasize that these components are predicates, we use the names **pre** and **post** instead of **fst** and **snd** to access the components of the pair.

We have used the word “postcondition” with two slightly different meanings. The postcondition of an anonymous function

$$\text{rec } f \ (x). \{p\} e \{q\}$$

is the formula q ; it is a predicate over *two* states, because it abstracts over the initial state and the final state of the function call. The postcondition of an application of the **wp** calculus

$$\text{wp}_s(e, p)$$

is the formula p ; it is a predicate over a single state, the state after executing p .

Let us continue to define the wp calculus. Branching expressions are rather simple to deal with: we simply have to express that, depending on the truth value of the test, only one branch is relevant.

$$\text{wp}_s(\text{if } v \text{ then } e_1 \text{ else } e_2, q) = ([v] = \text{true} \Rightarrow \text{wp}_s(e_1, q)) \wedge ([v] = \text{false} \Rightarrow \text{wp}_s(e_2, q))$$

We have assumed that e_1 and e_2 have the same effect, which is true when the branching expression is well-typed.

A let-expression which introduces a value does not pose any problems; we simply lift the value and compute the wp predicate inside the scope of a new let-binding.

$$\text{wp}_s(\text{let } x [\bar{\chi}] = v \text{ in } e, q) = \text{let } x [\bar{\chi}] = [v] \text{ in } \text{wp}_s(t, q)$$

When the let-binding does not bind a value, but the result of an expression with side effects, it becomes slightly more complicated. We now also have to deal with the state modifications introduced by both expressions:

$$\text{wp}_s(\text{let } x = e_{\tau, \varphi} \text{ in } e', q) = \text{wp}_s(e, \lambda s' : \langle \varphi \rangle. \lambda x : [\tau]. \text{wp}_{s'}(e', q))$$

Here we assume that both e and e' have the same effect φ , which can of course always be achieved using the SUB typing rule. We first compute the weakest precondition for e' , given an intermediate state s' and the return value of e called x . Using this predicate as postcondition, we compute the wp of e in the current state s .

Let us turn to the letregion construct. The difficulty here is to deal with the local region ϱ hidden by the construct. Let's say that the expression letregion ϱ in e is of type φ , then e may have an effect on the local region ϱ , so its effect can be $\varphi \cup \varrho$.

$$\text{wp}_s(\text{letregion } \varrho \text{ in } e, q) = \forall \varrho. \varrho \notin s \Rightarrow \forall s' : \langle \varrho \rangle. \text{wp}_{s \oplus s'}(e, \lambda t : \langle \varphi \cup \varrho \rangle. q \ t_{|\varphi})$$

The trick is to quantify over any region ϱ and any state s' that only contains region ϱ . We then enrich the current state s as well as the expected state in the postcondition using s' , because e may contain effects on ϱ . In addition, we can assume that ϱ is different from all regions in s .

The rule for the region construct is the following:

$$\text{wp}_s(\text{region } r \text{ in } e, q) = \text{wp}_{s \oplus \vartheta_r}(e, \lambda t : \langle \varphi \cup r \rangle. q \ t_{|\varphi})$$

It corresponds to an instantiation of the formula for the letregion case, where the region variable ϱ has been replaced by a concrete region r . Also, the quantification over the state containing ϱ is replaced with the state ϑ_r .

In the case of if-expressions and let expressions, we have argued that applications of the SUB typing rule can render the effects of the two subexpressions equal. However, if we want our wp-calculus to be precise, applications of this rule require a special treatment. We must not lose the information that an expression whose effect has been artificially increased using SUB does in fact modify a smaller part of the state. We write $(e : \varphi <: \varphi')$ to express an application of the SUB rule where φ has been augmented to φ' .

$$\text{wp}_s(e : \varphi <: \varphi', q) = \text{wp}_{s|_{\varphi}}(e, \lambda s' : \langle \varphi \rangle. q \ (s \oplus s'))$$

3. A Weakest Precondition Calculus

$$\begin{aligned}
\text{correct}(x \ [\bar{\kappa}]) &= \text{True} \\
\text{correct}(c \ [\bar{\kappa}]) &= \text{True} \\
\text{correct}(l) &= \text{True} \\
\text{correct}(\text{rec } f \ (x : \tau).\{p\}e_{\tau',\varphi}\{q\}) &= \forall x : [\tau].\forall f : [\tau \rightarrow^\varphi \tau'].f = [\text{rec } f \ (x : \tau) \dots] \\
&\Rightarrow \forall s : \langle \varphi \rangle.p \ s \Rightarrow \text{wp}_s(e, q \ s) \\
\text{wp}_s(v, q) &= \text{correct}(v) \wedge q \ s \ [v] \\
\text{wp}_s(v_{\tau' \rightarrow \varphi \tau} \ v', q) &= \text{correct}(v) \wedge \text{correct}(v') \wedge \text{pre } [v] \ [v'] \ s \wedge \\
&\forall s' : \langle \varphi \rangle.\forall x : [\tau].\text{post } [v] \ [v'] \ s \ s' \ x \Rightarrow q \ s' \ x \\
\text{wp}_s(\text{let } x \ [\bar{\chi}] = v \ \text{in } e, q) &= \forall \bar{\chi}.\text{correct}(v) \wedge \text{let } x \ [\bar{\chi}] = [v] \ \text{in } \text{wp}_s(e, q) \\
\text{wp}_s(\text{let } x = e_{\tau,\varphi} \ \text{in } e', q) &= \text{wp}_s(e, \lambda s' : \langle \varphi \rangle.\lambda x : [\tau].\text{wp}_{s'}(e', q)) \\
\text{wp}_s(\text{if } v \ \text{then } e_1 \ \text{else } e_2, q) &= ([v] = \text{true} \Rightarrow \text{wp}_s(e_1, q)) \wedge \\
&([v] = \text{false} \Rightarrow \text{wp}_s(e_2, q)) \\
\text{wp}_s(\text{letregion } \varrho \ \text{in } e, q) &= \forall \varrho.\varrho \notin s \Rightarrow \forall s' : \langle \varrho \rangle.\text{wp}_{s \oplus s'}(e, \lambda t : \langle \varphi \cup \varrho \rangle.q \ t_{|\varphi}) \\
\text{wp}_s(\text{region } r \ \text{in } e, q) &= \text{wp}_{s \oplus \vartheta_r}(e, \lambda t : \langle \varphi \cup r \rangle.q \ t_{|\varphi}) \\
\text{wp}_s(e : \varphi <: \varphi', q) &= \text{wp}_{s|_{\varphi}}(e, \lambda s' : \langle \varphi \rangle.q \ (s \oplus s'))
\end{aligned}$$

Figure 3.1: The weakest precondition calculus.

We restrict the current state s to the actual effect of e when calling wp recursively; also, we build a new postcondition by abstracting over a state s' with domain φ , and pass the combined state $s \oplus s'$ to q ; we therefore guarantee that the part of the store that is not modified by e , basically $\varphi' \setminus \varphi$, is the same in s and $s \oplus s'$.

Correctness obligations. There is something missing in our wp -calculus. Let us look at a simple example to see the problem. Let us call id the following anonymous function:

$$\text{id} = \text{rec } f \ (x : \text{int}).\{\lambda s.\text{True}\}x\{\lambda s.\lambda s'.\lambda r.\text{False}\}$$

It is essentially the identity function, but the specification is rather strange. Let us not worry about it; instead, let us apply this function to the integer 0; the result should of course be 0 again. However, we are able to prove that the result is 1:

$$\begin{aligned}
\text{wp}_s(\text{id } 0, \lambda s' \lambda r.r = 1) &= \text{pre } \text{id } 0 \ s \wedge \forall s'.\forall r.\text{post } \text{id } 0 \ s \ s' \ r \Rightarrow (\lambda s' \lambda r.r = 1) \ s' \ r \\
&\Leftrightarrow (\lambda x.\lambda s.\text{True}) \ 0 \ s \wedge \\
&\quad \forall s'.\forall r.(\lambda x.\lambda s.\lambda s'.\lambda r.\text{False}) \ 0 \ s \ s' \ r \Rightarrow r = 1 \\
&\Leftrightarrow \text{True} \wedge \forall s'.\forall r.\text{False} \Rightarrow r = 1
\end{aligned}$$

The last line can be easily proved because of the **False** premise. This means that in its current form, the wp calculus is not sound.

The problem is of course the specification of id itself, which is already wrong. We need a way to ensure that only correct values appear in the program. We therefore define a formula $\text{correct}(v)$, which, for any value v , expresses its correctness.

For variables, constants and memory locations, there is nothing to do; all these values are trivially correct and $\text{correct}(v)$ is True in this case. In the remaining case for anonymous recursive functions, there is some work to do. We have to guarantee that the specification of the function is coherent with its body. More precisely, when the precondition is true for some initial state, we want the postcondition to be true for any final state and result. This sounds familiar, and indeed, the most natural formulation of the correctness of values involves the wp-calculus itself. If we set

$$v = \text{rec } f (x : \tau) . \{p\} e_{\tau', \varphi} \{q\},$$

then

$$\text{correct}(v) = \forall x : [\tau] . \forall f : [\tau \rightarrow^\varphi \tau'] . f = [v] \Rightarrow \forall s : \langle \varphi \rangle . p \ s \Rightarrow \text{wp}_s(e, q \ s)$$

This requires a bit of explanation. On the right hand side, at the right of the first implication, we have the expected formula: for any state s , the precondition on s implies the weakest precondition of the body of the function and the postcondition, evaluated at s . Note that the postcondition q is applied to s before being passed to the wp function; this allows it to refer to the *initial* state of the function call. The part which precedes this formula may be surprising: we quantify not only over the argument x , but also over the recursive name f of the function. Additionally, we assume that f is equal to $[v]$. What does this mean?

First of all, as the function is potentially recursive, we need to quantify over f , as the formula obtained by wp may refer to f . What can we say about these recursive calls to f ? We claim (and will later prove) that it is sound to assume the correctness of the specifications of f for recursive calls to establish the adequacy of the body of f with respect to its specification — and ultimately the correctness of f itself. We assert that the specification of f is correct by stating the equality $f = [v]$. By unfolding $[v]$, we see that this corresponds to

$$f = (\lambda x . p, \lambda x . q),$$

so we actually *equate* f with its specification.

This seems to be a circular argument, but it really is not. We only assume the correctness of f for *recursive calls*, not for the body itself. In the context of *partial correctness*, this is enough. The reader does not need to believe us; we will set out to prove this soon.

Now, to connect the generation of correctness formulas with the generation of weakest preconditions, we simply require that each value in a program needs to be correct; we enrich the wp-calculus with the corresponding conjunctions. Fig. 3.1 summarizes the definitions of correct and wp .

The fact that we add correctness formulas for all functions in the program text poses a problem for the completeness proof. In the extreme case, this means that, if an unused function is incorrectly annotated, then the correctness of the program cannot be proved. Here is an example:

```
let f x = { True } x { False } in
1
```

We cannot prove that this program always returns 1, even if this property is trivial,

3. A Weakest Precondition Calculus

simply because we also have to prove the correctness of the annotations of f , which are trivially wrong. We see in Section 3.3 how the completeness proof deals with this problem.

Well-definedness of wp. There is an ambiguity in our definition of wp. If our program contains a monomorphic let-binding of a value, say $\text{let } x = v \text{ in } e$, which of the two rules concerning let applies? We briefly show that it does not matter which rule is applied, as in this case they collapse to equivalent formulas. In the case of the rule concerning the polymorphic let, we derive:

$$\text{wp}_s(\text{let } x = v \text{ in } e, q) = \text{correct}(v) \wedge \text{let } x = [v] \text{ in } \text{wp}_s(e, q)$$

and in the monomorphic case:

$$\begin{aligned} \text{wp}_s(\text{let } x = v \text{ in } e, q) &= \text{wp}_s(v, \lambda s' \langle \varphi \rangle. \lambda x : [\tau]. \text{wp}_{s'}(e, q)) \\ &= \text{correct}(v) \wedge (\lambda s' \langle \varphi \rangle. \lambda x [\tau]. \text{wp}_{s'}(e, q)) s [v] \\ &\Leftrightarrow \text{correct}(v) \wedge \text{wp}_s(e, q)[x \mapsto [v]] \\ &\Leftrightarrow \text{correct}(v) \wedge \text{let } x = [v] \text{ in } \text{wp}_s(e, q) \end{aligned}$$

References. The wp calculus does not mention regions or references explicitly. How can we deal with operations on references? In Chapter 2, we have seen that the operations concerning creation, assignment and reading of references are *not* built into the language; instead, they are modeled using the function constants ref , $!$, $:=$ and $:=_{r,l,\tau}$. And just as we had to give additional information concerning the semantics and the types of these constants, we now have to explain the meaning of these functions *in the logic*. We do so by defining the pre- and postconditions for each of these functions. Actually, all four preconditions are equal to True, so let us just fix the postconditions. For the four functions, we use state names s and s' to refer to the current (final) state and the initial state, and r to refer to the result.

$$\begin{aligned} \text{post } \text{ref } v \ s' \ s \ r &\Leftrightarrow r \notin s \wedge s = \text{set } r \ v \ s' \\ \text{post } ! \ x \ s' \ s \ r &\Leftrightarrow r = !!x \ s' \wedge s = s' \end{aligned}$$

The postcondition of ref states that the new state is equivalent to the old state, except that the location denoted by r is fresh, and is set to the value v . The postcondition of $!$ states that the initial and the final state are equal¹ and that the result of the memory access is precisely the content of the memory at the position described by the argument x .

For the assignment functions, the case of $:=_{r,l,\tau}$ is similar to the one of ref :

$$\text{post } :=_{r,l,\tau} \ v \ s' \ s \ r \Leftrightarrow s = \text{set } l \ v \ s'$$

For the assignment function $:=$, things are a bit more complicated, because the return value is a function.

$$\text{post } := \ x \ s' \ s \ g \Leftrightarrow g = (\text{True}, \lambda v. \lambda s'. \lambda s. \lambda r. s = \text{set } x \ v \ s')$$

¹We show in section Section 3.4.2 that a simple extension of our system, that distinguishes between read and write effects, does not need equality between states.

Here we state that the return value of $:=$ applied to a reference x is a function with precondition True and a postcondition that is identical to the one of $:=_{r,l,\tau}$.

Examples. Having given the specifications for the functions manipulating references, we can show some examples of a computation of a wp formula. We first consider the trivial example of an access to a reference. We read the reference x and we want to guarantee that the result of this access is the integer 1. Of course, we would like to obtain a precondition stating that x must contain 1 as well. We assume that x is an integer reference in region ϱ ; stated otherwise, x has type $\text{ref}_\varrho \text{ int}$. Here is the derivation, that combines pure unfolding of the definition of wp with simplifications of the obtained formula:

$$\begin{aligned} \text{wp}_s(!x, \lambda s' : \langle \varrho \rangle . \lambda r . r = 1) &= \text{correct}(!) \wedge \text{correct}(x) \wedge \text{pre} ! s x \wedge \\ &\quad \forall s' : \langle \varrho \rangle . \forall r . \text{post} ! x s s' r \Rightarrow r = 1 \\ &\Leftrightarrow \forall s' : \langle \varrho \rangle . \forall r . r = !x s \wedge s = s' \Rightarrow r = 1 \\ &\Leftrightarrow \forall r . r = !x s \Rightarrow r = 1 \\ &\Leftrightarrow !x s = 1 \end{aligned}$$

From the first line to the second line, we remove the parts of the formula that evaluate to True and unfold the formula $\text{post} ! x s s' r$. In the second step, we simplify the formula by removing the useless quantification over s' . Finally, we remove the quantification over r that is immediately followed by an equation defining r . The obtained precondition clearly states that in s , the state before executing the read, the reference x must contain 1 for the expected postcondition to be true.

Let us turn to a slightly more complicated case: assignment. We assign 1 to the reference x and state that x now contains 1. Of course, we want to obtain a precondition that is trivially true. Again, we combine unfolding of wp with logical simplifications. Remember that programs in \mathcal{W} are in A-normal form, so that $x := 1$ is actually a short-cut for $\text{let } f = (:= x) \text{ in } f \ 1$. We set q to be the desired postcondition:

$$q = \lambda s' : \langle \varrho \rangle . \lambda r : \text{unit} . !x s' = 1$$

We start with the unfolding of the definition of wp for let bindings:

$$\text{wp}_s(\text{let } f = (:= x) \text{ in } f \ 1, q) = \text{wp}_s(:= x, \lambda s' : \langle \varrho \rangle . \lambda f . \text{wp}_{s'}(f \ 1, q))$$

We now set

$$p = \lambda s' : \langle \varrho \rangle . \lambda f . \text{wp}_{s'}(f \ 1, q)$$

and observe that $:= x$ actually has empty effect. Therefore, we have to use the subeffecting rule, and we obtain

$$\text{wp}_\emptyset(:= x, \lambda s' : \langle \rangle . p (s \oplus s'))$$

As s' is the empty state, we can simplify $s \oplus s'$ to s and we obtain, unfolding the application case for wp and omitting trivial parts,

$$\forall s' : \langle \rangle . \forall f . \text{post} := x \ \emptyset \ s' \ r \Rightarrow p \ f \ s$$

3. A Weakest Precondition Calculus

Unfolding p again and applying the rule for application, we obtain

$$\forall s' : \langle \rangle. \forall f. \text{post} := x \ \emptyset \ s' \ f \Rightarrow \forall s'' : \langle \varrho \rangle. \forall r. \text{post} \ f \ 1 \ x \ s \ s'' \ r \Rightarrow !!x \ s'' = 1$$

Replacing $\text{post} :=$ by its definition, we obtain that

$$f = (\text{True}, \lambda v. \lambda s'. \lambda s. \lambda r. s = \text{set } x \ v \ s')$$

Rewriting this equation and dropping the useless quantification over s' , we obtain

$$\forall s'' : \langle \varrho \rangle. \forall r. s'' = \text{set } x \ 1 \ s \Rightarrow !!x \ s'' = 1$$

and this formula is valid.

This second example also shows that our wp calculus is similar to Dijkstra's (Dijkstra, 1975), even though it is not exactly the same rule, due to the indirection of the treatment of assignment, passing through state objects. In the same manner, our wp calculus has a similar form concerning other language constructs, such as the sequence and function application.

3.2. Soundness of the wp Calculus

We now want to prove several properties of our calculus, most importantly soundness. As stated in the introduction of this chapter, soundness states that if one establishes the wp of a program e and a postcondition q , then q is indeed true after executing e . The proof is very similar to the subject reduction proof in Chapter 2; we prove that each reduction step preserves the validity of the weakest precondition.

We recall that the letter ϑ stands for “the current state” and its domain depends on the store typing Σ . We also recall the notation ϑ_φ which stands for the restriction of ϑ to the domain φ . However, we will often omit this restriction when it is obvious from the context. At a few points in the proof, we annotate ϑ correctly to emphasize that the effect restriction changes.

We start by stating the obvious, namely that the wp calculus is *stable* with respect to type, effect and region substitutions.

Lemma 3.1 (Type Substitution Lemma). *The weakest precondition calculus and type, effect and region substitutions commute. Stated more formally, for any substitution ϕ ,*

$$\text{wp}_s(e, q)\phi \Leftrightarrow \text{wp}_s(e\phi, q\phi).$$

Proof. We omit this very simple and mechanical proof. □

The next step basically corresponds to the Substitution Lemma for type soundness (Lemma 2.19). We prove that if a value v is correct, then one can either compute the wp on e , and substitute $[v]$ for x in the result, or substitute v for x in e , and compute the wp .

Lemma 3.2 (Substitution Lemma for the wp calculus). *The weakest precondition calculus and value substitutions commute for correct values. The same is true for correctness assertions for values. Stated more formally, if $\forall \bar{x}.\text{correct}(v)$ is true, then*

$$\text{wp}_s(e, q)[x \mapsto \Lambda \bar{x}.\lceil v \rceil] \quad \Leftrightarrow \quad \text{wp}_s(e[x \mapsto \Lambda \bar{x}.v], q[x \mapsto \Lambda \bar{x}.\lceil v \rceil]).$$

and

$$\text{correct}(v')[x \mapsto \Lambda \bar{x}.\lceil v \rceil] \quad \Leftrightarrow \quad \text{correct}(v'[x \mapsto \Lambda \bar{x}.v])$$

Note how the substitution of $\lceil v \rceil$ becomes a substitution of v in program expressions.

Proof. We first prove the claim about expressions, assuming the one about values. We proceed by induction over the structure of e . We set $\phi = [x \mapsto \Lambda \bar{x}.\lceil v \rceil]$, but we will use it for substitution in logic terms *and* program expressions, where it actually substitutes $\Lambda \bar{x}.v$. It should always be clear which type of substitution is intended.

Case v Let $e = v'$. We derive

$$\begin{aligned} \text{wp}_s(v', q)\phi &= (\text{correct}(v') \wedge q \text{ s } [v'])\phi \\ &= \text{correct}(v')\phi \wedge q\phi \text{ s } [v']\phi \\ &= \text{correct}(v'\phi) \wedge q\phi \text{ s } [v'\phi] \\ &= \text{wp}_s(v'\phi, q\phi) \end{aligned}$$

We have used the claim about correctness and Proposition 2.32 to obtain the third line from the second.

Case v v Similarly, we have to unfold the definition of wp, push the substitution down, apply the claim about correctness and Proposition 2.32 and fold the definition of wp again.

Case polymorphic let, letregion, region, if, subeffecting Again, unfold the definition of wp, push the substitution down, and apply, as necessary, the claim about correctness, Proposition 2.32 and the induction hypothesis on subexpressions.

Case monomorphic let Let $e = \text{let } y = e_1 \text{ in } e_2$. We derive

$$\begin{aligned} \text{wp}_s(e, q)\phi &= \text{wp}_s(e_1, \lambda s : \langle \varphi \rangle.\lambda y : [\tau].\text{wp}_s(e_2, q))\phi \\ &= \text{wp}_s(e_1\phi, \lambda s : \langle \varphi \rangle.\lambda y : [\tau].\text{wp}_s(e_2, q)\phi) \\ &= \text{wp}_s(e_1\phi, \lambda s : \langle \varphi \rangle.\lambda y : [\tau].\text{wp}_s(e_2\phi, q\phi)) \\ &= \text{wp}_s(e\phi, q\phi) \end{aligned}$$

We have used the induction hypotheses for e_1 (second line) and e_2 (third line).

Now let us turn to the claim concerning correctness formulas for values. The claim is trivial for variables different from x , constants and memory locations, because they are not subject to the substitution, and they are trivially correct. Two cases remain.

3. A Weakest Precondition Calculus

Case $v' = x [\bar{\kappa}]$ We have

$$\begin{aligned} \text{correct}(v'\phi) &= \text{correct}(v[\bar{\chi} \mapsto \bar{\kappa}]) \\ &\Leftrightarrow \text{correct}(v)[\bar{\chi} \mapsto \bar{\varkappa}] \\ &\Leftrightarrow \text{True} \\ &\Leftrightarrow \text{correct}(v)\phi \end{aligned}$$

We have used Lemma 3.1 to obtain the second line. We can state that the second line is equivalent to `True` because we additionally have the hypothesis that $\forall \bar{\chi}.\text{correct}(v)$ holds.

Case of anonymous functions For recursive functions, the proof is similar to the cases showed above: unfold the definition of `correct`, push the substitution down, apply the claim for `wp` and finally fold back the definition of `correct`.

□

The next lemma states the quite intuitive fact that if we want to prove a stronger postcondition of our program, we need to prove a stronger precondition.

Lemma 3.3 (Weakening Lemma for the `wp` calculus). *For any expression $e_{\tau,\varphi}$, and for any q_1 and q_2 of type $\langle\varphi\rangle \rightarrow [\tau] \rightarrow \text{prop}$, if q_1 is stronger than q_2 :*

$$\forall s : \langle\varphi\rangle.\forall r : [\tau].q_1 \ s \ r \Rightarrow q_2 \ s \ r$$

then the weakest precondition of e and q_1 is stronger than the weakest precondition of e and q_2 :

$$\forall s : \langle\varphi\rangle.\text{wp}_s(e, q_1) \Rightarrow \text{wp}_s(e, q_2)$$

Proof. We proceed by induction over the structure of e . This lemma is simple to prove in most cases.

Cases v and $v \ v$ In this case, no recursive call takes place; the claim stems from the fact that the postcondition q is only used in positive position.

Cases `letregion`, `region`, `polymorphic let`, `if` and `subeffecting` In these cases, the claim can be established using the induction hypothesis for the recursive call, which appears in positive position.

Case `monomorphic let` This case is not more difficult, but more interesting than the others. We conduct the proof in more detail. We set $e = \text{let } x = e_1 \text{ in } e_2$ and we have

$$\text{wp}_s(e, q_1) = \text{wp}_s(e_1, \lambda s' : \langle\varphi\rangle.\lambda x : [\tau].\text{wp}_{s'}(e_2, q_1)).$$

Using the induction hypothesis for e_2 , and setting $f(q) = \text{wp}_{s'}(e_2, q)$, we can prove that

$$f(q_1) \Rightarrow f(q_2)$$

and also, setting $g(q) = \lambda s' : \langle \varphi \rangle . \lambda x : [\tau] . f(q)$, that

$$\forall s : \langle \varphi \rangle . \forall r : [\tau] . g(q_1) s r \Rightarrow g(q_2) s r$$

This is the required condition to apply the induction hypothesis for e_1 , which gives us the required result. □

Remark. Using Lemma 3.3, we can also prove that equivalent postconditions q_1 and q_2 lead to equivalent formulas $\mathbf{wp}_s(e, q_1)$ and $\mathbf{wp}_s(e, q_2)$. We simply split the equivalence between q_1 and q_2 into two implications, use the lemma on each implication, and obtain two implications in opposite directions between the wp formulas.

After these preliminary lemmas, we now proceed to the actual soundness proof. We want to show that if the weakest precondition of s, e and q is true, then the weakest precondition is true for any s', e' that can be attained from the configuration s, e . As we did already in Chapter 2, the first lemma deals with the (δ) rule. An additional hypothesis states that every value in the store s is correct. This requirement is obviously needed when proving the soundness of the specification of the dereferencing operator.

Definition 3.4. A store s is *correct*, noted $\mathbf{correct}(s)$, if all values in the store are correct. More formally, for all regions r and locations l , we can prove $\mathbf{correct}(s(r)(l))$ if $s(r)(l)$ is defined. correct (s)

Lemma 3.5. All three reduction relations \rightarrow , \longrightarrow and \rightarrow preserve the correctness of the store, if for the initial expression e , the formula $\mathbf{wp}_\vartheta(e, \mathbf{True})$ holds.

Proof. We simply observe that in the only two kinds of expressions that alter the store, namely $\mathit{ref} v$ and $:=_l v$, the correctness of v is implied by $\mathbf{wp}_s(\vartheta, e) \mathbf{True}$. Of course, we must again assume that δ does also have this property. □

We now proceed to prove the correctness property for each of the reduction relations. As in Section 2.1.4, we start by an assumption on the function δ .

Hypothesis 3.6. Let $e = c[\bar{\kappa}] v$ be a well-typed expression and s a correct state such that $\delta(s, c[\bar{\kappa}], v) = s', v'$ is defined. Then if $s \models \mathbf{wp}_\vartheta(e, q)$, we also have $s' \models \mathbf{wp}_\vartheta(v', q)$.

Proof for the constants of Definition 2.5. We proceed by case analysis on the form of the constant c .

Case ! Let $e = ![\tau r] l$. We have $s = s'$ and $v' = s(r)(l)$. The hypothesis can be unfolded:

$$s \models \mathbf{wp}_\vartheta(e, q) \Leftrightarrow s \models \forall r . r =!! l \vartheta \Rightarrow q \vartheta r.$$

Thanks to Definition 2.33 and Definition 2.34, we can simplify this to

$$s \models q \vartheta [s(r)(l)].$$

This corresponds to half of the claim, because we also need to prove $\mathbf{correct}(v')$. The assumption about the correctness of the store s guarantees this.

3. A Weakest Precondition Calculus

Case *ref* Let $e = \text{ref}[\tau r] v$. We have $s' = s[r \mapsto s(r)[l \mapsto v]]$ for some fresh l . We also have $v' = l$. Our hypothesis becomes

$$\begin{aligned} s \models \text{wp}_\vartheta(e, q) &\Leftrightarrow s \models \forall s'. r \notin s \wedge s' = \text{set } r [v] \vartheta \Rightarrow q \ s' \ r \\ &\Leftrightarrow s \models \forall r. r \notin s \Rightarrow q \ (\text{set } r [v] \vartheta) \ r \\ &\Rightarrow s \models q \ (\text{set } l [v] \vartheta) \ l \end{aligned}$$

and we have to prove:

$$s' \models \text{wp}_\vartheta(l, q) \Leftrightarrow s' \models q \ \vartheta \ l$$

Making the shift from s to s' , we see that the hypothesis implies the claim.

Case := This case is trivial.

Case := _{r, l, τ} This case is very similar to the one of *ref*.

□

We now prove the same claim for the reduction relation \rightarrow .

Lemma 3.7 (Soundness of Top Step). *Let $e_{\tau, \varphi}$ be a well-typed expression and s be a correct store such that $s, e \rightarrow s', e'$. Then if $s \models \text{wp}_\vartheta(e, q)$, we also have $s' \models \text{wp}_\vartheta(e', q)$.*

Proof. By case analysis over the last rule of the typing derivation of e . We do not need to consider the **VALUE** rule, because e is not a value.

Case APP We know that $e = v_1 v_2$. The value v_1 is either a constant c , or an anonymous function. In the first case, we can apply Hypothesis 3.6 and we are done. In the other case, we have $v_1 = \text{rec } f (x : \tau'). \{p_f\} e_f \{q_f\}$.

Let us unfold the definition for **wp** on e :

$$s \models \text{wp}_\vartheta(e, q) \Leftrightarrow s \models \text{pre } [v_1] [v_2] \vartheta \wedge \forall r : [\tau]. \forall t : \langle \varphi \rangle. \text{post } [v_1] [v_2] \vartheta \ t \ r \Rightarrow q \ t \ r$$

which is equivalent to

$$s \models p_f[x \mapsto [v_2]] \ \vartheta \ \wedge \ \forall r : [\tau]. \forall t : \langle \varphi \rangle. q_f[x \mapsto [v_2]] \ \vartheta \ t \ r \Rightarrow q \ t \ r \quad (3.1)$$

We have omitted the properties of correctness for v_1 and v_2 . We have to show that

$$s \models \text{wp}_\vartheta(e_f[x \mapsto v_2, f \mapsto v_1], q)$$

Now let us look more closely at the correctness statement for v_1 :

$$\text{correct}(v_1) = \forall x : [\tau']. \forall f : [\tau' \rightarrow^\varphi \tau]. f = [v_1] \Rightarrow \forall s : \langle \varphi \rangle. p_f \ s \Rightarrow \text{wp}_s(e_f, q_f \ s)$$

Let us instantiate this formula with $x = [v_2]$ and $f = [v_1]$. The first part $f = [v_1]$ is trivially true and we have established

$$\forall s : \langle \varphi \rangle. p_f[x \mapsto [v_2]] \ s \Rightarrow \text{wp}_s(e_f, q_f \ s)[x \mapsto [v_2], f \mapsto [v_1]]$$

If we again instantiate this formula with $s = \vartheta$, we see that the premise of the implication is true by (3.1), so we have

$$s \models \mathbf{wp}_\vartheta(e_f, q_f \vartheta)[x \mapsto [v_2], f \mapsto [v_1]].$$

Let us push the substitution inside the wp using Lemma 3.2 and the correctness of both values:

$$s \models \mathbf{wp}_\vartheta(e_f[x \mapsto v_2, f \mapsto v_1], q_f[x \mapsto [v_2]] \vartheta).$$

We have used the fact that the variable f cannot appear in q_f . This last formula looks almost the same as our goal, except the appearance of q_f instead of q . Fortunately, we know that $q_f[x \mapsto [v_2]] \vartheta$ is stronger than q because of (3.1), so we can conclude by Lemma 3.3.

Case LETPOLY We know that e is of the form $\text{let } x \ [\bar{\chi}] = v \text{ in } e_1$. We also know that $e' = e_1[x \mapsto \Lambda\bar{\chi}.v]$. Let us unfold the hypothesis:

$$s \models \mathbf{wp}_\vartheta(e, q) \quad \Leftrightarrow \quad s \models \forall \bar{\chi}. \text{correct}(v) \wedge \text{let } x \ [\bar{\chi}] = [v] \text{ in } \mathbf{wp}_\vartheta(e_1, q)$$

In this case, the soundness result is trivial: we simply remark that in L, a let statement is equivalent to the polymorphic substitution:

$$s \models \mathbf{wp}_\vartheta(e_1, q)[x \mapsto \Lambda\bar{\chi}.v]$$

Using the correctness of v and Lemma 3.2, we can easily prove the goal:

$$s \models \mathbf{wp}_\vartheta(e_1[x \mapsto \Lambda\bar{\chi}.v], q).$$

In particular, we have used the fact that q cannot contain x .

Case LET Our remark on the well-definedness of the wp calculus shows that this case is identical to the previous one.

Case IF We know that $e = \text{if } v \text{ then } e_1 \text{ else } e_2$. We also know that $v = \text{true}$ or $v = \text{false}$, depending on which reduction rule applies. Finally, we have $s = s'$. Let us only consider the case where $v = \text{true}$, the other one is symmetric. We have

$$s \models \mathbf{wp}_\vartheta(e, q) \quad \Rightarrow \quad s \models [v] = \text{true} \Rightarrow \mathbf{wp}_\vartheta(e_1, q).$$

As we have assumed that $v = \text{true}$, we can conclude.

In the following two cases we must be more careful about the domain of ϑ .

Case LETREG We know that $e = \text{letregion } \varrho \text{ in } e_1$. Assume the effect of e_1 to be φ' , with $\varphi = \varphi' \setminus \varrho$. We unfold the hypothesis:

$$s \models \mathbf{wp}_{\vartheta_\varphi}(e, q) \quad \Leftrightarrow \quad s \models \forall \varrho. \varrho \notin \vartheta_\varphi \Rightarrow \forall s : \langle \varrho \rangle. \mathbf{wp}_{\vartheta_\varphi \oplus s}(e_1, \lambda t : \langle \varphi' \rangle. q \ t|_\varphi) \quad (3.2)$$

We also know that $e' = \text{region } r \text{ in } e_1[\varrho \mapsto r]$, where r is the fresh region created by the reduction. Our goal is to prove

$$s' \models \mathbf{wp}_{\vartheta_\varphi}(e', q),$$

3. A Weakest Precondition Calculus

which is equivalent to

$$s' \models \mathbf{wp}_{\vartheta_\varphi \oplus \vartheta_r}(e_1[\varrho \mapsto r], \lambda t : \langle \varphi'[\varrho \mapsto r] \rangle . q \ t|_\varphi)$$

This can be rewritten to

$$s' \models \mathbf{wp}_{\vartheta_\varphi \oplus \vartheta_r}(e_1, \lambda t : \langle \varphi' \rangle . q \ t|_\varphi)[\varrho \mapsto r]$$

using Lemma 3.1 and the fact that q does not contain ϱ (it is out of the scope of the `letregion` binder). Finally, we argue that because r is fresh with respect to s , the hypothesis cannot mention r .² As s and s' only differ by a new region in r , we can carry over the hypothesis (3.2) from s to s' . The reformulated goal is now only an instantiation of (3.2), substituting r for ϱ and ϑ_r for s . The premise $r \notin \vartheta_\varphi$ of (3.2) is of course true, because r is fresh.

Case REGION We know that $e = \mathbf{region} \ r$ in v . Although we know that v can be typed without any effect, it would be incorrect to assume that the effect of v is empty. In particular, this would assume that no `SUB` rule has been applied before. We therefore assume the effect of v to be φ' , with $\varphi = \varphi' \setminus r$. Now let us unfold the hypothesis:

$$s \models \mathbf{wp}_{\vartheta_\varphi}(e, q) \Leftrightarrow \mathbf{wp}_{\vartheta_\varphi \oplus \vartheta_r}(v, \lambda t : \langle \varphi' \rangle . q \ t|_\varphi) \Leftrightarrow q \ \vartheta_\varphi \ [v] \wedge c(v)$$

We can conclude:

$$s' \models \mathbf{wp}_{\vartheta_\varphi}(e', q) \Leftrightarrow q \ \vartheta_\varphi \ [v] \wedge c(v)$$

because $e' = v$.

Case SUB We know that $e = (e_1 : \varphi' <: \varphi)$ with $s, e_1 \mapsto s', e'_1$. Let us unfold the hypothesis:

$$s \models \mathbf{wp}_{\vartheta_\varphi}(e, q) \Leftrightarrow s \models \mathbf{wp}_{\vartheta_{\varphi'}}(e_1, \lambda s : \langle \varphi' \rangle . q \ (\vartheta_\varphi \oplus s))$$

and the goal:

$$s' \models \mathbf{wp}_{\vartheta_\varphi}(e', q) \Leftrightarrow s' \models \mathbf{wp}_{\vartheta_{\varphi'}}(e'_1, \lambda s : \langle \varphi' \rangle . q \ (\vartheta_\varphi \oplus s))$$

The induction hypothesis is:

$$\forall f. s \models \mathbf{wp}_{\vartheta_{\varphi'}}(e_1, f) \Rightarrow s' \models \mathbf{wp}_{\vartheta_{\varphi'}}(e'_1, f)$$

and it allows us to conclude. □

We now proceed to prove the same property for the reduction relation \longrightarrow .

²Remember that we assume e and q to be well-typed in the same environment Γ and store typing Σ . As we also have $\Sigma \vdash s$, and r is fresh with respect to s , r must also be fresh with respect to q .

Lemma 3.8 (Soundness of One Step). *Let $e_{\tau, \varphi}$ be a well-typed expression and s a correct store such that $s, e \longrightarrow s', e'$. Then if $s \models \text{wp}_{\vartheta_{\varphi}}(e, q)$, we also have $s' \models \text{wp}_{\vartheta_{\varphi}}(e', q)$.*

Proof. We prove this lemma by induction over the form of reduction contexts. If the context is empty, *i.e.*, of the form \square , then the claim is already covered by Lemma 3.7. There are two cases remaining.

Case region We have

$$s, \text{region } r \text{ in } E[e] \longrightarrow s', \text{region } r \text{ in } E[e'],$$

and because of $s, e \longrightarrow s', e'$, and because of the CONTEXT rule, we also have

$$s, E[e] \longrightarrow s', E[e'].$$

We also denote by $\varphi' = \varphi \cup r$ the effect of $E[e]$. The induction hypothesis is

$$\forall f. s \models \text{wp}_{\vartheta_{\varphi'}}(E[e], f) \quad \Rightarrow \quad s' \models \text{wp}_{\vartheta_{\varphi'}}(E[e'], f) \quad (3.3)$$

Let us first unfold the hypothesis:

$$s \models \text{wp}_{\vartheta_{\varphi}}(\text{region } r \text{ in } E[e], q) \quad \Leftrightarrow \quad s \models \text{wp}_{\vartheta_{\varphi} \oplus \vartheta_r}(E[e], \lambda t : \langle \varphi' \rangle. q \ t_{|\varphi})$$

and the claim:

$$s' \models \text{wp}_{\vartheta_{\varphi}}(\text{region } r \text{ in } E[e'], q) \quad \Leftrightarrow \quad s' \models \text{wp}_{\vartheta_{\varphi} \oplus \vartheta_r}(E[e'], \lambda t : \langle \varphi' \rangle. q \ t_{|\varphi})$$

We use the fact that $\vartheta_{\varphi'} = \vartheta_{\varphi} \oplus \vartheta_r$ which follows from property (A7) of Section 2.2.3 and can conclude using (3.3).

Case let The term we are considering is of the form $e = \text{let } x \ [\bar{\chi}] = E[e_1] \text{ in } e_2$ and we have $s, e_1 \longrightarrow s', e'_1$. Therefore we also have $s, E[e_1] \longrightarrow s', E[e'_1]$, $E[e_1]$ cannot be a value and we know that the list $\bar{\chi}$ is empty. Our induction hypothesis is

$$\forall f. s \models \text{wp}_{\vartheta}(E[e_1], f) \quad \Rightarrow \quad s' \models \text{wp}_{\vartheta}(E[e'_1], f).$$

Now let us unfold the hypothesis

$$s \models \text{wp}_{\vartheta}(\text{let } x = E[e_1] \text{ in } e_2, q) \quad \Leftrightarrow \quad s \models \text{wp}_{\vartheta}(E[e_1], \lambda t : \langle \varphi \rangle. \lambda x : [\tau]. \text{wp}_t(e_2, q))$$

and the claim

$$s' \models \text{wp}_{\vartheta}(\text{let } x = E[e'_1] \text{ in } e_2, q) \quad \Leftrightarrow \quad s' \models \text{wp}_{\vartheta}(E[e'_1], \lambda t : \langle \varphi \rangle. \lambda x : [\tau]. \text{wp}_t(e_2, q))$$

Then we can conclude by induction hypothesis.

□

Finally, we can state the theorem about \twoheadrightarrow , the reflexive and transitive closure of \longrightarrow .

3. A Weakest Precondition Calculus

Theorem 3.9 (Soundness of **wp**). *Let $e_{\tau,\varphi}$ be an expression and s a correct store such that $s, e \rightarrow s', e'$. Then if $s \models \mathbf{wp}_{\vartheta}(e, q)$, we also have $s' \models \mathbf{wp}_{\vartheta}(e', q)$.*

Proof. We can now proceed by induction on the length n of the reduction sequence of $s, e \rightarrow s', e'$. If $n = 0$, the claim is trivial because $s = s'$ and $e = e'$. In the induction step, we have $s, e \rightarrow s', e'$ and $s', e' \rightarrow s'', e''$ and we have $s \models \mathbf{wp}_{\vartheta}(e, q)$. We obtain, by the induction hypothesis, that $s' \models \mathbf{wp}_{\vartheta}(e', q)$. We can conclude by Lemma 3.8. Theorem 2.23 and the accompanying lemmas help to establish the necessary well-typedness conditions and Lemma 3.5 guarantees the correctness of the store. \square

Corollary. *For any well-typed expression such that $\emptyset, e \rightarrow s', v$ and $s \models \mathbf{wp}_{\vartheta}(e, q)$, we also have $s' \models q \vartheta [v] \wedge \mathbf{correct}(v)$.*

Proof. This is a direct consequence of Theorem 3.9, the definition of **wp** and the fact that the empty store is correct. \square

3.3. Completeness

In this section, we prove the completeness result of our **wp** calculus. Completeness states a program e that is correct, *i.e.*, that is capable of guaranteeing a certain postcondition q under initial conditions described by p , can be annotated such that the **wp** of e and q is implied by p . While soundness is clearly the most important result, completeness is a close second. First, it guarantees to the user that he can prove his (correct) program using the **wp** calculus. Second, during a proof development, when a user does not manage to prove a program to be correct, then in principle four possibilities should be considered:

- The program is incorrect;
- the specification is incorrect;
- the proof attempts to prove the formula obtained by **wp** have not been sufficient;
- the program is out of scope for the **wp** calculus.

The completeness result eliminates the last point and the user can concentrate on the three other points.

A first particularity in this section is that we assume names of recursive functions not to appear in any annotations. It is already guaranteed by the typing relation that in a value v of the form $\mathbf{rec } f(x). \{p\}e\{q\}$, the formulas p and q do not contain the variable name f . Up to now, nothing did forbid annotations of local functions in e to refer to f . We now specifically disallow this. However, this restriction does not reduce the expressive power: if some formula g in e refers to f , we can simply replace f by $[v]$, namely by the pair (p, q) . The **wp** calculus does something similar anyway, by introducing the equation $f = [v]$ in the correctness formula, so this does not change much. As another remark, we also assume all stores s to be *correct*.

The first lemma we want to prove is somewhat technical and deals with recursive functions. Let us first give the intuition behind it. Say we have a recursive function f defined as follows:

```

let rec f x =
  { x = 0 }
  if x = 0 then 1 else f (x - 1)
  { r : r = 1 }

```

The specification of function f is correct; when it terminates, it always returns 1. The specification is unnecessarily restrictive, however; the postcondition is (partially) correct for any input, and not only for $x = 0$.

A mechanical way to improve the precondition of f is to use the **wp** calculus. Let us execute the **wp** calculus once over the body of f , assuming that f has the specification we have given to it. We obtain:

$$(x = 0 \Rightarrow 1 = 1) \wedge (x \neq 0 \Rightarrow (x - 1 = 0 \wedge \forall r. r = 1 \Rightarrow r = 1))$$

We can simplify the left hand side of the conjunction to **True** and the right hand side to:

$$x = 0 \vee x = 1$$

and we have indeed improved the precondition of f ; it now accepts not only 0, but also 1 as argument for x .

It is however not immediately clear that we can use the newly computed formula as precondition of f ; after all, f is recursive, so changing the specification of f changes the **wp** of its body. This is the motivation for the next lemma.

Lemma 3.10 (Improving specifications of recursive functions). *Let*

$$v = \text{rec } f(x). \{p\} e \{q\}$$

be a correct value such that the variable f does not appear in annotations in e . Then, setting

$$A = \lambda s. \text{wp}_s(e, q \ s) [f \mapsto [v]],$$

the value

$$v' = \text{rec } f(x). \{A\} e \{q\}$$

is also correct.

Remark. Before we give the proof, let us observe that that the substitution $[f \mapsto [v]]$ can be either placed, in this form, outside the **wp** transformer, or in the form $[f \mapsto v]$ inside, only applied to e . Lemma 3.2 guarantees that both are equivalent.

Proof. To establish the claim, we have to prove that v' is correct, stated otherwise, we need to prove that

$$\forall f. f = [v'] \Rightarrow \forall x s. A \ s \Rightarrow \text{wp}_s(e, q \ s).$$

Observing that A does not contain f , we unfold v' and A and substitute f to obtain:

$$\forall x s. \text{wp}_s(e, q \ s) [f \mapsto [v]] \Rightarrow \text{wp}_s(e, q \ s) [f \mapsto (A, q)].$$

We slightly generalize this claim by allowing *different* postconditions, as long as they do not contain f and one implies the other. Our new claim is that for any pair q, q' that do not contain f , if

$$\forall s, x. q \ x \ s \Rightarrow q' \ x \ s$$

3. A Weakest Precondition Calculus

then

$$\forall xs. \mathbf{wp}_s(e, q)[f \mapsto [v]] \Rightarrow \mathbf{wp}_s(e, q')[f \mapsto (A, q)]. \quad (3.4)$$

Formula (3.4) is the claim we have to prove; we add another claim about correctness:

$$\forall v_0. \mathbf{correct}(v_0)[f \mapsto [v]] \Rightarrow \mathbf{correct}(v_0)[f \mapsto (A, q)] \quad (3.5)$$

We now proceed to prove claims (3.4) and (3.5) by mutual induction over the structure of values and expressions. We start by (3.5).

Case of variables, constants and memory locations There is nothing to prove, $\mathbf{correct}(v_0)$ is always True in this case.

Case of recursive functions Set $v_0 = \mathbf{rec } g (y). \{p_0\} e_0 \{q_0\}$. We have

$$\mathbf{correct}(v_0) = \forall g. g = [v_0] \Rightarrow \forall ys. p_0 s \Rightarrow \mathbf{wp}_s(e_0, q_0 s)$$

As the specifications p_0 and q_0 do not contain f , the only place where the different substitutions can change anything is the rightmost subformula. To prove that

$$\mathbf{wp}_s(e_0, q_0 s)[f \mapsto [v]] \Rightarrow \mathbf{wp}_s(e_0, q_0 s)[f \mapsto (A, q)],$$

we simply apply the induction hypothesis on e_0 .

We now go on to prove claim (3.4). We show the proof only for selected cases.

Case of values Set $e = v_0$. Then

$$\mathbf{wp}_s(e, q) = q s [v_0] \wedge \mathbf{correct}(v_0)$$

Only the second part is subject to a substitution concerning f . But here we can use the induction hypothesis concerning correctness of values.

Case of application Let $e = v_1 v_2$. Then

$$\mathbf{wp}_s(e, q) = \mathbf{pre } [v_1] [v_2] s \wedge \forall s' r. \mathbf{post } [v_1] [v_2] s s' r \Rightarrow q s' r,$$

omitting correctness statements for v_1 and v_2 for which we can again apply the induction hypothesis. We now remark that this formula does not contain any occurrences of f , so the claim is trivial, using the implication between q and q' .

Case of let expressions We focus on the case where $e = \mathbf{let } y = e_1 \text{ in } e_2$. We have

$$\mathbf{wp}_s(e, q) = \mathbf{wp}_s(e_1, \lambda s' \lambda y. \mathbf{wp}_s(e_2, q))$$

We have to show that

$$\mathbf{wp}_s(e, q)[f \mapsto [v]] \Rightarrow \mathbf{wp}_s(e, q')[f \mapsto (A, q)]$$

Let B be the formula representing the inner \mathbf{wp} occurrence:

$$B = \mathbf{wp}_s(e_2, q)$$

We can state by the induction hypothesis that $B[f \mapsto [v]] \Rightarrow B[f \mapsto (A, q)]$. Again by the induction hypothesis, we obtain that

$$\text{wp}_s(e_1, \lambda s' \lambda y. B[f \mapsto [v]])[f \mapsto [v]] \Rightarrow \text{wp}_s(e_1, \lambda s' \lambda y. B[f \mapsto (A, q)])[f \mapsto (A, q)]$$

Let us push the substitutions inside, eliminating the double substitutions in the postconditions:

$$\text{wp}_s(e_1[f \mapsto [v]], \lambda s' \lambda y. B[f \mapsto [v]]) \Rightarrow \text{wp}_s(e_1[f \mapsto (A, q)], \lambda s' \lambda y. B[f \mapsto (A, q)])$$

We now can lift the substitutions back to obtain the claim:

$$\text{wp}_s(e_1, \lambda s' \lambda y. B)[f \mapsto [v]] \Rightarrow \text{wp}_s(e_1, \lambda s' \lambda y. B)[f \mapsto (A, q)]$$

The case where $e = \text{let } x [\bar{x}] = v \text{ in } e_2$ is very similar to the case of the (β) rule.

Case of if expressions Let $e = \text{if } v \text{ then } e_1 \text{ else } e_2$. We have

$$\text{wp}_s(e, q) = ([v] = \text{true} \Rightarrow \text{wp}_s(e_1, q)) \wedge ([v] = \text{false} \Rightarrow \text{wp}_s(e_2, q))$$

We see that the claim can be proved by applying the induction hypotheses on e_1 and e_2 .

Case of letregion expressions Let $e = \text{letregion } \varrho \text{ in } e_1$. We have

$$\text{wp}_s(e, q) = \forall \varrho'. \forall s' : \langle \varrho \rangle. \text{wp}_{s \oplus s'}(e_1, \lambda t : \langle \varphi \cup \varrho \rangle. q \ t_{|\varphi})$$

As q is stronger than q' , this is also true for the modified postconditions: set $g(q) = \lambda t : \langle \varphi \cup \varrho \rangle. q \ t_{|\varphi}$, then $g(q)$ is stronger than $g(q')$. This allows us to apply the induction hypothesis and to conclude.

Case of region expressions This case is entirely analogous to the previous one. □

We need another, less technical, proposition to prove completeness. This time, we state that annotating an expression or value is always possible. This proposition is useful when we need to annotate a value or expression that is actually not needed at all.

Proposition 3.11. *An unannotated value v can always be annotated such that $\text{correct}(V)$ is true.*

Proof. It is important to remark that this proposition is only about finding an *arbitrary* specification for expressions and values. This proposition does not try to find the best or most suitable annotation. Therefore, this proposition is trivial: values other than anonymous functions are trivially true. For anonymous functions, we can set its precondition to `False` to obtain its correctness. □

3. A Weakest Precondition Calculus

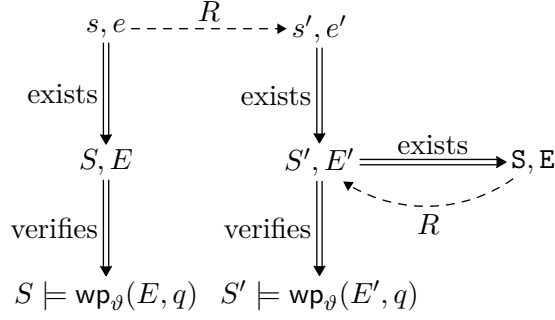


Figure 3.2: An illustration of the structure of the completeness claim.

We now can proceed to prove the completeness lemmas. Another way to express completeness is to prove that the formula computed by wp is indeed the weakest precondition. This is equivalent to saying that if a predicate p guarantees that a postcondition q is true after executing e , then the formula computed by wp is *weaker*, e.g., it is implied by p .

The technical difficulties and the relative heaviness of the statements of the following lemmas stems from the annotations. In presence of annotations, wp clearly does *not* compute the weakest precondition. As an example, consider the following definition:

```
let f x =
  { x > 0 }
  x
  { r : r = x }
```

Function f is just the identity function for integers, but artificially restricted to positive ones. Now, any precondition is sufficient to guarantee that the function call $f\ 0$ returns 0; but the wp of $f\ 0$ for the postcondition $\lambda s \lambda r. r = 0$ is equivalent to False , because of f 's precondition. So the annotations pose a problem for the statement of the lemmas. But we cannot drop them, because wp cannot be computed without annotations.

Before we continue, we again need to introduce additional notation. In the completeness proof, we consider modifications of values and expressions where only the annotations change. We therefore introduce the following convention, valid for the remainder of this section: starting from values, expressions and states written in lower case, such as v , e and s , derived annotated values and expressions are written in upper case, such as V , E and S . The operation of removing annotations from an expression is written $[e]$. We also introduce the convention that when a letter, such as e , is used in lower case and upper case in the same context, then we have $[e] = [E]$, *i.e.*, the expressions differ only by their annotations. Finally, if several annotations of the same term are important in a proof, we may also use a typewriter font such as \mathbf{V} or \mathbf{E} , with the same convention if the same letter is used in lower case as well.

The idea of the completeness proof is to consider executions (reductions) of expressions in reverse. It is illustrated in Fig. 3.2. Let us first restrict our attention to expressions that *can* reduce.

subject to

Definition 3.12. An configuration s, e is *subject to* a reduction relation $R \in \{\delta, \rightarrow$

, \rightarrow , $\rightarrow\}$ if there is a state s' and an expression e' such that

$$(s, e)R(s', e').$$

Now let us characterize the fact that, for a given configuration (s, e) , all possible images (s', e') w.r.t. the reduction relation R can be annotated to verify the weakest precondition of e' and some postcondition q .

Definition 3.13. For a relation R , an expression $e_{\tau, \varphi}$, a state s and a postcondition $q_{\langle \varphi \rangle \rightarrow [\tau] \rightarrow \text{prop}}$, we define the predicate $P(R, s, e, q)$ to be equivalent to the property that for all s' and e' , if

$$(s, e)R(s', e')$$

then there exist S' and E' such that

$$S' \models \text{wp}_{\vartheta}(E', q) \wedge \text{correct}(S').$$

Additionally, S' and E' must be coherent in the sense that there exist \mathbf{S} and \mathbf{E} such that

$$(\mathbf{S}, \mathbf{E})R(S', E').$$

The property P is illustrated in Fig. 3.2 on the right hand side. This property is the *premise* of our completeness lemmas and theorems. The claim of these lemmas and theorems is simply the following: If $P(R, s, e, q)$ is true, then there are annotations S and E such that $S \models \text{wp}_{\vartheta}(E, q)$. It is illustrated on the left hand side of Fig. 3.2.

Let us illustrate on several examples why such complicated machinery is necessary to prove completeness.

- We cannot prove $s \models \text{wp}_{\vartheta}(e, q)$ directly, because it is false in general. This claim would be: If (s, e) always reduces to (s', e') such that $s' \models \text{wp}_{\vartheta'}(q)$, then we also have $s \models \text{wp}_{\vartheta}(e, q)$. This is false: consider the expression

$$e = (\text{rec } f(x). \{x = 0\}x \{r : r = x\}) 1$$

i.e., the identity function, requiring its argument to be zero, applied to 1. We can easily prove that any reduction of this expression is equal to 1, but we cannot prove it. We need to change the annotations of the anonymous function.

- We cannot require $s' \models \text{wp}_{\vartheta}(e', q)$ directly. In the theorem concerning the reduction relation \rightarrow , we have to chain reasoning steps such as the one illustrated in Fig. 3.2. Therefore, we cannot require more in the premise than we prove. We only prove that a reannotation E of e verifies $S \models \text{wp}_{\vartheta}(E, q)$, so we can only require that a reannotation E' requires $S' \models \text{wp}_{\vartheta}(E', q)$.
- We do need to require the existence of \mathbf{S} and \mathbf{E} such that $(\mathbf{S}, \mathbf{E})R(S', E')$. This is necessary because we only know that there *exist* S' and E' . However, they should be obtained *consistently*. As an example, consider the reduction relation δ and the expression

$$e = e =![\tau r] l$$

3. A Weakest Precondition Calculus

so that e' is of the form $s(r)(l)$ and $s' = s$. Now S' and E' are obtained from s' and e' by changing the annotations, but nothing guarantees that we have $E' = S'(r)(l)$. The requirement

$$\delta(\mathbf{S}, \mathbf{E}) = (S', E')$$

provides a solution.

- We cannot expect S and E to be equal to \mathbf{S} and \mathbf{E} . This also means that we cannot expect the relation

$$(S, E)R(S', E')$$

to hold in general. A counterexample has been discussed in preparation for Lemma 3.10: the annotation of a recursive function can be incomplete (too restrictive) for, say, an integer argument $n + 1$, but it can be sufficient for a reduced expression, in which only smaller calls occur, say, with integer argument n . In many parts of the proofs, however, we can and will set $S = \mathbf{S}$ and $E = \mathbf{E}$.

To be more precise about the notations being used, we use typewriter font such as \mathbf{V} for the annotations that are *given* by the premise, for example in \mathbf{E} . On the other hand, we use italic letters such as V if we refer to annotations we have to *find* to establish the claim, as in E .

As usual, we start with a hypothesis concerning δ , the function that deals with reduction of constants.

Hypothesis 3.14. *Let an expression $e = c v$ and a correct state s form a configuration subject to δ , and let $q_{\langle\varphi\rangle \rightarrow [\tau] \rightarrow \text{prop}}$ be a predicate on a final state and the return value of e . Then, the premise*

$$P(\delta, s, e, q)$$

implies that there are annotations E of e and S of s such that

$$S \models \text{wp}_\vartheta(E, q).$$

Proof for the constants of Definition 2.5. As s, e is subject to δ , we can exhibit appropriate s' and e' with which we instantiate the premise. As e' is necessarily a value, we introduce $e' = v'$. We exhibit annotations V' and S' , with the properties specified in Definition 3.13, in particular

$$S' \models \text{wp}_\vartheta(E', q) \Leftrightarrow S' \models q \vartheta [V'] \wedge \text{correct}(V')$$

In this proof, we can set $S = \mathbf{S}$ and $E = \mathbf{E}$. Now we proceed by case analysis.

Case ! Let $e = ![\tau r] l$. In fact, e cannot contain any annotations, so we have to prove:

$$\begin{aligned} S \models \text{wp}_\vartheta([\tau r] l, q) &\Leftrightarrow S \models \text{pre } ! l \vartheta \wedge \forall s' r. \text{post } ! l \vartheta s' r \Rightarrow q s' r \\ &\Leftrightarrow S \models \forall s' r. \vartheta = s' \wedge r = !! l s' \Rightarrow q s' r \\ &\Leftrightarrow S \models q \vartheta (!! l \vartheta) \end{aligned}$$

As $S = \mathbf{S}$ and because of $\delta(S, E) = S', E'$, we have $S'(r)(l) = V'$. Because of Definitions 2.33 and 2.34, the claim is equivalent to the formula guaranteed by the premise.

Case *ref* Let $e = \text{ref}[\tau r] v$, and $v' = l$, for some fresh l . Also, $s' = s[r \mapsto s(r)][l \mapsto v]$. Now let us unfold the claim:

$$\begin{aligned} S \models \text{wp}_\vartheta(\text{ref}[\tau r] v, q) &\Leftrightarrow S \models \text{correct}(V) \wedge \forall s'. r.\text{post } \text{ref } [V] \vartheta s' r \Rightarrow q s' r \\ &\Leftrightarrow S \models \text{correct}(V) \wedge \forall r. r \notin s \Rightarrow q (\text{set } r [V] \vartheta) r \end{aligned}$$

Using our hypothesis that

$$S' \models q \vartheta l,$$

we can prove the right hand side of our claim. Indeed, the quantification over r in the claim corresponds to the fact that the hypothesis is true for *any* fresh l (because of the quantification over s'). We can now lift the hypothesis from S' to S , replacing ϑ by $(\text{set } r [v] \vartheta)$. The left hand side of the claim stating the correctness of V can be obtained from the correctness of S' .

Case := This case is trivial.

Case :=_{r,l,\tau} This case is very similar to the one of *ref*, but even simpler, because there is no quantification over the memory location involved. □

Lemma 3.15. *Let an expression e and a correct state s form a configuration subject to \rightarrow , and let $q_{\langle \varphi \rangle \rightarrow [\tau] \rightarrow \text{prop}}$ be a predicate on a final state and the return value of e . Then, the premise*

$$P(\rightarrow, s, e, q)$$

implies that there are annotations E of e and S of s such that

$$S \models \text{wp}_\vartheta(E, q).$$

Proof. We want to show

$$S \models \text{wp}_\vartheta(E, q)$$

for some annotation (S, E) of (s, e) . As before, we exhibit appropriate s' and e' to instantiate the premise, obtain S' and E' such that

$$S' \models \text{wp}_\vartheta(E', q).$$

In this proof, we cannot always set $E = \mathbf{E}$. We now proceed by case analysis over the reduction using \rightarrow .

Case (β) Then $e = v_1 v_2$ with $v_1 = \text{rec } f(x). \{p_x\}_{e_x} \{q_x\}$. We have to find an annotated version of e ; we already introduce names for those annotated components: set $E = V_1 V_2$, where $V_1 = \text{rec } f(x). \{P_x\}_{E_x} \{Q_x\}$. The V_2 , P_x and Q_x have to be determined. Using these names, we have to show:

$$s \models \text{correct}(V_1) \wedge \text{correct}(V_2) \wedge P_x[x \mapsto [V_2]] \vartheta \wedge \forall sr. Q_x[x \mapsto [V_2]] \vartheta s r \Rightarrow q s r \quad (3.6)$$

3. A Weakest Precondition Calculus

Let us set $S = S'$, because the (β) rule does not change the state. The premised guarantees

$$S \models \text{wp}_\vartheta(E', q). \quad (3.7)$$

We know that $E' = \mathbf{E}_x[f \mapsto \mathbf{V}_1, x \mapsto \mathbf{V}_2]$, where the values and expressions in typewriter font are the ones taken from \mathbf{E} . Our aim is now to find V_1 and V_2 such that (3.6) becomes true.

Let us first set $E_x = \mathbf{E}_x$. Now, we can find V_2 as follows. If x occurs in e_x , set $V_2 = \mathbf{V}_2$. The value V_2 is correct because of (3.7) and the fact that it appears in E' . If x does not occur in e_x , we can find a suitable annotation using Proposition 3.11.

For V_1 , we have to find pre- and postconditions P_x and Q_x that make (3.6) true. We simply set Q_x to $\lambda_.q$ and P_x to $\lambda s. \text{wp}_s(E_x[f \mapsto \mathbf{V}_1], q_x s)$. By construction, Q_x implies q if applied to ϑ . For the claim concerning P_x , we have:

$$\begin{aligned} s \models \text{wp}_\vartheta(E', q) &\Leftrightarrow s \models \text{wp}_\vartheta(E_x[f \mapsto \mathbf{V}_1, x \mapsto \mathbf{V}_2], q) \\ &\Leftrightarrow s \models \text{wp}_\vartheta(e_x, q)[f \mapsto [\mathbf{V}_1], x \mapsto [\mathbf{V}_2]] \end{aligned}$$

and we want to prove:

$$s \models \text{wp}_\vartheta(e_x, q)[f \mapsto [V_1], x \mapsto [V_2]]$$

Comparing to the available hypothesis, we see the difference in the substitution for x , once it is substituted by $[V_2]$, and once by $[\mathbf{V}_2]$. This is only a problem when $x \in e_x$, but in this case we have $V_2 = \mathbf{V}_2$ by construction.

The correctness of V_1 is guaranteed by Lemma 3.10.

Case (*let*) Now, $e = \text{let } x [\bar{\chi}] = v \text{ in } e_x$. We set $S = S'$. Our hypothesis becomes

$$S \models \text{wp}_\vartheta(\mathbf{E}_x[x \mapsto \Lambda \bar{\chi}.V], q). \quad (3.8)$$

Defining $E = \text{let } x [\bar{\chi}] = V \text{ in } E_x$, we have to prove:

$$S \models \text{wp}_\vartheta(E, q)$$

which unfolds to

$$S \models \forall \bar{\chi}. \text{correct}(V) \wedge \text{let } x [\bar{\chi}] = [V] \text{ in } \text{wp}_\vartheta(E_x, q)$$

The proof is now similar to the previous case of (β) , but much simpler. We simply set $E_x = \mathbf{E}_x$ and, if $x \in e_x$, we set $V = \mathbf{V}$. Otherwise we simply make up annotations for V using Proposition 3.11. We observe that (3.8) takes care of half the claim. The correctness of V is guaranteed by construction.

Case (*iftrue*) Set $e = \text{if } v \text{ then } e_1 \text{ else } e_2$ and $S = S..$ Assume that $v = \text{true}$, the other case is of course symmetric. Then $e' = e_1$, and we have

$$S \models \text{wp}_\vartheta(\mathbf{E}_1, q).$$

Defining $E = \text{if } V \text{ then } E_1 \text{ else } E_2$, we need to prove:

$$S \models ([V] = \text{true} \Rightarrow \text{wp}_\vartheta(E_1, q)) \wedge ([V] = \text{false} \Rightarrow \text{wp}_\vartheta(E_2, q))$$

Fortunately, v does not need any annotations (it is a variable or a constant), so $V = v$. We can set $E_1 = \mathbf{E}_1$. For E_2 , *any* annotation works, because the hypothesis $[V] = \text{false}$ is always false, according to our initial assumption.

Case (δ) This is covered by Hypothesis 3.14.

Case **region** We have $e = \text{region } r \text{ in } v$. Then $e' = v$ and we set $S = S'$. The hypothesis becomes

$$S \models \text{wp}_\vartheta(\mathbf{V}, q) \Leftrightarrow q \vartheta [V] \wedge \text{correct}(\mathbf{V})$$

We set $V = \mathbf{V}$ and unfold the claim:

$$\begin{aligned} S \models \text{wp}_\vartheta(\text{region } r \text{ in } \mathbf{V}, q) &\Leftrightarrow s \models \text{wp}_{\vartheta \oplus \vartheta_r}(\mathbf{V}, \lambda t : \langle \varphi \cup r \rangle . q \ t|_\varphi) \\ &\Leftrightarrow S \models (\lambda t : \langle \varphi \cup r \rangle . q \ t|_\varphi) \vartheta_{\varphi \cup r} [V] \wedge \text{correct}(\mathbf{V}) \\ &\Leftrightarrow S \models q \vartheta_\varphi [V] \wedge \text{correct}(\mathbf{V}) \end{aligned}$$

The last line corresponds to the hypothesis.

Case **letregion** Let $e = \text{letregion } \varrho \text{ in } e_1$. We have $s' = s[r \mapsto \emptyset]$ for a fresh region name r , and $e' = \text{region } r \text{ in } e_1[\varrho \mapsto r]$. Our hypothesis is

$$S' \models \text{wp}_\vartheta(\text{region } r \text{ in } e_1[\varrho \mapsto r], q) \Leftrightarrow S' \models \text{wp}_{\vartheta_{\varphi \cup r}}(e_1[\varrho \mapsto r], \lambda t : \langle \varphi \cup r \rangle . q \ t|_\varphi) \quad (3.9)$$

We have to prove that

$$S \models \text{wp}_\vartheta(\text{letregion } \varrho \text{ in } e_1, q),$$

for some S , which is equivalent to

$$S \models \forall \varrho. \varrho \notin s \Rightarrow \forall s' : \langle \varrho \rangle . \text{wp}_{\vartheta \oplus s'}(e_1, \lambda t : \langle \varphi \cup \varrho \rangle . q \ t|_\varphi).$$

Note that (3.9) is correct for *any* fresh r and *any* S' such $S' = S[r \mapsto \emptyset]$. This quantification and the freshness condition can be translated to quantification in the claim.

□

We now prove essentially the same lemma for the reduction relation \longrightarrow .

Lemma 3.16. *Let an expression e and a correct state s form a configuration subject to \longrightarrow , and let $q_{\langle \varphi \rangle \rightarrow [\tau] \rightarrow \text{prop}}$ be a predicate on a final state and the return value of e . Then, the premise*

$$P(\longrightarrow, s, e, q)$$

implies that there are annotations E of e and S of s such that

$$S \models \text{wp}_\vartheta(E, q).$$

3. A Weakest Precondition Calculus

Proof. As in the preceding proofs, we want to show

$$S \models \text{wp}_\vartheta(E, q)$$

for some annotation (S, E) of (s, e) . As before, we exhibit appropriate s' and e' to instantiate the premise, obtain S' and E' such that

$$S' \models \text{wp}_\vartheta(E', q).$$

We also obtain \mathbf{S} and \mathbf{E} such that $S, E \longrightarrow S', E'$. In this proof, we cannot set $E = \mathbf{E}$. We proceed by induction over the form of the reduction context which we call F here in order to avoid confusion.

Case $F = []$ This case is covered by Lemma 3.15.

Case *let* Here, $e = \text{let } x = F[e_1] \text{ in } e_2$. The expression e' is of the form

$$e' = \text{let } x = F[e'_1] \text{ in } e_2,$$

where $s, e_1 \rightarrow s', e'_1$. By hypothesis, we have

$$S' \models \text{wp}_\vartheta(\text{let } x = F[E'_1] \text{ in } \mathbf{E}_2, q)$$

Setting $E_2 = \mathbf{E}_2$, we have to prove

$$S \models \text{wp}_\vartheta(\text{let } x = F[E_1] \text{ in } \mathbf{E}_2, q)$$

for suitable E_1 . Unfolding this, we have to prove

$$S \models \text{wp}_\vartheta(F[E_1], \lambda t. \lambda x. \text{wp}_t(\mathbf{E}_2, q)) \tag{3.10}$$

Applying the induction hypothesis for $F[e_1]$, we can reduce this to proving $P(\longrightarrow, s, F[e_1], \lambda t. \lambda x. \text{wp}_t(\mathbf{E}_2, q))$, *i.e.*, the premise of the lemma, where e has been replaced by $F[e_1]$ and q by $\lambda s'. \lambda x. \text{wp}_{s'}(\mathbf{E}_2, q)$.

Therefore assume s, s' and e' (different from the previously introduced ones for the sake of justifying the application of the induction hypothesis) such that $s, F[e_1] \longrightarrow s', e'$. We immediately derive that

$$s, \text{let } x = F[e_1] \text{ in } e_2 \longrightarrow s', \text{let } x = e' \text{ in } e_2,$$

which means that we can apply the premise on e and obtain

$$S' \models \text{wp}_\vartheta(\text{let } x = F[E'_1] \text{ in } \mathbf{E}_2, q) \Leftrightarrow s' \models \text{wp}_\vartheta(F[E'_1], \lambda t. \lambda x. \text{wp}_t(\mathbf{E}_2, q))$$

This is exactly what we need (compare with the postcondition of (3.10)).

Case *region* similarly, using the induction hypothesis.

□

The main result is now a similar claim concerning \rightarrow . However, to avoid the theorem to become trivial, we do not use the relation \rightarrow directly in connection with the predicate P . If we did, we could simply set $e' = e$ and $s' = s$ to obtain the claim. Instead, we introduce the family of relations \rightarrow_n , that describe reduction sequences using \rightarrow of length n .

Theorem 3.17. *Let e be an expression and s a correct state, and n an integer such that (s, e) is subject to \rightarrow_n , and let $q_{(\varphi) \rightarrow [\tau] \rightarrow \text{prop}}$ be a predicate on a final state and the return value of e . Then, the premise*

$$P(\rightarrow_n, s, e, q)$$

implies that there are annotations E of e and S of s such that

$$S \models \text{wp}_\vartheta(E, q).$$

Proof. By induction on n . As we have seen, the claim is trivial for $n = 0$, because then $s = s'$ and $e = e'$. Now let us show the claim for $n + 1$.

We exhibit s', e' such that $s, e \rightarrow_{n+1} s', e'$, and we cut this reduction sequence in two parts, as follows:

$$s, e \rightarrow s'', e'' \rightarrow_n s', e'$$

We now want to apply the induction hypothesis to the right side of the reduction sequence, the reduction of length n . To do this, we first need to prove $P(\rightarrow_n, s'', e'', q)$, but this follows from $P(\rightarrow_{n+1}, s, e, q)$, because each sequence of length n starting from (s'', e'') is a sequence of length $n + 1$ starting from (s, e) . We now have obtained S'' and E'' such that

$$S'' \models \text{wp}_\vartheta(E'', q).$$

We just have shown that this is true for *any* such s'' and e'' . So we have shown $P(\rightarrow, s, e, q)$, and we can use Lemma 3.16 to obtain the final claim. \square

A corollary that more closely follows the the usual completeness result is now a simple consequence of Theorem 3.17.

Corollary. *Let $e_{\tau, \varphi}$ be an expression and $q_{(\varphi) \rightarrow [\tau] \rightarrow \text{prop}}$ be a predicate. For all s and v , if*

$$\emptyset, e \rightarrow s, v$$

and if

$$\text{correct}(s) \wedge s \models q \vartheta [v] \wedge \text{correct}(v),$$

then we can find an annotation E of e such that

$$\emptyset \models \text{wp}_\vartheta(E, q).$$

Proof. Let n be the the length of the reduction sequence. We apply Theorem 3.17 on \rightarrow_n . We only need to prove $P(\rightarrow_n, \emptyset, v, q)$. We simply set $S' = s$ and $V = v$. We also set $E = e$ and $S = \emptyset$. Then this predicate is trivially true. \square

3.4. Extensions

We now discuss a few possible extensions to the core system of W and L that make writing programs or reasoning about programs easier. We are less rigorous in this section and only sketch the arguments for the soundness proof for each extension. We do not prove completeness for the extensions, but we believe that completeness would still hold.

3.4.1. Logical Symbols in Programs

One aspect of the original Hoare logic (see Section 1.3.1) that makes it particularly convenient to work with is the fact that arithmetic expressions are part of the programming language as well as the logic. The advantage is that if a program uses logical functions instead of program functions, one does not need to reason about pre- and postconditions; one can directly use the same function symbol in the logic as well.

This is not possible in W and L as presented, because of the different structure of types in both languages. This introduces some artificial difficulties. As an example, consider the addition function $+$. If we cannot use logical symbols in programs, we first have to decide if $+$ is a logical function or a program function. In Chapter 2 we decided that it is a program function of type $\text{int} \rightarrow^{\emptyset} \text{int} \rightarrow^{\emptyset} \text{int}$. The question is now, what is the specification of $+$? We first need another symbol, say $\hat{+}$, this time a logical symbol representing integer addition. Now we can say that the postcondition of $+$ states that the result of the function call $+ x y$ is always equal to $x\hat{+}y$.

However, there is a simple way out. The inclusion of logical terms into programs can be easily achieved if one extends the types of programs τ to include logical functions of type $\tau \rightarrow \tau$. Syntactically, few things need to change: we only need to integrate pure anonymous functions of the form

$$\lambda(x : \tau).v$$

into the syntax of values in W . Note that an effectful function with empty effect, of type $\tau \rightarrow^{\emptyset} \tau'$, is different from a *pure*, logical function of type $\tau \rightarrow \tau'$. In particular, an effectful function may not terminate, while logical functions are assumed to always terminate.

Following these definitions, a new notion of value emerges. *Syntactic* values are what we simply called *values* before; they are defined syntactically and the reduction stops when encountering such a syntactic value. To adapt the semantics, we simply need to extend the relation \rightarrow with a rule (β') to account for the reduction of anonymous pure functions. The new notion of *logical values* describes expressions that are formed only of syntactic values or *pure* applications, *i.e.*, applications of functions whose type is of the form $\tau \rightarrow \tau'$. These “values” continue to reduce during the execution of the program, but from a logical point of view we do not need to decompose these function calls. In particular, the lifting operation $\lceil \cdot \rceil$ is defined as follows for pure applications:

$$\lceil v_{\tau \rightarrow \tau'} v' \rceil = \lceil v \rceil \lceil v' \rceil$$

We see that the application remains intact in the logic.

As stated, from the point of view of the wp calculus, logical applications are values. Therefore, they are subject to correctness formulas. They are computed as follows:

$$\text{correct}(v_1 v_2) = \text{correct}(v_1) \wedge \text{correct}(v_2)$$

In the definition of wp itself, these applications are treated in the case for values; they are carried over unchanged (using $\lceil \cdot \rceil$) to the logic side.

As a result, we now can decide that $+$ is a logical function, but we can still use it in programs. The reasoning about these function calls is greatly simplified.

3.4.2. Read-Write Effects

Currently, effects in W do not distinguish between read and write effects. In principle, any function that reads some reference has to state in its postcondition that this region remains unchanged. We have done this for example in the specification of $!$. In practice, these specifications become rapidly tedious. A way to get rid of them is to distinguish between read and write effects; any region that is mentioned in read effects and not write effects is guaranteed to be unchanged.

More formally, we propose here to replace the effect φ , which is a simple list of region and effect variables (also called *basic effects*), by a pair (φ_1, φ_2) of lists of basic effects. The left component is intended to be the read effect, the right component is the write effect. The effect operations \cup and \setminus are redefined to work pointwise:

$$\begin{aligned} (\varphi_1, \varphi_2) \cup (\varphi_a, \varphi_b) &= (\varphi_1 \cup \varphi_a, \varphi_2 \cup \varphi_b) \\ (\varphi_1, \varphi_2) \setminus \varrho &= (\varphi_1 \setminus \varrho, \varphi_2 \setminus \varrho) \end{aligned}$$

For the weakest precondition calculus to work, it is necessary to enforce that the write effect is always contained in the read effect. Fortunately, this property is maintained by both the union and the region removal operation, so we only need to verify that it is true for function constants.

To obtain type soundness of the modified system, we simply use the generalized results of Section 2.1.5. Indeed, both operations still are stable under substitution as required. We therefore obtain type soundness for free.

It is important to note that the definition of logical types remains the same: the domain of state types $\langle \varphi \rangle$ is still a flat list of basic effects. When lifting function types to logical types using $\lceil \cdot \rceil$, the effect that is retained for the state types is the read effect:

$$\lceil \tau_1 \rightarrow^{(\varphi_1, \varphi_2)} \tau_2 \rceil = (\lceil \tau_1 \rceil \rightarrow \langle \varphi_1 \rangle \rightarrow \mathbf{prop}) \times (\lceil \tau_1 \rceil \rightarrow \langle \varphi_1 \rangle \rightarrow \langle \varphi_1 \rangle \rightarrow \lceil \tau_2 \rceil \rightarrow \mathbf{prop})$$

To exploit the more precise effect information, we need to modify the weakest precondition calculus for function applications:

$$\begin{aligned} \text{wp}_s(v_{\tau' \rightarrow (\varphi_1, \varphi_2) \tau} v', q) = \\ \text{pre } \lceil v \rceil \lceil v' \rceil s \wedge \forall s' : \langle \varphi_2 \rangle. \forall x : \lceil \tau \rceil. \text{post } \lceil v \rceil \lceil v' \rceil s (s \oplus s') x \Rightarrow q (s \oplus s') x \end{aligned}$$

The idea is that we only need to quantify over the *modified* part of the store. We use the expression $s \oplus s'$, the state after executing v , as the poststate. This expression uses

3. A Weakest Precondition Calculus

the *initial* state s containing all read and written regions, updated using s' , which only contains the written regions.

To prove the correctness of this modification of the **wp** calculus, one modification has to be applied. First, it is easy to see that Lemma 3.1 concerning type substitution and Lemma 3.2 concerning value substitution are still correct. Lemma 3.3 has to be modified so that the implication in the hypothesis quantifies only over written regions:

Lemma 3.18 (Weakening Lemma for the **wp** calculus). *For any expression of type τ and read-write effect (φ_1, φ_2) , for a state t of type $\langle \varphi_1 \rangle$ and for any q_1 and q_2 of type $\langle \varphi_1 \rangle \rightarrow [\tau] \rightarrow \text{prop}$, if q_1 is stronger than q_2 :*

$$\forall s : \langle \varphi_2 \rangle. \forall r : [\tau]. q_1 (t \oplus s) r \Rightarrow q_2 (t \oplus s) r$$

then the weakest precondition of e and q_1 is stronger than the weakest precondition of e and q_2 :

$$\forall s : \langle \varphi_1 \rangle. \text{wp}_s(e, q_1) \Rightarrow \text{wp}_s(e, q_2)$$

Proof. The proof is very similar to the one of Lemma 3.3. □

Using this lemma, we can prove Lemma 3.7 for this calculus; the only difference in the proof is that we use Lemma 3.18 to justify soundness in the case APP.

3.4.3. Algebraic Data Types and Pattern Matching

Adding algebraic data types and pattern matching (see Section 1.3.2) to the language W is quite simple and orthogonal to the features that are already present. First, we need to add them to the language:

$$\begin{aligned} K & \quad \text{constructor} \\ e ::= & \quad \dots \mid \text{match } v \text{ with } \bar{c} \\ c ::= & \quad p \rightarrow e \\ p ::= & \quad x \mid K \bar{p} \end{aligned}$$

A pattern matching is introduced by the keyword **match**, followed by a value (recall that we present W in A-normal form), the keyword **with** and finally a list of *cases*. A case is a *pattern*, followed by an arrow, and the expression corresponding to the pattern. Finally, a pattern is either a variable or a constructor applied to a list of patterns.

We do not give the semantics and typing rules here; they are standard and can be found, for example, in the book by Pierce (2002).

Algebraic data types and pattern matching (see Section 1.3.2) are useful not only in programming languages, but also in the logic. A few provers and interactive proof assistants, for example Coq, propose these constructs. Thus, it is reasonable to assume the existence of these features in the logic. Under this assumption, an integration of these features in the weakest precondition calculus is very simple.

$$\text{wp}_s(\text{match } v \text{ with } \overline{p \rightarrow \bar{e}}, q) = \text{match } [v] \text{ with } \overline{p \rightarrow \text{wp}_s(\bar{e}, q)}$$

where the bar is used to denote lists of subexpressions, as usual.

If we assume that the semantics of the pattern matching in the programming language and the logic are identical, then the soundness and completeness of this additional rule is obvious. One can also see it as a generalization of the rule for the if-then-else construct.

4. A Language without Aliasing

We have presented the programming language W with its annotation language L in Chapter 2. We also have presented a correct wp-calculus for annotated W programs. However, this calculus is lacking a few important properties that help reasoning about larger programs. In particular, the functions `get`, `set`, `combine` and `restrict` are central for the manipulation of state, but the potential simplifications suggested by the properties (A3), (A5) and (A6) in Section 2.2.3 are protected by premises on regions and domains. For example, property (A5) requires that a certain region ρ be disjoint from some domain described by an effect φ . But of course, if ρ is a region variable, or if φ contains region or effect variables, we cannot know for sure, unless we have assumed them to be disjoint.

In practice, this means that programs which manipulate many regions and effects with effect variables need to be annotated with separation predicates about regions and effects; this is very tedious in practice. So tedious that many mechanisms have been developed to avoid writing these predicates, or to do so more succinctly. In Section 4.1, we present an extension to the type system of W that corresponds to the proposal of Hubert and Marché (2007), but extended to a higher-order setting. Their idea was to ensure that at all time, two different region variables always stand for different regions. In our higher-order setting, we have to go one-step further and state that, additionally, all effect variables stand for disjoint effects. We call this modified system W without region aliasing.

In W , even without region aliasing, there is a second imprecision: a region may contain several references. This means, from the point of view of the specification, that when a single reference in a certain region is modified, all references in that region have potentially been modified, and if they have not, this has to be stated explicitly. The reason is that the precision of the type system is precisely captured by the notion of region; statically, we do not know if two references of the same region are equal or different, and we do not even know how many references are contained in a region.

Let us give a simple example that illustrates the difficulty. Consider the expression

$$\text{if } v \text{ then } x \text{ else } y \tag{4.1}$$

where x and y are two references. In the type system of W , actually in most type systems, both branches of an if-expression must be of the same type, and this is also the type of the entire expression. As x and y are references, this means that the expression is of type $\text{ref}_\rho \tau$ for some type τ and some region ρ . Now it depends on the boolean value of v if x or y is returned. Our type system cannot express that the exact reference which is returned depends on v .

In a type system based on regions, there are two possible answers to the problem posed in the previous paragraph. One solution is to accept that a region contains several references. Then it is up to the Hoare logic specifications to deal with this fact,

4. A Language without Aliasing

and it should be possible to deduce which reference has been returned by expression (4.1). This is what the unrestricted system \mathbb{W} and the system \mathbb{W} without region aliasing do.

The other possibility is to enforce that each region contains only a single reference. Such a region is called a *singleton region*. In this case, if we give the type $\text{ref}_\rho \tau$ to expression (4.1), its return value is necessarily the single reference contained in region ρ , regardless of the truth value of v . This means that x and y are actually the same reference, which can also be seen by the fact that both have the same type $\text{ref}_\rho \tau$. Of course, if x and y have the same value, then the if-expression makes no sense. On the other hand, if x and y are different references, they must have different region annotations in their type, and expression (4.1) is ill-typed.

In Section 4.2, we present an extension of \mathbb{W} without region aliasing that enforces singleton regions. Again, the result is a restriction of the input language in exchange for simpler proof obligations. It should be noted that the first restriction (\mathbb{W} without region aliasing) is not a very serious restriction. It often corresponds to good programming style (avoiding aliasing of mutable objects) and can be circumvented when necessary. \mathbb{W} with singleton regions, however, represents a severe restriction and indeed rules out an entire class of programs, notably those with shared mutable state. Indeed, consider the type of lists, for example. In ML, lists are homogeneous, *i.e.*, they contain elements of the same type. In a system with singleton regions, this means that a list of references can only contain a *single* reference, potentially several times. In this setting, computations with shared mutable state, such as mutable lists or graph- and tree-like mutable structures, are ill-typed.

\mathbb{W} with singleton regions is an extension of \mathbb{W} without region aliasing. In the presence of region aliasing, the system of singleton regions would be unsound.

4.1. Excluding Aliasing of Regions

The problems described in the opening paragraph actually result from *aliasing* of regions, a phenomenon similar to the aliasing of mutable variables described in Section 1.3.1. We briefly describe two solutions to this problem that are the source of inspiration to our proposal.

In Why (Filliâtre, 2003), aliasing of mutable program variables is excluded using a restriction of the application of functions to effectful variables. If f is a function with latent effect φ , then the application $f x$ is only allowed if x is immutable or if x does not appear in φ (notice that in Why, effects are composed of variable names instead of regions). The consequence is that different program variables always refer to different memory locations. Another necessary restriction is that references cannot be stored in data structures. Therefore, many algorithms based on *sharing* cannot be implemented directly in Why.

Hubert and Marché (2007) use the same idea, but in a setting with regions. Now the critical point is not the application of variables, but the *instantiation* with regions. In their system, a region polymorphic variable f can only be instantiated with a region ρ if the type of f does not already contain ρ before instantiation. If a variable is polymorphic with respect to several regions, all those regions must be instantiated with

different region arguments. The consequence is that two different region variables always refer to different memory locations. Two mutable variables of the same region may still refer to the same memory location, though. Therefore, algorithms with sharing can be implemented. This system does not contain effect variables, so generic reasoning about higher-order functions is not possible.

We extend this basic idea (restricting the application/instantiation of effects) to programs with higher-order functions and effect polymorphism. The idea is that all effect and region instantiations for a given variable f must be disjoint from each other and from the effects already present in the type of f . Again, this guarantees that two different region variables point to different regions; in addition, it guarantees that in a given context, a region or region variable must be disjoint from any effect variable. As a consequence, simplifications of expressions using the functions `restrict` and `combine` can be applied statically; they are not needed anymore in the logic.

Restriction of effect and region instantiations. By looking at properties (A5) and (A6), the premise about the domain of s_2 does not allow us to simplify state combining expressions. To illustrate this, consider a state s of type $\langle r \rangle$ and a state s' of type $\langle \varrho\varepsilon \rangle$, where r is a concrete region, ϱ is a region variable and ε an effect variable. Now consider the expression

$$\text{get } (\text{combine } s \ s') \ x$$

where x is a reference in region r . We cannot reduce this expression to `get s x` even though it looks like we could; after all, the domain of s' is $\varrho\varepsilon$ and does not *seem* to contain r . However, we would like properties (A5) and (A6) to be true regardless of substitutions. But of course, there are two cases of substitutions where the simplification breaks down: the substitution $[\varrho \mapsto r]$ and any substitution $[\varepsilon \mapsto \varphi]$ such that φ contains r will render the assumption $r \notin \varrho\varepsilon$ false.

On the other hand, this kind of simplification is actually one of the reason we used regions in the first place. If two memory cells are from different regions, it is a huge advantage to know that they are different; also, if a function effect does not mention a certain region, it should not modify it, even by substitution of effect variables.

The solution is an extension of the region based separation analysis of Hubert and Marché (2007): we simply disallow certain instantiations. More precisely, we disallow instantiations that would introduce region or effect names into types that already contain them.

Definition 4.1. A region substitution $[\varrho \mapsto \rho']$ is *compatible* with any type that does not contain ρ' . Similarly, an effect substitution $[\varepsilon \mapsto \varphi]$ is compatible with any type that does not contain any component of φ . A substitution $[\bar{\chi} \mapsto \bar{\alpha}]$ or $[\bar{\chi} \mapsto \bar{\kappa}]$ is compatible with a type τ if all effect and region images of the substitution are disjoint of each other and do not occur in τ . Compatibility is denoted using the \sim symbol: $[\bar{\chi} \mapsto \bar{\kappa}] \sim \tau$.

compatible

Now, we modify both the typing rules of W programs and L terms containing instantiation to be restricted to compatible instantiations. The new rules are represented in Fig. 4.1.

Remark. By restricting the typing relation, we have reduced the set of typable programs, but do we keep the expressive power of W? The answer is two-fold.

4. A Language without Aliasing

$$\begin{array}{c}
\text{PVAR} \frac{\Gamma(x) = \forall \bar{\chi}. \tau \quad \phi = [\bar{\chi} \mapsto \bar{\kappa}] \quad \phi \sim \tau}{\Gamma; \Sigma \vdash_v x [\bar{\kappa}] : \tau \phi} \\
\text{PCONST} \frac{\text{Typeof}(c) = \forall \bar{\chi}. \tau \quad \phi = [\bar{\chi} \mapsto \bar{\kappa}] \quad \phi \sim \tau}{\Gamma; \Sigma \vdash_v c [\bar{\kappa}] : \tau \phi} \\
\text{L-VAR} \frac{\Delta(x) = \forall \bar{\chi}. \sigma \quad \phi = [\bar{\chi} \mapsto \bar{\varkappa}] \quad \phi \sim \sigma}{\Delta; \Sigma \vdash_l x [\bar{\varkappa}] : \sigma \phi} \\
\text{L-CONST} \frac{\text{LogicTypeof}(c) = \forall \bar{\chi}. \sigma \quad \phi = [\bar{\chi} \mapsto \bar{\varkappa}] \quad \phi \sim \sigma}{\Delta; \Sigma \vdash_l c [\bar{\varkappa}] : \sigma [\bar{\chi} \mapsto \bar{\varkappa}]}
\end{array}$$

Figure 4.1: The modified rules for variables and constants in W and L.

If one has the whole program at ones disposal, then the answer is yes: Every ML program, *i.e.*, a program that does not contain region and effect annotations, **letregion** and effect and region polymorphism, that is typable in W, *i.e.*, by adding region and effect annotations, is typable in W without region aliasing. One simply uses a single region for any mutable state, and does not use effect or region polymorphism. Put otherwise, whenever the instantiation restriction poses a problem, one can solve it by merging regions, as long as one has the entire program available. On the other hand, when part of the program has already been typed, using effect and region polymorphism to define functions, then the instantiation restriction disallows certain uses of these functions. The solution proposed by Hubert and Marché (2007) has the same restriction.

Type soundness. As we have modified the typing rules, a few modifications have to be applied to the correctness proofs. It should be noted, however, that only statements about values are concerned. We now go through the lemmas and theorems of Chapter 2 and prove that we still have type soundness for this restricted language.

The modified rules type *strictly less* programs and logical terms. This automatically guarantees that Theorem 2.15 (Progress) is still true. Proposition 2.12 concerning canonical values is obviously still true.

In our type soundness proof, Lemma 2.18 concerning type, region and effect substitution, is used at two places, once for justifying the **let** reduction rule, which substitutes polymorphic values for variables, and in the case concerning the **letregion** rule, because we also need to perform a region substitution there. Using our new typing rules, both substitutions must be compatible, while there was no such restriction before. The one in the **letregion** case is compatible because it is of the form $[\varrho \mapsto r]$ for a fresh r that does not occur in any expression or type. The one in the **let** case is compatible because all variable instantiations now must be compatible.

We can therefore restrict the statement of Lemma 2.18 concerning value typing to substitutions that are compatible with the type of the value. To adapt the proof of

Lemma 2.18 to our new setting, it remains to prove that compatibility is “transitive”:

Proposition 4.2. *if $\phi \sim \tau$ and $\phi' \sim \tau\phi$, then also $\phi\phi' \sim \tau$.*

Proof. Let us first assume that $\phi = [\varrho \mapsto \varrho_1]$ and $\phi' = [\varrho_2 \mapsto \varrho_3]$, i.e., that ϕ and ϕ' are simple region substitutions. We know that $\phi \sim \tau$, which means that $\varrho_1 \notin \tau$. We also know that $\varrho_3 \notin \tau\phi$. We have to prove that the image of $\phi\phi'$, which is the set $\{\varrho_1, \varrho_3\}$, is not contained in τ . As we know this already for ϱ_1 , it remains to prove the claim for ϱ_3 .

First we remark that substitutions in ML are always idempotent, even composed ones. An idempotent substitution can be applied twice to a term or a type without any further effect: $\tau\phi\phi = \tau\phi$. The reason is that the domain of a substitution in ML stems from the generalization of type, effect and region variables. The names of these type variables can be chosen freely (the Barendregt convention) and are always different from free variables. Instantiation always happens outside the scope of this generalization. Therefore, the image of a substitution in ML cannot contain variables from its domain. Such substitutions are always idempotent. We can derive that $\varrho_3 \neq \varrho$.

To continue our proof, we know that $\varrho_3 \notin \tau\phi$ and this implies that either $\varrho_3 \notin \tau$ or $\varrho_3 = \varrho$. We have seen that we can disprove the latter and conclude with the former.

The proof can be done in a similar way when ϕ or ϕ' , or both, are effect substitutions for a single effect variable.

Finally, we can generalize this argument to substitutions with a larger domain. Actually, an idempotent substitution $\phi = [\chi_1 \mapsto \kappa_1 \cdots \chi_n \mapsto \kappa_n]$ can be decomposed into n atomic substitutions ϕ_i of the form $[\chi_i \mapsto \kappa_i]$, such that $\phi = \phi_1 \cdots \phi_n$. We now can finish the proof by a straightforward induction over the length of the decomposition of ϕ' . \square

This proposition is to be used in the case for the typing rule VAR in the modified proof of Lemma 2.18. The proofs of the substitution lemma (Lemma 2.19) and the subject reduction lemmas can now remain unchanged; when evoking Lemma 2.18 in the case for values, the additional hypothesis $\phi \sim \tau$ has to be used.

Soundness and completeness of the wp calculus. The soundness proof for the wp calculus given in Section 3.2 remains correct; it applies to well-typed expressions, so the additional restriction does not change this. The completeness proof basically computes weakest specifications using the wp calculus. To obtain its validity even in presence of the modifications introduced in this section, we simply observe that the wp predicate contains the same effect and region instantiations as the expression e and postcondition q it receives as input. Therefore, the completeness proof of Section 3.3 does not depend on features that would be removed by activating these restriction.

Exploiting the absence of aliasing. The modifications of the type system that have been introduced in the previous section restrict the set of well-typed programs so that fewer programs are accepted. In exchange, we obtain an advantage: we now can apply the properties (A5) and (A6) directly, because we can decide statically if a given region is contained in an effect. To illustrate this more clearly, let us look at an instance of

4. A Language without Aliasing

property (A5), with full type, region and effect instantiations for some region ρ and some effects φ_1, φ_2 and φ_3 :

$$\text{get } [\rho, \varphi_1\varphi_2\varphi_3] (\text{combine } [\rho\varphi_1, \varphi_2, \varphi_3] s_1 s_2) x = \text{get}[\rho, \varphi_1\varphi_2] s_1 x$$

with the side condition that

$$\text{region}(x) \notin \text{domain}(s_2).$$

Looking at the types of x and s_2 , we can rewrite this side condition as

$$\rho \notin \varphi_2\varphi_3.$$

The point is now that simply by choosing these particular region and effect instantiations when writing down this instance of the axiom, we have fixed that only the effect represented by $\rho\varphi_1$ (the first instantiation of `combine`) can contain ρ ; any other instantiation of `combine` where φ_2 or φ_3 contain ρ would be ill-typed because the resulting substitution would be incompatible with the type of `combine`. In this case, the side condition is always true and we can statically simplify the expression to the one on the right hand side.

A similar reasoning would have allowed us to use property (A6) to simplify the expression if we had used the instantiation $[\varphi_1, \varphi_2, \rho\varphi_3]$, for example, with the `combine` function.

In practice, these results imply that we can simplify expressions of the form

$$\text{get } (\text{combine } s_1 s_2) x$$

statically, because we know which of s_1 and s_2 are relevant. The same is true for expressions of the form

$$\text{get } (\text{set } s x v) y$$

if x and y are in different regions. This results in many simplifications in generated proof obligations.

The Frame Rule

We now want to show that these additional properties, that simplify formulas in practice, also have a theoretical impact on our system. In particular, the so-called *frame rule*, which is central to many Hoare logic systems, now becomes admissible in our system. It states that the parts of the store that are not modified by a given program remain the same. This sounds like a triviality, but it is a crucial property of any system to guarantee this in a way that is the simplest possible for the user. In the original Hoare logic (see Section 1.3.1), this property is expressed using the *conjunction rule*:

$$\text{CONJ } \frac{\{ P \} C \{ Q \}}{\{ P \wedge R \} C \{ Q \wedge R \}}$$

where R is a formula that does not mention variables that are modified by C . This side condition is precisely the *frame* condition; it states that R and C live in different

frames or regions of the store. As a consequence, the CONJ rule is able to state that the validity of R is not modified by C .

An equivalent of the frame rule is crucial for any verification system. It guarantees that the reasoning about any piece of code can be limited to the effects of that code, and does not need to consider its context, as long as this context is not necessary for correctness. Using the notation of the CONJ rule, the specification of the program C does not need to mention the logical context R of its usage, but only the formulas P and Q that are relevant to C .

We now show that an equivalent of the frame or conjunction rule is provable in our wp calculus. We start by showing that the validity of formulas that do not depend on the state or the result of the expression, is left unchanged by the wp calculus.

Proposition 4.3. *For an expression $e_{\tau, \varphi}$ and two formulas p_{prop} and $q_{\langle \varphi \rangle \rightarrow [\tau] \rightarrow \text{prop}}$, the following formula holds:*

$$\text{wp}_s(e, q) \wedge p \Rightarrow \text{wp}_s(e, \lambda s : \langle \varphi \rangle. \lambda r : [\tau]. (q \ s \ r \ \wedge \ p))$$

Note that well-typedness conditions imply that p cannot contain occurrences of the bound variables r and s .

Proof. The proof is somewhat similar to Lemma 3.3; we proceed by induction over the structure of e .

Cases v and $v \ v$ In this case, the claim follows from the fact that the postcondition of wp is only used in positive position.

Cases letregion, region, polymorphic let, if and subeffecting In these cases, the claim can be established using the induction hypothesis for the recursive call(s), which appear(s) in positive position.

Case monomorphic let Let us set $f = \lambda s : \langle \varphi \rangle. \lambda r : [\tau]. q \ s \ r \ \wedge \ p$. We also set $e = \text{let } x = e_1 \text{ in } e_2$ and we have

$$\text{wp}_s(e, f) = \text{wp}_s(e_1, \lambda s' : \langle \varphi \rangle. \lambda x : [\tau']. \text{wp}_{s'}(e_2, f)).$$

We know that $\text{wp}_{s'}(e_2, q) \wedge p$ implies $\text{wp}_{s'}(e_2, f)$ by induction hypothesis. By Lemma 3.3, we obtain that

$$\text{wp}_s(e_1, \lambda s' : \langle \varphi \rangle. \lambda x : [\tau']. \text{wp}_{s'}(e_2, q) \wedge p) \tag{4.2}$$

implies

$$\text{wp}_s(e_1, \lambda s' : \langle \varphi \rangle. \lambda x : [\tau']. \text{wp}_{s'}(e_2, f)) \tag{4.3}$$

Again by the induction hypothesis, we can also state that

$$\text{wp}_s(e_1, \lambda s : \langle \varphi \rangle. \lambda x : [\tau']. \text{wp}_{s'}(e_2, q)) \wedge p \tag{4.4}$$

implies (4.2). This closes the chain: (4.4) is equal to the left hand side of the claim; it implies formula (4.2), which itself implies (4.3). This last term is equal to the right hand side $\text{wp}_s(e, f)$. \square

4. A Language without Aliasing

Comparing Proposition 4.3 with the CONJ rule, we see that p in the proposition already plays the rôle of R in the CONJ rule. However, the condition on p is too strong: Proposition 4.3 basically requires p to be independent of the current state.

The following lemma leverages this restriction. It has the same form as Proposition 4.3, but introduces a predicate p that may depend on a portion of the store that is disjoint from the effect of the program e .

Theorem 4.4 (Frame Rule). *Let φ_1 and φ_2 be two disjoint effects. Let e_{τ, φ_1} be an expression and $p_{(\varphi_2) \rightarrow \text{prop}}$ and $q_{(\varphi_1) \rightarrow [\tau] \rightarrow \text{prop}}$ be formulas. Then*

$$(\text{wp}_{s|\varphi_1}(e, q) \wedge p_{s|\varphi_2}) \Rightarrow \text{wp}_s(e, \lambda s : \langle \varphi_1 \cup \varphi_2 \rangle. \lambda r : [\tau]. (q_{s|\varphi_1} r \wedge p_{s|\varphi_2})).$$

Proof. We simply unfold the definition of wp on the right. Setting

$$f = \lambda s : \langle \varphi_1 \cup \varphi_2 \rangle. \lambda r : [\tau]. q_{s|\varphi_1} r \wedge p_{s|\varphi_2}$$

we derive:

$$\begin{aligned} \text{wp}_s(e, f) &= \text{wp}_{s|\varphi_1}(e, \lambda s' : \langle \varphi_1 \rangle. f(s' \oplus s)) \\ &= \text{wp}_{s|\varphi_1}(e, \lambda s' : \langle \varphi_1 \rangle. \lambda r : [\tau]. q_{(s \oplus s')|\varphi_1} r \wedge p_{(s \oplus s')|\varphi_2}) \\ &= \text{wp}_{s|\varphi_1}(e, \lambda s' : \langle \varphi_1 \rangle. \lambda r : [\tau]. q_{s'} r \wedge p_{s|\varphi_2}) \\ &= \text{wp}_{s|\varphi_1}(e, \lambda s' : \langle \varphi_1 \rangle. \lambda r : [\tau]. q_{s'} r) \wedge p_{s|\varphi_2} \end{aligned}$$

In the last line of the derivation, we can factor out the formula $p_{s|\varphi_2}$ using Proposition 4.3, because it does not contain s' nor r . \square

Remark. It should be remarked that Theorem 4.4 is a theoretical result; it does not get directly applied in a particular phase of the weakest precondition calculus. It is simply a property that every Hoare logic system should possess.

An example. Let us study the consequences of the region aliasing restriction on an example. Here is the code of the simple function `apply_reset`:

```
let apply_reset [ρε] (f : unit →ε unit) (x : refρ α) =
  { pre f () cur|ε }
  x := 0;
  f ()
  { post f () old|ε cur|ε () ∧ !!x = 0 }
```

The function takes a function f and a reference x as arguments, sets x to 0 and calls f . We do not want to go into details about the specification of `apply_reset` here; we see that the precondition states that we need to have the precondition of f on the current state. The postcondition states that we have the postcondition of f of the initial and the current state, and that $x = 0$. Is `apply_reset` specified correctly?

To answer this question, we need to compute the formula $\text{correct}(\text{apply_reset})$ and see if it is valid. Let e be the body of the function and q its postcondition, then the correctness of apply_reset can be unfolded as follows:

$$\text{correct}(\text{apply_reset}) = \forall s : \langle \varrho \varepsilon \rangle. \forall x. \text{pre } f () s|_{\varepsilon} \Rightarrow \text{wp}_s(e, q s)$$

We continue to unfold the weakest precondition formula:

$$\begin{aligned} & \text{wp}_s(e, q s) \\ \Leftrightarrow & \forall x. \text{wp}_s(e, \lambda s'. \text{post } f () s|_{\varepsilon} s'|_{\varepsilon} () \wedge \text{get } s x = 0) \\ \Leftrightarrow & \text{wp}_s(x := 0, \lambda s' : \langle \varrho \varepsilon \rangle. \lambda(). \text{pre } f () s'|_{\varepsilon} \wedge \forall s'' : \langle \varepsilon \rangle. \text{post } f () s'|_{\varepsilon} s'' () \Rightarrow q s (s'' \oplus s')) \\ \Leftrightarrow & s' = \text{set } x 0 s \Rightarrow \text{pre } f () s'|_{\varepsilon} \wedge \forall s'' : \langle \varepsilon \rangle. \text{post } f () s'|_{\varepsilon} s'' () \Rightarrow q s (s'' \oplus s') \end{aligned}$$

Cutting the proof obligations into pieces, we have to prove that

$$\text{pre } f () s|_{\varepsilon} \Rightarrow s' = \text{set } x 0 s \Rightarrow \text{pre } f () s'|_{\varepsilon}$$

and

$$\text{post } f () s'|_{\varepsilon} s'' () \Rightarrow s' = \text{set } x 0 s \Rightarrow \text{post } f () s|_{\varepsilon} (s'' \oplus s')|_{\varepsilon} ()$$

and finally

$$s' = \text{set } x 0 s \Rightarrow \text{get } (s'' \oplus s') x = 0. \quad (4.5)$$

In W without any restrictions, all these proof obligations are false, because the specification of apply_reset assumes that the effect ϱ does not belong to the effect described by ε . But, depending on the effect instantiations, this assumption may be false.

The proof obligations become provable if we add to each of them the hypothesis that $\varrho \notin \varepsilon$. Taking for example the obligation (4.5), the conclusion can be simplified to

$$\text{get } s' x = 0$$

and the proof obligation becomes trivial. Similar considerations are true for the other formulas.

If we activate the region aliasing exclusion, then the additional hypothesis $\varrho \notin \varepsilon$ comes for free. However, this comes at a cost, as one cannot use apply_reset with a function f whose effect contains ϱ . On the one hand, this restriction is a natural one whenever higher-order functions and mutable state play together. The classical example are iterators over mutable data structures. It is a common requirement (Krishnaswami, 2006) of such functions, often stated informally, that the iteration function does not modify the structure to be iterated on. On the other hand, one could also explicitly increase the effect of f when defining apply_reset , to $\varrho \varepsilon$, for example.

4.2. Singleton Regions

As we have already discussed, it is necessary for reasoning about shared mutable state to allow multiple reference cells per region. On the other hand, if the program at hand does not exhibit this kind of behavior, it would be nice to obtain more precise tracking of regions, at the cost of losing shared mutable state. A solution is to introduce

4. A Language without Aliasing

singleton regions, that only allow a single memory location in each region, in addition to excluding region aliasing.

To see how this improves the precision of the region analysis, let us look at the type of the identity function:

$$id : \forall \alpha. \alpha \rightarrow \alpha$$

If we apply the identity function to a reference x of type $\text{ref}_\rho \alpha$, we obtain another object, say r , of the same type. The postcondition of id probably states that the result r is identical to the argument x , so we can *prove* that x and r are actually the same reference.

On the other hand, if we only deal with singleton regions, we do not even need the postcondition of id ; both x and r are in the *same* region, so they necessarily denote the same reference. In this setting, the variable names of references become irrelevant; the only important information is the name of the region.

The idea presented here to obtain singleton regions is very simple: if for each region, the function ref is called only once, there cannot be multiple memory locations in the same region. As a consequence, regions and memory locations become equivalent.

To implement this restriction, we use once again our effect system, with a modified representation (see Section 2.1.5) of effects. In particular, our effects now are pairs (φ, ω) , where φ is a usual effect expression, and ω is a *creation effect*. This component describes the set of regions in which an expression allocates a reference. The effect ω has the same form as φ , *i.e.*, it is composed of regions and effect variables.

The region removal operation $(\varphi, \omega) \setminus \rho$ is somewhat peculiar: for region variables ϱ , it works as expected, applying the corresponding operation on both components:

$$(\varphi, \omega) \setminus \varrho = (\varphi \setminus \varrho, \omega \setminus \varrho)$$

But for region constants r , the operation does not erase a potential creation effect in r :

$$(\varphi, \omega) \setminus r = (\varphi \setminus r, \omega)$$

The idea behind this definition is that the `letregion` construct still entirely hides a region from the outside, but the `region` construct only hides read and write effects, not creation effects. This makes a soundness proof possible.

The effect union of these modified effects, denoted \uplus , is a partial function and is defined as follows:

$$(\varphi_1, \omega_1) \uplus (\varphi_2, \omega_2) = (\varphi_1 \cup \varphi_2, \omega_1 \cup \omega_2) \quad \text{if } \omega_1 \text{ is disjoint from } \omega_2$$

The idea is that we only have the right to build the union of effects if their creation effects are disjoint. Looking at the typing rules of Fig. 2.5 (page 52), the only place where effect unions appear is the LET rule:

$$\text{LET} \frac{\Gamma; \Sigma \vdash e_1 : \tau', \varphi_1 \quad \Gamma, x : \tau'; \Sigma \vdash e_2 : \tau, \varphi_2}{\Gamma; \Sigma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau, \varphi_1 \cup \varphi_2}$$

This means that our modified effect union operation disallows chaining expressions if they have common creation effects. In turn, this means that no region can contain more than one reference.

We need to apply another modification to the typing rules if we want to ensure this. The `ref` function needs to actually produce a creation effect:

$$\text{Typeof}(\text{ref}) := \forall \alpha \varrho. \alpha \rightarrow^{(e, \varrho)} \text{ref}_{\varrho} \alpha$$

All other previously introduced function constants have an empty creation effect.

Soundness of the Restriction to Singleton Regions

We want to prove the soundness of the extension, *i.e.*, that each region contains only one memory location. Of course, we also want to keep type soundness as stated in Theorem 2.23. But this theorem does not hold with creation effects. Let us see why. Consider the configuration

$$s, \text{letregion } \varrho \text{ in } e, \quad (4.6)$$

where e has a creation effect in ϱ . This effect is hidden because the region removal operation for region variables indeed removes the variable ϱ from the read/write effect and the creation effect. Now, consider the reduction of configuration (4.6) using \rightarrow . We obtain

$$s[r \mapsto \emptyset], \text{region } r \text{ in } e[\varrho \mapsto r]. \quad (4.7)$$

Of course, $e[\varrho \mapsto r]$ now has a creation effect in r . But this effect is no longer hidden: the region removal operation for region constants leaves the creation effect unchanged. Therefore, the effect has *increased* when reducing configuration (4.6) to (4.7).

We cannot expect to prove type soundness as stated in Theorem 2.23. Though, to share as much as possible with the proof of that theorem, we exploit the generalization detailed in Section 2.1.5. While we cannot use this generalization to obtain type soundness directly, we can establish an equivalent to Lemma 2.19 easily, if we can prove stability of the union operation and stability of region variable removal when the substitution does not contain the variable in question.

When aliasing of regions is possible, \uplus cannot be stable under region substitutions; think of the creation effects $\{\varrho_1\}$ and $\{\varrho_2\}$ that are only disjoint if ϱ_1 and ϱ_2 are never substituted for the same variable. However, if the substitution ϕ is *compatible* with $\varphi_1 \uplus \varphi_2$, then

$$(\varphi_1 \uplus \varphi_2)\phi = \varphi_1\phi \uplus \varphi_2\phi$$

because, by definition of compatibility, distinct regions and effect variables cannot overlap after the substitution. Therefore, we obtain the stability of \uplus by requiring the region aliasing restriction.

The region removal operation $\varphi \setminus \rho$ is stable when ρ is a fresh region variable. In summary, we obtain a result equivalent to Lemma 2.19, where a simple effect φ is replaced by a composed effect (φ, ω) . We cannot obtain type soundness (subject reduction) like this, because we do not have the property that

$$\varphi \setminus \varrho = \varphi[\varrho \mapsto r] \setminus r$$

which is required in the proof of Chapter 2. This is not surprising as the corresponding result is actually wrong. Given our definition of the effect operations, we cannot expect to give the same effect to all expressions in a reduction chain. In particular, when

4. A Language without Aliasing

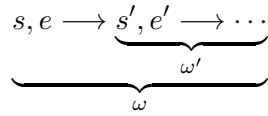


Figure 4.2: A reduction sequence with creation effects.

a `letregion` construct reduces to a `region` construct, the creation effect of that region becomes visible, and thus the overall effect of the expression increases. Our subject reduction lemma has to reflect this fact. In particular, the creation effect ω of an expression has to be in correspondence with the nonempty regions of the store.

$NE(s)$

Definition 4.5. For a store s , the set $NE(s)$ is the set of nonempty regions in s .

Let us look at the problem from another angle. The creation effect ω of an expression e describes the set of regions in which the configuration (s, e) , while reducing, can create a reference. When (s, e) reduces to (s', e') , e' can be typed with another creation effect ω' . Fig. 4.2 shows this situation. We have seen that ω' is not necessarily smaller than ω , because hidden creation effects may have been uncovered. However, the important property is that no creation effect happens in a nonempty region. So the property to be maintained by the reduction relation is that $NE(s) \cap \omega = \emptyset$ for any configuration s, e , where e has creation effect ω . This means that, for any creation effect of e , the corresponding region in s must be empty, so the region will become at most singleton. If we can guarantee that for any configuration (s, e) , any region in s will be singleton or empty. Going back to Fig. 4.2, this means that

$$NE(s) \cap \omega = NE(s') \cap \omega' = \emptyset.$$

The second idea is that for any two configurations (s, e) and (s', e') where the first one is related to the second one by one of the reduction relations \rightarrow or \longrightarrow , then we want to guarantee that $NE(s') \setminus NE(s)$ is contained in the creation effect ω of e ; said otherwise, all regions that become nonempty between s and s' are mentioned in ω .

We proceed just as in the proof of subject reduction; for each reduction rule, we prove that for the new store s' , we can associate an appropriate creation effect ω' that also correctly describes the effect of the new expression. Because there are quite a few hypotheses, let us introduce a predicate that groups these hypotheses together.

Definition 4.6. For a typing environment Γ , a store typing Σ , an expression e , a type τ , an effect φ , a creation effect ω and a store s , we define the predicate $H(\Gamma, \Sigma, e, \tau, \varphi, \omega, s)$ to mean the conjunction of the following properties:

1. $\Gamma; \Sigma \vdash e : \tau, (\varphi, \omega)$,
2. $\Sigma \vdash s$,
3. s contains only empty or singleton regions,
4. $NE(s) \cap \omega = \emptyset$.

Our goal is to prove that predicate H is maintained during the reduction of an expression, when going from creation effect ω to ω' . However, to be able to prove this result by case analysis of the reduction sequence, we cannot accept *any* creation effect ω' . In fact, only certain modifications of ω are permitted to obtain ω' .

Definition 4.7. For any store s , we define the relation R_s between two creation effects ω and ω' to hold if one of the following conditions is true:

 $R_s(\omega, \omega')$

1. $\omega' = \omega$;
2. $\omega' = \omega \cup \{r\}$ where r is fresh with respect to s ;
3. $\omega' = \omega \setminus \{r\}$ where $r \in \omega$.

At each reduction step, we have to find a ω' that is in relation with ω . In the following proof, we only state that there *exists* such a ω' , but it could be defined depending on the reduction rule applied. Whenever the store remains the same, or only read/write effects occur, we can set $\omega' = \omega$. When a creation effect happens, *i.e.*, on a call to *ref*, we can remove it from the effect and set $\omega' = \omega \setminus \{r\}$. When a hidden creation effect is uncovered, *i.e.*, on the reduction of a *letregion* expression, we need to add it to the creation effect: $\omega' \cup \{r\}$.

As we have already done in the previous chapters, we start with a hypothesis on the constant reduction function δ .

Hypothesis 4.8. For each defined mapping $(s, c, \bar{v} \mapsto s', v')$ in δ , if we set $e = c v_1 \dots v_n$ and $e' = v'$, then $H(\Gamma, \Sigma, e, \tau, \varphi, \omega, s)$ implies $H(\Gamma, \Sigma', e', \tau, \varphi, \omega', s')$ for some Σ' and ω' such that $R(\omega, \omega')$. Additionally, we have $NE(s') \setminus NE(s) \subseteq \omega$.

Proof for the constants of Definition 2.5. As usual, we can only check these properties for our partial definitions of δ and *Typeof()*. While the well-typing properties and the store typing Σ' can be obtained using Hypothesis 2.17, we need to give a new creation effect ω' . Whenever the set of nonempty regions in s is unchanged, we can choose $\omega' = \omega$; this fulfills the properties of H and v' can be given any effect including ω' . The only special case, as expected, is the case of *ref*.

The function *ref* creates a reference in a region. In our proof, we must be sure that this cannot happen in a region that already contains a memory location. But of course, if the region instantiation of *ref* is r , then $r \in \omega$, and as $NE(s) \cap \omega = \emptyset$, we know that $r \notin NE(s)$, stated otherwise, $s(r)$ is empty. If we choose $\omega' = \omega \setminus \{r\}$, ω' is of one of the expected forms, and all the side conditions hold. In particular, $NE(s') \cap \omega' = \emptyset$ because basically r moved from ω to s' , and $NE(s') \setminus NE(s) = r \subseteq \omega$ as already stated. Finally, we can type e' with creation effect ω' as the result is a value and can be typed with any creation effect. \square

Remark. The hypothesis on δ basically states that there are no constants that do create several references in a single region, and every constant that *does* create a reference has to declare this in its creation effect.

We now prove the same property for the relation \multimap .

4. A Language without Aliasing

Lemma 4.9 (Top Level Subject Reduction for Singleton Regions). *For any s, e and s', e' such that $s, e \rightarrow s', e'$, then $H(\Gamma, \Sigma, e, \tau, \varphi, \omega, s)$ implies $H(\Gamma, \Sigma', e', \tau, \varphi, \omega', s')$ for suitable Σ' and ω' such that $R_s(\omega, \omega')$ holds. Additionally, we have $NE(s') \setminus NE(s) \subseteq \omega$.*

Proof. We remark that, while the corresponding variant of Lemma 2.21 does not hold, its *proof* is still useful here; we can use it to find an appropriate Σ' to type e' . So, for each possible reduction, we only need to find a ω' that fulfills the necessary conditions.

Cases (β) and (let) The conclusion follows easily from the variant of Lemma 2.19 (Substitution). We can set $\omega' = \omega$.

Case (δ) We can conclude by Hypothesis 4.8.

Cases (iftrue) and (iffalse) We obtain the statement $\Gamma; \Sigma \vdash e' : \tau, (\varphi, \omega)$ by inversion of the typing derivation of $e = \text{if } v \text{ then } e_1 \text{ else } e_2$. We can set $\omega' = \omega$ because s is unchanged.

Case (region) The value v can be typed with an empty effect; we can of course increase this effect and set $\omega' = \omega$; this works because s remains unchanged.

Case (letregion) Let $e = \text{letregion } \varrho \text{ in } e_1$. In this case, a new empty region is created, $s' = s[r \mapsto \emptyset]$ for a fresh r . We therefore set $\omega' = \omega \cup \{r\}$ so that this region can be populated with a single memory location. The proof of Lemma 2.21 gives us an appropriate store typing Σ' , and we can prove $\Gamma; \Sigma' \vdash \text{region } r \text{ in } e_1 : \tau, (\varphi, \omega')$ because we can always obtain that r is included in the creation effect of e_1 . As $s'(r)$ is empty, we still have $NE(s') \cap \omega' = \emptyset$, and $NE(s') \setminus NE(s) = \emptyset$.

□

Before we proceed to the next lemma, let us prove a little property on creation effects.

Proposition 4.10. *For any creation effects ω_1, ω'_1 and ω_2 , and store s such that $R_s(\omega_1, \omega'_1)$, if ω_1 and ω_2 are disjoint, then ω'_1 and ω_2 are disjoint as well and we have $R_s(\omega_1 \cup \omega_2, \omega'_1 \cup \omega_2)$.*

Proof. By case analysis on the three different ways to be related that are listed in Definition 4.7.

Case $\omega_1 = \omega'_1$: In this case there is nothing to prove.

Case $\omega'_1 = \omega_1 \setminus \{r\}$ with $r \in \omega_1$: Obviously, ω'_1 and ω_2 are disjoint. We also have $r \notin \omega_2$, and therefore $\omega'_1 \cup \omega_2 = (\omega_1 \cup \omega_2) \setminus \{r\}$ with $r \in \omega_1 \cup \omega_2$.

Case $\omega'_1 = \omega_1 \cup \{r\}$ with r fresh: The region constant r is fresh and cannot be contained in ω_2 . Therefore, ω'_1 and ω_2 are still disjoint. We have the same relation between the resulting sets.

□

We proceed to prove the required property for the relation \longrightarrow .

Lemma 4.11 (One Step Subject Reduction for Singleton Regions). *For any s, e and s', e' such that $s, e \rightarrow s', e'$, then $H(\Gamma, \Sigma, e, \tau, \varphi, \omega, s)$ implies $H(\Gamma, \Sigma', e', \tau, \varphi, \omega', s')$ for suitable Σ' and ω' such that $R_s(\omega, \omega')$. Additionally, we have $NE(s') \setminus NE(s) \subseteq \omega$.*

Proof. We proceed by structural induction over the reduction context E . In the case of the empty context \square , we can conclude by Lemma 4.9. Two cases remain.

Case let In this case, we have $e = \text{let } x = E[e_1] \text{ in } e_2$ and we have $s, e_1 \rightarrow s', e'_1$. The typing derivation for e looks like this:

$$\text{LET} \frac{\Gamma; \Sigma \vdash E[e_1] : \tau', (\varphi_1, \omega_1) \quad \Gamma, x : \tau'; \Sigma \vdash e_2 : \tau, (\varphi_2, \omega_2)}{\Gamma; \Sigma \vdash \text{let } x = E[e_1] \text{ in } e_2 : \tau, (\varphi_1, \omega_1) \uplus (\varphi_2, \omega_2)}$$

We can derive that ω_1 and ω_2 are disjoint and we have $NE(s) \cap (\omega_1 \cup \omega_2) = \emptyset$. By consequence, we also have $NE(s) \cap \omega_1 = \emptyset$. By the induction hypothesis applied to $E[e_1]$, we obtain

$$\Gamma; \Sigma \vdash E[e'_1] : \tau', (\varphi_1, \omega'_1)$$

for some ω'_1 with $NE(s') \cap \omega'_1 = \emptyset$ and $NE(s') \setminus NE(s) \subseteq \omega_1$. We know that ω'_1 is related to ω_1 in one of the ways described in Definition 4.7. By Proposition 4.10, $\omega_1 \cup \omega_2$ and $\omega'_1 \cup \omega_2$ are related in the same way and ω'_1 and ω_2 are disjoint. We derive that e is well-typed with this creation effect. It is also easy to see that $NE(s') \cap (\omega'_1 \cup \omega_2) = \emptyset$ and $NE(s) \setminus NE(s') \subseteq \omega'_1 \cup \omega_2$. Finally, the fact that s' contains only empty or singleton regions is derived from the induction hypothesis.

Case region We have $e = \text{region } r \text{ in } E[e_1]$ and $s, e_1 \rightarrow s', e'_1$. The typing derivation looks like this:

$$\text{REGION} \frac{\Gamma; \Sigma \vdash E[e_1] : \tau, (\varphi_1, \omega_1)}{\Gamma; \Sigma \vdash \text{region } r \text{ in } E[e_1] : \tau, (\varphi_1, \omega_1) \setminus r}$$

Looking at the definition of the region removal operation for region constants, we can see that the `region` construct does not change the creation effect of an expression. We therefore obtain that $NE(s) \cap \omega_1 = \emptyset$ and we can apply the induction hypothesis on $E[e_1]$, obtaining a creation effect ω'_1 for $E[e'_1]$. We can set $\omega'_1 = \omega_1$ and are done. □

We can now prove the main result for the reduction relation \rightarrow :

Theorem 4.12 (Subject Reduction for Singleton Regions). *For any reduction sequence $s, e \rightarrow s', e'$, the hypothesis $H(\Gamma, \Sigma, e, \tau, \varphi, \omega, s)$ implies $H(\Gamma, \Sigma', e', \tau, \varphi, \omega', s')$ for some store typing Σ' and some creation effect ω' .*

Proof. By induction over the length of the reduction chain. Note that we can no longer expect $R_s(\omega, \omega')$ because there can be a bigger difference between ω and ω' than one creation effect. However, this property is not required in the proof. □

4. A Language without Aliasing

Corollary. *For any expression e such that $\emptyset; \emptyset \vdash e : \tau, (\varphi, \omega)$ and $\emptyset, e \rightarrow s, v$, the state s contains only singleton regions and empty regions. The same is true for any store s' that appears in this reduction sequence.*

Proof. The first claim of this corollary is a special case of Theorem 4.12. The second claim concerning intermediate stores can be obtained by the following reasoning. During execution, the domain of the store, and the domain of each region of the store, only grows and can never shrink. As a consequence, the region domain of an intermediate store s' must be contained in the domain of s , and for each region in s' , the domain of that region in s' must be contained in the domain of that region in s . Hence, any region in s' is at most singleton. \square

Exploiting singleton regions. There are many equivalent ways in which the fact that all regions are singleton can be exploited. The simplest one would be to state with an axiom that, when the region is fixed, `get` becomes a constant function:

$$\forall \alpha \varrho \varepsilon. \forall x, y : \text{ref}_{\varrho} \alpha. \forall s : \langle \varrho \varepsilon \rangle. \text{get } x \ s = \text{get } y \ s$$

Another possibility is to apply this axiom whenever possible in terms; one might chose a reference variable x as the canonical reference of a certain region ϱ , and replace all accesses to that region by accesses using x .

A simple example of a situation that leads to simpler proof obligations is the following program:

```

let id [ $\alpha$ ] ( $a : \alpha$ ) = { ... } a { ... }
let idref [ $\alpha \varrho$ ] ( $x : \text{ref}_{\varrho} \alpha$ ) =
  { }
  let  $y = \text{id } x$  in
  ! $y$ 
  {  $r : r = !!x$  }

```

Both x and y have type $\text{ref}_{\varrho} \alpha$. In a setting with singleton regions, this information is already enough to conclude that x and y are actually the same reference, and that the access to x and to y must return the same value. We can derive this *without any annotations of the function id*. In a setting with group regions, we would need an annotation stating that the returned value is identical to the argument.

Mixing group regions and singleton regions. After the previous sections, the natural question to ask is whether singleton regions and group regions can coexist in a single system. A user could then decide to use singleton regions for the majority of the mutable state of the program, and group regions when reasoning about shared mutable state is necessary. Such a combination would combine the best of both worlds.

While we have not worked out the details, we believe that this is possible. When designing such a system, one should take care that one never assumes a region to be singleton when this is not the case, and one should not lose precision by “forgetting” that a region is singleton. A simple solution would consist in letting the user specify which regions are singleton and which are not. This would be verified on region instantiation, for example. Effect union would be more complex, because now the disjointness of

creation effects must be checked, but only for singleton regions. The axiom for singleton regions could then be added only for the regions that are indeed singleton regions. Automatic transformation of proof obligations would be more complex, however.

Capabilities (Smith et al., 2000) also provide the coexistence of singleton and group regions using mechanisms such as adoption and focus, at the cost of a slightly more complex type system and more effect annotations. Compared to our hypothetical system that combines singleton and group regions, the mechanisms of adoption and focus permit to *merge* a singleton region with a group region, and to *focus* on a particular reference in a group region to obtain, in fact, a singleton region. See also page 33 for a more detailed discussion of capabilities.

5. Implementation and Case Studies

In the preceding chapters, we have described a theoretical system to prove properties of higher-order programs with side effects. We now want to demonstrate that our system is useful in practice. We therefore present our implementation of the system, called *Who*, detail its internals, present practical aspects that have been eluded from the discussion so far and give examples of programs that have been proved correct using *Who*.

5.1. The *Who* Tool

The *Who* tool (Kanig and Filiâtre, 2009; Kanig, 2010) is an implementation of the techniques presented so far. *Who* consists of an implementation of the programming language W and the specification language L , with syntactic sugar that has partly been presented in Chapter 2. *Who* also contains an implementation of the weakest precondition calculus presented in Chapter 3. *Who* programs have to respect the region aliasing restriction from Section 4.1; the singleton region restriction of Section 4.2 is optional, on a per-program basis.

The *Who* tool accepts W programs with specifications written in L , and outputs proof obligations in a standard higher-order logic that we call L_0 , a logic that is identical to L , with the exception of the absence of constructs related to effects, *i.e.*, in particular state types of the form $\langle\varphi\rangle$.

The Architecture of *Who*

Internally, *Who* consists of several intermediate languages, and functions that translate expressions of one language to expressions of another. We now discuss each intermediate language and each function briefly, and refer to the sections in which they are discussed in more detail. Fig. 5.1 gives an overview of the architecture of the tool.

The input language. *Who* does accept programs in a language that is a bit richer and more convenient than W . In particular, it accepts arbitrary application between expressions, it does a limited form of type, region and effect inference and incorporates a number of syntactic conveniences, such as the ones that have been discussed in Section 2.1.1. This includes fixing the names of the state variables in pre- and postconditions to *cur* and *old*, permitting function definitions using *let* and *let rec*, *for*-loops and more. Language features that are not present in W include algebraic data types and pattern matching, inductive definitions of predicates, declarations of type constants, definitions of type abbreviations and declarations of logic functions.

We call the inference mechanism limited, because in particular effect instantiations have to be given for each use of an effect polymorphic variable, when several solutions

5. Implementation and Case Studies

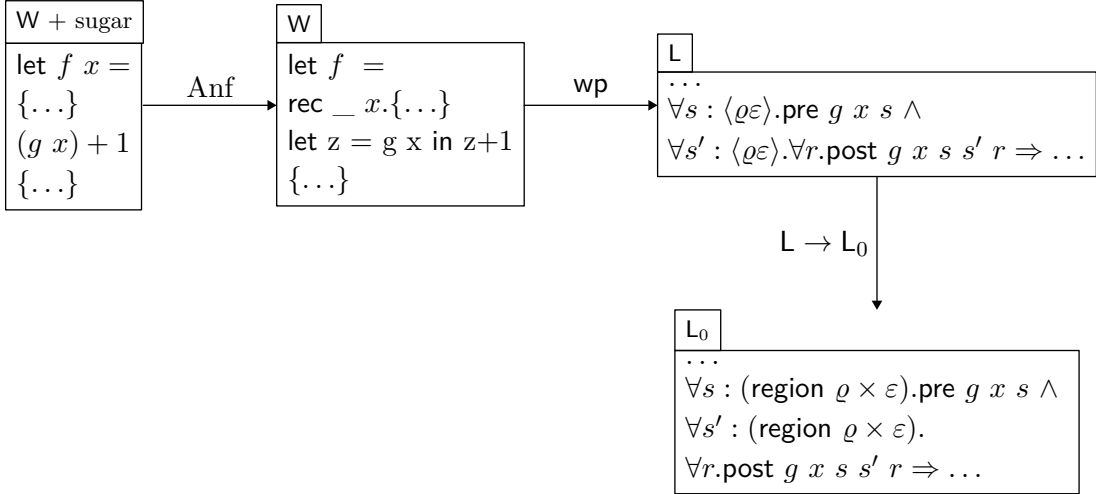


Figure 5.1: The Architecture of the Who tool.

exist. In these ambiguous cases, we choose to insist on a choice by the user, because different effect instantiations can lead to proof obligations of varying difficulty.

Maybe the most important syntactic sugar that has not been discussed yet are *Hoare triples* in the logic. In *Who* annotations, the following syntax is accepted:

$$\{ p \} e \{ q \}$$

where p and q , as in function annotations, may refer to the state variables `old` and `cur`. The meaning of this formula is that if p holds for some initial state, then q holds after executing e in that state. Using our weakest precondition calculus, this is of course equivalent to the formula

$$\forall s. p \ s \Rightarrow \text{wp}_s(e, q \ s). \quad (5.1)$$

In the common case where e is the application between two variables f and x , this can be translated (“desugared”) to the following conjunction:

$$\forall s. p \ s \Rightarrow \text{pre } f \ x \ s \ \wedge \ \forall s' r. \text{post } f \ x \ s \ s' \ r \Rightarrow q \ s \ s' \ r$$

We use Hoare triples extensively in Section 5.4, which discusses programs that have been proved correct using *Who*.

A-normal form. The aim of the transformation that is called “Anf” in Fig. 5.1 is to remove all the syntactic sugar and to put programs in A-normal form (see Section 2.1.1). The output language of this transformation is very close to *W*, with the exception of minor superficial differences and Hoare triples, which are removed in the next phase. The A-normal form transformation itself is continuation-based and is an extension of the one by Flanagan et al. (1993).

Weakest preconditions. The weakest precondition calculus, that we have described in detail in Chapter 3, is used in *Who* to transform an expression in *W* into a formula in

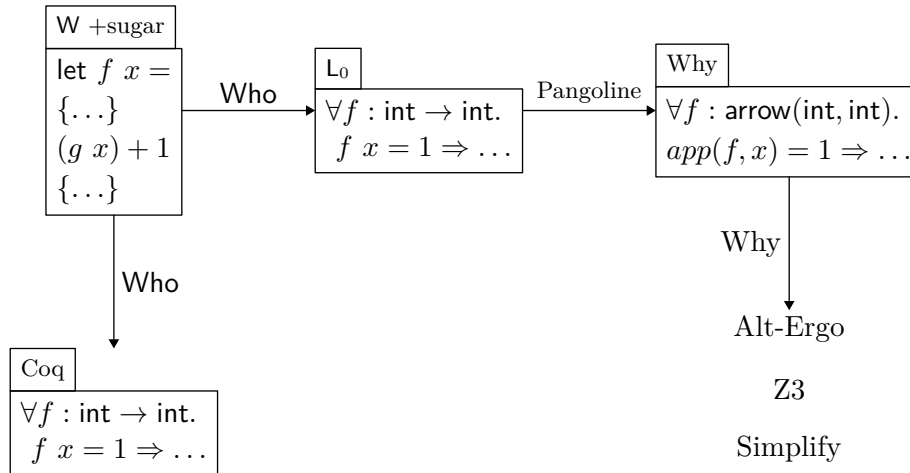


Figure 5.2: Using Who, Pangoline, Why, Coq and automated provers to discharge proof obligations.

L. The calculus implemented in Who is identical to the one in Chapter 3, including the extensions concerning read/write effects and algebraic data types. During this phase, Hoare triples of the form

$$\{ p \} e \{ q \}$$

are replaced by the corresponding formula in L given by (5.1).

Obtaining formulas in L_0 . We have said that Who outputs formulas in the subset L_0 of L that does not contain state types. We show in Section 5.2 how formulas in L can be translated to formulas in L_0 .

What to do with the produced formulas. Who is only a verification condition *generator*, *i.e.*, it simply takes an annotated program and outputs verification conditions, or proof obligations, that imply the correctness of the input program with respect to its specification. Who itself, however, does not do any attempt to *prove* them, except the most trivial ones. If the formula to be proved is already contained in the hypotheses, or is equal to True, or if False is contained in the hypotheses, and in very few other trivial cases, Who can automatically discharge proof obligations.

In general, however, other tools are necessary to prove these obligations, given in the language L_0 , a polymorphic higher-order logic. L_0 is a subset of the logics of most interactive theorem provers, which could be used to prove the obligations manually. However, currently only Coq syntax is supported directly by Who.

Another desirable way to prove these obligations would be to use automated theorem provers. There is a big problem however: most automated provers only support first-order logic, while L_0 is a higher-order logic. We therefore propose an *encoding* of L_0 to polymorphic first-order logic; the logic of the Why tool is an example of a first-order logic that contains everything we need. We have developed and implemented

this encoding in collaboration with Yann Régis-Gianas in a tool called Pangoline. The translation on which Pangoline is based is detailed in Section 5.3.

It is interesting for us to target the Why language, because Why is capable of generating files for many different automated and interactive provers, which can then all be used to discharge proof obligations.

The connection between the different languages and tools is described in Fig. 5.2. In practice, one does not need to execute all the different tools by hand. Instead, a script permits to run automated provers on all the proof obligations generated from a Who file, using the chain Who – Pangoline – Why. Selected proof obligations — usually the ones no automated prover could prove — can then be generated in Coq format for manual proof.

5.2. Translation from L to L₀

As we have seen in the introduction, the first obstacle for proving the proof obligations of a program is that such obligations are terms in the logic L which contains non-standard state types. Automated or interactive provers, even the ones with higher-order logic, do not deal with these types. In this section, we therefore describe a practical translation from L to a sublanguage of L without state types. This sublanguage, called L₀, is a subset of common higher-order logics such as Coq’s or Isabelle’s. In this section, we assume the aliasing regions restriction of Section 4.1 to be applied.

The logic L₀. We do not give an explicit definition with semantics and typing rules for L₀. Instead, we say that L₀ is the subset of L that does not contain state types of the form $\langle \varphi \rangle$, nor region and effect polymorphism and region and effect instantiations. We also remove the “current state” ϑ and memory locations l ; they have only been introduced into L to be able to carry out the soundness and completeness proofs of the wp calculus, because in this proof we have to manipulate programs and annotations that are partially evaluated. Here, we are only interested in annotated programs that have not been evaluated yet. This means that the store s and the store typing Σ are empty. The current state ϑ is therefore useless and memory locations l cannot appear in a well-typed formula. As a result, L₀ is very close to (subsets of) other logics, such as Coq’s logic without dependent types, the logic of HOL (Gordon, 2000) and the logic used in the work of Régis-Gianas and Pottier (2008).

n -ary tuples. In this section, we assume that L₀ possesses n -ary tuple types or n -tuples, of the form $\tau_1 \times \tau_2 \times \dots \times \tau_n$. This is not a very strong requirement; first, types of this form are present in many provers. Second, n -ary tuples can of course be encoded using pairs. Pairs are in L and L₀, so adding n -tuples does not add any expressive power to the language, but can be seen as syntactic sugar. We also assume that there are constants mk_n that construct tuples, for each n , and projection constants π_i^n that return the i th component of an n -tuple, where i is an integer between 1 and n .

$$\begin{aligned} mk_n &: \forall \alpha_1 \dots \alpha_n. \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \alpha_1 \times \dots \times \alpha_n \\ \pi_i^n &: \forall \alpha_1 \dots \alpha_n. \alpha_1 \times \dots \times \alpha_n \rightarrow \alpha_i \end{aligned}$$

We follow the common usage to write (t_1, \dots, t_n) instead of $mk_n t_1 \cdots t_n$. In most cases, we omit the parameter n of both types of functions; usually, it can easily be derived from the context, and plays no role in the proofs.

The idea of the translation. The aim of the translation we are about to describe is to eliminate state types of the form $\langle \varphi \rangle$ from our formulas. There are other possibilities, but we decide to represent state types $\langle \varphi \rangle$ in \mathbf{L} by n -ary tuple types in \mathbf{L}_0 , where n is the number of elements in the effect expression φ , and objects of type $\langle \varphi \rangle$ are represented by n -tuples. Let us denote the translation function from \mathbf{L} to \mathbf{L}_0 by $\llbracket \cdot \rrbracket$. Then, as an example,

$$\llbracket \langle \rho_1 \rho_2 \varepsilon \rangle \rrbracket = \llbracket \rho_1 \rrbracket \times \llbracket \rho_2 \rrbracket \times \llbracket \varepsilon \rrbracket,$$

where $\llbracket \rho_1 \rrbracket$, $\llbracket \rho_2 \rrbracket$ and $\llbracket \varepsilon \rrbracket$ are the types that result from the translation of the region and effect variables.

Effect variables can simply be encoded using type variables. For the images of effect variables, instead of maintaining an environment that keeps track of the mapping from effect variables to type variables that represent them, we simply use the name of the effect variable as the name of the corresponding type variable. Now, we can write

$$\llbracket \varepsilon \rrbracket = \varepsilon,$$

meaning that the *effect* variable ε is translated to the *type* variable ε . This is an abuse of notation, but it should be clear what is meant by this equation.

Region *names* can be translated using the same trick, and region polymorphism can be expressed using type polymorphism. However, we need to express *regions*, the parts of the store, differently. For this, we use a unary type constructor **region**, such that **region** ϱ represents the region corresponding to ϱ . We write:

$$\llbracket \varrho \rrbracket = \mathbf{region} \ \varrho$$

Again, on the right hand side, ϱ is a type variable.

On our example, the state type $\langle \rho_1 \rho_2 \varepsilon \rangle$ in \mathbf{L} becomes

$$\llbracket \langle \rho_1 \rho_2 \varepsilon \rangle \rrbracket = \mathbf{region} \ \rho_1 \times \mathbf{region} \ \rho_2 \times \varepsilon,$$

where ρ_1 , ρ_2 and ε are now *type* variables.

There are two problems with our translation. The first one is that in the effect calculus, the two types $\langle \rho_1 \rho_2 \rangle$ and $\langle \rho_2 \rho_1 \rangle$ are considered the same, because effects are sets and both sets contain the same elements. However, in our translation based on tuples, the order becomes crucial: The type **region** $\rho_1 \times \mathbf{region} \ \rho_2$ is different from the type **region** $\rho_2 \times \mathbf{region} \ \rho_1$. In this form, the problem can be easily dealt with: simply fix an arbitrary order, written $<$, on all basic effects, *i.e.*, region variables and effect variables, and always order the representation of effects based on this ordering. So, assuming that $\rho_1 < \rho_2$, the effect expression $\rho_2 \rho_1$ actually stands for $\rho_1 \rho_2$, and both state types are translated to the same tuple type.

But this resolves only half the issue. The more serious problem is that our translation does not behave well in connection with effect substitutions. To see the problem, consider our state type $\tau_1 = \langle \rho_1 \rho_2 \varepsilon \rangle$ and instantiate the effect variable ε with the effect

5. Implementation and Case Studies

$\rho_0\rho_3$. We assume the region variables to be ordered such that $\rho_i < \rho_j$ whenever $i < j$. Now, set

$$\tau_2 = \tau_1[\varepsilon \mapsto \rho_0\rho_3] = \langle \rho_0\rho_1\rho_2\rho_3 \rangle.$$

The translation of τ_2 is the type

$$\llbracket \tau_2 \rrbracket = \text{region } \rho_0 \times \text{region } \rho_1 \times \text{region } \rho_2 \times \text{region } \rho_3,$$

while the translation of type τ_1 is

$$\llbracket \tau_1 \rrbracket = \text{region } \rho_1 \times \text{region } \rho_2 \times \varepsilon.$$

If we translate the *effect* substitution $[\varepsilon \mapsto \rho_0\rho_3]$ to a *type* substitution in L_0 , such as $\phi = [\varepsilon \mapsto \text{region } \rho_0 \times \text{region } \rho_3]$, then

$$[\tau_1]\phi = \text{region } \rho_1 \times \text{region } \rho_2 \times (\text{region } \rho_0 \times \text{region } \rho_3).$$

It is important to note that the parentheses around the last tuple are necessary, and that this type indeed describes a *nested tuple*, *i.e.*, a tuple inside a tuple. So not only has the problem of order reappeared, but also the *structure* of the types is not the same.

While the structure is not the same, both types “morally” represent the same objects, namely the ones that consist of a component of type $\text{region } \rho_i$ for each i between 0 and 3. And indeed, it is easy to find a conversion function between both types: The term

$$f = \lambda s : [\tau_2].(\pi_2 s, \pi_3 s, (\pi_1 s, \pi_4 s))$$

is a function of type $[\tau_2] \rightarrow [\tau_1]\phi$, and similarly we could construct a function of inverse type.

Extending the conversion between state types to all types. Before we explain the details of the translation, we want to show that the construction of bijections that we just sketched can be carried over to terms of more complex types. In this paragraph, we assume that we have a set of bijective conversion functions $f_{a \rightarrow b}$ for a set of pairs (a, b) of types. This can be, for example, the function f that converts between the different tuple structures that we have described in the previous paragraph.

We can now define a conversion function that converts between types, say τ_1 and τ_2 , that differ only by types a and b , for which we have a conversion function $f_{a \rightarrow b}$. This means that τ_2 can be obtained from τ_1 by substituting occurrences of a in τ_1 by b .

Our approach is that we define a meta-function M that, given two such types τ_1 and τ_2 , and a term of type τ_1 , returns a term of type τ_2 :

$$\begin{aligned} M(\tau, \tau, t) &= t \\ M(a, b, t) &= f_{a \rightarrow b} && \text{if } f_{a \rightarrow b} \text{ exists} \\ M(\iota \bar{\sigma}, \iota \bar{\sigma}', t) &= \text{map}_{\sigma} \overline{(\lambda x : \sigma. M(\sigma, \sigma', s))} t \\ M(\text{ref } \rho \sigma, \text{ref } \rho \sigma', t) &= \text{mapref } \overline{(\lambda x : \sigma. M(\sigma, \sigma', s))} t \\ M(\tau_1 \rightarrow \tau'_1, \tau_2 \rightarrow \tau'_2, t) &= \lambda x : \tau_2. M(\tau'_1, \tau'_2, t M(\tau_2, \tau_1, x)) \end{aligned}$$

For polymorphic constants and for reference types, we assume that a mapping function has been given which takes the conversion function(s) in argument and returns an object of the expected type.

$$\begin{aligned}
\llbracket \iota \bar{\sigma} \rrbracket &= i \overline{\llbracket \sigma \rrbracket} \\
\llbracket \alpha \rrbracket &= \alpha \\
\llbracket \sigma \rightarrow \sigma \rrbracket &= \llbracket \sigma \rrbracket \rightarrow \llbracket \sigma \rrbracket \\
\llbracket \text{ref}_\varrho \tau \rrbracket &= \text{ref } \varrho \llbracket \tau \rrbracket \\
\llbracket \langle \varrho_1 \cdots \varrho_m \varepsilon_1 \cdots \varepsilon_n \rangle \rrbracket &= \text{region } \varrho_1 \times \cdots \times \text{region } \varrho_m \times \varepsilon_1 \times \cdots \times \varepsilon_n
\end{aligned}$$

Figure 5.3: The translation of L types to L₀ types

Translating types. The translation of types from L to L₀ is very simple and is summarized in Fig. 5.3. It is simply a traversal of the type structure until we encounter a type of the form $\langle \varphi \rangle$. At this point, we apply the translation to tuples we have discussed earlier. The translation of reference types merits a comment: the type $\text{ref}_\varrho \tau$, a special type in L, is translated to the application of a binary type constant ref to the type variable ϱ and the type $\llbracket \tau \rrbracket$.

Translating state manipulations. As we are replacing state types by tuples, we must also replace the functions manipulating state types by something else. Now, we can concretely implement these functions, specialized for each application. The idea is very simple, but is relatively cumbersome to write formally. We first define a metafunction *find* that, given a region or effect variable, returns the corresponding projection in the translation of a state object s . Let us write effects using a list of basic effects χ and let us assume that s is a state object of type $\langle \chi_1 \cdots \chi_n \rangle$. Then

$$\text{find } \chi_i s_{\langle \chi_1 \cdots \chi_n \rangle} = \pi_i \llbracket s \rrbracket.$$

The function *find* is of course only defined when χ_i belongs to the domain of s ; in the following, we use *find* only in situations where this is the case. In the definition of *find*, the translation function on terms, $\llbracket \cdot \rrbracket$, is used on s .

We now use this function to build tuples of the right form:

$$\begin{aligned}
\llbracket (\text{combine } s_1 s_2) \langle \bar{\chi} \rangle \rrbracket &= mk \overline{(\text{if } \chi \in \varphi_2 \text{ then } \text{find } \chi s_1 \text{ else } \text{find } \chi s_2)} \\
\llbracket s_{\langle \bar{\chi} \rangle} \rrbracket &= mk \overline{(\text{find } \chi s)}
\end{aligned}$$

We have again used χ to denote effect and region variables, and the list syntax using an overline.

In L, the function *get* acts on the store directly and is of type

$$\text{get} : \forall \alpha \varrho \varepsilon. \langle \varrho \varepsilon \rangle \rightarrow \text{ref}_\varrho \alpha \rightarrow \alpha.$$

As, with the region aliasing restriction, we know statically which region of the store is concerned, we replace this function in L₀ by one that directly acts on regions:

$$\text{get} : \forall \alpha \varrho. \text{region } \varrho \rightarrow \text{ref } \varrho \alpha \rightarrow \alpha.$$

5. Implementation and Case Studies

To use this function, we simply have to pick the right component of a state tuple and apply the function:

$$\llbracket \text{get } s_{\langle \varphi \rangle} x_{\text{ref}_e} \tau \rrbracket = \text{get } (\text{find } \varrho \varphi) \llbracket x \rrbracket$$

Remark. We have presented the translation of state manipulating functions in their fully applied form. However, only the instantiation of the function variables is important and guides the translation process. Partial applications of `get`, `restrict` and `set` are perfectly possible.

The actual translation. We now can define the translation from L to L_0 . It is actually very simple and consists in translating all forms of polymorphism to type polymorphism and translating all state types to tuple types. The only difficulty is that due to effect instantiations, conversion functions have to be inserted to make the term well-typed. In the definition of the translation, we expect all terms in L to be well-typed, and we assume the variables and constants to be annotated with their type schemes.

Let us start with the most difficult case concerning variables (the one for constants is identical). Let us try to translate the term $t = x [\overline{z}]$, where x is a variable of type scheme $\forall \overline{\chi}. \tau_x$. A first try would be to simply translate the instantiations, to obtain the term

$$t' = x \llbracket \overline{z} \rrbracket,$$

and if we assume that the type scheme of x has been translated to $\forall \overline{\chi}. \llbracket \tau_x \rrbracket$, the type of this term is

$$\tau' = \llbracket \tau_x \rrbracket [\overline{\chi} \mapsto \llbracket \overline{z} \rrbracket].$$

The type of term t in L is $\tau_x [\overline{\chi} \mapsto \overline{z}]$, and if we translate this type to L_0 , we obtain

$$\tau = \llbracket \tau_x [\overline{\chi} \mapsto \overline{z}] \rrbracket.$$

So we have obtained a term of type τ' , but to be able to build a correct typing derivation which contains t' , we need to obtain a term of type τ . However, we have seen that there are situations where τ is different from τ' . We therefore insert a conversion:

$$\llbracket t \rrbracket = M(\tau', \tau, t').$$

And this term has the right type. The case of constants is identical.

All other cases are very simple, and consist of a simple traversal of the term structure, converting polymorphic variables and variable instantiations on the way. We recall that all polymorphic variables in L_0 are type variables, so for example in the case

$$\llbracket \forall \overline{\chi}. t \rrbracket = \forall \overline{\chi}. \llbracket t \rrbracket$$

we have effect variables, region variables and type variables on the left, but only type variables on the right. Fig. 5.4 summarizes the translation for terms.

Remark. The reader may be worried that this translation clutters the formula with tuples and projections everywhere. However, in practice, many simplifications can apply. Projections applied to concrete tuples can be reduced as follows

$$\pi_i (mk_n t_1 \cdots t_n) \rightarrow t_i.$$

5.3. Translation from Higher-Order Logic to First-Order Logic

$$\begin{aligned}
\llbracket c_{\forall \bar{\chi}. \tau} \bar{z} \rrbracket &= M(\llbracket \tau \rrbracket [\bar{\chi} \mapsto \overline{\bar{z}}], \llbracket \tau [\bar{\chi} \mapsto \bar{z}] \rrbracket, c \llbracket \overline{\bar{z}} \rrbracket) \\
\llbracket x_{\forall \bar{\chi}. \tau} \bar{z} \rrbracket &= M(\llbracket \tau \rrbracket [\bar{\chi} \mapsto \overline{\bar{z}}], \llbracket \tau [\bar{\chi} \mapsto \bar{z}] \rrbracket, x \llbracket \overline{\bar{z}} \rrbracket) \\
\llbracket t_1 t_2 \rrbracket &= \llbracket t_1 \rrbracket \llbracket t_2 \rrbracket \\
\llbracket \lambda x : \sigma. t \rrbracket &= \lambda x : \llbracket \sigma \rrbracket. \llbracket t \rrbracket \\
\llbracket \forall x : \sigma. t \rrbracket &= \forall x : \llbracket \sigma \rrbracket. \llbracket t \rrbracket \\
\llbracket \forall \bar{\chi}. t \rrbracket &= \forall \bar{\chi}. \llbracket t \rrbracket \\
\llbracket \text{let } x \llbracket \bar{\chi} \rrbracket = t_1 \text{ in } t_2 \rrbracket &= \text{let } x \llbracket \bar{\chi} \rrbracket = \llbracket t_1 \rrbracket \text{ in } \llbracket t_2 \rrbracket \\
\llbracket \text{get } s_{\langle \varphi \rangle} x_{\text{ref}_{\varrho} \tau} \rrbracket &= \text{get } (\pi_{\text{find}_{\varrho} \varphi} \llbracket s \rrbracket) \llbracket x \rrbracket \\
\llbracket (\text{combine } s_1 s_2)_{\langle \bar{\chi} \rangle} \rrbracket &= mk \overline{(\text{if } \chi \in \varphi_2 \text{ then find } \chi s_1 \text{ else find } \chi s_2)} \\
\llbracket s_{\llbracket \bar{\chi} \rrbracket} \rrbracket &= mk \overline{(\text{find } \chi s)}
\end{aligned}$$

Figure 5.4: The translation of L terms to L₀ terms.

To maximize the number of concrete tuples, one can replace quantifications over tuples by quantifications over the components:

$$\forall x : \tau_1 \times \dots \times \tau_n. p(x) \quad \rightarrow \quad \forall x_1 : \tau_1. \dots. \forall x_n : \tau_n. p((x_1, \dots, x_n)).$$

Using only these two transformations, the *Who* tool manages to eliminate all tuples due to state types on first-order and second-order functions. Tuple types that cannot be eliminated appear when proving properties about third-order functions or higher.

5.3. Translation from Higher-Order Logic to First-Order Logic

In this section, we present a translation from a higher-order logic to a first-order logic. This translation has been implemented in a tool called *Pangoline*, in collaboration with Yann Régis-Gianas.

5.3.1. Motivation

As we have seen, *Who* uses a higher-order logic to express annotations and proof obligations. Given the state of the art of automated theorem proving, we cannot expect to discharge all proof obligations of even moderately complex programs automatically. Manual proofs can be done in interactive theorem provers such as *Coq* ([The Coq Development Team, 2008](#)), and this task is made easier by built-in decision procedures, like congruence closure or the tactic *omega* ([Crégut, 2001](#)) to decide properties of linear arithmetic. On the other hand, these decision procedures combine very badly in *Coq* and in practice the user has to progress step by step, preparing the goal manually for a certain decision procedure. Outside the world of interactive theorem provers, solvers that efficiently combine such decision procedures exist. In program proof, a large number of proof obligations are actually relatively easy, often requiring only a few instantiations of lemmas or axioms, as well as a combination of arithmetic reasoning,

5. Implementation and Case Studies

congruence closure and propositional reasoning. We would like to discharge those easy proof obligations automatically, so that the user can concentrate on the difficult ones.

There are two main families of automated theorem provers today.

- *Resolution-based* provers such as Vampire (Riazanov and Voronkov, 2002), Spass (Weidenbach et al., 2009) or the E prover (Schulz, 2004). This is the class that is usually referred to by the term *automated theorem provers* in the literature. These tools are often refutationally complete and very powerful for logical reasoning. However, they are considered less suited for equational reasoning and even less suited for arithmetic reasoning. To our knowledge, with one notable exception (see below), no resolution-based prover supports higher-order logic.
- *SMT-solvers*, which are usually implemented using a SAT-solver, built-in equality and arithmetic reasoning and an instantiation mechanism to make use of lemmas and axioms. Well-known provers are Z3 (de Moura and Bjørner, 2009), Yices (de Moura and Dutertre, 2009), Alt-Ergo (Conchon and Contejean, 2008) Simplify (Detlefs et al., 2005) and CVC3 (Barrett and Tinelli, 2007). While these provers often use complete decision procedures for ground terms and formulas, these tools are generally incomplete in the presence of quantifiers. However, they deal very well with arithmetic reasoning, and some provers also have other built-in theories, such as the theory of arrays, which is very interesting for programs that manipulate arrays. To our knowledge, there is no SMT-solver that deals with higher-order logic.

For the sake of completeness, we need to mention that one automated prover has limited support for higher-order logic: Otter, developed by Beeson (2006). Following Meng and Paulson (2008), the connection of Otter’s logic with the one of Coq or Isabelle, is not clear, however. Meng and Paulson also argue that they do not want to depend on a single automated prover, and we do not want to either. To be able to use all the other provers, we need some encoding of the higher-order features of our logic, to obtain equivalent first-order formulas that can be sent to those automated provers.

Polymorphism. While the logic L_0 , the output format of Who and the input format of Pangoline, is polymorphic, in this section we define a monomorphic higher-order logic that only provides polymorphic *constants*. The user cannot define his own polymorphic functions. The same is true for the target first-order logic. The removal of polymorphism from both languages is motivated by a simpler presentation of the encoding. With one exception (see page 139), the presence of polymorphism does not add much complication, but the notations become much heavier. While the input and target language provide polymorphic constants, we leave the instantiation implicit in both languages.

Polymorphism in the input language requires polymorphism in the target language, if no special encoding for polymorphism is applied. A polymorphic first-order logic as target language could be considered to be problematic. After all, most of the previously cited automated provers, with the exception of Alt-Ergo, accept only a simply-typed, or even an untyped logic. However, the Why tool, which itself accepts formulas in polymorphic first-order logic, contains several different encodings for the different provers,

depending on whether they are simply typed or untyped (Couchot and Lescuyer, 2007). We simply build on this capacity. Meng and Paulson (2008) and Hurd (2003) directly translate from a monomorphic higher-order logic to untyped resolution-based provers.

5.3.2. An Overview of the Encoding

The input language we define in this section is slightly simpler than L_0 : in particular, we remove tuple types and type polymorphism from the discussion. To differentiate this language from L_0 , we call it “higher-order logic” (HOL).

There are two main differences between HOL and first-order logic (FOL, we will define more precisely these two languages soon). The first one is obviously the “higher-order” part of HOL. One can abstract and quantify over functions and predicates, one can build anonymous functions, one can partially apply functions. As an example, in HOL one can write

$$\begin{aligned} &\text{let } \mathit{apply} \ (f : \text{int} \rightarrow \text{int}) \ (x : \text{int}) = f \ x \\ &\forall f : \text{int} \rightarrow \text{int}. \mathit{apply} \ f = (\lambda(x : \text{int} \rightarrow \text{int}). x) \ f \end{aligned}$$

to express that *apply* is the identity function for one-argument integer functions.¹ One cannot express this kind of properties in FOL.

The second difficulty is the usage of the type **prop**, the type of propositions. In HOL, **prop** is just a type like any other. In most definitions of FOL, there is no such type. However, in FOL one syntactically distinguishes between terms, which have a certain type, and formulas, which do not have a type. In a way, formulas are like terms of type **prop**, but formulas may also contain logical connectives and quantifiers, while terms cannot. Similarly, one has to distinguish between function symbols, which have a list of argument types and a return type, and predicate symbols, which have only a list of argument types, because their return type **prop** is understood. A natural translation from HOL to FOL maps functions whose return type is **prop** to predicates. As there are syntactic restrictions on the occurrences of functions and predicates, an encoding has to take them into account.

First, let us make clear that we do not claim our encoding to be original. The general concept of using a binary function symbol *app* to encode higher-order application goes at least back to Reynolds (1998). The other main ideas come from the work of Meng and Paulson (2008) and Pottier and Gauthier (2006). Our contribution here is the effort to obtain very natural first-order formulas and the observation that the encoding can be ad-hoc justified simply by evaluation of the function symbols that have been introduced by the encoding.

The coding process can be divided into three parts. The first one prepares the HOL term for the actual encoding by eliminating quantifiers and anonymous λ -abstractions, using *λ -lifting*. The second part consists of the actual encoding, translating from HOL to FOL. The third part applies a number of simple optimizations to the FOL formula, to obtain a reasonably natural formula.

¹Actually, this property is only true if one assumes the so-called axiom of *functional extensionality*, which states that two functions are equal if and only if they yield the same result when applied to the same arguments.

5. Implementation and Case Studies

A central concept of our encoding, as of the previous encodings by others, is the usage of a predefined first-order function app , which simulates higher-order application using first-order application. The function app takes a function object f as first argument and the function's argument x as second argument, and returns an object which corresponds to the higher-order application $f x$. As an example, the HOL term $f x y$ could be encoded as $app(app(f, x), y)$. However, if f has a natural first-order signature, then this term is not well-typed; one cannot use function symbols like this in FOL. This fact forces f to have some type representing a function object. Therefore, we add another twist, inspired by Pottier/Gauthier (Pottier and Gauthier, 2006). The symbol f gets a natural first-order type; for example the addition function $+$ has type $int \rightarrow int \rightarrow int$ in HOL, in FOL we give it its natural type $int, int \rightarrow int$. Basically, we replace the top-level arrows by commas, except the last arrow. Additionally, we declare a symbol \hat{f} which has the function object type we mentioned. And whenever the function f is applied to all its argument in the HOL formula, we can use directly f in FOL. Only when it is partially applied, or when f itself is a function argument, we use \hat{f} . Now, to establish the connection between the two symbols, we have to state as an axiom that \hat{f} , when applied to all its arguments, gives the same result as f .

Concerning the second difficulty of **prop**, we use a second type **tprop**, which represents **prop** at places where **prop** cannot appear in FOL. We also have to add a predicate symbol $evalp$ which takes one argument of type **tprop** for the conversion in one direction. For the conversion in the other direction, the aforementioned encoding of higher-order application is actually sufficient.

As an example, consider the following theory in HOL:

$$\begin{aligned} &\text{let } idprop (x : \mathbf{prop}) = x \\ &\text{logic } p : \mathbf{int} \rightarrow \mathbf{prop} \\ &\forall x : \mathbf{int}. p\ x \Rightarrow idprop\ p\ x \end{aligned}$$

It would be translated to the following theory in FOL:

$$\begin{aligned} &\text{predicate } idprop (x : \mathbf{tprop}) = evalp(x) \\ &\dots \\ &\text{predicate } p : \mathbf{int} \\ &\text{logic } \hat{p} : < \text{function object type} > \\ &\forall x : \mathbf{int}. p(x) \Rightarrow idprop (app(\hat{p}, x)) \end{aligned}$$

The idea here is that even though \hat{p} is applied to all its arguments in the last line, we cannot reduce it to $p(x)$, because p is a predicate symbol, and cannot appear in term position. It is understood that $app(\hat{p}, x)$ has type **tprop**.

We now develop more formally the encoding.

5.3.3. The Source Language

Our source language is a standard higher-order logic, defined in Fig. 5.5.

A type τ is either **prop**, or a function type, or an instantiated type symbol. The propositional constants as well as the logical connectives are represented as predefined constants. An HOL term t is either such a constant, a variable, an application of two

5.3. Translation from Higher-Order Logic to First-Order Logic

$$\begin{aligned}
\tau &::= \text{prop} \mid \tau \rightarrow \tau \mid \iota [\bar{\tau}] \\
Q &:: \forall \mid \exists \\
t &::= c \mid x \mid f \mid t \ t \mid \lambda(x : \tau).t \mid Q(x : \tau).t \\
c &::= \wedge \mid \Rightarrow \mid \text{true} \mid \text{false} \mid \dots \\
\text{decl} &::= \text{let } f \overline{(x : \tau)} : \tau = t \mid \text{axiom } t \\
\Delta &::= \emptyset \mid \Delta, x : \tau \mid \Delta, f : \tau
\end{aligned}$$

Figure 5.5: Simplified HOL.

$$\begin{array}{c}
\text{CONST} \frac{\text{Typeof}(c) = \tau}{\Delta \vdash c : \tau} \quad \text{LOCALVAR} \frac{\Delta(x) = \tau}{\Delta \vdash x : \tau} \quad \text{GLOBALVAR} \frac{\Delta(f) = \tau}{\Delta \vdash f : \tau} \\
\\
\text{APP} \frac{\Delta \vdash t_1 : \tau' \rightarrow \tau \quad \Delta \vdash t_2 : \tau'}{\Delta \vdash t_1 \ t_2 : \tau} \quad \text{ABS} \frac{\Delta, x : \tau' \vdash t : \tau}{\Delta \vdash \lambda(x : \tau').t : \tau' \rightarrow \tau} \\
\\
\text{QUANT} \frac{\Delta, x : \tau' \vdash t : \text{prop}}{\Delta \vdash Q(x : \tau').t : \text{prop}} \quad \text{LET} \frac{\Delta, \overline{x : \tau'} \vdash t : \tau'}{\Delta \vdash \text{let } f \overline{(x : \tau)} : \tau' = t} \quad \text{AXIOM} \frac{\Delta \vdash t : \text{prop}}{\Delta \vdash \text{axiom } t} \\
\\
\text{DECL-LET} \frac{\Delta \vdash \text{let } y \overline{(x_i : \tau_i)} : \tau' = t \quad \Delta, y : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau' \vdash th}{\Delta \vdash \text{let } y \overline{(x_i : \tau_i)} : \tau' = t, th} \\
\\
\text{DECL-AXIOM} \frac{\Delta \vdash \text{axiom } t \quad \Delta \vdash th}{\Delta \vdash \text{axiom } t, th}
\end{array}$$

Figure 5.6: Simplified HOL: the typing rules.

HOL terms, an abstraction or a quantification. We syntactically distinguish between global and local variables; a local variable x is introduced by λ -abstractions, quantifiers or function arguments, while a global variable f can only be introduced using a top-level definition. However, both kinds of variable have the same logical status. In addition to these constructs, that are all already present in L_0 , we now explicitly add top-level declarations. Such a top-level declaration decl is either a function definition or an axiom. An HOL theory is simply a list of declarations. Finally, a typing context Δ is just a mapping from global and local variable names to types. As we have mentioned already, HOL does not provide polymorphism, *i.e.*, type variables or a generalization or instantiation mechanism.

We give the typing rules for simplified HOL in Fig. 5.6. They are pretty standard, with the exception that they reflect the syntactic distinction between global and local variables. For HOL itself, this distinction is unnecessary; global and local variables behave the same and have the same logical status. However, in FOL this is no longer

true, and our encoding behaves differently for global and local variables. To be able to formalize this behavior easily, the distinction must already appear in HOL.

5.3.4. Elimination of Quantifiers

The first simplification we apply is to eliminate the quantifiers and to replace them by predefined constants `forall` and `exists`. This is so natural in the context of higher-order logics that many developments define HOL with these constants and without explicit quantifiers. The constant `forall` is of type $(\alpha \rightarrow \text{prop}) \rightarrow \text{prop}$, and the term `forall` $(\lambda(x : \tau).p(x))$ is intended to represent the statement $\forall x : \tau.p(x)$. Our simplification simply consists in replacing quantifiers by an application of the these constant:

$$\begin{aligned} \forall x : \tau.t &\rightarrow \text{forall } (\lambda(x : \tau).t) \\ \exists x : \tau.t &\rightarrow \text{exists } (\lambda(x : \tau).t) \end{aligned}$$

To keep the meaning of the formula, we have to define both constants, for example with an axiom:

$$\begin{aligned} \forall f : \tau \rightarrow \text{prop}.\text{forall } f &\Leftrightarrow \forall x : \tau.f x \\ \forall f : \tau \rightarrow \text{prop}.\text{exists } f &\Leftrightarrow \exists x : \tau.f x \end{aligned}$$

5.3.5. Elimination of λ -Abstractions

In HOL, one can define anonymous functions using λ -abstractions. In FOL, the only way to build functions is to define them at the top-level. A possible step towards FOL is thus to replace λ -abstractions by top-level definitions. The process that achieves this is called *λ -lifting* (Johnsson, 1985).

Our λ -lifting, written $\llbracket t \rrbracket^d$ (see Fig. 5.7), expects a term t and a current list of declarations d . It returns an abstraction-free term t' and a possibly enriched context d' . In the case of variables and constants, there is nothing to do; the term and the context are left unchanged. When we encounter an application $t_1 t_2$, we would like to leave it as is, because we only want to eliminate abstractions. But of course, both terms may contain abstractions themselves so we have to apply the transformation to the two subterms obtaining t_a and t_b , threading the context through these recursive calls. At the end, we return the application $t_a t_b$ and the richest environment d_2 .

When encountering a sequence of abstractions, we first compute the free variables of the whole term, *i.e.*, not counting the abstracted variables. We then call $\llbracket \cdot \rrbracket$ recursively on the body of the abstraction, obtaining an abstraction-free body t_1 . We now build a new top-level definition, using a fresh name f , whose body is t_1 and whose arguments are the free variables we computed before and the variables of the abstraction. This new top-level definition is appended to the context. The returned term is the application of our new top-level function f to the free variables.

It is easy to see that this transformation is *correct*, *i.e.*, that the obtained term is logically equivalent to the original term. Simply observe that if $\llbracket t \rrbracket^d = t', d'$, then $t = t'$ modulo δ -equivalence, that is definition unfolding. This is trivial for the case of variables and constants, and very easy for the case of applications, by induction. In the case of a

$$\begin{array}{l}
 \boxed{\llbracket t \rrbracket^d = t', d'} \\
 \boxed{\llbracket \text{decl} \rrbracket^d = \text{decl}'} \\
 \\
 \begin{array}{lcl}
 \llbracket x \rrbracket^d & = & x, d \\
 \llbracket c \rrbracket^d & = & c, d \\
 \llbracket t_1 t_2 \rrbracket^d & = & \text{let } t_a, d_1 = \llbracket t_1 \rrbracket^d \text{ in} \\
 & & \text{let } t_b, d_2 = \llbracket t_2 \rrbracket^d \text{ in} \\
 & & t_a t_b, d_2 \\
 \llbracket \lambda \overline{(x : \tau)}. t \rrbracket^d & = & \text{let } \overline{(y : \tau)} = fv(\lambda \overline{(x : \tau_1)}. t) \text{ in} \\
 & & \text{let } t_1, d_1 = \llbracket t \rrbracket^d \text{ in} \\
 & & \text{let } d_2 = d_1; \text{let } f \overline{(y : \tau)} \overline{(x : \tau_1)} = t_1 \text{ in} \\
 & & f \overline{y}, d_2 \\
 \llbracket \text{let } y \overline{(x : \tau)} : \tau = t \rrbracket^d & = & \text{let } t_1, d_1 = \llbracket t \rrbracket^d \text{ in} \\
 & & \text{let } d_2 = d_1; \text{let } y \overline{(x : \tau)} : \tau = t_1 \text{ in} \\
 & & d_2 \\
 \llbracket \text{axiom } t \rrbracket^d & = & \text{let } t_1, d_1 = \llbracket t \rrbracket^d \text{ in} \\
 & & \text{let } d_2 = d_1; \text{axiom } t_1 \\
 & & d_2
 \end{array}
 \end{array}$$

 Figure 5.7: The λ -Lifting transformation.

lambda-abstraction, simply unfold the newly obtained definition and use the fact that t and t_1 are already equal modulo δ -equivalence.

The transformation also succeeds in removing every λ -expression, as can be easily verified by induction and by observing that none of the result terms contains a λ -expression.

λ -lifting and polymorphism. The λ -lifting algorithm is the only part of the encoding where the presence of polymorphism leads to a more complex algorithm. Let us consider as an example the following function skeleton:

$$\begin{array}{l}
 \text{let } f [\alpha\beta] (x : \alpha) = \\
 \quad \dots (\lambda z : \beta. \dots x \dots) \dots
 \end{array}$$

When lifting the anonymous function in f , to define a new top-level function g , we have to add the variable x to the arguments of g . But there are also the type variables α and β . In fact, g must be polymorphic, and in f , we must instantiate g with the right type variables:

$$\begin{array}{l}
 \text{let } g [\alpha\beta] (x : \alpha) (z : \beta) = \dots x \dots \\
 \text{let } f [\alpha\beta] (x : \alpha) = \\
 \quad \dots g [\alpha\beta] x \dots
 \end{array}$$

In a polymorphic system such as L_0 , we also have a quantifier over type variables of the

5. Implementation and Case Studies

$$\begin{aligned}
\kappa &::= \iota \bar{\kappa} \\
\sigma &::= \text{predicate } \bar{\kappa} \mid \text{function } \bar{\kappa} \rightarrow \kappa \\
t &::= x(\bar{t}) \\
\circ &::= \wedge \mid \Rightarrow \mid \dots \\
p &::= x(\bar{t}) \mid \forall x : \kappa. p \mid p \circ p \mid \dots \\
\text{decl} &::= \text{function } y \overline{(x : \kappa)} : \kappa = t \mid \text{predicate } y \overline{(x : \kappa)} = p \mid \text{axiom } p \mid f : \sigma \\
\Gamma &::= \emptyset \mid \Gamma, x : \sigma
\end{aligned}$$

Figure 5.8: FOL: the syntax.

form

$$\forall \alpha. f$$

to bind local type variables. These binders can also interfere with λ -lifting, and sometimes a suitable quantification over local type variables must be added to the body of the lifted function. In some cases, this also leads to the removal of a quantification over types at the place where the lifted function comes from.

5.3.6. The Target Language: FOL

Before we turn to the actual encoding of HOL, we have to introduce the target language. Its syntax is summarized in Fig. 5.8. First-order types are written κ ; they are simply type constructors of a given arity. Note that **prop** is *not* a first-order type. A first-order signature σ is either a predicate signature, consisting of a list of argument types, or a function signature which additionally contains a return type. A term t is an n -ary application of a function symbol x to a list of terms. A formula p is either an n -ary application of a predicate symbol x to a list of terms, or a quantification, or a formula with a logical connective. There are now four different top-level definitions: let-declarations in HOL are split in function declarations with a return type and a term body and predicate definitions without return type and with a formula body. One can also declare axioms and function symbols. Logically, an axiom corresponds to a new premise, and a declaration of a function symbol to a universal quantification.

Fig. 5.9 summarizes the typing rules for FOL. A typing environment Γ is a mapping from variables to either predicate or function signatures. In terms, one can only use function symbols (variables whose signature in Γ is a function signature), and in formulas one can only use predicate symbols. Variables of first-order type κ , introduced by quantifiers and function or predicate arguments, are encoded as nullary function symbols using the syntax `function` ($\rightarrow \kappa$).

The translation to FOL requires a few predefined type constructors and function

5.3. Translation from Higher-Order Logic to First-Order Logic

$$\begin{array}{c}
\text{T-APP} \frac{\Gamma(x) = \text{function } \bar{\kappa} \rightarrow \kappa' \quad \Gamma \vdash_t t_i : \kappa_i}{\Gamma \vdash_t x(\bar{t}) : \kappa'} \\
\\
\text{P-APP} \frac{\Gamma(x) = \text{predicate } \bar{\kappa} \quad \Gamma \vdash_t t_i : \kappa_i}{\Gamma \vdash_p x(\bar{t})} \qquad \text{FQUANT} \frac{\Gamma, x : \text{function } (\rightarrow \kappa) \vdash_p p}{\Gamma \vdash_p \forall x : \kappa.p} \\
\\
\text{FLOP} \frac{\Gamma \vdash_p p_1 \quad \Gamma \vdash_p p_2}{\Gamma \vdash_p p_1 \circ p_2} \qquad \text{FEMPTY} \frac{}{\Gamma \vdash_d \emptyset} \\
\\
\text{FSEQFUN} \frac{\Gamma, x : \text{function } (\rightarrow \kappa) \vdash_t t : \kappa' \quad \Gamma, y : \text{function } \bar{\kappa} \rightarrow \kappa' \vdash_d \overline{\text{decl}}}{\Gamma \vdash_d \text{function } y (x : \kappa) : \kappa' = t; \overline{\text{decl}}} \\
\\
\text{FSEQPRED} \frac{\Gamma, x : \text{function } (\rightarrow \kappa) \vdash_p p \quad \Gamma, y : \text{predicate } \bar{\kappa} \vdash_d \overline{\text{decl}}}{\Gamma \vdash_d \text{predicate } y (x : \kappa) = p; \overline{\text{decl}}} \\
\\
\text{FSEQAXIOM} \frac{\Gamma \vdash_p p \quad \Gamma \vdash_d \overline{\text{decl}}}{\Gamma \vdash_d \text{axiom } p; \overline{\text{decl}}} \qquad \text{FSEQDECL} \frac{\Gamma, f : \sigma \vdash_d \overline{\text{decl}}}{\Gamma \vdash_d f : \sigma; \overline{\text{decl}}}
\end{array}$$

Figure 5.9: FOL : the typing rules.

symbols. Here they are:

type	term	α
type	tprop	
type	arrow	(α, β)
app	:	function term arrow (α, β) , term $\alpha \rightarrow$ term β
eval	:	function term $\alpha \rightarrow \alpha$
reflect	:	function $\alpha \rightarrow$ term α
evalp	:	predicate term tprop

The intuition behind these symbols is the following. The type constructor **term** represents *reified* higher-order terms, *i.e.*, objects of type **term** κ are intended to represent higher-order objects of a type corresponding to κ . Every form of types in HOL that does not exist in FOL is represented by a FOL type constructor: **prop** corresponds to **tprop**, and the type arrow \rightarrow corresponds to **arrow**. So this means that the objects of type **term** **tprop** are reified versions of higher-order predicates, and objects of type **term** **arrow** (κ_1, κ_2) are reified functions. To use such functions, we dispose of a symbol **app**, which takes a reified function and a reified object of the expected type and returns a reified object of the return type of the function. Finally, we have two functions **eval** and **reflect**, converting to and from reified objects. We assume the equation

$$\text{eval}(\text{reflect}(t)) = t$$

5. Implementation and Case Studies

to be true for all instances. There is a special case in the case of reified predicates; in this case we can use the predicate symbol `evalp` to obtain a first-order predicate.

5.3.7. The Encoding

The language we have obtained after λ -lifting does not contain quantifiers nor λ -abstractions, but it still contains higher-order types and partial application. Also, the HOL-type `prop` may be used at places where it cannot be used in FOL, for example as argument type. We now define the encoding which takes care of these features.

The first definition concerns types. As indicated before, we use first-order type constructors to handle the problematic cases:

$$\begin{aligned} \llbracket \text{prop} \rrbracket &= \text{tprop} \\ \llbracket \iota \bar{\tau} \rrbracket &= \iota \llbracket \bar{\tau} \rrbracket \\ \llbracket \tau \rightarrow \tau' \rrbracket &= \text{arrow} (\llbracket \tau \rrbracket, \llbracket \tau' \rrbracket) \end{aligned}$$

Here, `prop` is dealt with by `tprop`, and the arrow \rightarrow by `arrow`. Other type constructors are taken over unchanged.

Given the syntactical distinction between local and global variables, the encoding of terms is very simple:

$$\begin{aligned} \llbracket f \rrbracket &= \hat{f} \\ \llbracket x \rrbracket &= \text{reflect}(x) \\ \llbracket t_1 t_2 \rrbracket &= \text{app}(\llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket) \end{aligned}$$

An application is encoded using the first-order function `app`. Local variables are *reflected*. For global variables f , we directly use the reflection \hat{f} to represent them. We will see in a minute how the name \hat{f} is introduced. An important property of this encoding is that if t is of type τ in HOL, then $\llbracket t \rrbracket$ is of type `term` $\llbracket \tau \rrbracket$. This property will be useful to prove the well-typedness of the overall encoding.

Finally let's look at how contexts are encoded:

$$\begin{aligned} \llbracket \text{let } f \overline{(x : \tau)} : \tau' = t \rrbracket &= \text{function } f \overline{x : \llbracket \tau \rrbracket} : \llbracket \tau' \rrbracket = \text{eval}(\llbracket t \rrbracket) \\ &\quad \text{function } \hat{f} : \text{term } \llbracket \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau' \rrbracket \\ \llbracket \text{axiom } A : t \rrbracket &= \text{axiom } A : \text{evalp}(\llbracket t \rrbracket) \end{aligned}$$

Axioms are simply translated to axioms; however as $\llbracket t \rrbracket$ is of type `term` `tprop`, we must add an application of `evalp`. Top-level definitions are similarly translated to top-level definitions, where an application of `eval` is added. However, this is not sufficient, as in FOL the symbol f cannot be partially applied anymore, nor be passed to other functions. The `reflect` function is of no use, as it can only be applied to local variables. We therefore directly introduce the reified version of f , called \hat{f} . Its type is the reification of the higher-order type of f .

5.3.8. Optimizations of the Encoding

The encoding we have shown up to now is very simple and easy to prove correct. However, it produces unnecessarily large terms, and it contains unnecessarily contrived formulations of first-order concepts. For example, up to now, we have made no use of the

5.3. Translation from Higher-Order Logic to First-Order Logic

formula constructs of FOL; even if the HOL formula contained logical connectors, well-known predicates like equality or quantifiers, these have been encoded as any other HOL function or predicate. This is a problem, as automated provers deal in an optimized way with these constructs. Any form of encoding for these constructs severely slows down their performance. Our job in this section is to undo as much as possible of the encoding so that we gain the original structure of the formula. In the best case, we would like to obtain the original HOL formula, whenever it was a first-order formula.

Introducing predicates. A first step consists in transforming function declarations of return type `tprop` into predicates:

$$\text{function } f : (\tau_1, \dots, \tau_n) \rightarrow \text{tprop}$$

becomes

$$\text{predicate } \hat{f} : (\tau_1, \dots, \tau_n).$$

We can do this because the encoding process has eliminated all occurrences of f in the remaining list of declarations, only occurrences of \hat{f} can appear. So changing the signature of f cannot change the well-typedness of the rest of the theory. We do not need to change the type in the declaration of \hat{f} .

Recovering the logical structure of the formula. If we want to obtain a relatively natural looking first-order formula, we need to re-obtain the logical connectives and the quantifiers. The following rewriting rules take care of this:

$$\begin{aligned} \text{evalp}(\text{app}(\text{app}(\hat{\circ}, f_1), f_2)) &\rightarrow \text{evalp}(f_1) \circ \text{evalp}(f_2) \\ \text{evalp}(\text{app}(\text{forall}, f)) &\rightarrow \forall x. \text{evalp}(\text{app}(f, \text{reflect}(x))) \\ \text{evalp}(\text{app}(\text{exists}, f)) &\rightarrow \exists x. \text{evalp}(\text{app}(f, \text{reflect}(x))) \end{aligned}$$

We simply rewrite full applications of $\hat{\wedge}$ to the connective \wedge and so on. It is however important to note that these rewriting steps can only be applied under an application of `evalp`. Otherwise we would be trying to build formulas where a formula cannot syntactically occur.

Reestablishing full applications. An occurrence of a reified global function symbol \hat{f} is *fully applied* when it appears in a term such as

$$\text{app}(\dots (\text{app}(\hat{f}, x_1), \dots), x_n)$$

where n is also the arity of f . In this case, it is possible to replace the whole term by the first-order application of f to the arguments x_1 to x_n . The same is obviously possible for predicate symbols. However, for the term to be well-typed, we need to inject calls to `reflect`, so that the rewriting rules look like this:

$$\begin{aligned} \text{eval}(\text{app}(\dots (\text{app}(\hat{f}, x_1), \dots), x_n)) &\rightarrow f(\text{reflect}(x_1), \dots, \text{reflect}(x_n)) & \text{arity}(f) = n \\ \text{evalp}(\text{app}(\dots (\text{app}(\hat{p}, x_1), \dots), x_n)) &\rightarrow p(\text{reflect}(x_1), \dots, \text{reflect}(x_n)) & \text{arity}(f) = n \end{aligned}$$

5. Implementation and Case Studies

Rewriting. In this and the previous paragraph, we have essentially defined a rewriting system on first-order terms. A rewriting system is a set of rules, just as the ones we have given here, that can be applied in an arbitrary order at arbitrary places in a term. To be useful, a term rewriting system must be *terminating* and *confluent*. A rewriting system is terminating if for any given term, only a finite number of rules can be applied, one after the other, to obtain a term which does not permit applications of the rules anymore. Such a term is called a *normal form*. A rewriting system is confluent if, when a term t permits applications of several rules to obtain, say, t_1 and t_2 , then t_1 and t_2 permit a sequence of applications of rewriting rules to obtain the same normal form, say t' .

The property of termination guarantees that one obtains a normal form after a finite number of steps. The property of confluence guarantees that the order of application of the rewriting rules is not important. Together, they imply that we can apply the rewriting rules in any order in the term, and will find the normal form eventually.

The CiME tool (Contejean et al., 2007) can prove termination and confluence for rewriting system. We have entered the above rewriting rules in this tool and were able to prove both properties. Due to limitations of the CiME tool, we had to simplify the problem a bit, but we believe that this does not change the termination and confluence proofs. First, we have limited ourselves to a single pair of function f and corresponding constant \hat{f} , of binary arity. Second, and maybe more seriously, we have replaced the quantification in the rewriting rules concerning the constants $\hat{\text{forall}}$ and $\hat{\text{exists}}$ by unary functions `forall` and `exists`, and the variable x by a constant.

Inlining function symbols that have been created by the translation. Each quantification of the form

$$\forall x.p$$

is transformed at the beginning to

$$\text{forall } (\lambda x.p)$$

, and then, by λ -lifting, to

$$\begin{array}{l} \text{let } f \ x = p \\ \text{forall } f \end{array}$$

After the encoding and the simplifications, we obtain the formula

$$\begin{array}{l} \text{predicate } f \ x = \llbracket p \rrbracket \\ \forall x.f \ x \end{array}$$

The result is relatively clear, but the additional predicate symbol is superfluous and may hinder the proof process. We therefore decide to inline predicate symbols that originate from λ -lifted abstractions, when possible. In our example, we obtain (almost) the original formula:

$$\forall x.\llbracket p \rrbracket$$

Adding the necessary axioms. A last item is missing; up to now, we have never linked the reified symbols \hat{f} to the original symbols f . We do that now. We simply have to state that the n -ary application of \hat{f} using `app` corresponds to the n -ary first-order application of f . To do this, for every declaration of a function symbol:

$$\text{function } f : (\tau_1, \dots, \tau_n) \rightarrow \tau$$

we have to add an axiom

$$\forall x_1 : \tau_1 \dots \forall x_n : \tau_n. \text{eval}(\text{app}(\dots(\text{app}(\hat{f}, \text{reflect}(x_1)), \dots), \text{reflect}(x_n))) = f(x_1, \dots, x_n)$$

and for every declaration of a predicate symbol:

$$\text{predicate } p : (\tau_1, \dots, \tau_n)$$

we have to add an axiom:

$$\forall x_1 : \tau_1 \dots \forall x_n : \tau_n. \text{evalp}(\text{app}(\dots(\text{app}(\hat{p}, \text{reflect}(x_1)), \dots), \text{reflect}(x_n))) \Leftrightarrow p(x_1, \dots, x_n)$$

It is important to add these axioms only at the end of the encoding process. Otherwise, the right-hand sides of the axioms would be simplified by the rewriting rules we have given in the previous paragraphs. To be useful, the axioms have to be left untouched.

5.3.9. An Example

Let us show the encoding on the example that has been discussed at the beginning of Section 5.3.2, on page 135:

```
let apply (f : int → int) (x : int) = f x
goal : ∀ f : int → int. apply f = (λ(x : int → int). x) f
```

The second line is the goal to prove. Introducing the quantifier constant, we obtain

```
let apply (f : int → int) (x : int) = f x
goal forall (fun (f : int → int). apply f = (λ(x : int → int). x) f)
```

After λ -lifting, we obtain

```
let apply (f : int → int) (x : int) = f x
let id (x : int) = x
let p (f : int → int) = apply f = id f
goal : forall p
```

We have given the names `id` and `p` to the lifted abstractions. Now the actual encoding is applied, to obtain the following FOL theory:

```
function apply (f : arrow (int, int)) (x : int) = eval(app(f, reflect(x)))
 $\hat{\text{apply}}$  : term arrow (arrow (int, int), arrow (int, int))
function id (x : int) = x
 $\hat{\text{id}}$  : term arrow (int, int)
function p (f : arrow (int, int)) =
```

5. Implementation and Case Studies

```

    eval(app(app(≐, app(apply, reflect(f))), app(id, reflect f))
  p̂ : term arrow (arrow (int, int), tprop)
  goal : evalp(app(forall, p̂))

```

In the body of the identity function *id*, the application of *eval* to the term *reflect(x)* has been simplified to *x*. We assume that the context contains constants $\hat{=}$ and \hat{forall} , that represent the reified versions of the equality predicate and the quantification constant, respectively.

We can now apply the simplifications and obtain a much cleaner formula:

```

function apply (f : arrow (int, int)) (x : int) = eval(app(f, reflect(x)))
apply : term arrow (arrow (int, int), arrow (int, int))
function id (x : int) = x
id : term arrow (int, int)
predicate p (f : arrow (int, int)) = apply(f) = id(f)
p̂ : term arrow (arrow (int, int), tprop)
goal : ∀(f : arrow (int, int)).p f

```

A useful heuristics to obtain a more natural formula is then to inline predicates that originated from quantifications:

```

function apply (f : arrow (int, int)) (x : int) = eval(app(f, reflect(x)))
apply : term arrow (arrow (int, int), arrow (int, int))
function id (x : int) = x
id : term arrow (int, int)
goal : ∀(f : arrow (int, int)).apply(f) = id(f)

```

We finally can add the axioms that link reified symbols to the original ones, to obtain the final result:

```

function apply (f : arrow (int, int)) (x : int) = eval(app(f, reflect(x)))
apply : term arrow (arrow (int, int), arrow (int, int))
axiom apply_equiv :
  ∀ (f : arrow (int, int)) (x : intml).
    eval(app(app(apply, reflect(f)), reflect(x))) = apply(f, x)
function id (x : int) = x
id : term arrow (int, int)
axiom id_equiv : ∀ (id : int). eval(app(id, reflect(x))) = id(x)
goal : ∀(f : arrow (int, int)).apply(f) = id(f)

```

Note that this goal is only provable using functional extensionality, but this was already the case in the HOL theory.

5.3.10. Justifying the Encoding

Proving the different transformations correct is not very difficult. An interesting challenge, however is to generate a machine-checkable proof that the obtained list of FOL declarations is indeed equivalent to the initial formula in HOL. Our translation mechanism is designed to easily obtain machine-checkable proofs. We want to show this using the proof assistant Coq. We do so by embedding the input logic HOL and the output

5.3. Translation from Higher-Order Logic to First-Order Logic

logic FOL in Coq. To avoid introducing yet another syntax, we use the syntax of HOL to describe Coq terms.

We have seen that HOL is a subset of Coq, so we can directly type HOL terms in Coq. For FOL, this is less obvious, but still easily definable. We simply define the following embedding:

$$\begin{aligned}
\llbracket \text{function } \kappa_1 \cdots \kappa_n, \kappa \rrbracket &= \kappa_1 \rightarrow \cdots \rightarrow \kappa_n \rightarrow \kappa \\
\llbracket \text{predicate } \kappa_1 \cdots \kappa_n \rrbracket &= \kappa_1 \rightarrow \cdots \rightarrow \kappa_n \rightarrow \text{prop} \\
\llbracket x (t_1, \dots, t_n) \rrbracket &= x t_1 \cdots t_n \\
\llbracket p_1 \circ p_2 \rrbracket &= \circ p_1 p_2 \\
\llbracket \forall x : \kappa. p \rrbracket &= \forall x : \llbracket \kappa \rrbracket \llbracket p \rrbracket \\
\llbracket \text{function } f \overline{(x : \kappa)} : \tau' = t \rrbracket &= \text{let } f \overline{(x : \kappa)} : \tau' = \llbracket t \rrbracket \\
\llbracket \text{predicate } f \overline{(x : \kappa)} = t \rrbracket &= \text{let } f \overline{(x : \kappa)} : \text{prop} = \llbracket t \rrbracket
\end{aligned}$$

This definition is really straightforward; it simply encodes the n -ary function and predicate types by n type arrows and n -ary first-order application by n unary higher-order applications. Type constants, even polymorphic ones, are directly embeddable in Coq.

There are a number of symbols in FOL for which we need to give a definition in Coq. These are the type constants `term`, `tprop`, `arrow` and the function constants `app`, `eval`, `reflect` and `evalp`, but also the function symbols \hat{f} introduced by the encoding itself. We could postulate their existence, with corresponding Coq types, and in this way obtain well-typed terms in Coq. But to obtain a machine-checkable proof of the correctness of the encoding, we give a *definition* of these functions. The aim is that an *evaluation* of the term obtained by the encoding returns a term that is identical to the initial term in Coq. As, in Coq, a proof of equality can in particular be carried out by evaluation, the correctness proof becomes trivial. When we speak of evaluation, we mean in particular the β and δ rules in the calculus of Coq.

We now give definitions in Coq for the missing constants; we use the syntax of HOL that we have already introduced.

$$\begin{aligned}
\text{type term } \alpha &= \alpha \\
\text{type tprop} &= \text{prop} \\
\text{type arrow } \alpha \beta &= \alpha \rightarrow \beta \\
\text{let app } f x &= f x \\
\text{let eval } x &= x \\
\text{let reflect } x &= x \\
\text{let evalp } x &= x
\end{aligned}$$

Finally, for every function symbol \hat{f} that has been introduced by the encoding, we add the definition

$$\text{let } \hat{f} = f.$$

First of all, let us convince us that the given implementation is indeed correct. For this, we need to check the axioms of the form

$$\forall x_1 : \tau_1 \dots \forall x_n : \tau_n. \text{eval}(\text{app}(\dots(\text{app}(\hat{f}, x_1), \dots), x_n)) = f(x_1, \dots, x_n)$$

5. Implementation and Case Studies

for functions, and the equivalent axioms using `evalp` for predicates. Actually, we need to check their equivalent in Coq:

$$\forall x_1 : \tau_1 \dots \forall x_n : \tau_n. \text{eval} (\text{app} (\dots (\text{app } \hat{f} x_1) \dots) x_n) = f x_1 \dots x_n$$

where we have simply changed the n -ary applications into unary ones. Looking at the definitions of `app` and `eval`, and the definition of \hat{f} , the axiom is entirely trivial. Not only it is trivial for a human, but the equality can be proved simply by unfolding the definitions on the left hand side of `eval`, `app` and \hat{f} . In Coq, this is called proof by reflection and is built-in in the core calculus. It also becomes clear that the “optimizations” in Section 5.3.8 are simply unfoldings of `evalp`, `eval`, `app` and \hat{f} .

The encoding in Section 5.3.7 concerning types and terms is basically the inverse of the definitions we have given to the FOL constants; the correctness can thus be proved once again by unfolding. The same is true for function definitions. If the initial term in HOL was

$$\text{let } f \overline{(x : \tau)} : \tau' = t$$

then its encoding in FOL, again rendered in HOL, gives

$$\text{let } f \overline{(x : \llbracket t \rrbracket)} : \llbracket \tau' \rrbracket = \text{eval } \llbracket t \rrbracket$$

We have already established the equality by evaluation of the encodings of types and terms, and `eval` evaluates to the identity function.

The only missing steps now are the elimination of anonymous functions using λ -lifting and the elimination of quantifiers. Both are simple to justify, and in both cases the justification is again definition unfolding. λ -lifting simply introduces additional definitions in the context; it can be justified by unfolding them. In our translation, quantifiers are replaced by application of the constant `forall`. If we define this constant in this way:

$$\text{let forall } f = \forall x. f x$$

this quantifier elimination is again justified by unfolding of a definition. Defining a constant in this way is entirely legitimate in Coq.

In summary, we have seen that with the given definitions for the symbols that are introduced by the encoding, we can justify the equation

$$f = \llbracket f \rrbracket$$

in Coq, simply by evaluating the encoded term, and this for any given input theory f . Note, however, that this is not a machine-checkable proof of the encoding itself. It only represents a machine-checkable proof for each *instance* or *execution* of the encoding on a particular input.

The deep reason for this limitation is that here we only consider a *shallow embedding* of HOL and FOL formulas, *i.e.*, we directly use Coq syntax to represent terms of these languages. A much heavier, but more powerful possibility would consist in *representing* the formulas of HOL and FOL by objects of a Coq data type, for example an algebraic data type, or inductive type in Coq. One could then *implement* the encoding we just have presented in Coq and *prove* that it always preserves the validity of the input formula. We do not follow this approach. Instead we propose to use a shallow embedding

to generate a proof for each execution of the encoding that shows that input and result are equivalent.

This result could be used to obtain more confidence in the answers of automated theorem provers, that in most cases are limited to “yes” or “no”. If there were a first-order prover for first-order logic that answered with a machine-checkable proof, then this approach could lift the proof of the first-order theorem to the original problem in higher-order logic. Currently, very few first-order provers output proof traces. A notable exception is Zenon (Bonichon et al., 2007), that is able to output Coq proofs. Another work in this direction is the work by Paulson and Susanto (2007). They call powerful resolution-based provers to obtain a list of used lemmas in the proof. This list does not represent a machine-checkable proof because of the lack of details. However, they use this list to call a home-made, slower prover called Metis, initially developed by Hurd (2003). Metis has been programmed with proof traces in mind; it lacks many optimizations of state-of-the-art theorem provers, but it is capable of generating a checkable proof trace in Isabelle/HOL syntax. Because Metis knows the list of used lemmas, it can ignore all other lemmas and does in effect solve a much smaller problem. Therefore, despite the fact that Metis is slower than other automated provers, it can construct a proof in reasonable time.

A long term objective of Pangoline is to use Zenon or an approach similar to the one followed by Paulson and Susanto (2007) to obtain proof traces for higher-order theorems.

Formulas that are already first-order formulas. It is easy to see that our encoding leaves (almost) unchanged functions and formulas that are already in first-order form. To realize this, we first need to define what it means to be first-order for a term in the higher-order logic HOL. But this is easy; we simply declare a term to be first-order if it is the image of a first-order term under the embedding defined in the previous section. It is easy to see that in all cases, the encoding will be undone by the simplification phase: all function and predicate symbols are fully applied in FOL, quantified formulas are re-obtained by application of the rewrite rules and inlining the corresponding predicate symbols. Still, we cannot say that the embedding (\cdot) to HOL, together with the encoding $\llbracket \cdot \rrbracket$ to FOL, is the identity function on FOL theories, because of the hatted function symbols and the corresponding axioms. They can still lead to more work for an automated prover.

The presented encoding is implemented in the tool Pangoline. The implementation is very close to the described process, and as presented, it consists of a sequence of relatively small encoding and rewriting steps.

5.4. Case Studies

In this section, we show a few examples, with increasing difficulty, of proofs that can be carried out in our calculus. The examples are first given in ML-like syntax, *i.e.*, without annotations of any kind, such as pre- and postconditions, regions or effects, to ease the understanding of the code. Then, a fully annotated version is presented and explained.

5. Implementation and Case Studies

All programs have actually been proved correct using our tool *Who* and several automatic provers, as well as the interactive proof assistant *Coq*, when necessary.

5.4.1. Introductory Examples

We first start with a few simple examples that do not need any additional type and function constants.

The function *apply*. One of the simplest higher-order functions is the identity function on functions:

```
let apply f x = f x
```

It simply takes the function *f* and its argument *x* and applies *f* to *x*.

In *Who*, we have to account for several difficulties: *apply* has to be effect polymorphic to allow its application to functions of any effect, and it has to use the specification of *f* to be as generic as possible. Here is the code in *Who*:

```
let apply [ $\alpha\beta\varepsilon$ ] (f :  $\alpha \rightarrow^\varepsilon \beta$ ) (x :  $\alpha$ ) =  
  {pre f x cur}  
  f x  
  {r : post f x old cur r}
```

As required, *apply* not only generalizes over the argument and return types of *f*, but also over the effect ε . Its overall type is:

$$\textit{apply} : \forall\alpha\beta\varepsilon. (\alpha \rightarrow^\varepsilon \beta) \rightarrow^\emptyset \alpha \rightarrow^\varepsilon \beta$$

For such a simple function, the pre- and postcondition are trivial: we simply require the precondition of *f* to hold in the initial state, and we guarantee the postcondition of *f* to be true for the two relevant states and the returned value. This very simple example is proved by automated provers.

The for loop. The for-loop is a very useful tool, especially in connection with arrays, present in almost any practical programming language. However, the core syntax of *W* does not contain such a construct. But it is easy to define something very similar to a for-loop: a higher-order function taking the limits of the loop and a function to be executed between these limits. Here is the implementation (without annotations) of this idea:

```
let forfun s e f =  
  let rec aux i =  
    if i ≤ e then begin f i; aux (i + 1) end  
    else ()  
  in  
  aux s
```

The function *forfun* takes two integers *s* and *e* (for “start” and “end”) and executes the function call *f* *i* for each integer *i* between *s* and *e*, in increasing order. We use a local

```

let forfun [ε] (inv : < ε > → int → prop)
  (s : int) (e : int) (f : int →ε unit) =
  { inv cur s ∧ ∀ i. s ≤ i ≤ e → {inv cur i} f i {inv cur (i + 1)} }
  let rec aux (i : int) =
    {s ≤ i ∧ inv cur i}
    if i ≤ e then begin f i; aux (i + 1) end
    { (i ≤ e + 1 → inv cur (e + 1)) ∧ (i > e → inv cur i) }
  in
  aux s
  { inv cur (max s (e + 1)) }

```

Figure 5.10: The implementation of *forfun* in Who.

recursive function *aux* to obtain the looping behavior. Note that the function *forfun* can be used to simplify the syntactic sugar introduced in Section 2.1.

Contrary to the example of *apply*, it is not sufficient to require the precondition of *f* to be true initially. Indeed, *f* may be called many times, in different states, with different integer arguments. Actually, it is quite clear that, as we are implementing a *loop*, *f* must maintain some kind of invariant. The exact invariant to be maintained depends of course on the body of the *for*-loop, *i.e.*, the function *f*. We solve this problem here by taking an additional, *logical* argument *inv*, a predicate that precisely states the invariant maintained by *f*. It has to be given when calling *forfun*, just as an invariant has to be given for *for*-loops in Hoare logic. The invariant is of type $\langle \varepsilon \rangle \rightarrow \text{int} \rightarrow \text{prop}$; it depends on the current state and current count of the loop. The annotated version of *forfun* is presented in Fig. 5.10. Once we have that additional argument representing the invariant, everything becomes quite simple. The precondition simply states that the invariant must be initially true and that *f* must maintain the invariant: if it is true for *i* before calling *f*, it is true for *i* + 1 after the call. In the postcondition, we guarantee that the invariant is true for either *s* or *e* + 1, whichever is larger.

Let us now explain the body of *forfun* in more detail, in particular the local function *aux*. This function achieves the looping behavior: as long as $i \leq e$, the recursive call will be executed. However, *aux* is designed to work also when this condition is initially false; in particular, it is not mentioned in the precondition of *aux*. The code does not have to do anything in particular to make it work. However, the postcondition of *aux* needs to consider both cases.

Finally, let us explain how one can obtain a correctly specified program starting from a *for*-loop:

```
for i = s to e do { inv } body done
```

In Section 2.1.1, we explained how to desugar a *for*-loop without annotations into a *W* program without annotations. Here, we follow a similar approach, with two modifications. First, we use the *forfun* function, and second, we need to insert specifications into the desugared code. To simplify, let us assume that *s* and *e* are variables.

```
let f i = { p } body { q } in
```

5. Implementation and Case Studies

forfun inv s e f

We now only need to define p and q . The *body* is only executed when the counter i is indeed between the bounds s and e ; additionally, the invariant must be true:

$$p = s \leq i \leq e \wedge \text{inv cur } i$$

The postcondition establishes the invariant for the next value of i .

$$q = \text{inv cur } (i + 1)$$

All proof obligations are proved by automated provers.

Landin's knot. The technique called Landin's knot or *backpatching* is a technique to obtain recursion without using a recursion mechanism built-in in the language. We have already presented it in Section 2.1. The idea is, when defining a function f which needs to be recursive, to replace each recursive call by a call to a function stored in some reference, say x . Initially, x contains some dummy value, for example the identity function. Once the function f has been defined, one replaces this dummy value in x by f , and the recursion has been achieved. In Chapter 2, we gave the following code for the factorial function implemented using this technique:

```
letregion  $\varrho$ 
let circfact =
  let id  $n = n$  in
  let  $x = \text{ref } [\varrho] \text{ id}$  in
  let  $f \ n = \text{if } n = 0 \text{ then } 1 \text{ else } n \times (!x) (n - 1)$  in
   $x := f$ ;
  ! $x$ 
```

It is possible to generalize this technique and to implement a generic fixed point combinator. Its unannotated code (omitting also region information) might look like this:

```
let backpatch ( $f : \text{ref } (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ ) =
  let id  $x = x$  in
  let  $x = \text{ref id}$  in
   $x := f \ x$ ;
  ! $x$ 
```

This “generic” backpatching function takes a higher-order function f as an argument. This function f expects a reference to a function in argument; its return value is another function of the same type, which is then affected to x . The argument of f needs to be a reference so that the effect of reading x can be delayed until *after* the assignment $x := f \ x$.

A problem of our function is that it is not entirely generic; we had to fix the type of the obtained function to $\alpha \rightarrow \alpha$ because we need to have a dummy value of the right type. We could also have asked the caller of *backpatch* to provide such a value.

Fully annotated, we obtain the code in Fig. 5.11. First of all, the interesting bits happen in a region ϱ , which has to be created outside of *backpatch*. A difference with the ML version is that we need a predicate p in argument; it takes a function and a state in argument. The precondition of *backpatch* is the most interesting part: it states

```

let backpatch [ $\alpha \rho$ ] ( $p : [\alpha \rightarrow^e \alpha] \rightarrow \langle r \rangle \rightarrow \text{prop}$ )
  ( $f : \text{ref}_r (\alpha \rightarrow^e \alpha) \rightarrow^\emptyset \alpha \rightarrow^e \alpha$ ) =
  {  $\forall x. \{ \} f x \{ g : \forall (s : \langle r \rangle). !!x s = g \rightarrow p g s \} \}$ 
  let  $id (x : \alpha) = \{ \} x \{ \}$  in
  let  $x = \text{ref} [r] id$  in
   $x := f x;$ 
  ! $x$ 
  {  $r : p r \text{ cur}$  }

```

Figure 5.11: The implementation in Who of *backpatch*.

that f must return a function g that, when x is equal to g , verifies p . This is the crucial property; it allows us to establish the postcondition, stating that the return value r also verifies p . This single non-trivial proof obligation of this third-order function is proved automatically.

We have already emphasized the fact that the first argument of f is a *reference* to a function. We could also replace the line

$$x := f x;$$

by the following line, where the partial application $f x$ has been eta-expanded and x is dereferenced instead to be passed to f directly.

$$x := (\text{fun } z \rightarrow \{ \dots \} f !x z \{ \dots \});$$

It would be an interesting challenge to find the specification for this anonymous function. For the function to be well-typed, we must change the type of f to

$$(\alpha \rightarrow^e \alpha) \rightarrow^\emptyset \alpha \rightarrow^e \alpha.$$

5.4.2. Memoization Functions

Let us now turn to the problem of memoization. The idea is to avoid repeated calls of a (pure) function f applied to the same argument, say x . When first calling f with argument x , the return value of f is stored in a table and looked up the next time we call f with the same argument. The hope is that, if executing f is expensive, then these table lookups are more efficient than a second call to f .

Implementing memoization requires some form of table that stores key-value pairs. We do not detail the implementation of such a table here; instead, we simply give an interface for a pure map from keys to values, accompanied by defining axioms, in Fig. 5.12. The type *map* represents pure maps from keys α to values β . We have the empty map, a function to test presence of a key, a function to get the value associated to a key and finally a function to store a key with a given value. If the key was already present, its value becomes overwritten with the new value. The access function returns an *option* type, so it returns *None* when there is no value for a given key in the map.

We first define a predicate *stores* $f m$, which states that the map m stores key-value pairs that correspond indeed to argument-result pairs of the function f .

5. Implementation and Case Studies

```

type map ( $\alpha, \beta$ )
empty :  $\forall \alpha \beta. \text{map } (\alpha, \beta)$ 
get :  $\forall \alpha \beta. \alpha \rightarrow \text{map } (\alpha, \beta) \rightarrow \beta \text{ option}$ 
set :  $\forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \text{map } (\alpha, \beta) \rightarrow \text{map } (\alpha, \beta)$ 
axiom get_set_eq :
   $\forall \alpha \beta. \forall (k : \alpha) (v : \beta) (\text{map} : \text{map } (\alpha, \beta)).$ 
    get k (set k v map) = Some v
axiom get_set_neq :
   $\forall \alpha \beta. \forall k_1 k_2 v (\text{map} : \text{map } (\alpha, \beta)).$ 
     $k_1 <> k_2 \rightarrow \text{get } k_1 (\text{set } k_2 v \text{ map}) = \text{get } k_2 \text{ map}$ 

```

Figure 5.12: The theory of maps in Who.

```

let memo [ $\alpha \beta \rho$ ] (table : ref $_{\rho}$  map( $\alpha, \beta$ )) (f :  $\alpha \rightarrow \beta$ ) (x :  $\alpha$ ) =
  {stores f !!table}
  match get x !table with
  | Some r  $\rightarrow$  r
  | None  $\rightarrow$ 
    let z = f x in
    table := set x z !table;
    z
  {r : r = f x  $\wedge$  stores f !!table}

```

Figure 5.13: The implementation in Who of *memo*.

```

let stores [ $\alpha \beta$ ] (f :  $\alpha \rightarrow \beta$ ) (m : map( $\alpha, \beta$ )) =
   $\forall (x : \alpha).$ 
    match get x m with
    | None  $\rightarrow$  True
    | Some v  $\rightarrow$  v = f x

```

Now we can give the code directly with its specification, in Fig. 5.13. The only precondition of *memo* assures that the table is valid w.r.t. *f*. This function first checks if the argument *x* is a key in the memo table; if it is, the corresponding value is returned. If it is not, *f* is called, the result *z* is stored in the table (better: the updated table is stored in the reference *table*), and finally *z* is returned. The postcondition clearly states that the result is the same as if one had called *f* directly, and additionally, that the invariant on the table is still valid. This last point is necessary if one wants to call *memo* several times. The proof obligations of *memo* are proved automatically.

Memoization across recursive calls. Sometimes one wants to define a recursive function with memoization for recursive calls. The classic example is the function *Fib*(*n*), computing the *n*th element of the well-known Fibonacci sequence. Implemented naively,

```

let ymemo table ff =
  let rec f x =
    match get x !table with
    | Some r → r
    | None →
      let z = ff f x in
      table := set x z !table;
      z
  in
  f

```

Figure 5.14: An implementation of *ymemo* in ML.

the complexity of the recursive function is exponential. But if the recursive calls are memoized, the process becomes linear. We show the implementation and the specification of a memoizing fixed point combinator which we call *ymemo*. The ML implementation is shown in Fig. 5.14. Similarly to *backpatch*, the function *ymemo* is a third-order function, expecting a functional (*ff* in this case) as argument. Indeed, both are fixed-point combinators. Inside *ymemo*, we define a recursive function *f*, which is somewhat similar to *memo* shown before. The main difference is the call to *ff* when the key *x* is not yet in the table; the function *f* passes itself to *ff* to allow recursion.

We start by defining a predicate *realizes table f₀ f* which is true when the effectful function *f* has the same results as the pure function *f₀*, as long as the *table stores f* (see the *memo* example).

$$\text{let } \textit{realizes} [\alpha\beta\varrho] (\textit{table} : \textit{ref}_\varrho \textit{map}(\alpha, \beta)) (f_0 : \alpha \rightarrow \beta) (f : \alpha \rightarrow^e \beta) = \\ \forall (x : \alpha). \{ \textit{stores } f_0 \textit{ !!table} \} f x \{ r : r = f_0 x \wedge \textit{stores } f_0 \textit{ !!table} \}$$

We can now give the fully annotated code of *ymemo*, in Fig. 5.15. It has an additional logical argument representing the computed mathematical function. Let us look first at the specification of the local function *f*. It is obvious that the predicate *realizes table f₀ f* is true if the specification is correct. In the precondition of *ymemo*, we precisely require that if *k* realizes *f₀*, then *ff k* also realizes *f₀*. We now want to check that the specification of *f* is indeed correct. The critical point is the call to *ff*, but by the discussion above we know that *f* validates the precondition of *ff*. We therefore can conclude that the value *z* computed by the call *ff f x* is the value that corresponds to the key *x*, and therefore the specification of *f* is correct. Note that we have assumed the correctness of *f* for recursive appearances while proving its body. A few proof obligations of *ymemo* are proved automatically, the more difficult ones have to be proved manually in Coq, but the proofs are short.

Hiding effects is not possible in Who. An aspect is common to the functions *backpatch*, *memo*, *ymemo*: an effect on a single reference is used locally to produce a certain result, but from the outside, the functions behave as if they were pure. The *backpatch* function, for example, is simply an alternative implementation of the classical fixed-point

5. Implementation and Case Studies

```

let ymemo [ $\alpha\beta\rho$ ] ( $f_0 : \alpha \rightarrow \beta$ ) ( $table : \text{ref}_\rho \text{map}(\alpha, \beta)$ )
  ( $ff : (\alpha \rightarrow^e \beta) \rightarrow^\emptyset \alpha \rightarrow^e \beta$ ) =
  {  $\forall (k : \alpha \rightarrow^e \beta). \{ \text{realizes } table \ f_0 \ k \} \text{ff } k \{ r : \text{realizes } table \ f_0 \ r \} \}$ 
  let rec f ( $x : \text{int}$ ) =
    { stores  $f_0$  !! $table$  }
    match get  $x$  ! $table$  with
    | Some  $r \rightarrow r$ 
    | None  $\rightarrow$ 
      let  $z = \text{ff } f \ x$  in
       $table := \text{set } x \ z$  (! $table$ );
       $z$ 
    {  $r : r = f_0 \ x \wedge \text{stores } f_0$  !! $table$  } in
  f
  {  $rf : \text{realizes } table \ f_0 \ rf$  }

```

Figure 5.15: The implementation in Who of *ymemo*.

combinator, which has the same type as *backpatch*, but no effects. Is it possible to give *backpatch* a type that does not mention any effect? The answer is no, at least in Who. First of all, there is no technical device to do that in W. The only way to hide effects in W is *letregion*, but it only permits to hide a region from an effect when the region does not appear in the type of the expression. Also, because of the Barendregt convention, it can never hide a region that appears in the typing environment — after all, *letregion* is a binding construct, so the region ρ bound there is always different from free regions. As in any of the cited functions, the region already appears in the argument types, it is impossible to hide it in W. This is a drawback of our system compared to recent work (Schwinghammer et al., 2010).

5.4.3. The Array Module

Arrays, *i.e.*, contiguous blocks of memory, are central to imperative programming, because they provide constant time random access. W does not provide arrays directly, but it is easy to model them following same strategy that has been applied to the *maps* in the previous section. It is also the same idea that is used to model arrays in the Why system.

The theory of arrays. In our programs dealing with arrays, instead of using actual arrays, which the language does not provide, we use references to *pure* arrays. A pure array cannot be modified; the *set* operation simply returns a new array, leaving the old unchanged. The theory of arrays is summarized in Fig. 5.16. If we compare this signature with the one for maps given earlier, the main difference is that *get* now does not return an object of *option* type, to be closer to the type of the access function of arrays in ML. The other important difference is that pure arrays are only valid if the index is between 0 and the length of the array. All the axioms are therefore protected

```

type array  $\alpha$ 
empty :  $\forall \alpha. \text{array } \alpha$ 
create :  $\forall \alpha. \text{int} \rightarrow \alpha \rightarrow \text{array } \alpha$ 
get :  $\forall \alpha. \text{int} \rightarrow \text{array } \alpha \rightarrow \alpha$ 
set :  $\forall \alpha. \text{int} \rightarrow \alpha \rightarrow \text{array } \alpha \rightarrow \text{array } \alpha$ 
length :  $\forall \alpha. \text{array } \alpha \rightarrow \text{int}$ 
axiom getseteq :
   $\forall \alpha. \forall (i : \text{int}) (v : \alpha) (a : \text{array } \alpha).$ 
     $0 \leq i < \text{length } a \rightarrow \text{get } i (\text{set } i v a) = v$ 
axiom getsetneq :
   $\forall \alpha. \forall (i j : \text{int}) (v : \alpha) (a : \text{array } \alpha).$ 
     $0 \leq i < \text{length } a \wedge 0 \leq j < \text{length } a \wedge i \neq j$ 
     $\rightarrow \text{get } i (\text{set } j v a) = \text{get } i a$ 
axiom length_set :
   $\forall \alpha. \forall (i : \text{int}) (v : \alpha) (a : \text{array } \alpha).$ 
     $\text{length } (\text{set } i v a) = \text{length } a$ 
axiom length_empty :  $\forall \alpha. \text{length } (\text{empty}) = 0$ 
axiom get_create :
   $\forall \alpha. \forall (n : \text{int}) (v : \alpha). 0 \leq i < n \rightarrow \text{get } (\text{create } n v) = v$ 

```

Figure 5.16: The theory of arrays in Who.

by hypotheses over the indices in question. Finally, the function *create* allows to create an array of a given length, filled with a value *v*.

The term *create n v* stands for an array of length *n*, filled with a value *v* (for any integer *i*, *get i (create n v) = v*), while the term *empty* stands for the empty array, of length 0.

Mutable arrays are now modeled by references to such pure arrays. Additionally, we introduce program functions using *get* and *set*, that artificially introduce restrictions on the indices to be used (Fig. 5.17). We see that modifying the array in place is modeled by an assignment of the new array value to the reference. The preconditions are artificial in the sense that the correctness of these functions could be proved without them; but then our modelization would not reflect usual mutable arrays precisely. Finally, we can introduce syntactic sugar: we write *a.(i)* instead of *_get i a* and *a.(i) ← v* instead of *_set i v a*.

Iteration over arrays. Apart from accessing and modifying elements, other common operations include appending (concatenating copies of two arrays to form a new, larger array), filling (part of) the array with a certain value, extracting a sub array, copying (part of) one array into another, sorting, and so on. These are all first-order operations and have been dealt with in many systems, many times. The new possibilities provided by W deal with higher-order functions. One such case is the higher-order iteration function, commonly called *iter*:

5. Implementation and Case Studies

```

let _get [ $\alpha\rho$ ] (i : int) (a :  $\text{ref}_\rho$  (array  $\alpha$ )) =
  {  $0 \leq i < \text{length } a$ 
    get i !a
    {  $r : r = \text{get } i !!a$  }
let _set [ $\alpha\rho$ ] (i : int) (v :  $\alpha$ ) (a :  $\text{ref}_\rho$  (array  $\alpha$ )) =
  {  $0 \leq i < \text{length } a$ 
    a := set i v !a
    {  $!!a = \text{set } i v (!!a \text{ old})$  }

```

Figure 5.17: Wrappers for *get* and *set*.

```

let iter [ $\alpha\rho\varepsilon$ ] (inv :  $\langle\rho\varepsilon\rangle \rightarrow \text{int} \rightarrow \text{prop}$ ) (a :  $\text{ref}_\rho$  (array  $\alpha$ )) (f :  $\alpha \rightarrow^{\rho\varepsilon} \text{unit}$ ) =
  { inv cur 0  $\wedge$ 
     $\forall(i : \text{int}). 0 \leq i < \text{length } !!a \rightarrow$ 
      { inv cur i } f (get i !a) { inv cur (i + 1)  $\wedge \text{length } !!a = \text{length } !!a|\text{old}$  }
  }
  for i = 0 to length !a - 1 do
    { inv cur i }
    f a.(i)
  done
  { inv cur (length !!a) }

```

Figure 5.18: The implementation in *Who* of *iter*.

```

let iter f a =
  for i = 0 to length a - 1 do
    f a.(i)
  done

```

In this particular case, the function is simply a **for**-loop; still, using *iter* instead of directly writing the loop helps avoid programming errors on the two indices. Another reason for the existence of this function is the uniformity of interfaces in a language; iteration functions should exist for all data structures, and in most cases they are not as simple as the one presented here.

Let us now go on to specify this function. Fig. 5.18 gives the annotated code. From a typing point of view, we again use effect polymorphism to characterize the effect of *f*. For simplicity, we say that *f* also potentially modifies the region of array *a*. From a specification point of view, the main idea is again an additional logical argument representing the invariant *inv* maintained by the iteration. It takes the current state as well as the integer index up to which we have iterated as arguments. Conveniently, we can simply put the predicate *inv cur i* as the **for**-loop invariant. The postcondition is also very simple: it states that we obtain the invariant for the end of the array. The precondition is slightly more complicated because of the presence of the Hoare triple: it

```

let map f a =
  let l = length a in
  if l = 0 then ref empty
  else
    let k = f a.(0) in
    let r = ref (create l (f a.(0))) in
    for i = 1 to l - 1 do
      r.(i) ← f a.(i)
    done ;
  r

```

Figure 5.19: The *map* function for Arrays in ML.

states that f , applied to the i th element of the array, takes the invariant from one state and index i to another state and index $i + 1$. Finally, we have to require the invariant for the index 0.

The reader has certainly noticed that we also require the function f in argument to preserve the length of the input array a . This is a consequence of our modelization of arrays; we cannot assume that *set* is the only function that is used to modify an array that is stored in a reference. For example, a user might simply write

$$a := \text{empty}$$

to change the length of an array reference. The additional requirement in the post-condition of f avoids such pathological cases. All proof obligations of *iter* are proved automatically.

Mapping over arrays. Another usage of higher-order functions in combination with arrays in ML-like languages is mapping. Entirely analogous to the *map* function for lists, this map function accepts an array a and a function f and returns another array of identical length whose i th element is the result of the call $f a.(i)$. The function f is called with cells in increasing order. The ML code is shown in Fig. 5.19. We first remark that in the special case where the length of the array is 0, we simply return the empty array. In the other case, we call f on the first cell of the input array, create a new array r of the full length, filled with this value. We then iterate, starting from 1 and not from 0, till the end of the input array, and fill r with the right values. Finally, we return the array r .

The difficulty of specifying *map* for arrays is the fact that during the entire function call, except at the very end of the execution, the result array r contains invalid entries, originating from the call to *create*, that have not been overwritten yet by the *for*-loop. We call $r.(i)$ when it not the result of a call of the form $f a.(i)$. Initially, this is only true for $i = 0$, and, as stated, it is only true for all values in r when the *for*-loop has terminated.

It would therefore be wrong to blindly apply the previous trick, which consists in stating that f maintains some invariant *inv*. Let us try for a moment to see the

5. Implementation and Case Studies

```

let map [ $\alpha\beta\rho_1\rho_2\varepsilon$ ] (inv :  $\langle\rho_1\varepsilon\rangle \rightarrow \text{array } \beta \rightarrow \text{int} \rightarrow \text{prop}$ )
  (f :  $\alpha \rightarrow^{\rho_1\varepsilon} \beta$ ) (a :  $\text{ref}_{\rho_1}(\text{array } \alpha)$ ) =
  { inv cur $_{\rho_1\varepsilon}$  empty 0  $\wedge$ 
     $\forall(k : \beta \text{ array}) (i : \text{int}). 0 \leq i < \text{length } !!a \rightarrow$ 
      { inv cur (sub k i) i } f (get i !a) { r : inv cur (sub (set i r k) (i+1)) (i+1) }
    }
  let l = length !a in
  if l = 0 then ref $_{\rho_2}$  empty
  else
    let k = f a.(0) in
    let r = ref $_{\rho_2}$  (create l k) in
    for i = 1 to l - 1 do
      { inv cur $_{\rho_1\varepsilon}$  (sub !!r i) i }
      r.(i)  $\leftarrow$  f a.(i)
    done ;
    r
  { r : inv cur $_{\rho_1\varepsilon}$  !!r (length !!a) }

```

Figure 5.20: The implementation in Who of *map*.

problem. The result of *map* is an array, so at the very least, *inv* takes an array and an integer (the index up to which the array is valid) as arguments. However, nothing prevents the invariant to depend on invalid parts. We need some way to provide the invariant with only the valid part of the array, which has already been filled with return values of *f*.

A straightforward solution is the introduction of a logical function *sub*, of type

$$\forall\alpha.\text{array } \alpha \rightarrow \text{int} \rightarrow \text{array } \alpha,$$

such that *sub a n* returns an array of length *n* such that for all indices smaller than *n*, the access to the sub-array returns the same result as an access to *a*. If we pass such a shortened array to the invariant, it cannot access any invalid parts. Fig. 5.20 shows the (relatively complex) specification. First, the precondition requires the invariant to be true for the empty array; this is required when the input array is already empty: the postcondition and the first part of the precondition are identical in this case. The second part of the precondition expresses that the call *f (get i !a)* maintains the invariant. As stated above, we cannot simply pass the array *k* to the invariant, because *k* may contain garbage. Instead, we pass the smaller array *sub k i*; now, this array only contains relevant fields.

Notice that we have assumed *extensionality* for pure arrays: pure arrays that contain the same elements are equal. For example, to prove the above function, we must have the equality

$$\text{sub } a \ 0 = \text{empty}$$

for any array *a*. All proof obligations of the *map* function are proved automatically.

```

let rec mapinv f l =
  match l with
  | Nil → Nil
  | Cons (x, rl) →
    let tl = mapinv f rl in
    let hd = f x in
    Cons hd tl

```

Figure 5.21: The function *mapinv* in ML.

5.4.4. The List Module

The other container data type that exists in all ML-like languages is the *list*. We have already seen the definition in ML of this data type in Chapter 1, let us repeat it here:

```

type list α =
  | Nil
  | Cons of α * list α

```

So, a list in ML is either empty, or a head element “consed” (attached) to a list tail.

Many operations on lists are naturally expressed higher-order functions, because of the high genericity of the type definition. This includes iteration (similar to iteration on arrays) and filtering, *i.e.*, building a new list that contains only the elements of a given list that verify a certain predicate, among others.

A very common operation on lists is the *map* function. This function takes a function f and a list l as argument and calls f on each element of the list. It returns a list r that contains all the return values of the calls to f , in the same order as the elements in l . In other words, if l contains the elements x_1, \dots, x_n , *map* builds a new list of same length, with the elements $f x_1, \dots, f x_n$. Let us consider an implementation this idea in a function that we call *mapinv* (Fig. 5.21) for reasons that will become clear soon. The code is quite simple: *mapinv* simply traverses the list and calls f on each node, building the output list using the results of f . The important detail of this implementation is that the recursive call happens *before* the call to f . In practice, this means that f is called on the last element of the input list first.

Let us specify *mapinv*. It is quite simple to specify because the construction of the output list and the calls to f are synchronized. Fig. 5.22 shows the specification. We again generalize over the invariant to be maintained; it is a predicate over a state and two lists. In the case of a non-empty list, it is easy to see that if the recursive call establishes *inv s rl tl*, where s is the state after the recursive call, then the precondition on f guarantees the postcondition.

A problem of *mapinv* is that f is called on the last element of the input list first. It would be easier to understand *map f l* as the list $f x_1, f x_2, \dots, f x_n$, where the function f is called in *that order*. The actual implementation of *map* in the standard library achieves this by forcing the recursive call to happen *after* the call to f . The implementation of *map* in ML is given in Fig. 5.24.

5. Implementation and Case Studies

```

let rec mapinv [ $\alpha\beta\varepsilon$ ] (inv :  $\langle\varepsilon\rangle \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list} \rightarrow \text{prop}$ )
  (f :  $\alpha \rightarrow^\varepsilon \beta$ ) (l :  $\alpha \text{ list}$ ) =
  { ( $\forall s. \text{inv } s \text{ Nil Nil}$ )  $\wedge$ 
     $\forall x l l_2. \{ \text{inv cur } l l_2 \} f x \{ r : \text{inv cur (Cons } x l) (\text{Cons } r l_2) \}$  }
match l with
| Nil  $\rightarrow$  Nil
| Cons (x, rl)  $\rightarrow$ 
  let tl = mapinv inv f rl in
  let hd = f x in
  Cons hd tl
{ r : inv cur l r }

```

Figure 5.22: The implementation in Who of *mapinv*.

```

let rec map [ $\alpha\beta\varepsilon$ ] (ia :  $\langle\varepsilon\rangle \rightarrow \alpha \text{ list} \rightarrow \text{prop}$ )
  (ib :  $\langle\varepsilon\rangle \rightarrow \langle\varepsilon\rangle \rightarrow \beta \text{ list} \rightarrow \text{prop}$ )
  (f :  $\alpha \rightarrow^\varepsilon \beta$ ) (l :  $\alpha \text{ list}$ ) =
  { ia cur l  $\wedge$  ( $\forall s s'. \text{ib } s s' \text{ Nil}$ )  $\wedge$ 
    ( $\forall s_1 s_2 s_3 x r l. \text{ib } s_2 s_3 l \wedge \text{post } f x s_1 s_2 r \rightarrow \text{ib } s_1 s_3 (\text{Cons } r l)$ )  $\wedge$ 
     $\forall x l. \{ \text{ia cur (Cons } x l) \} f x \{ r : \text{ia cur } l \}$ 
  }
match l with
| Nil  $\rightarrow$  Nil
| Cons (x, rl)  $\rightarrow$ 
  let hd = f x in
  Cons hd (map ia ib f rl)
{ r : ia cur Nil  $\wedge$  ib old cur r }

```

Figure 5.23: The implementation in Who of *map* for lists.

```

let rec map f l =
  match l with
  | Nil → Nil
  | Cons (x, rl) →
      let hd = f x in
      Cons hd (map f rl)

```

Figure 5.24: The implementation of *map* in ML.

The specification of *map* is more complicated, because now the effect of *f* and the effect of the recursive call are mixed. We give its specification in Fig. 5.23. The main difficulty is that now *three* states are involved in the case of a non-empty list. The initial state, the state after calling *f* and the state after the recursive call. We decide here to split the invariant into two parts. The part of the invariant maintained by *f* is called *ia*. This becomes clear in the precondition, where we state that *f* maintains *ia*. The same condition also makes it clear that the list argument of *ia* is intended to be the input list, because it becomes smaller after the call to *f*. The second part of the invariant is represented by *ib* and corresponds to the recursive call. The *ib* predicate is intended to relate the initial state s_1 and the final state s_3 of the call to *map*. However that the recursive call only establishes this predicate for the intermediate state s_2 and s_3 . The precondition now establishes the necessary link, if s_1 and s_2 are pre- and poststate of a call to *f*, respectively.

The only difference between the functions *mapinv* and *map* is the order in which the effects happen. If the function *f* does not have any side effects, the functions are of course equivalent. All proof obligations are discharged automatically.

5.4.5. Koda and Ruskey's Algorithm

The development presented in this section has been published (Kanig and Filliâtre, 2009).

The algorithm considered in this section is due to Koda and Ruskey (1993) — Knuth (2001) has given a very efficient implementation. From a mathematical point of view, the algorithm enumerates the ideals of certain finite partially ordered sets — namely, those whose Hasse diagram is a forest—as a Gray code. Expressed in different terms, the task is to enumerate all *colorings* of a given, arbitrary forest. A coloring consists in marking every node as either black or white, with the only constraint that all descendants of a white node be white as well. For instance, the following forest:



admits exactly 15 distinct colorings, all of which are given in Fig. 5.25. By definition, a sequence of colorings forms a Gray code if and only if every coloring of the forest appears exactly once in it and two consecutive colorings differ by the color of exactly one node.

5. Implementation and Case Studies

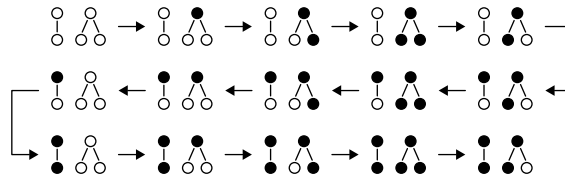


Figure 5.25: Koda and Ruskey’s algorithm applied to the forest (5.2).

Let us illustrate the algorithm’s functioning on the forest (5.2). The main idea is to interleave the sequences of colorings which correspond to each of the trees that form the forest. Here, one must interlace the sequence of the three colorings of the left-hand tree, namely:

$$\begin{array}{c} \circ \\ | \\ \circ \end{array} \quad \begin{array}{c} \bullet \\ | \\ \circ \end{array} \quad \begin{array}{c} \bullet \\ | \\ \bullet \end{array} \quad (5.3)$$

with the sequence of the five colorings of the right-hand tree, given below:

$$\begin{array}{c} \circ \\ | \\ \circ \end{array} \quad \begin{array}{c} \bullet \\ | \\ \circ \end{array} \quad \begin{array}{c} \bullet \\ | \\ \bullet \end{array} \quad \begin{array}{c} \bullet \\ | \\ \bullet \end{array} \quad \begin{array}{c} \bullet \\ | \\ \bullet \end{array} \quad (5.4)$$

Thus, the first line of Fig. 5.25 exhibits the first coloring of the left-hand tree, combined successively with all colorings of the right-hand tree. The second line shows the second coloring of the left-hand tree, again combined with all colorings of the right-hand tree, but this time in reverse order — indeed, it is clear that the mirror image of a Gray code remains a Gray code. Finally, the third line exhibits the third coloring of the left-hand tree and all colorings of the right-hand tree, this time again in their initial order.

There remains to explain how to enumerate all colorings of a tree. Let the first coloring be uniformly white. Then, to obtain the remainder of the sequence, color the root node black and enumerate all colorings of the forest formed by its children. The sequence thus obtained is indeed a Gray code, because (i) the first and second colorings differ only by the color of the root node and (ii) from then on, the root node remains unaffected, and the sequence of the colorings of the children forms a Gray code by construction. This process is illustrated by (5.3) and (5.4) above. Note that the coloring where every node is black does not necessarily appear last in a sequence.

Functional implementation. We consider an OCaml implementation of Koda and Ruskey’s algorithm which makes use of higher-order functions (Filliâtre and Pottier, 2003). First, we introduce the types for trees, forests and colors as follows:

```
type tree = Node of int * forest
and forest = tree list
type color = White | Black
```

A tree is thus a term $\text{Node}(i, f)$ where i is the index of its root and f the forest of its sub-trees. A forest is simply a list of trees. The current coloring of the considered forest will be materialized in a global array *bits*, which is assumed to be large enough to contain all indices of the forest.

```

let rec enumforest k f =
  match f with
  | [] →
    k ()
  | Node (i, f') :: f →
    let k () = enumforest k f in
    if bits.(i) = White then begin k (); bits.(i) ← Black; enumforest k f'
    end else begin enumforest k f'; bits.(i) ← White; k ()
    end
end

```

Figure 5.26: An OCaml implementation of Koda and Ruskey’s algorithm.

A nice way to implement Koda and Ruskey’s algorithm is to use a continuation-based approach, using a recursive function *enumforest* with the following type:

$$\text{enumforest} : (\text{unit} \rightarrow \text{unit}) \rightarrow \text{forest} \rightarrow \text{unit}$$

It takes a continuation *k* and a forest *f* as arguments. Then it enumerates all colorings of *f*, applying continuation *k* once for each different coloring of *f*.

The code for *enumforest* is given in Fig. 5.26 and proceeds as follows. If the forest is empty, we simply call the continuation *k*. Otherwise, the forest contains at least one tree, say *Node*(*i*, *f'*), next to a sub-forest *f*. We first build a new continuation *k* which enumerates the colorings of *f*, using the old continuation *k*. Then we consider the tree itself. The function must be able to enumerate the colorings in both directions (as explained in the next section). To determine which, we look up the color of the tree’s root, that is *bits*(*i*). If it is currently white, then the whole tree must be white. We have a complete coloring, so we signal the continuation *k*; then, we color the root black and enumerate its children’s colorings using *enumforest*. If, on the other hand, the root is currently black, we do the converse. That is, we first use *enumforest* to enumerate the children’s colorings in reverse order, which leaves all of the children entirely white; then, we color the root white, and signal the continuation *k*.

Formal specification. We are now going to give a formal specification to this functional implementation. In particular, we should characterize what is the effect of continuation *k*. Obviously, it modifies the contents of array *bits*, since it is precisely used to do so in recursive calls. But *k* may have other effects, if for instance the initial continuation is used to print the current coloring or to record it in some array². Therefore, we use effect polymorphism to indicate that *k* may have some effect ε , disjoint from *bits*:

$$\text{enumforest} : \forall \varepsilon. (\text{unit} \xrightarrow{\text{bits}, \varepsilon} \text{unit}) \rightarrow \text{forest} \xrightarrow{\text{bits} \varepsilon} \text{unit}$$

To specify the behavior of *enumforest*, we must also exhibit the forest whose colorings

²For the purpose of drawing pictures such as the one in Fig. 5.25, *enumforest* can be used with a continuation *k* producing pictures.

5. Implementation and Case Studies

```

let rec enumforest [ε] (f0 : forest) (k : unit →bits ε unit) (f : forest) =
  { validf (append f f0) ∧ anyf bits (append f f0) ∧
    { validf f0 ∧ anyf bits f0 } k () { mirrorf bits|old bits f0 ∧ eqout bits|old bits f0 }
  }
  match f with
  | [] → k ()
  | Node (i, f') :: f →
    let k () =
      { validf (append f f0) ∧ anyf bits (append f f0) }
      enumforest f0 k f
      { mirrorf bits|old bits (append f f0) ∧ eqout bits|old bits (append f f0) }
    in
    if iswhite (get !bits i) then
      (k (); bits := set !bits i Black ; enumforest (append f f0) k f')
    else
      (enumforest (append f f0) k f'; bits := set !bits i White; k ())
  { mirrorf bits|old bits (append f f0) ∧ eqout bits|old bits (append f f0) }

```

Figure 5.27: Who implementation of Koda and Ruskey’s algorithm.

are enumerated by the continuation, as an additional argument, say f_0 . Thus the Who implementation of Koda and Ruskey’s algorithm has three parameters, and looks like

```

let rec enumforest [ε] (f0 : forest) (k : unit →bits, ε unit)
  (f : forest) = ...

```

The additional argument f_0 is *logical*, since it only participates to the specification and not to the computation.

We now turn to the specification itself. Here, we focus on the behavior of *enumforest* with respect to the current coloring, *i.e.*, we characterize the conditions under which the function can be called and its effect on the contents of array *bits*. We do not prove that the set of all colorings form a Gray code, but this could be deduced without too much effort. The Who code for function *enumforest* is given in Fig. 5.27 and its annotations are detailed in the remaining of this section.

The first requirement is a sanity condition over forests f and f_0 , which says that they do not contain duplicate indices. We write $i \in t$ (resp. $i \in f$) when i is an index occurring in tree t (resp. in forest f). We also write *valid* t (resp. *valid* f) to characterize a tree t (resp. a forest f) where all indices are different. These two notions of occurrence and validity are easily defined inductively over trees and forests³. If *append* denotes the concatenation of forests, we thus require

valid (*append* f f_0)

³In the formal development, we distinguish *validt* for trees and *validf* for forests, since there is no overloading; in this description, we simply write *valid* for greater clarity. We proceed similarly for other predicates defined on trees and forests.

as a precondition.

The next requirement is a condition over the current coloring, *i.e.*, the state of *bits*, for *enumforest* to execute correctly. Requiring the nodes to be all colored in white is a too strong condition, since recursive calls are going to be used to “decolor” some trees, as in the second row of Fig. 5.25. We must thus characterize the final coloring of a tree or a forest. Obviously, the parity of the number of colorings plays a role. Indeed, in a forest containing two trees, say t_1 and t_2 in that order, the final coloring of t_2 will be all white if t_1 admits an even number of colorings, and will be itself a final coloring of t_2 otherwise. We introduce the predicates **even** and **odd**, over trees and forests, to indicate an even (resp. odd) number of colorings. They are inductively defined as follows:

$$\begin{array}{ccc} \frac{\text{even } t}{\text{even } (t :: f)} & \frac{\text{even } f}{\text{even } (t :: f)} & \frac{\text{odd } f}{\text{even } (\text{Node}(i, f))} \\ \\ \frac{}{\text{odd } []} & \frac{\text{odd } t \quad \text{odd } f}{\text{odd } (t :: f)} & \frac{\text{even } f}{\text{odd } (\text{Node}(i, f))} \end{array}$$

We can now define the notions of initial and final colorings. In the following, s stands for a possible state of array *bits*, that is an array of colors. The predicate $\mathbf{I } s f$ characterizes an initial state s for a given forest f , as being all-white:

$$\mathbf{I } s f \stackrel{\text{def}}{=} \forall i, i \in f \Rightarrow s(i) = \text{White}$$

Similarly, the predicate $\mathbf{F } s t$ (resp. $\mathbf{F } s f$) characterizes a final state s for a tree t (resp. a forest f):

$$\begin{array}{ccc} \frac{\mathbf{F } s t \quad \text{even } t \quad \mathbf{I } s f}{\mathbf{F } s (t :: f)} & & \frac{\mathbf{F } s t \quad \text{odd } t \quad \mathbf{F } s f}{\mathbf{F } s (t :: f)} \\ \\ \frac{}{\mathbf{F } s []} & & \frac{s(i) = \text{Black} \quad \mathbf{F } s f}{\mathbf{F } s (\text{Node}(i, f))} \end{array}$$

The precondition of *enumforest* requires each tree of the forest to be either in an initial or final state, which can be defined as follows:

$$\mathbf{any } s [f_1; \dots; f_n] \stackrel{\text{def}}{=} \forall i, \mathbf{I } s f_i \vee \mathbf{F } s f_i$$

More precisely, the precondition requires that **any** holds on the concatenated forest *append* $f f_0$. We also require that **valid** and **any** are sufficient conditions to ensure the precondition of k . Finally, the new continuation k which is built in *enumforest* is given the same requirement.

We now turn to the postcondition of *enumforest*. Simply speaking, we want to state that it switches the coloring of the forest from initial to final and conversely. As for the precondition, it would be a too strong requirement and we need to characterize the effect of *enumforest* more subtly. Again, the parity of the number of colorings is playing a role, since an even number of colorings for the first tree would result in an unchanged coloring for the remaining of the forest and, conversely, an odd number for the first tree

5. Implementation and Case Studies

would result in a switch for the remaining of the forest. To denote unchanged colorings, we introduce the following predicate over two different states s_1 and s_2 :

$$\text{same } s_1 \ s_2 \ f \stackrel{\text{def}}{=} \forall i, i \in f \Rightarrow s_1(i) = s_2(i)$$

Then we can define the effect of *enumforest* between pre-state s_1 and post-state s_2 , as the following, inductively defined predicate **mirror**:

$$\frac{\frac{\text{I } s_1 \ t \quad \text{F } s_2 \ t}{\text{mirror } s_1 \ s_2 \ t} \quad \frac{\text{F } s_1 \ t \quad \text{I } s_2 \ t}{\text{mirror } s_1 \ s_2 \ t}}{\text{mirror } s_1 \ s_2 \ []}$$

$$\frac{\text{mirror } s_1 \ s_2 \ t \quad \text{odd } t \quad \text{mirror } s_1 \ s_2 \ f}{\text{mirror } s_1 \ s_2 \ (t :: f)}$$

$$\frac{\text{mirror } s_1 \ s_2 \ t \quad \text{even } t \quad \text{same } s_1 \ s_2 \ f}{\text{mirror } s_1 \ s_2 \ (t :: f)}$$

This is the expected postcondition for *enumforest*, and thus also a requirement over continuation k .

As such, the specification of *enumforest* is incomplete, as it does not say anything about the colors for the indices which are not in f . Since the state of *bits* is considered as a whole, these colors could have been changed. But for the correctness proof of *enumforest*, it is necessary to know that recursive calls will not modify the color of node i . Thus we need a stronger postcondition, which “frames” the effects on array *bits*. For that purpose, we introduce the predicate

$$\text{eqout } s_1 \ s_2 \ f \stackrel{\text{def}}{=} \forall i, i \notin f \Rightarrow s_1(i) = s_2(i)$$

and use it in the postcondition of *enumforest* and k .

Formal proof. When processed with Who, the code in Fig. 5.27 results in 17 proof obligations. They have been discharged using the Coq proof assistant, with the use of several auxiliary lemmas over predicates **even**, **any**, **mirror**, etc., which have been proved in Coq as well. Proving that the obtained enumeration is indeed a Gray code would be interesting future work.

5.4.6. A Challenge for the Who Tool

We have succeeded in proving a number of programs where higher-order features and effects are intricately connected. It is a long-term objective to prove larger programs, where also other aspects play a rôle, such as being able to modularly organize a proof. We believe that our system and tool does have all the necessary properties that permit modular reasoning, even on a larger scale. The ingredients that render this possible are effect and region polymorphism, the possibility to express properties about functional values, and the effect analysis. From a theoretical point of view, the frame rule (Theorem 4.4) guarantees that one can concentrate on the part of the state that has been modified by a given expression.

Still, a larger example than the ones we have presented would be rewarding to prove correct. A possible candidate is the lazy fixed-point computation proposed by Pottier (2009). The goal of this program is to compute the least fixed-point of a set of recursive equations between variables. The interesting point of the algorithm is that these equations are not given *syntactically*, *i.e.*, by a representation as a data type in ML, but *semantically*, *i.e.*, simply by ML expressions. One could also say that Pottier proposes a shallow embedding instead of the usual deep embedding (see also the discussion on page 148).

Usually, a system of equations is given in the following syntactic form:

$$\begin{aligned} x_1 &= E_1(\bar{x}) \\ x_2 &= E_2(\bar{x}) \\ \dots & \\ x_n &= E_n(\bar{x}) \end{aligned}$$

where the x_i are variables and the right-hand sides E_i are expressions containing these variables. A fixed-point of such a system is an assignment of all variables such that the equations hold.

Pottier now takes a semantic approach, *i.e.*, the user can give the equations directly in ML syntax. He achieves this by defining a *valuation* to be a function from variables to values:

type valuation = variable → value

A right-hand side *rhs* is a function from a valuation to a value:

type rhs = valuation → value

And finally, a set of equations is a mapping from a variable to an *rhs*.

type equations = variable → rhs

The advantage of this approach is that the user can define an *rhs* as an ML function

```
let rhsi (v : valuation) =
  Ei(v y1, ..., v yn)
```

where he only has to replace occurrences of the variables y_i by calls to the valuation v , with the variable as argument.

Now, Pottier provides a function

lfp : equations → valuation

that computes the least fixed-point of the list of equations. Unfolding all the type definitions, we see that *lfp* is a function of third order. Internally, the function does a certain number of side effects to minimize the number of computations.

A central aspect of the implementation of *lfp* is how it detects that a right-hand side depends on a certain variable. We do not explain the exact implementation here, instead we give the intuition. Imagine that we have a function

$$f : \text{int} \rightarrow \text{int},$$

5. Implementation and Case Studies

but we do not have its definition, and we want to know if the function f uses its argument. If it does not, it must be a constant function. The problem, stated as-is, cannot be solved in ML, but by slightly modifying it we can propose a solution. Let us change the type of f :

$$f : (\text{unit} \rightarrow \text{int}) \rightarrow \text{int}$$

where we have modified the type of the argument of f from int to $\text{unit} \rightarrow \text{int}$. The integer 1 would be represented as the constant function

$$\lambda().1$$

Now, whenever f wants to access the value of f , it needs to *execute* its argument. This gives us a way to *detect* if f uses its argument. By passing an *effectful* function to f , we can observe if f executes its argument, for example as follows:

```
let b = ref false in
let x () = b := true; 1 in
f x
```

Now, if b is true after executing f , it must have read x ; otherwise it hasn't.

Pottier presents his program as a challenge to the program verification community. We firmly believe that it is possible to prove that *lfp* computes a fixed point of the initial set of equations.

6. Conclusion and Outlook

It is time to recapitulate the achievements, see what they can be used for and finally sketch possible improvements and future work. In this last chapter of the thesis, we first summarize briefly our contributions. We then discuss how close we have come to the initial goal of the thesis, namely to be able to prove realistic programs in an existing ML dialect; here, we focus on OCaml. Finally, we discuss remaining challenges and possible improvements.

6.1. A Summary of the Contributions of this Thesis

The two central contributions of this thesis are

- a theoretical system that allows to specify effectful higher-order programs and to generate proof obligations that imply the correctness of the program w.r.t. its specifications,
- and an *implementation* of this system called *Who*, that allows to actually prove the generated obligations using standard automated and interactive theorem provers.

The theoretical system comprises

- an ML-like programming language called *W* with type and effect system very close to the one by Talpin and Jouvelot (1994),
- a new specification language called *L* that extends a standard higher-order logic,
- a new weakest precondition calculus that combines features of the Why system (Filliâtre, 2003) and the Pangolin system (Régis-Gianas and Pottier, 2008), accompanied by a soundness and completeness proof,
- two restrictions of the initial system, namely the exclusion of region aliasing and the restriction to singleton regions, that generalize ideas by Hubert and Marché (2007) and Filliâtre (2003), respectively.

The system allows a modular specification of higher-order functions in the presence of side effects. This modularity is achieved by using a higher-order logic as annotation language and by providing effect polymorphism, which allows to abstract over the effect of a functional argument.

The *Who* tool contains

- an implementation of all parts of the theoretical system,
- an encoding of the non-standard parts of *L* to a standard higher-order logic, and a means to export proof obligations to Coq (The Coq Development Team, 2008),

6. Conclusion and Outlook

- an encoding of standard higher-order logic to first-order logic, more precisely the logic of the Why tool (Filliâtre and Marché, 2007), to be able to use all automated and interactive provers that are supported by the Why tool, to discharge proof obligations.

Using this tool, a number of programs, that mix higher-order functions and effects in interesting ways, have been proved correct:

- higher-order functions on arrays and lists, such as *iter* and *map*, applied to functions that produce side effects,
- third-order functions such as *ymemo* and *backpatch* (Landin’s knot) that are known to be difficult to reason about,
- Koda and Ruskey’s algorithm, in a continuation-based variant.

Higher-order iterators in presence of effects have already been discussed by others, for example in the Ynot system (Nanevski et al., 2008), in a less general way by Honda et al. (2005), and more recently by Borgström et al. (2010). Only in Ynot, there is an actual implementation and an associated proof. Honda et al. (2005) also discuss Landin’s knot, but only in a concrete case (the factorial function), not in its general implementation as a fixed-point combinator. To our knowledge, Koda and Ruskey’s algorithm has not been subject to program verification before.

6.2. Using Who to Verify OCaml Programs

A natural question to ask is whether we can directly apply the results of this thesis to prove programs in a concrete implementation of ML, such as OCaml, for example. In theory, the answer is mostly yes, but in practice a few more obstacles are to overcome. Let us elaborate.

Language features that are already supported or easy to add. Many of the language constructs of OCaml are also present in *W*, or can be easily added or encoded. *W* already contains the functional core of ML, recursive functions and references, and higher-order functions. We have also briefly discussed that algebraic data types and pattern matching are easy to add, and both are present in the *Who* tool. The same is true for tuple types. In Section 5.4.3, we have shown that the arrays of OCaml can be easily modeled in *Who*.

OCaml also provides records. A record type definition looks like this:

```
type t = { f1 : τ1 ; ... ; fn : τn }
```

A value that belongs to this type is called a *record*, and is a mapping from the *fields* *f*_{*i*} to a value of type *τ*_{*i*}. The names *f*_{*i*} of the fields are also used for accessor functions to the corresponding value of a field.

In principle, records can be easily dealt with by *Who*. Simply replace the record type definition by a declaration of an algebraic data type with a single constructor and an argument for each field of the record type:

$$\text{type } t = \text{TRec of } \tau_1 \times \cdots \times \tau_n$$

The accessor functions can now be defined by pattern matching.

Other features of OCaml require a bit more work, but seem to be easy to add. We do not deal directly with other effects such as exceptions and input/output. However, the effect mechanisms in the literature (effects, monads, capabilities) have all been shown to address these effects easily, and [Filliâtre \(2003\)](#) has shown this in the context of program verification. We believe that similar adjustments can be easily applied to *W* and *Who*. The challenge of OCaml's *polymorphic variants* ([Garrigue, 1998](#)) seems to lie only in their complex typing relation, but they can otherwise be treated much like regular algebraic data types.

Mutable records. The encoding we just showed for OCaml's record types does not take into account the fact that record fields can be declared *mutable*, *i.e.*, in-place modifiable. A simple modification of our encoding consists of declaring a *reference type* for such a field, *i.e.*, the declaration

$$\text{type } t = \{ f_1 : \tau_1 ; \text{mutable } f_2 : \tau_2 \}$$

is encoded to the region polymorphic type

$$\text{type } t [\rho] = \text{TRec of } \tau_1 \times \text{ref}_\rho \tau_2$$

This encoding works, but it does not reflect an important fact of records: the mutable fields of two *different* records, of same type or not, can *never* be aliased, while in the encoding, we can easily store the same reference of the right type in many different records.

A possible solution is to change the point of view and encode references using records instead of the other way around. The type declaration

$$\text{type } \text{ref } [\alpha] = \{ \text{mutable } \text{contents} : \alpha \}$$

is the definition in OCaml of the reference type. A slight generalization of our system could deal with mutable records directly, instead of references, and associate regions to mutable record fields. All regions that are associated to the field *contents* correspond to references, while other regions may correspond to mutable fields of other records. In this model, which corresponds to the Burstall-Bornat memory model ([Burstall, 1972](#); [Bornat, 2000](#)), it is impossible for two different mutable fields (of the same or of different records) to exist in the same region.

Modules. Another central feature of existing ML programming languages is the module system. Modules are simply groups of type definitions, values and functions. Additionally, modules can be parameterized w.r.t. other modules; such parameterized modules are called *functors*. Functors can be instantiated with concrete modules, and such instantiations, just as function applications, can have side effects.

The fact that our proof system is very modular, thanks to higher-order logic and effect polymorphism, implies that in principle, there should be no major obstacle to the support of modules in *Who*. After all, a module is barely more than a record of values and functions, and functor application is very similar to function application. So, from a purely technical point of view, modules could be supported.

6. Conclusion and Outlook

However, the notion of module also comes with the notion of encapsulation. Often, types in modules are rendered *abstract*, *i.e.*, outside of the module, the definition of the type is unknown. One would like to hide effects that are only needed for the internal implementation of the module, but are not observable outside of it. Often, one would like to attach an *invariant* to an abstract type, stating that all values of that type, seen from the outside of the module, verify the invariant — for example, that a list is sorted. Finally, one would like to present the specifications of the module’s functions using a *model*. For example, if a module M implements a type t representing sets of integers, a value *empty* that represents the empty set and functions *add* and *remove* with the obvious meaning, one would like to say that *add* adds an element to a set, without saying how this is exactly achieved. In the implementation, this can correspond to adding a node to a tree or flipping a bit in some array, but these details are irrelevant outside of the module.

So, in practice, the addition of a module system to *Who* would require mechanisms that allow to

- hide effects,
- define a model that corresponds to a type,
- attach invariants to types.

We have seen in Section 5.4.2 that effect hiding is not possible; recent work by Schwinghammer et al. (2010) presents theoretical systems in which effect hiding is possible.

The object system of OCaml. OCaml contains another language feature, namely a sophisticated object system (Rémy and Vouillon, 1997). A complete treatment of this feature would certainly require other features of the logic and/or the type system. There is a large body of literature concerning object-oriented programming and formal verification. It would be interesting to see whether it can be applied to the object system of OCaml. A paper where the notion of type invariant is applied to verification of object-oriented programs is the one of Barnett et al. (2004a).

Surface syntax and annotation inference. Another obstacle is of more practical nature. The *Who* tool uses its own input syntax for the W language, even though it is relatively similar to the OCaml syntax. The differences between W and the subset of OCaml that corresponds to the features in W essentially concern effect annotations and specifications, both of which do not exist in OCaml. A first step towards proof of OCaml programs would be to put effect annotations and specifications in special comments, as is the case for the JML and Krakatoa specification languages. This would permit to treat the file with the proof tool, but also with a regular OCaml compiler. However, to be as convenient as possible for the programmer, one should strive to *infer* as much information as possible, just as type inference does for types. Effect and region inference, just as type inference, is possible and is implemented in *Who*. However, region and effect annotations can change the way a program must be annotated, and can change proofs. Currently, effect generalization and instantiation is therefore explicit in *Who*,

while type and region annotations are unnecessary. An adaptation of the work by [Hubert and Marché \(2007\)](#), that finds the “best” region instantiation for a given program, to effect polymorphism would be helpful. Finally, there are many techniques capable of discovering program specifications such as loop invariants automatically. There are incomplete because the problem is undecidable, but in practice they can be of great help.

Other Extensions and Ideas for Future Work

It would be challenging to specify and prove correct larger programs involving higher-order functions and effects. One such example is the fixed point computation detailed in [Section 5.4.6](#). Other interesting applications are iterator functions and folds over more complex data structures — hash tables, trees, trees with invariants — than the ones we have presented (arrays and lists).

The type and effect system of W can be improved. We have already briefly discussed that extending it to deal with exceptions and input/output would be useful. But also the part of effects dealing with references could be improved. At the end of [Section 4.2](#), we already sketched a system in which singleton regions and group regions can be used simultaneously, but do not interact. The realization of such a system, its improvement it to be able to mix singleton and group regions freely, is still future work. Currently, *Who* implements the system of [Section 4.1](#), and the restriction [Section 4.2](#) can be switched on only for the entire program.

A. Résumé en Français

A.1. Introduction

Depuis des décennies maintenant, l'informatique et le logiciel s'installent partout dans notre vie. C'est de plus en plus le cas aussi pour les systèmes dits *critiques*, des systèmes dont la défaillance peut mettre en péril une grande somme d'argent, la vie d'un humain, ou les deux. Des exemples pour de tels systèmes critiques sont les logiciels de contrôle des satellites et avions, des centrales nucléaires, et encore les logiciels sur les marchés financiers.

Il devient donc impératif de s'assurer du bon fonctionnement de ces programmes. Pour cela, il faut d'abord fixer dans quelles situations le programme doit opérer, et quel est son comportement attendu. Le *test* est une manière simple de vérifier si un programme correspond à ces attentes. Il s'agit de fixer un certain nombre d'*entrées* du programme et de déterminer, par exemple manuellement, le résultat attendu. Le test va simplement exécuter le programme avec chacune de ces entrées et vérifier si le résultat du programme correspond au résultat attendu.

Le test, sous une de ses nombreuses formes, est certainement la méthode la plus adaptée dans l'industrie pour s'assurer du bon fonctionnement d'un programme. Cependant, le test ne peut garantir la correction d'un programme pour un nombre fini d'entrées. Or, il y a généralement une infinité de situations possibles dans lesquelles peut se trouver un programme, même dans des cas très simples. Ou, pour le dire avec Dijkstra, le test ne peut montrer que la *présence* d'erreurs, mais pas leur absence. Il est également difficile de prévoir toutes les situations les plus critiques qui peuvent apparaître en pratique.

A.1.1. La logique de Hoare

La logique de Hoare (Hoare, 1969) est une approche qui fait parti des *méthodes formelles* (Monin, 2002), qui visent de *prouver* avec des méthodes mathématiques qu'un programme est correct vis-à-vis de sa spécification, c'est-à-dire qu'il se comporte comme attendu dans toutes les situations dans lesquelles le programme est censé fonctionner. Cette approche est supérieure au test dans le sens qu'elle peut fournir de garanties bien plus importantes. Cependant, elle exige un travail très important du programmeur ou d'un expert ; elle est donc plus coûteuse que le test.

La logique de Hoare établit ce qu'on appelle un *triplet de Hoare* : étant donné un programme C et des formules logiques P et Q , on dit que le triplet de Hoare

$$\{ P \} C \{ Q \}$$

est valide si, quand la formule P est vraie initialement, la formule Q est toujours vraie après l'exécution du programme C . Dit autrement, si C est le programme à vérifier,

alors P exprime les conditions dans lesquelles le programme est censé fonctionner, et Q exprime les propriétés des résultats de l'exécution du programme.

A.1.2. L'ordre supérieur

Dans les langages dits de premier ordre, seuls des objets simples comme les entiers, les booléens ou encore les chaînes de caractères sont considérés comme des *valeurs*, c'est-à-dire ils peuvent être passés à une fonction, être retournés par une fonction, ou encore être stockés dans une structure de données. On dit aussi que ce sont des objets de *première classe*; toute opération du langage peut s'appliquer à ces objets. En particulier, dans un langage de premier ordre, les fonctions ne sont pas de première classe; souvent, on peut seulement les définir à des endroits bien précis d'un programme, et les appeler ensuite.

Dans un langage *d'ordre supérieur*, les fonctions deviennent des objets de première classe, on peut donc les manipuler comme n'importe quelle valeur. En particulier, une fonction peut maintenant prendre une autre fonction en argument. Une telle fonction est appelée une fonction d'ordre supérieur.

A.1.3. Le langage ML

Parmi les langages de programmation existants, le langage ML occupe une position privilégiée. Ceci est dû d'abord à sa sûreté. En effet, un système de types puissant garantit qu'un programme ML bien-typé ne peut s'arrêter à cause d'un accès invalide à la mémoire ou encore des erreurs de typage. Néanmoins, ce système de types est suffisamment riche pour permettre la plupart des applications qui sont possibles dans d'autres langages. En particulier, le polymorphisme de types, les fonctions d'ordre supérieur, et enfin les effets de bords grâce aux *références* sont possibles en ML. Enfin, ML est très simple dans sa définition, ce qui le rend attractif pour une étude théorique de ses propriétés.

A.1.4. Les systèmes à effets

Dans certains cas, il est utile de connaître *l'effet* d'une partie d'un programme c'est-à-dire l'ensemble des emplacements de mémoire que celui-ci va modifier. Cette information peut servir à un compilateur, qui peut ou non appliquer certaines optimisations au code généré. Elle peut aussi servir à simplifier le raisonnement nécessaire dans la logique de Hoare. Dans le langage initialement présenté par Hoare (1969), le calcul des effets d'une instruction est trivial. Dans des langages plus complexes, en particulier en présence de fonctions, il devient nécessaire d'intégrer la notion d'effet dans le système de types. On parle alors de systèmes de types et effets. Enfin, si le langage de programmation permet de l'ordre supérieur, un tel système nécessite une notion de *polymorphisme d'effet*. Le système de types et effets dont nous nous inspirons a été publié par Talpin and Jouvelot (1994).

A.1.5. Cette thèse

Le sujet de cette thèse est la preuve de programmes d'ordre supérieur en présence d'effets de bord. Nous avons choisi la méthode de la logique de Hoare comme cadre. Quant au langage de programmation à retenir, ML semble représenter le choix idéal, d'une part parce que il est proche d'être le plus petit langage contenant à la fois de l'ordre supérieur et des effets de bord, d'autre part parce que son typage fort garantit déjà une partie des propriétés d'un programme. D'autres langages de programmation, comme le C, nécessitent la preuve de propriétés liées à la validité des pointeurs, par exemple.

Il reste à choisir un mécanisme permettant de tracer les effets de bord et l'état modifié par le programme. Nous avons choisi un système de types et effets avec polymorphisme d'effets. D'abord, cela semble être une extension naturelle des systèmes existants, basés sur la logique de Hoare et une analyse d'effets, qui ne traitent que le premier ordre. Ensuite, cette technique permet de raisonner de manière modulaire sur les programmes. Enfin, la combinaison d'un système à effets comme celui que nous utilisons avec un système de logique de Hoare n'a jamais été réalisés dans le cadre des programmes d'ordre supérieur.

A.2. Le langage de programmation W et la logique L

Comme langage de programmation qui sert de base de discussion pour tout le document, nous introduisons le langage W, avec un système de règles de typage et une sémantique à petits pas. Le langage est très proche de ML, mais présente néanmoins quelques particularités liées au fait que W contient un système à effets similaire à celui de Talpin and Jouvelot (1994). Tout d'abord, la notion d'*effet* est centrale. Un effet est un ensemble de *régions*, une région étant une notion statique correspondant à une ou plusieurs adresses de mémoire, et décrit ce que peut modifier une expression. Les types des fonctions sont également annotés par un effet : une fonction qui accepte un entier et retourne un entier, tout en modifiant une cellule de mémoire dans une région ρ a le type $\text{int} \rightarrow^\rho \text{int}$. Afin de pouvoir calculer l'effet de chaque expression, les références (cellules de mémoire) doivent être annotées avec la région à laquelle elles appartiennent. Un effet peut aussi contenir des variables d'effet ε et des variables de régions ρ .

Nous définissons un système de types et effets avec deux jugements de typage. Le jugement :

$$\Gamma; \Sigma \vdash_v v : \tau$$

décrit qu'une valeur v a le type τ sous l'environnement de typage Γ et le *store typing* Σ . Le store typing sert à donner un type et une région aux adresses de mémoire créées pendant l'exécution d'un programme. Le jugement :

$$\Gamma; \Sigma \vdash e : \tau, \varphi$$

décrit qu'une expression e a le type τ et produit l'effet φ sous l'environnement de typage Γ et le store typing Σ . Nous prouvons la *correction* du typage : un programme bien typé va boucler ou réduire vers une valeur de même type. L'effet de l'expression peut également être maintenu lors de la réduction d'une expression.

Le langage W permet trois types de polymorphisme : le polymorphisme de types, comme en ML , mais aussi le polymorphisme d'effets et de régions. Dans notre présentation de W , nous avons choisi de rendre ce polymorphisme explicite, c'est-à-dire l'utilisateur doit indiquer les variables de types, effets et régions qui doivent être généralisées, et aussi indiquer l'instanciation des variables, lors de l'utilisation d'un symbole polymorphe.

Nous définissons également un langage de spécifications pour les programmes écrits en W . Ce langage est très proche de la logique d'ordre supérieur standard, avec essentiellement deux différences à savoir les *types état* paramétrés par un effet, notés $\langle\varphi\rangle$, et le polymorphisme de régions et effets comme dans les programmes. Un objet de type $\langle\varphi\rangle$ représente un état qui contient toutes les régions contenues dans l'expression d'effet φ . On peut accéder à ces états à l'aide de l'opération `get`, les modifier avec la fonction `set`, les combiner avec la fonction `combine`, et restreindre leur domaine (l'effet φ) avec la fonction `restrict`.

Une idée centrale, inspirée par le système Pangolin (Régis-Gianas and Pottier, 2008), est de plonger les types et les valeurs des programmes vers des types et valeurs dans la logique. Comme W est un langage d'ordre supérieur, les fonctions, potentiellement avec effets de bord, sont aussi des valeurs. Nous traduisons un type $\tau \rightarrow^\varphi \tau'$ par une paire de prédicats de type

$$\tau \rightarrow \langle\varphi\rangle \rightarrow \text{prop} \times \tau \rightarrow \langle\varphi\rangle \rightarrow \langle\varphi\rangle \rightarrow \tau' \rightarrow \text{prop},$$

où la première composante représente la *précondition* d'une fonction, c'est-à-dire un prédicat concernant l'argument de la fonction et l'état initial. La deuxième composante est la *postcondition*, rélatant l'argument, les états initial et final et la valeur renvoyée par la fonction. Les états ne contiennent que ce qui est relevant à l'exécution de la fonction, donc ce qui apparaît dans son effet.

Les valeurs peuvent également être plongées dans la logique. Une fonction en W , annotée avec une précondition p et une postcondition q , est traduite par le couple (p, q) .

Les types de références peuvent être plongés dans la logique sans modification, mais bien sûr les fonctions de lecture et écriture par effet de bord sont devenues inaccessibles ; leur traduction dans la logique est une paire, qui ne peut pas être appliquée. Elles sont remplacées par les fonctions correspondantes qui manipulent des états.

Grâce à ce plongement des valeurs et types des programmes, l'utilisateur peut exprimer dans la logique des propriétés de n'importe quelle valeur d'un programme.

A.3. Le calcul de la plus faible précondition

Jusqu'ici nous avons présenté le langage de programmation W et le langage de spécifications L , mais aucune vérification est faite si les annotations sont correctes et ont du sens. Cette vérification prend la forme d'un *calcul de la plus faible précondition*. Ce calcul prend une formule q et une expression e et calcule une formule p qui garantit que si p est vraie initialement, q est vraie après l'exécution de l'expression e . Nous écrivons

$$\text{wp}_s(e, q)$$

pour la plus faible précondition de e et q , qui peut mentionner l'état s . L'utilisateur doit maintenant prouver la formule p pour obtenir la correction de son programme vis-à-vis de sa spécification.

Parallèlement à la notion de plus faible précondition, nous introduisons aussi la notion de *correction* d'une valeur, qui exprime qu'une valeur ne contient que des spécifications correctes. Cette notion est définie pour toutes les valeurs, cependant elle n'est non triviale que pour les fonctions. Une fonction de pré- et postcondition p et q et de corps e est correcte si, pour un état s , la précondition pour s implique la plus faible précondition pour s , e et q :

$$\forall s. p \ s \Rightarrow \text{wp}_s(e, q \ s)$$

L'état initial de la postcondition est également l'état s .

Le calcul de plus faible précondition suit largement les définitions classiques, mais tire avantage des informations d'effets des expressions. Par exemple, pour justifier qu'une application d'une fonction f à un argument x garantit une condition q , il faut prouver la précondition de f , appliquée à x et l'état courant, et il faut prouver que la postcondition de f implique la condition q dans l'état qui est établi par la fonction f . Cette définition classique est raffinée par le fait que l'état initial et final ne peuvent différer qu'en les régions qui sont mentionnées dans l'effet de la fonction f .

Nous prouvons la correction du calcul de plus faible précondition, c'est-à-dire que la formule obtenue par ce calcul implique en effet la correction du programme. Plus précisément, nous prouvons que si $\text{wp}_s(e, q)$ est valide, alors à partir de l'état s , e boucle ou se réduit vers un état s' et une valeur v qui vérifient la postcondition q . Nous acceptons aussi la non-terminaison du programme, nous nous intéressons donc seulement à la correction *partielle* du calcul de plus faible précondition.

Nous prouvons également la complétude de ce calcul, c'est-à-dire la propriété qu'étant donné un programme e et une condition q , si e se réduit toujours vers une valeur qui vérifie q , alors on peut modifier les annotations contenues dans e , pour obtenir une expression e' , identique d'un point de vue calculatoire, pour laquelle nous pouvons prouver la plus faible précondition, exprimée par $\text{wp}_s(e, q)$.

A.4. Les restrictions d'alias

Le système présenté dans les deux sections précédentes est simple d'un point de vue théorique, et possède de propriétés souhaitables tel que la correction et la complétude. Néanmoins, il n'est pas très agréable d'utilisation en pratique. Prenons comme exemple une fonction f avec le type suivant :

$$f : \forall \varrho_1, \varrho_2. \text{ref}_{\varrho_1} \text{ int} \rightarrow^{\emptyset} \text{ref}_{\varrho_2} \text{ int} \rightarrow^{\varrho_1 \varrho_2} \text{ int}$$

Si on veut raisonner sur une telle fonction, il faut considérer trois cas :

- les deux régions ϱ_1 et ϱ_2 sont distinctes, et par conséquent, les références le sont aussi ;
- les deux régions ϱ_1 et ϱ_2 sont en fait les mêmes (sont *aliasés*), mais les deux références sont distinctes

A. Résumé en Français

- les deux références sont identiques, et les régions ϱ_1 et ϱ_2 sont également identiques.

Nous proposons ici deux simplifications du système initial qui restreignent l'ensemble de programmes acceptés, mais réduisent le nombre de cas à considérer.

A.4.1. L'exclusion d'aliasing de régions

Une bonne intuition, adaptée dans un cadre sans polymorphisme d'effets par [Hubert and Marché \(2007\)](#), est de dire que les variables de région et variables d'effets sont toujours indépendants. En pratique, cela veut dire que

- deux variables de régions différentes représentent des régions différentes ;
- deux variables d'effets représentent deux effets disjoints ;
- l'effet représenté par une variable d'effet ne peut contenir une région représentée par une variable de région.

Prenons l'exemple d'une fonction qui a un schéma de type comme celui-ci :

$$\forall \varrho_1 \varrho_2 \varepsilon. \dots$$

alors les variables de région ϱ_1 et ϱ_2 ne peuvent pas être instanciées par la même région, et la variable d'effet ε ne peut pas être instanciée avec un effet qui contient les régions qui ont servi à instancier ϱ_1 et ϱ_2 . Pour obtenir ce comportement, seules deux règles de typage doivent être modifiées légèrement.

D'un point de vue théorique, cette restriction permet la formulation de la *règle de frame*, qui dit qu'un programme ne peut toucher que les régions qui sont mentionnées dans son effet, et tout ce qui n'est pas mentionné reste inchangé. Ce théorème était vrai sans la restriction, mais l'aliasing entre régions rendait délicat la décision si une région était mentionnée dans un effet ou pas. La condition $\varrho \notin \varphi$ ne suffisait pas, parce que la variable de région ϱ pouvait être remplacée plus tard par une région contenue dans φ , ou une éventuelle variable d'effet dans φ pouvait être remplacée par un effet qui contient ϱ . Avec la restriction d'aliasing de régions, cela n'est plus possible, et cette décision d'inclusion peut être prise statiquement. La condition $\varrho \notin \varphi$ est suffisante pour conclure que si φ est l'effet d'une expression, alors la région ϱ ne sera pas touchée par cette expression. En pratique, ce résultat théorique permet de nombreuses simplifications des spécifications, les obligations de preuves et les preuves elles-mêmes.

Cette restriction réduit le nombre de programmes acceptés, mais ne réduit pas l'expressivité du langage, dans le sens que si un programme pouvait être typé dans le système d'origine, une modification des annotations de régions et d'effets peut être trouvée telle que le programme ainsi obtenu peut être typé dans le système avec restriction.

Pour résumer, et revenir à notre fonction f du début de la section, cette restriction propose de distinguer *statiquement* le cas où les deux régions sont différentes des deux autres cas. Si f a le type

$$\forall \varrho_1, \varrho_2. \text{ref}_{\varrho_1} \text{ int} \rightarrow^{\emptyset} \text{ref}_{\varrho_2} \text{ int} \rightarrow^{\varrho_1 \varrho_2} \text{ int},$$

alors les deux régions sont différentes et les deux références le sont forcément aussi. Si le programmer donne le type

$$\forall \rho. \text{ref}_{\rho} \text{ int} \rightarrow^{\emptyset} \text{ref}_{\rho} \text{ int} \rightarrow^{\rho} \text{ int}$$

à f , alors les références sont dans la même région et donc potentiellement égales. Il est à l'utilisateur de différencier les deux situations à l'aide d'annotations.

A.4.2. Les régions singletons

Une deuxième restriction, qui se rajoute à la première, est la restriction des régions à des *régions singleton*. Une région singleton est une région qui contient au plus une seule référence. La distinction entre région et référence disparaît alors d'un point de vue logique.

D'un point de vue technique, un simple changement de la notion d'effet permet d'obtenir cette restriction. Nous introduisons la notion d'effet de *création*, c'est-à-dire la création d'une référence dans une région. L'objectif est de ne permettre qu'un seul effet de création par région, ainsi chaque région ne peut contenir une seule référence. Cette restriction est obtenue en interdisant l'union d'effets dont les effets de création ne sont pas disjoints.

Cette restriction rajoute encore une simplification ; on peut maintenant décider statiquement si deux références sont identiques, à savoir quand elles sont dans la même région. Un certain nombre d'obligations de preuves deviennent encore plus simple dans ce cadre.

Contrairement à la première restriction, celle-ci restreint fortement l'expressivité du langage. Un certain nombre de programmes ne peuvent plus être traités par ce système simplifié. Cela concerne tous les programmes qui nécessitent un traitement dynamique des références. Ceci inclut les références qui contiennent des références, des références dans les listes, les expressions qui renvoient une référence ou une autre selon la situation. Tous ces programmes peuvent être écrits, mais leur comportement devient alors trivial. Une référence sur une autre référence doit forcément contenir toujours la même, à tout point de programme ; une liste de références ne peut contenir deux références distinctes, et l'expression renvoie toujours la même référence.

Pour reprendre l'exemple de notre fonction f , si le programmeur donne le type suivant pour f :

$$\forall \rho_1, \rho_2. \text{ref}_{\rho_1} \text{ int} \rightarrow^{\emptyset} \text{ref}_{\rho_2} \text{ int} \rightarrow^{\rho_1 \rho_2} \text{ int}$$

alors, comme avant, les deux références sont forcément distinctes. Si le programmeur donne le type

$$\forall \rho. \text{ref}_{\rho} \text{ int} \rightarrow^{\emptyset} \text{ref}_{\rho} \text{ int} \rightarrow^{\rho} \text{ int}$$

à f , alors les deux références sont *forcément identiques* ; en conséquence, la fonction f n'a plus besoin d'avoir deux arguments de type référence.

Il s'agit alors d'un choix entre expressivité du système et simplicité des obligations de preuve. Il est imaginable de faire cohabiter les deux restrictions dans le même système, permettant la création de plusieurs références dans la même région, si cela est nécessaire, mais cette combinaison n'a pas été étudiée dans cette thèse.

A.5. L’outil et des exemples

Le système théorique et ses extensions ont été implantés dans un outil nommé *Who*, écrit en OCaml. L’outil prend en entrée un programme dans le langage *W*, avec des annotations en langage *L*. À l’aide du calcul de la plus faible précondition, il engendre, à partir de ce programme annoté, une formule en *L*, qui contient toutes les obligations de preuve nécessaires. Ensuite, cette formule est traduite vers une logique d’ordre supérieur standard, qui est acceptée par un certain nombre de prouveurs interactifs. L’utilisateur peut maintenant prouver cette formule pour prouver la correction du programme de départ vis-à-vis de sa spécification.

A.5.1. Traduction vers la logique du premier ordre

Il est souhaitable de pouvoir se servir des nombreux prouveurs automatiques afin de décharger les obligations de preuve plus facilement. Or, la grande majorité des prouveurs automatiques supportent seulement une logique du premier ordre. En collaboration avec Yann Régis-Gianas, nous avons donc développé une traduction de la logique d’ordre supérieur vers la logique du premier ordre. La cible de cette traduction est l’outil *Why* (Filliâtre and Marché, 2007). Cet outil accepte des formules en logique du premier ordre dans une syntaxe particulière, et peut traduire ces formules dans les différentes syntaxes de nombreux prouveurs automatiques. Grâce à la traduction vers le premier ordre, et grâce à l’outil *Why*, nous pouvons utiliser les prouveurs du premier ordre pour décharger les obligations de preuve générées par *Who*. Le principe de la traduction présentée n’est pas nouveau, mais nous y apportons quelques modifications qui réduisent cette traduction à l’identité si la formule de départ est déjà une formule du premier ordre.

A.5.2. Études de cas

Pour démontrer la capacité de *Who* à prouver la correction de programmes d’ordre supérieur avec effets, nous avons écrits et spécifié quelques programmes en *W*, et nous les avons prouvés corrects :

- une implantation de la boucle *for*, utilisant une fonction d’ordre supérieur ;
- le nœud de Ladin, qui consiste à réaliser une fonction récursive, sans utiliser le mécanisme de récursion fourni par le langage ;
- deux fonctions de mémoïsation, qui utilisent un effet de bord sur une table pour garder en mémoire le résultat d’une fonction pure et éviter de l’appeler de nouveau avec le même argument ;
- deux fonctions d’ordre supérieur, *iter* et *map*, concernant les tableaux, et la fonction *map* pour les listes ;
- L’algorithme de Koda et Ruskey (Koda and Ruskey, 1993), qui énumère certaines coloration d’une forêt.

Dans tous ces cas, nous présentons une implantation en ML, sans annotations, suivi d'une implantation en W, avec annotations. La majorité des programmes peuvent être prouvés corrects en utilisant uniquement les prouveurs automatiques. Les autres programmes, comme l'algorithme de Koda et Ruskey, nécessitent des preuves manuelles, qui ont été effectuées dans l'assistant de preuve Coq.

A.6. Conclusion

Les deux contributions centrales de cette thèse sont :

- un système théorique permettant de spécifier des programmes impératifs d'ordre supérieur et de générer des obligations de preuve qui impliquent la correction du programme par rapport à sa spécification ;
- une *implantation* de ce système, nommée *Who*, qui permet, en pratique, de prouver ces obligations générées à l'aide de prouveurs existants, automatiques et interactifs.

Le système théorique inclut :

- un langage de programmation que nous appelons W, similaire à ML, avec un système de types et effets qui ressemble celui de [Talpin and Jouvelot \(1994\)](#) ;
- un nouveau langage de spécifications que nous appelons L, qui étend la logique d'ordre supérieur standard,
- un nouveau calcul de la plus faible précondition, qui combine les capacités du système de Why ([Filliâtre, 2003](#)) et du système de Pangolin ([Régis-Gianas and Pottier, 2008](#)) ;
- des preuves de correction et complétude de ce calcul ;
- deux modifications du système initial, à savoir l'exclusion d'aliasing entre régions, et la restriction à des régions singletons. Les deux modifications généralisent des idées par [Hubert and Marché \(2007\)](#) et [Filliâtre \(2003\)](#), respectivement.

Ce système permet la spécification modulaire de fonctions d'ordre supérieur en présence d'effets de bord. Cette modularité est obtenue grâce à l'utilisation d'une logique d'ordre supérieur en tant que langage de spécification, mais aussi grâce au polymorphisme d'effets présent en W, qui permet de s'abstraire de l'effet d'un argument fonctionnel.

L'outil *Who* contient :

- une implantation de tout ce qui a été présenté dans la partie théorique ;
- un encodage des parties non-standards de L vers la logique d'ordre supérieur standard, et un moyen d'exporter les obligations de preuve vers Coq ([The Coq Development Team, 2008](#)) ;

A. Résumé en Français

- un encodage de la logique d'ordre supérieur vers le premier ordre, plus précisément la logique de l'outil Why (Filliâtre and Marché, 2007). Cet encodage permet de décharger les obligations de preuve à l'aide des nombreux prouveurs automatiques et interactifs dont Why connaît le format d'entrée.

À l'aide de l'outil Who, nous avons prouvé un certain nombre de programmes qui mélangent effets de bord et ordre supérieur pour obtenir de résultats intéressants :

- Des fonctions d'ordre supérieur assez standard sur les listes et tableaux, tel que *iter* et *map*, appliquées à des fonctions qui produisent des effets de bord ;
- Des fonctions d'ordre trois, tel que *ymemo* et *backpatch* (le nœud de Landin), réputés pour être difficile à spécifier ;
- L'algorithme de Koda and Ruskey (1993), dans une variante à base de continuations (Filliâtre and Pottier, 2003; Kanig and Filliâtre, 2009).

Les itérateurs d'ordre supérieur, en combinaison avec des effets de bord, ont déjà été discuté dans la littérature, par exemple dans le système Ynot (Nanevski et al., 2008), de manière moins générale par Honda et al. (2005), et plus récemment par Borgström et al. (2010). Seul le système Ynot, et le notre, peuvent présenter une implantation prouvée de tels itérateurs. Honda et al. (2005) discutent le nœud de Landin, mais seulement dans un cas concret d'application, et non pas dans sa forme générale de point fixe. À notre connaissance, nous avons été les premiers à soumettre l'algorithme de Koda et Ruskey à une vérification formelle.

Bibliography

- Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0262011530.
- Jean-Raymond Abrial. *The B-Book, Assigning Programs to Meaning*. Cambridge University Press, 1996.
- Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*. Academic Press, 1986.
- Krzysztof R. Apt. Ten Years of Hoare's Logic : A Survey. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.
- Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering Formal Metatheory. *SIGPLAN Not.*, 43(1):3–15, 2008.
- J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language algol 60. *Commun. ACM*, 3(5):299–314, 1960.
- H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North Holland, Amsterdam, 2nd ed., 1984. ISBN 0-444-87508-5.
- Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology*, 3:27–56, 2004a.
- Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69, 2004b.
- Sylvain Baro. Introduction to Paf!, a Proof Assistant for ML Programs Verification. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2003.
- Sylvain Baro and Pierre Manoury. Un Système à Reasonner Formellement sur les Programmes ML. In Jean-Christophe Filliâtre, editor, *JFLA*, Collection Didactique, pages 49–62. INRIA, 2003.
- Clark Barrett and Cesare Tinelli. CVC3. In [Damm and Hermanns \(2007\)](#), pages 298–302.
- Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN 0321278658.

Bibliography

- Michael Beeson. Mathematical Induction in Otter-Lambda. *J. Autom. Reason.*, 36(4): 311–344, 2006.
- Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. METEOR : A Successful Application of B in a Large Project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of FM'99: World Congress on Formal Methods*, Lecture Notes in Computer Science (Springer-Verlag), pages 369–387. Springer Verlag, September 1999.
- Martin Berger, Kohei Honda, and Nobuko Yoshida. A Logical Analysis of Aliasing in Imperative Higher-order Functions. *J. Funct. Program.*, 17(4-5):473–546, 2007.
- Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astrée Static Analyzer, 2003. URL <http://www.astree.ens.fr/>.
- Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.
- Johannes Borgström, Juan Chen, and Nikhil Swamy. Verified Programming with an Affinity for Hoare Types. Technical Report MSR-TR-2010-95, Microsoft Research, July 2010.
- Richard Bornat. Proving Pointer Programs in Hoare Logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
- Rod Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. *Machine Intelligence*, 7:23–50, 1972.
- Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic Type Soundness Results for the Region Calculus. *Inf. Comput.*, 173(2):199–221, 2002.
- Robert Cartwright and Derek Oppen. The Logic of Aliasing. *Acta Informatica*, 15(4): 365–384, August 1981.
- Arthur Charguéraud. Program Verification Through Characteristic Formulae. In *ACM SIGPLAN International Conference on Functional Programming*, September 2010.
- Arthur Charguéraud and François Pottier. Functional Translation of a Calculus of Capabilities. In *Hook and Thiemann (2008)*, pages 213–224.
- Adam Chlipala. A Verified Compiler for an Impure Functional Language. In *Hermenegildo and Palsberg (2010)*, pages 93–106.
- Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Proofs for Higher-order Imperative Programs. In Andrew Tolmach, editor, *14th International Conference on Functional Programming, Edinburgh, Scotland, Proceedings*. ACM Press, September 2009.

- Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.
- Edmund Melson Clarke, Jr. Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Axiom Systems. *J. ACM*, 26(1):129–147, 1979.
- Sylvain Conchon and Évelyne Contejean. The Alt-Ergo Automatic Theorem Prover, 2008. URL <http://alt-ergo.lri.fr/>.
- Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. CiME3, 2007. URL <http://cime.lri.fr>. <http://cime.lri.fr>.
- Catarina Coquand and Thierry Coquand. *Structured Type Theory*. 1999.
- Jean-François Couchot and Stéphane Lescuyer. Handling Polymorphism in Automated Deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.
- Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.
- Pierre Crégut. Une Procédure de Décision Réflexive pour l'Arithmétique de Presburger en Coq. Deliverable, Projet RNRT Calife, 2001.
- Haskell B. Curry. *Combinatory logic / [by] Haskell B. Curry [and] Robert Feys. With two sections by William Craig*. North-Holland Pub. Co., Amsterdam, 1958.
- Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985. Technical report CST-33-85.
- Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.
- Werner Damm and Holger Hermanns, editors. *Computer Aided Verification*, volume 4590 of *LNCS*, Berlin, Germany, July 2007. Springer Verlag.
- Werner Damm and Bernhard Josko. A Sound and Relatively Complete Hoare-logic for a Language with Higher Type Procedures. *Acta Informatica*, 20(1):59–101, October 1983.
- Nikolas. G. de Bruijn. Lambda Calculus with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Proc. of the Koninklijke Nederlands Akademie*, 75(5):380–392, 1972.
- Leonardo de Moura and Nikolaj Bjørner. Z3, an Efficient SMT Solver, 2009. <http://research.microsoft.com/projects/z3/>.
- Leonardo de Moura and Bruno Dutertre. Yices: An SMT Solver, 2009. <http://yices.csl.sri.com/>.

Bibliography

- David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a Theorem Prover for Program Checking. *J. ACM*, 52(3):365–473, 2005.
- Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.
- Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. *SIGPLAN Not.*, 37(5):13–24, 2002.
- Jean-Christophe Filliâtre. Verification of Non-functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.
- Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Damm and Hermanns (2007)*, pages 173–177.
- Jean-Christophe Filliâtre and François Pottier. Producing All Ideals of a Forest, Functionally. *Journal of Functional Programming*, 13(5):945–956, September 2003.
- Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. *SIGPLAN Not.*, 28(6):237–247, 1993.
- Robert W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.
- Jacques Garrigue. Programming with Polymorphic Variants. In *ACM SIGPLAN Workshop on ML*, Baltimore, Maryland, USA, 1998.
- J.-Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, Univ. Paris VII, France, 1972.
- Mike Gordon. From LCF to HOL: A Short History. *Proof, language, and interaction: essays in honour of Robin Milner*, pages 169–185, 2000.
- Robert Harper. A Simplified Account of Polymorphic References. *Information Processing Letters*, 51(4):201 – 206, 1994.
- Leon Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15:81 – 91, 1950.
- Manuel V. Hermenegildo and Jens Palsberg, editors. *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, 2010. ACM.
- C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.
- C. A. R. Hoare. An Axiomatic Definition of the Programming Language Pascal. In *Proceedings of the International Symposium on Theoretical Programming*, pages 1–16, London, UK, 1974. Springer-Verlag. ISBN 3-540-06720-5.

- Kohei Honda, Nobuko Yoshida, and Martin Berger. An Observationally Complete Program Logic for Imperative Higher-order Functions. In *In Proc. LICS'05*, pages 270–279, 2005.
- James Hook and Peter Thiemann, editors. *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, 2008. ACM.
- William A. Howard. The Formulas-as-Types Notion of Construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article.
- Thierry Hubert and Claude Marché. Separation Analysis for Deductive Verification. In *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, March 2007.
- Joe Hurd. First-order Proof Tactics in Higher-order Logic Theorem Provers. In *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.
- Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4.
- Cem Kaner, Hung Q. Nguyen, and Jack L. Falk. *Testing Computer Software*. John Wiley & Sons, Inc., New York, NY, USA, 1993. ISBN 0442013612.
- Johannes Kanig. Who - A Verification Condition Generator for Imperative Higher-order Programs, 2010. URL <http://www.lri.fr/~kanig/who.html>.
- Johannes Kanig and Jean-Christophe Filliâtre. Who: A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, Edinburgh, Scotland, UK, August 2009.
- C. Keller and B. Werner. Importing HOL Light into Coq. In M. Kaufmann and L. Paulson, editors, *LNCS*, volume Proceedings of the Interactive Theorem Proving conference, LNCS 2010, Edinburgh, UK, July 11-14, 2010, 2010.
- B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley Professional, 3 edition, July 1997. ISBN 0201896834.
- Donald E. Knuth. *The Art of Computer Programming*, volume 4, Pre-Fascicle 2a: A Draft of Section 7.2.1.1: Generating all n -tuples. Addison-Wesley, September 2001. Circulated electronically. <http://www-cs-staff.stanford.edu/~knuth/news.html>.

Bibliography

- Yasunori Koda and Frank Ruskey. A Gray Code for the Ideals of a Forest Poset. *Journal of Algorithms*, 15(2):324–340, September 1993.
- Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 83–86, New York, NY, USA, 2006. ACM.
- Peter J. Landin. The Next 700 Programming Languages. *Commun. ACM*, 9(3):157–166, 1966.
- Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual, 2009. URL <http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/>.
- Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- Xavier Leroy and François Pessaux. Type-based Analysis of Uncaught Exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000.
- Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.11, Documentation and user's manual*, November 2008. <http://caml.inria.fr/pub/docs/manual-ocaml/>.
- J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57, New York, NY, USA, 1988. ACM.
- Séverine Maingaud, Vincent Balat, Richard Bubel, Reiner Hähnle, and Alexandre Miquel. Specifying Imperative ML-like Programs using Dynamic Logic. In *International Conference on Formal Verification of Object-Oriented Software*, June 2010.
- Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a Verified Relational Database Management System. In *Hermenegildo and Palsberg (2010)*, pages 237–248.
- Daniel Marino and Todd D. Millstein. A Generic Type-and-effect System. In Andrew Kennedy and Amal Ahmed, editors, *TLDI*, pages 39–50. ACM, 2009.
- Conor McBride and James McKinna. The View from the Left. *J. Funct. Program.*, 14(1):69–111, 2004.
- Jia Meng and Lawrence C. Paulson. Translating Higher-order Clauses to First-order Clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.
- Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, Hemel Hempstead, 1992.
- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000. ISBN 0136291554.
- Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT, May 1997. ISBN 0-262-63181-4.

- Jean-François Monin. *Understanding Formal Methods*. Springer Verlag, 2002. Foreword by G. Huet, ISBN 1-85233-247-6.
- Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN 0471469122.
- Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and Separation in Hoare Type Theory. In John Reppy and Julia Lawall, editors, *11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 62–73, Portland, Oregon, USA, 2006. ACM.
- Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the Awkward Squad. In [Hook and Thiemann \(2008\)](#), pages 229–240.
- Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.
- Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.
- Lawrence C. Paulson and Kong Woei Susanto. Source-level Proof Reconstruction for Interactive Theorem Proving. In *TPHOLs'07: Proceedings of the 20th international conference on Theorem proving in higher order logics*, pages 232–245, Berlin, Heidelberg, 2007. Springer-Verlag.
- Simon Peyton Jones et al. The Haskell 98 Language and Libraries: The Revised Report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. <http://www.haskell.org/definition/>.
- Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.
- Randy Pollack. Closure under Alpha-Conversion. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 313–332, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.
- François Pottier. Lazy least fixed points in ML. Unpublished, December 2009. URL <http://gallium.inria.fr/~fpottier/publis/fpottier-fix.pdf>.
- François Pottier and Nadji Gauthier. Polymorphic Typed Defunctionalization and Concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.
- Yann Régis-Gianas and François Pottier. A Hoare Logic for Call-by-value Functional Programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08)*, pages 305–335, July 2008.
- Didier Rémy and Jérôme Vouillon. Objective ML: a Simple Object-Oriented Extension of ML. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium*

Bibliography

- on Principles of programming languages*, pages 40–53, New York, NY, USA, 1997. ACM.
- John C. Reynolds. Syntactic Control of Interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, New York, 1978. ACM.
- John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.
- John C. Reynolds. Definitional Interpreters for Higher-order Programming Languages. *Higher-Order and Symbolic Computation*, 11:363–397, 1998. ISSN 1388-3690.
- John C. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEECS, 2002.
- Alexandre Riazanov and Andrei Voronkov. The Design and Implementation of Vampire. *AI Commun.*, 15(2-3):91–110, 2002.
- S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.
- Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. A semantic foundation for hidden state. In C.-H. L. Ong, editor, *Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2010)*, volume 6014 of *Lecture Notes in Computer Science*, pages 2–17. Springer, March 2010.
- Frederick Smith, David Walker, and J. Gregory Morrisett. Alias Types. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 366–381, London, UK, 2000. Springer-Verlag. ISBN 3-540-67262-1.
- Matthieu Sozeau. Program-ing Finger Trees in Coq. In Ralf Hinze and Norman Ramsey, editors, *12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pages 13–24, Freiburg, Germany, 2007. ACM.
- Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991. ISBN 0-201-53992-6.
- Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. *Inf. Comput.*, 111(2):245–296, 1994.
- The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. URL <http://coq.inria.fr>.
- Laurent Théry. A Tour of Formal Verification with Coq:Knuth’s Algorithm for Prime Numbers. Technical Report RR-4600, INRIA, 10 2002.
- Mads Tofte. Type Inference for Polymorphic References. *Information and Computation*, 89(1):1–34, 1990.

- Mads Tofte and Jean-Pierre Talpin. Region-based Memory Management. *Inf. Comput.*, 132(2):109–176, 1997.
- David Walker, Karl Crary, and Greg Morrisett. Typed Memory Management via Static Capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, 2000.
- Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass Version 3.5. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.
- J. B. Wells. Typability and Type Checking in System F Are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1998.
- Ryan Wisnesky, Gregory Malecha, and Greg Morrisett. Certified Web Services in Ynot. In *5th International Workshop on Automated Specification and Verification of Web Systems*, July 2009.
- Andrew K. Wright. Simple Imperative Polymorphism. *LISP and symbolic computation*, 8:343–355, 1995.
- Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.

Notations

Metavariables

c	Program or logic constant, page 40
e	Expression in W , page 40
l	Memory location, page 41
R	Region in a store, page 45
r	Region constant, page 39
s	Store, page 45
v	Value in W , page 41
Σ	Store typing, page 51
α, β, γ	Type variable, page 40
δ	Function that defines the semantics of constants in W , page 45
ε	Effect variable, page 39
ι	Type constant, page 40
κ	Instantiation metavariable in W , page 40
ρ	Region metavariable, page 39
τ	Type, page 40
φ	Effect, page 39
χ	Generalization metavariable, page 40
\varkappa	Instantiation Metavariable in L , page 63
ϱ	Region variable, page 39

Substitutions

$[\chi \mapsto \kappa]$	Type, effect, or region substitution in W , page 40
$[\chi \mapsto \varkappa]$	Type, effect, or region substitution in L , page 40
$[x \mapsto \Lambda \bar{\chi}.v]$	Polymorphic substitution of a value for a variable in W , page 46

Reduction relations

\mapsto	Top step reduction relation for W , page 46
\longrightarrow	Single step reduction relation for W , page 47
\twoheadrightarrow	General reduction relation for W , page 47

Typing

$\Gamma; \Sigma \vdash_v v : \tau$	Typing relation for values in W , page 52
$\Gamma; \Sigma \vdash e : \tau, \varphi$	Typing relation for expressions in W , page 52
$\Delta; \Sigma \vdash_l t : \sigma$	Typing relation for terms in L , page 65
$\Sigma \vdash s$	Compatibility of store typing Σ and store s , page 55
$[\bar{\chi} \mapsto \bar{\kappa}] \sim \tau$	The substitution $[\bar{\chi} \mapsto \bar{\kappa}]$ is <i>compatible</i> with type τ , page 109
$\langle \varphi \rangle$	State type, page 63
$[v]$	A value v lifted from W to L , page 72
$[\tau]$	A type τ , lifted from W to L , page 65
$LogicTypeof()$	Function to type constants in L , page 64
$Typeof()$	Function to type constants in W , page 51

Bibliography

Notation in proofs

$s _{\varphi}$	$\text{restrict}_{\varphi} s$, page 76
$s_1 \oplus s_2$	combine $s_1 s_2$, page 76
ϑ_{φ}	$\text{restrict}_{\varphi} \vartheta$, page 73
$NE(s)$	The set of nonempty regions in store s , page 118
$R_s(\omega, \omega')$	Relation between two creation effects, page 119
Syntactic sugar in programs	
cur	Current state, in pre- and postconditions, page 70
old	Initial state, in postconditions, page 70
$\{ p \} e \{ q \}$	Hoare triple in annotations in L , page 126

Index

- Abstract interpretation, 4
- Affine, 33
- Agda, 8
- Algebraic data type, 25, 104
- Aliasing, 16, 107
- Allocation, 6
- Annotation, 69
- Array, 156
 - iter*, 157
 - map*, 159
 - theory, 156
- Arrow type, 22
- Auxiliary Variable, 13
- Axiom, 11

- B method, 6
- Barendregt convention, 12
- Black box testing, 3
- Bound Variable, 12
- Bounded model checking, 4

- Canonical values, 55
- Capability, 33, 123
- Cast, 7
- Characteristic formula, 36
- Compatible substitution, 109
- Completeness, 5, 75, 90
- Cons*, 26
- Constructor, 25
- Contracts, 3
- Copy rule, 19
- Coq, 8, 35, 37
- Correctness obligation, 78
- Coverage, 3
- Curry-Howard isomorphism, 8

- de-Bruijn indices, 12
- Dependent types, 8

- Dynamic types, 7

- Effect, 15, 29, 33, 39
- Effect system, 29
- Effect variable, 39
- Epigram, 8
- Evaluation context, 47
- Expressions, 40

- First-order logic, 133
- Fixed-point combinator, 24, 154
- Formal methods, 4
- Frame rule, 112
- Free variable, 12
- Function type, 22
- Functions as values, 19

- Garbage collector, 7
- Group region, 30, 122

- Haskell, 8
- Higher-order function, 19
- Higher-order logic, 63, 133
- Hoare logic, 5, 10
- Hoare triple, 5, 20

- Inference rule, 5
- Integration testing, 3
- Invariant, 12

- Judgment, 11

- Koda and Ruskey's Algorithm, 163

- L, 62
- Label, 13
- λ -lifting, 138
- Landin's Knot, 44
- Landin's knot, 53, 152

Index

- Latent effect, 29
- letregion, 41, 46
- Lifting, 72
- Linear, 33
- List, 161
- List type, 26
- Locally nameless representation, 12
- Location, 28, 48, 64
- Logic type, 63

- Memoization, 153, 154
- Memory leaks, 6
- Memory management, 6
- Memory model, 17
- Metavariable, 11
- ML, 8
- ML, 20
- Model checking, 4
- Monomorphic, 23
- Multiple argument function, 70

- Nil*, 26
- Nonempty region, 118

- Object orientated programming, 7
- Option type, 25

- Paf, 37
- Partial correctness, 12
- Pattern matching, 25, 104
- Polymorphic substitution, 46
- Polymorphic type system, 23
- Predicate transformer, 13
- Preservation, 23, 54
- Progress, 23, 54, 55
- Proof trace, 146
- Proof tree, 11

- Read-Write effects, 103
- Recursion, 24
- Reduction Context, 21
- ref*, 28
- Reference, 28
- Refinement, 6
- Region, 29, 39, 45
- letregion, 77
- region, 41, 46, 77

- Region aliasing, 108
- Region variable, 39
- Regression testing, 3

- Semantics, 21, 45, 66
- Separating conjunction, 18
- Separation logic, 18
- Simple types, 22
- Singleton region, 30, 108, 115, 122
- Small step semantics, 21
- Soundness, 75, 82, 117
- State object, 63
- State type, 63
- Static analysis, 4
- Static types, 7
- Store, 45
- Store typing, 51
- Strong update, 33
- Structural rule, 15
- Stuck, 47
- Subject reduction, 23, 54, 56, 61
- Substitution, 21
- Substructural system, 33, 38
- Surface language, 48
- Syntactic category, 11
- Syntactic interference, 17
- Syntax, 21, 39, 63
- System F, 24

- Test driven development, 3
- Testing, 2
- Type inference, 24
- Type safety, 7
- Type system, 7
- Type theory, 8
- Type variable, 23, 40
- Typing, 50, 64
- Typing environment, 22

- Unit testing, 3

- Validity, 72
- Value, 40
- Value restriction, 28
- Variable binding, 11
- Variant, 12
- Void, 40

W, 39
Weakening, 84, 104
Weakest precondition calculus, 6, 13, 75
WHILE language, 10
White box testing, 3

Ynot, 37