UNIVERSITY
of
GLASGOW

Computing Science
Ph.D. Thesis

Expression Refinement

Alexander Bunkenburg

Submitted for the degree of

Doctor of Philosophy

Thesis
10806
Copy 1

Abstract

This thesis presents a refinement calculus for expressions.

The aim of refinement calculi is to make programming a mathematical activity, and thereby improve the correctness of programs. To achieve this, a refinement calculus provides a formal language and a set of rules that allow transformations of the language terms. Using a refinement calculus, to produce a correct program, the programmer writes a possibly non-algorithmic or inefficient term that nevertheless obviously describes the intended program. This term is the specification, and it is transformed into an efficient program by syntactic transformation, using the rules of the refinement calculus. This transformation is refinement.

Refinement is done iteratively and stepwise. By *iterative* refinement, we mean the specification is transformed into the program via a series of intermediate terms. By *stepwise* we mean a term may be refined by refining one of its subterms, in a piecemeal fashion. Formally, refinement is a partial order on language terms, and (most) language constructs are monotone with respect to that order.

The syntactic transformation rules comprise rules about logic, and rules for each of the language constructs. The programmer needs only these rules. A semantic interpretation of the expressions is not necessary to verify the refinement steps. All the rules, whether they are logical rules, or about the specificational and algorithmic language constructs, are at the same level. This is achieved by blurring the distinction between propositions and boolean expressions, and it leads to a unified deductive calculus of programming.

Our specification language is a wide-spectrum language: it contains non-algorithmic, specificational constructs as well as an algorithmic programming language. The refinement calculus presented here produces functional programs, and accordingly, the programming sublanguage is a rich functional programming language, similar to Haskell. The specificational constructs of the language are adapted from the imperative refinement calculus [Bac80, Mor87, Mor88b]. However, there are no assignments in our language. The variables never vary. In this sense, the terms of the language are like the expressions of the imperative refinement calculus, and we speak of 'expression refinement'.

The language has constructs that are nondetermined. In particular, it is possible to specify the desired outcome of a program by giving a property it should satisfy. Clearly, this need not determine a unique outcome. However, the nondeterminacy does not pervade the whole language. It is contained at variable binding. We have 'singular semantics', that is, each variable stands for one value, not for any of a choice of values.

However, variables may stand for the special value nontermination, or datastructures that contain nontermination. Function application is lazy: the body of the function must be evaluated before the argument is evaluated. We chose lazy semantics rather than the maybe more easily implementable strict semantics, because a lazy language is more expressive. In it, one can break an algorithm into parts without having to ensure that the intermediate values are terminating.

The rules of our calculus ensure that the final program delivers the desired outcome, and indeed terminates. That is, we have a total correctness calculus.

Imperative programming techniques are made possible in our language of expressions by the state monad. The state monad can be seen as a collection of primitives manipulating the abstract data type 'state'. By combining these primitives one can perform destructive updates, and yet variables of the language never vary.

An iterative calculation from non-algorithmic, mathematical specifications via functional programs to imperative programs is proposed. Strategies of derivation, and algebraic refinement laws are given to support this approach. The approach is illustrated by a few example derivations of programs from different problem domains.

We provide a denotational semantics of the language to clarify an informal understanding of the language constructs, and to show that the refinement axioms given are sound. The existence of the given model shows they are consistent. The semantics uses Smyth powerdomains to capture nondeterminacy. However, the programmer need never look at the semantics.

Declaration

This dissertation is submitted in accordance with the regulations for the degree of Doctor of Philosophy in the University of Glasgow. No part of it has been previously submitted by the author for a degree at any university. The derivation of the line drawing algorithm in chapter 7 is based on a paper by the author and Sharon Flynn [BF94].

Acknowledgements

I thank my first supervisor Joe Morris. We spent more time in critical discussions than he got Brownie points for. His search for elegance in simplicity and his relaxed confidence were good influences on me.

I thank Phil Wadler, John Launchbury, and John O'Donnell for supervision. Thanks to Ewen Denney and Sharon Flynn for discussions and ideas. Watch out for both their theses on expression refinement!

I also thank the Engineering and Physical Sciences Research Council[1] for financial support, in the form of a Research Studentship, during the period October 1992 to September 1995, and the Computing Science Department of Glasgow University for a discretionary grant from October 1995 to September 1996.

[1]previously the Science and Engineering Research Council

Contents

Chapter 1

Introduction

Good software should be correct, and fast. Compilers for the old imperative programming languages like Pascal and C are by now so mature and optimised that programs written in these languages usually satisfy the second requirement, speed, but since programming has not become easier, often fail the first, correctness.

The programmers can err in two ways: They can misunderstand the customer's requests and implement something different than their customer asked for. Or they can make errors in the programming and implement something different than they themselves intended. A refinement calculus tries to avert both kinds of error, the first one by a formal specification as a contract between the customer and the programmer, and the second by giving the programmer a means by which to derive the program mathematically from its specification.

The language of the refinement calculus should therefore be expressive enough to make it easy to write understandable yet precise specifications that characterise the desired properties of the result, without saying how the result may be calculated. At the same time, the language should include the target programming language as a subset so that the transition from logical specification to algorithmic program can be done iteratively and stepwise.

"The Refinement Calculus" is such a calculus. It is based on Dijkstra's guarded command language [Dij76], which is given a semantics in terms of predicate transformers. The Refinement Calculus was proposed independently by Back, Morris, and Morgan [Bac80, Mor87, Mor88b]. That language is an imperative language, best suited for developing programs in such languages as Pascal.

Imperative languages are easy to implement because their essence, destructively updatable variables, reflects the architecture of current computers. The changing state of the machine, however, makes such programs difficult to reason about, since we have to consider time, in the form of execution order, and in the form of change in the meaning of variables. For instance addition of two Pascal 'function' calls is not necessarily commutative. Instead, they could change a global variable in ways depending on which 'function' call is executed first. For example, given global variable a : *integer* and **function** $f(x : \mathit{integer}) : \mathit{integer}$; **begin** $a := x$; $f := x$ **end**, the expressions $f(1) + f(2)$ and $f(1) + f(2)$ both evaluate to 3, but leave different values in a. Because of these side-effects, Pascal 'functions' are not really functions. The same is true for ML with its references [MTH90].

A functional program, on the other hand, is a mathematical object without time. If it is expressed sufficiently algorithmically, it may be executed, remaining mathematically equivalent to the result. Furthermore, the order in which its parts are executed is given by data dependencies and is otherwise unconstrained. Unlike the variables in imperative programs the formal variables in functional programs don't actually vary.

In these two ways – no fixed execution order and unchanging variables – functional programs are more similar to mathematical logic than imperative programs are. Specifications, too, are written in mathematical logic. Therefore, it is appropriate to develop *functional* programs from them. In addition, current functional languages like Haskell [HW89, HPW91, HPW92, PH$^+$96] provide many convenient ways of defining and using data structures and higher order functions, so that the programmer has to travel less of the journey from logical specification to machine code.

Therefore calculi for functional programs or "expressions" have been developed. For instance, there is "Squiggol", also called the "Bird Meertens Formalism", inspired by [Bir84, Mee86] and developed further by Bird [Bir87, Bir88, Bir90] and others. Towards the end of the 1980s, Hoare suggested the Squiggolists forsake functions for relations, in order to capture nondeterminacy better. He also promoted category theory around that time [Hoa89], which is related to Squiggol functions and datatypes in [Mal89, Spi89], and in the theses [Jeu93, dM92]. The Squiggolists follow Hoare's suggestion, and relations are used in Ruby [JS90b] (a related calculus for hardware), and Squiggol-developments [BdBM$^+$91, ABH$^+$92, Hoo93]. It turns out that indeed the laws of functional Squiggol have parallels in relational Squiggol. Bird himself comes round to relations in [BdM93a, BdM93b].

In Squiggol, an efficient program is developed from a less efficient, but obviously correct one. Even the less efficient Squiggol program, however, is already executable in principle, and no specification constructs are employed. Functional Squiggol offers no nondeterminacy, whereas relational Squiggol does offer nondeterminacy in the form of non-functional relations, but unfortunately, the close similarity to functional programming languages is lost!

There are moreover researchers who have taken concepts and syntax from the imperative refinement calculus to produce a refinement calculus of expressions. These concepts include nondeterminacy and specification expressions.

Recently expression refinement calculi have been proposed. In [Mor90a, Mor90b], Morris explores the usefulness of such a calculus by example derivations. Norvell and Hehner [NH93] formalise an expression refinement calculus by an axiomatic semantics. In his thesis [War94] Ward presents an expression refinement calculus whose semantics are inspired by the predicate transformer semantics of the imperative calculus, but are more complicated, and not as neatly composable as predicate transformers.

However, in these languages there are no constructs for imperative programming at all, which is felt to be a loss, since one may want to program imperatively. Here are four possible reasons why one may want to employ imperative programming techniques.

Firstly, there is a theoretical reason. Some algorithms depend on sharing for their efficiency; such algorithms cannot be written functionally with the same time complexity.

Examples are the combinator graph reducer derived in this thesis and the depth-first-forest construction in [KL93].

Secondly, there is a reason of style. Some algorithms are most naturally expressed as imperative programs, that is as repeated modification of a set of variables. Examples are Bresenham's line and sphere drawing algorithms, and the fast sphere drawing algorithm derived in this thesis.

Thirdly, there is a practical reason. Sometimes speed is so important that an imperative program is needed since it can be mapped to imperative hardware more directly, and therefore is executed faster.

Fourthly, there is a reason of purpose. Programs that interact with the real world by nature describe a process rather than a value. Processes are described by imperative programs, whereas functional programs describe values. Interaction with the real world may be user interaction, communication with another program, in particular an operating system or libraries for particular tasks, or communication with input or output devices.

For these reasons, the target language of the refinement calculus in this thesis is an expression language with imperative threads in it.

There have been a number of proposals for an integration of functional and imperative techniques. The difficulty is adding state in a 'pure' way: the value of an expression must not depend on the state. Pascal functions and ML functions don't satisfy this, whereas the state monad does. Indeed, one can think of the state monad as just a library of operations that mimic denotational semantics of an imperative language. Since the state is an abstract data type and can be manipulated only by the provided operations, it is guaranteed that the underlying state of the machine can be used to implement the state. In the state monad model, a state transformer is a function from an old state to a pair of a result and a new state, which makes tying the connections between the functional and the imperative worlds easy.

The state monad is based on the categorial notion of a monad, utilised for computing science by [Mog89]. The idea was taken up by Wadler in a series of papers [Wad92b, Wad92a, Wad92c], in which monads are put to various uses, including imperative programming. The state monad and the related IO monad are treated in [PJW93, Lau93]. In the latter paper, and in [LJ94], the expressiveness resulting from the combination of laziness and imperative programming is explored. [LP95] gives a comprehensive treatment of the use of monads to capture imperative programming in Haskell.

The novelty of this thesis is a refinement calculus of expressions, with the state monad to capture imperative programming techniques. The language includes both non-algorithmic specification constructs and the algorithmic constructs of modern functional programming languages, so that all stages in a program derivation, from the first specification to the final program can be expressed in the language. Unlike the imperative refinement calculus, the state monad allows dynamic use of state: the number of updatable variables is not fixed by the program text; rather true references (pointers) are available.

The big aim is to develop a unified deductive calculus of programming. Such a calculus consists of a formal language, an interpretation of the language, and a set of axioms and inference rules describing theorems. The language contains a logic, an algorithmic language, and non-algorithmic specificational constructs.

Logic is the basis of any formal proof, in this case proofs about programs and specifications. Logical expressions also occur within programs and specifications, for example the condition in an **if then else** expression or the property in a prescription. We achieve a unified calculus by not distinguishing between these boolean expressions within expressions and the propositions about expressions. It turns out that for reasoning about specifications and programs, two-valued classical logic is not sufficient. We will come back to this point later.

The language also contains algorithmic expressions, that is, a programming language. Crucially, this language includes recursion, and therefore expressions that we interpret as nonterminating programs. Consequently, we must add the nonterminating truth value $\bot$ to the usual *True* and *False*.

The language also includes constructs to write non-algorithmic specifications, in particular, the generalised choices and guards. They can be combined in expressions that describe their possible outcomes by a property that they must satisfy. Clearly, such an expression may therefore have no, one, or many different possible outcomes. The specificational constructs thus introduce miracles (no possible outcome) and nondeterminacy (many possible outcomes). Since generalised choice already introduces nondeterminacy, there is no harm and some convenience in adding binary nondeterministic choice $\sqcap$ as well. Consequently, we must introduce more truth values: the miraculous truth value $\top$, and nondetermined choices between all truth values so far, in particular, *True* $\sqcap$ *False*. The way we choose to interpret choice we get $E \sqcap \top \equiv E$ and $E \sqcap \bot \equiv \bot$ for arbitrary expression E, and therefore, there are no further truth values beside the five $\bot$, *True* $\sqcap$ *False*, *True*, *False*, $\top$. However, the miraculous truth value $\top$ is of little interest, since it can be prevented by syntactic restrictions.

The language (not surprisingly) contains a binary relation equivalence $\equiv$, which we have already used above. Furthermore, to compare specifications and programs, it contains the refinement relation $\sqsubseteq$.

The interpretations of these language constructs come from extending the interpretation of classical two-valued logic to our five values, from extending the interpretation of a programming language with nondeterminacy, and interpreting the specificational language constructs. Obviously, there are many design choices here. The interpretations of equivalence $\equiv$ and refinement $\sqsubseteq$ are (formalisations of): If $E \equiv F$, then – as far as correctness is concerned – a customer asking for an implementation of E will be satisfied when given an implementation of F, and vice versa. If $E \sqsubseteq F$, then a customer asking for E will be satisfied when given F, but not necessarily the other way around.

The interpretation (model theory) is given to provide the programmers with some intuitive understanding of the language. This enables them to write the initial specification, and decide what program they should aim for in the derivation. In some aspects, it is useful to have the interpretation of a program correspond to the way that a computer will execute that program. However, for each individual refinement step in a derivation, no semantic understanding is required. Such a step is a purely syntactic application of a rule.

There are postulated rules (axioms) for each of the language constructs. Often, they come in pairs of rules that introduce and eliminate the language construct.

We have given a formal language that covers logic, specifications, and programs. The choice of language constructs, syntax, and interpretation, is somewhat influenced by personal prejudice and open to argument. The language given serves its purpose, but may well be revised in future work. Axiomatising a four- or five-valued logic is not straightforward: there are many choices with unexpected consequences. In general, the aim is to preserve as many familiar theorems from classical logic as possible.

The rest of this document is organised as follows. This was chapter 1. Chapter 2 introduces the specification language by giving an informal explanation of each of the language constructs. Chapter 3 lists and discusses the refinement axioms. These axioms characterise the language constructs algebraically. In derivations, they will be used to transform specifications to programs. Chapter 4 introduces imperative expressions and gives axioms for them. Chapter 5 describes data refinement of expressions. Chapter 6 gives the (partial) derivations of four simple programs. Its purpose is to familiarise the reader with the specification language. Imperative programming is used in the examples. Chapter 7 derives three related graphics programs: Bresenham's line and circle drawing algorithms and the Fast Sphere drawing algorithm. They are expressed imperatively for reasons of style, and depending on the system, speed. Chapter 8 gives the most involved derivation of the thesis: a graph reducer for a simple combinator language. This program must use state to achieve a desirable time complexity. Chapter 9 details the denotational semantics of the language. Some axioms are proven sound with respect to this semantics. Chapter 10 summarises, draws conclusions, and sketches possible further work.

Chapter 2

The Specification Language

This chapter introduces the specification language, except for the imperative expressions, which are discussed separately in chapter 4. Section 2.1 gives the general ideas and motivations behind the language, such as what it can express, and how we relate the expressions of the language to each other. Section 2.2 introduces each of the non-imperative language constructs informally, with small examples demonstrating their use. Section 2.3 is a short note on the sets and lists of the language. The final section gives grammars for the language, and the typing rules.

2.1 General Ideas

This section outlines the principles behind the specification language. Its subsections describe the scope of the language, define the notions of determinacy and feasibility, and informally describe the type system. The last subsection characterises expressions that are acceptable as complete specifications.

2.1.1 Scope of the Language

The specification language we'll use is very broad: it includes a logic, a programming language, and non-algorithmic specificational expressions. It is typed with implicit Hindley-Milner polymorphism [Hin69, Mil78].

The specification language contains the programming language so that programs can be derived from specifications iteratively. We won't say exactly what parts of the specification language the programming language excludes. It depends how much detail is required in a particular derivation. Sometimes one may be satisfied with a final program using fairly high-level constructs like floating point operations or lists. At other times one may be aiming for low-level code that can be mapped onto hardware directly. Then, we would eliminate floating point numbers, for example, and favour multiplications by powers of 2 over general multiplication. In general, the programming language contains the algorithmic, determined expressions – as the real programming languages do – whereas it definitely excludes the non-algorithmic language constructs.

2.1.2 Definedness, Determinacy, and Feasibility

Each expression is associated with a set of possible outcomes, also called values. An outcome is either a *proper* value, or the special outcome *undefined.* Undefined is a fictitious outcome of 'meaningless' expressions like $\frac{4}{0}$, or of expressions whose evaluation does not terminate because of infinite recursion.

If an expression E cannot yield the undefined outcome, we say E is *defined*, and write $\mathbf{def}E$. Otherwise, we say E is *undefined*, and write $\neg\mathbf{def}E$. We view $\mathbf{def}E$ itself as an expression of the specification language, rather than as a metalinguistic predicate on expression E.

For each type, the outcomes are partially ordered. When we talk of 'worst' and 'better' outcomes, we are referring to this order. The undefined outcome is worse than any other outcome.

Variables are bound to *outcomes*, not to expressions, which does make a difference in the presence of nondeterminacy. See the discussion of functions and λ abstractions in subsection 2.2.5. In particular, a variable in an expression may be bound to the undefined outcome. Since the semantics of the language are 'lazy', that needn't make the outcome of the whole expression undefined. To emphasise the differences, we will use capital letters $E, F, G, \ldots$ for expressions, and small letters x, y, z or identifiers starting with small letters for variables.

The sets of outcomes associated with the expressions are all 'upward closed', that is, if a set contains a given outcome, it also contains all outcomes better than it. We classify expressions by the number of their possible outcomes:

- If an expression has no possible outcome, we say it's a *miracle* or an *infeasible* expression. Miracles are not executable of course.
- If an expression has at least one possible outcome, we say it is *feasible.*

We write $\mathbf{feas}E$ to say that expression E is feasible, and $\neg\mathbf{feas}E$ to say that E is miraculous. The feasible expressions are further categorised by the worst outcomes in their associated sets. An outcome in a set is a *minimum* if it is worse than every other outcome in the set. An outcome is *minimal* in a set of outcomes if the set does not contain an outcome worse than it.

- If an expression has a minimum possible outcome, we say it is *determined.*
- If an expression does not have a minimum possible outcome, we say it is *nondetermined.* Typically, nondetermined expressions have two or more incomparable minimal outcomes.

So for example, the expression 3 is determined. Its minimum outcome is the outcome 3. The set of possible outcomes of the expression $\frac{4}{0}$ contains the undefined outcome, and every outcome better than it, that is, all other (numerical) outcomes. Therefore $\frac{4}{0}$ is also determined. Its minimum outcome is the undefined outcome. On the other hand, the expression $\pm\sqrt{2}$ is not determined. Its set of outcomes comprises the incomparable minimal outcomes $\sqrt{2}$ and $-\sqrt{2}$. We write $\mathbf{det}E$ to say that expression E is determined, and $\neg\mathbf{det}E$ to say that E is nondetermined.

Our specification language will contain only feasible expressions. They may however have infeasible subexpressions. Later, we'll characterise some feasible expressions syntactically. We'll also characterise some determined expressions syntactically.

2.1.3 Equivalence and Refinement

Expressions can be related by equivalence ($\equiv$) and by refinement ($\sqsubseteq$). Two expressions are equivalent if their sets of possible outcomes are equal. Expression E is refined by expression F, written $E \sqsubseteq F$, if the set of outcomes of E is a superset of the set of outcomes of F. It follows trivially that $\equiv$ is indeed an equivalence relation (it's reflexive, symmetric, and transitive), and that $\sqsubseteq$ is a partial order (reflexive, antisymmetric, and transitive).

The intuition for equivalence and refinement is this. If two programs are equivalent, the customer asking for one of them will be happy when given the other, and the other way round. If $E \sqsubseteq F$, then the customer asking for E will be happy when given F.

The expression $\perp_T$ ('bottom' or 'fail') may yield any possible outcome of type T, even the undefined outcome. Therefore its set of possible outcomes is a superset of any other set of outcomes of that type, and thus $\perp_T$ is refined by every expression E of type T. We write just $\perp$ and let the appropriate type be implicit.

$$\perp \sqsubseteq E$$

Our calculus is a total correctness calculus; we are always interested in the worst outcome of an expression. An expression that may yield the undefined outcome, for example $\perp$, is as bad as one that definitely yields the undefined outcome.

The extreme expression opposite $\perp$ is $\top$ ('top' or 'miracle'), which has no possible outcomes: It's a miracle indeed. Every expression is refined by miracle.

$$E \sqsubseteq \top$$

Again, there's a miracle $\top_T$ for every type T, but usually, we'll leave the type implicit. Of course miracles are not implementable, and in fact can be excluded by simple syntactic restrictions. If miracles were not excluded, the programmer could simply refine any specification to $\top$, but that would not be acceptable to the customer!

2.1.4 Types

The expressions in the language are typed. Types are helpful in programming in that they document the code, make a lot of programming errors detectable as type errors, and provide some guidance in program derivation. The type system we use here is essentially the same as the one used in Haskell [PH+96]. We write $E : T$ to express that the expression E has the type T. We use the capital letters $T, U, V, \ldots$ for arbitrary type expressions. The types include some primitive types like boolean, characters, natural numbers, integers, and real numbers, written as $\mathbb{B}, \mathit{Char}, \mathbb{N}, \mathbb{Z}, \mathbb{R}$ respectively. A type can also be a product of types, so for example, the expression $(5, \mathit{False})$ has type $\mathbb{N} \times \mathbb{B}$. Taking the product of types is not associative, so $(\mathbb{N} \times \mathbb{B}) \times \mathbf{Char}$ and $\mathbb{N} \times \mathbb{B} \times \mathbf{Char}$ are not the same type. A 1-tuple is just the same as its single element. The 0-tuple type is called the *unit type*, and written $()$. Its only proper value is the empty tuple $()$. The type of functions from arguments of type T to results of type U is $T \to U$. The arrow associates to the right, so the brackets in $T \to (U \to V)$ are superfluous.

Types can be named by a *type definition*, and then used anywhere. We use identifiers starting with capital letters for named types. For instance, we can define pixels as pairs of integers by:

$$\mathbf{type}\ \mathit{Pixel} \ \hat{=}\ \mathbb{Z} \times \mathbb{Z}.$$

We are then free to use *Pixel* as a type anywhere else. Naming types serves to document code. Using *Pixel* marks integer pairs that represent a pixel from pairs with different roles. There is, however, no semantic distinction. Type definitions can have type arguments (we use small letters $a, b, c, \ldots$), as in

$$\mathbf{type}\ \mathit{Label}\ a\ b \ \hat{=}\ a \times b \times \mathbb{N}.$$

We can then say that $(5.95, \text{`a'}, 0)$ has type $\mathit{Label}\ \mathbb{R}\ \mathbf{Char}$ and $(\text{`b'}, \text{`c'}, 1)$ has type $\mathit{Label}\ \mathbf{Char}\ \mathbf{Char}$.

Algebraic data types must be named before they are used. The values of an algebraic data type are generated by applying *constructors* to expressions. Constructors are identifiers starting with a capital letter. The definition of an algebraic data type lists the constructors of the type, and fixes how many arguments of what type they take. For instance, the most common algebraic data type is the boolean type.

$$\mathbf{type}\ \mathbb{B} \ \hat{=}\ \mathit{False} \mid \mathit{True}$$

It has two constructors, each of which has no arguments. We assume $\mathbb{B}$ as predefined. Another example for an algebraic data type are the extended real numbers, which could be modelled by

$$\mathbf{type}\ \mathit{ExtReal} \ \hat{=}\ \mathit{NegInf} \mid \mathit{Fin}\ \mathbb{R} \mid \mathit{PosInf}.$$

This type has two constructors with no arguments, and one with one real argument. The proper values of this type are $\mathit{NegInf}, \mathit{Fin}\ r, \mathit{PosInf}$, where r is of type $\mathbb{R}$.

If two proper values of an algebraic data type are constructed from distinct constructors, then they are considered distinct. For example, if we modelled complex numbers by

$$\mathbf{type}\ \mathit{Complex} \ \hat{=}\ \mathit{Cartesian}\ \mathbb{R}\ \mathbb{R} \mid \mathit{Polar}\ \mathbb{R}\ \mathbb{R},$$

then the two values $\mathit{Cartesian}\ \sqrt{2}\ \sqrt{2}$ and $\mathit{Polar}\ 1\ \frac{\pi}{4}$ would be considered not equal, and we would therefore probably want to define an equality-function that does identify them.

Comparing constructors to functions, we would say constructors are injective. However, unlike functions, constructors must always be fully applied, so *Cartesian* 3 is not a valid expression. Constructors are not strict: A constructor applied to an undefined argument is not undefined.

A type may include *type variables.* To express that the expression E has type T, whatever the type variable a in T stands for, we write $E : \forall\, a. T$[1], which is abbreviated to $E : T$, with the silent assumption that all free type variables in T are universally quantified on the outside. We say E is *polymorphic.* Using the type definition

$$\textbf{type}\ \mathit{Sum}\ a\ b \ \hat{=}\ \mathit{Inl}\ a \mid \mathit{Inr}\ b,$$

the expression *Inr* 5 is polymorphic. Its type is *Sum* a $\mathbb{N}$, whatever type a stands for.

We will use type definitions and algebraic data types to make *recursive* types. For example, lists would be defined

$$\textbf{type}\ \mathit{List}\ a \ \hat{=}\ \mathit{Empty} \mid \mathit{Cons}\ a\ (\mathit{List}\ a).$$

The type that is being defined (*List* a) occurs on the right hand side of its own definition. The meaning of such a definition is the limit of its unfoldings. Mutually recursive types are allowed.

With an appropriate formalisation of this type system, it can be shown that every expression has a unique 'most general' type.

2.1.5 Complete Expressions

The specification language consists of *complete* expressions, that is, those expressions that are feasible, have a (possibly polymorphic) type, and no free variables.

We insist on feasibility because miracle can never be implemented. Nevertheless it is a refinement of any expression. If miracle were not excluded, the programmer could accidentally refine too much and then later find they cannot implement the resulting infeasible expression on a computer. Such an accident is prevented by syntactic restrictions that guarantee feasibility of complete expressions. However, a complete expression still may have infeasible subexpressions.

We insist on typing because often typing guides a program derivation, and mechanical type checking exposes many programming errors. Furthermore type annotations document the code.

Although technically a complete expression should have no free variables, we allow free variables that stand for some 'standard constants'. In real programming languages these standard constants are built-in primitives, or are defined in 'prelude' files. During a derivation we will also often make auxiliary definitions. Technically, the standard constants and auxiliary definitions are attached to an expression by use of a **let** expression. We make the convention that the defining expressions of all auxiliary definitions must be determined, which allows folding and unfolding the definitions freely in calculations.

[1]This is not a dependent function type.

This convention is very mild, since most auxiliary definitions introduce determined functions, and even those that are not functions are generally determined.

2.1.6 Discussion

It is worth noting that refinement (and therefore equivalence) captures only the correctness of a program. It gives information about the sets of possible outcomes of the two compared expressions. Firstly, it says nothing about whether an expression is in the programming sublanguage of the specification language. So $E \sqsubseteq F$ does not mean 'F uses fewer specification-constructs than E'. Secondly, it says nothing about the length and syntactic complexity of the code. So $E \sqsubseteq F$ does not mean 'F is shorter/more elegant/more readable than E'. Thirdly, it says nothing about the time or space complexity. So $E \sqsubseteq F$ does not mean 'F is more efficient than E'. It is left to the programmer to make sure the derivation ends up in an elegant/readable, efficient program.

Total correctness is a guarantee that a program will terminate, and yield the desired outcome. Partial correctness on the other hand guarantees only that if a program terminates then it will yield the desirable outcome, but not that it does terminate. Obviously total correctness is the more useful guarantee, and therefore our calculus is a total correctness calculus. However, in some settings, for instance when dealing with parallel communicating processes, it is useful to factor a total correctness proof into a partial correctness proof and a termination proof. Partial correctness proofs in general are simpler than total correctness proofs.

Some argue that a total correctness guarantee is useless, unless we are also given a time bound for termination [Heh94], and therefore build timing into their calculus. Instead, we assume that all terminating programs do terminate in a reasonable length of time, and leave it to the programmer to shorten that time while refining the program, if necessary.

2.2 The Language Constructs Informally

In this section, each of the language constructs will be introduced informally, with some simple examples of their use. Unless mentioned otherwise, all expressions are possibly infeasible. Expressions will be denoted by capital letters, and formal variables by small letters.

2.2.1 Comparing and Classifying Expressions

We have already mentioned expressions of the forms

$$\mathbf{def}E, \mathbf{det}E, \mathbf{feas}E$$
$$E \sqsubseteq F, E \equiv F,$$

expressing that E is defined, determined, or feasible, and that E is refined by F, or equivalent to it. All these expressions are of type $\mathbb{B}$, and are defined, determined, and feasible, whatever E and F are. In the last two, E and F must have the same type.

2.2.2 The Miracle Buster if .. fi

The meaning of **if** E **fi** is the same as that of E, except when E is a miracle, in which case **if** E **fi** is undefined.

$$\begin{aligned} \textbf{if}\,\top\,\textbf{fi} &\equiv \bot \\ \textbf{if}\,E\,\textbf{fi} &\equiv E,\ \text{if} E \not\equiv \top \end{aligned}$$

In this way **if** .. **fi** is a miracle-buster, and clearly not monotone with respect to refinement. It turns a possibly infeasible expression into a guaranteed feasible expression. The type of **if** E **fi** is just that of E.

2.2.3 Choice

The outcome of the expression $E \sqcap F$ will be an outcome of E or of F. We have no more information. For example, the expression

$$\left(-\frac{p}{2}+\sqrt{\frac{p^2}{4}-q}\right) \sqcap \left(-\frac{p}{2}-\sqrt{\frac{p^2}{4}-q}\right)$$

denotes a (possibly complex) root of the quadratic equation $x^2 + px + q = 0$, that is one of two possibly distinct values (assuming $0 \leq \sqrt{\frac{p^2}{4} - q}$). Mathematicians commonly write it as $-\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$.

The two arguments of choice must have the same type. Choice is the greatest lower bound with respect to refinement. For arbitrary (possibly infeasible) expression X, we have:

$$(X \sqsubseteq E) \wedge (X \sqsubseteq F) \equiv (X \sqsubseteq E \sqcap F).$$

It follows that $E \sqcap \bot \equiv \bot$, for any expression E. We say choice is $\bot$-seeking, or *demonic*. It also follows that choice is $\top$-avoiding, that is, $E \sqcap \top \equiv E$. Choice is also idempotent, associative, and commutative.

Demonic choice $E \sqcap F$ is the greatest lower bound of E and F with respect to refinement. We do not use angelic (that is $\bot$-avoiding) choice, but it can be added as least upper bound with respect to refinement. The least upper bound gives $E \sqcup \bot \equiv E$ as required, and also $3 \sqsubseteq 3 \sqcup 4$. Refinement means reducing demonic nondeterminacy and increasing angelic nondeterminacy.

Choice is not selective: We don't have $E \sqcap F \equiv E$ or $E \sqcap F \equiv F$ for arbitrary expressions E and F. Selective choice (also known as 'cold nondeterminacy') has the desirable property that the λ rule is an equivalence, that is $(\lambda\, x.E)\ F \equiv E[F/x]$ for any E and F, but unfortunately function application does not distribute over it, and that is an unacceptable hindrance to calculation.

2.2.4 Generalised Choice

We generalise binary choice $\sqcap$ to a quantifier $\bigsqcap$, in the same way that conjunction $\wedge$ is generalised to universal quantification $\forall$. The form of a generalised choice is $\bigsqcap x : T.E$. Free occurrences of x in the body E are bound in $\bigsqcap x : T.E$. If the type is obvious, we may omit it. The meaning of the generalised choice $\bigsqcap x : T.E$ is the meaning of E with x bound to an arbitrary outcome of type T, including the undefined outcome. For example, $\bigsqcap r : \mathbb{R}.r^2$ yields the square of an arbitrary real number, or the undefined real number.

It is obvious that generalised choice introduces nondeterminacy. However, it preserves feasibility: $\bigsqcap x.E$ is feasible if E is feasible (for arbitrary x). Generalised choices become useful in combination with guards, which will be discussed shortly.

The symbol $\bigsqcap$ is chosen to remind us that generalised choice is the greatest lower bound with respect to refinement:

$$(\forall x : T.X \sqsubseteq E) \equiv (X \sqsubseteq \bigsqcap x : T.E),$$

where x is not free in X. The type of $\bigsqcap x : T.E$ is simply that of E, under the assumption that $x : T$.

2.2.5 Functions

Function application will be denoted by a space, or by enclosing the argument in brackets. Function application associates to the left, so the brackets in $(E\ F)\ G$ are superfluous.

If E has type $T \to U$, and $F : T$, then $E\ F : U$.

Not all functions yield $\perp$ when applied to $\perp$; those that do are called *strict*. But $\perp$ applied to an expression is $\perp$:

$$\perp E \quad \equiv \quad \perp.$$

Functions are constructed by λ abstractions. An abstraction has the form $\lambda x : T.E$, where E is a feasible expression in which x, of type T, may appear. (We may omit the type.) Furthermore, we require that E is monotone in x, that is, if $F \sqsubseteq G$, then $E[F/x] \sqsubseteq E[G/x]$[2]. This restriction is necessary to avoid inconsistency; see the later subsection 3.3.4 for explanation. When the function $\lambda x.E$ is applied to an expression, the formal variable x is bound to the outcome of the argument expression, and we receive an outcome of E. Formal variables are bound to *outcomes*, not to expressions. We say formal variables have *singular semantics*, in the sense of [SS92]. For example the possible outcomes of $(\lambda x.x + x)(0 \sqcap 1)$ are 0 and 2, but not 1. That means the β reduction rule of the λ calculus is not generally true. The refinement

$$E\,[F/x] \quad \sqsubseteq \quad (\lambda x.E)\ F$$

[2]Throughout the thesis, we'll use the meta syntax $E[F/x]$ for the expression E with all free occurrences of variable x replaced by expression F, and similarly $T[U/a]$ for types, subject, of course, to variable capture restrictions.

is valid for any F, but only if F is determined is it an equivalence. Many determined expressions can be recognised syntactically, namely as closed expressions drawn from grammar $\mathcal{D}$, given at the end of this chapter. In particular $\perp$ is determined, and so $E[\perp/x] \equiv (\lambda\, x.E)\perp$.

If E has type U under the assumption that $x : T$, then $(\lambda\, x : T.E) : T \rightarrow U$.

We will assume lots of standard logical and mathematical functions as predefined. They could be built from abstraction, application, and variables, as in the pure λ calculus. In section 2.3 these predefined functions will be joined by standard functions producing data structures.

The domain of a defined function $E : T \rightarrow U$ is the set of values on which the function is defined, that is, $\operatorname{dom} E \mathrel{\hat{=}} \{x : T \mid E\ x \not\equiv \perp_U\}$. The domain of the undefined function is the undefined set, whereas the domain of $\lambda\, x.\perp$ is the empty set.

A one-point function override is written $E[F \mapsto G]$. This is the function that maps F to G and otherwise agrees with function E. Obviously, the types must fit: if $E : T \rightarrow U$, then $F : T$ and $G : U$.

Some standard functions (with symbols rather than identifiers) $\oplus : T \rightarrow U \rightarrow V$ are written between their arguments, as $E \oplus F$ instead of $\oplus\ E\ F$. Such functions may be *sectioned*, that is applied partially:

$$\begin{aligned}(E\oplus) &\mathrel{\hat{=}} \lambda\, x.E \oplus x\\ (\oplus F) &\mathrel{\hat{=}} \lambda\, x.x \oplus E\\ (\oplus) &\mathrel{\hat{=}} \lambda\, x.\lambda\, y.x \oplus y.\end{aligned}$$

The brackets may be omitted when there is no confusion.

The undefined function $\perp_{T\rightarrow U}$ is refined by any function (of type $T \rightarrow U$). A defined function E is refined by a second defined function F if its applications are refined by the corresponding applications of F:

$$(E\ X \sqsubseteq F\ X) \Rightarrow (E \sqsubseteq F),$$

for arbitrary expression X.

2.2.6 Guarded Expressions

Guarded expressions have the form $G \rightarrow E$ where G is a feasible boolean expression, called the *guard*, and E is a feasible expression. The guarded expression guarantees that G is *True*, and delivers the outcome of E. If G is not equivalent to *True* (it could be $\perp$, *True* $\sqcap$ *False*, or *False*), we are guaranteed a miracle!

$$\begin{aligned}\mathit{True} \rightarrow E &\equiv E\\ G \rightarrow E &\equiv \top,\ \text{if } G \not\equiv \mathit{True}\end{aligned}$$

Therefore, guarded expressions are possibly infeasible. The type of $G \rightarrow E$ is that of E, and G must have type $\mathbb{B}$.

We write $G \rightarrow E$ to express that the program evaluation may only use E if G yields true. Guards act like emergency brakes and have to be checked by a human evaluator. For instance, an emergency brake may be used to prevent applying functions to arguments outside their domain, as in $(x \neq 0) \to \frac{17}{x}$ or $(x \neq 0) \to \ln x$ or $(x \neq 0 \vee y \neq 0) \to x^y$. The guard records knowledge that may be used to refine E.

Since $\top$ is the identity of choice, we can use guarding and choice to build conditional expressions like

$$\mathit{odd}\ x \to 1 \sqcap \mathit{even}\ x \to 0,$$

which computes the sign of x. The general form is

$$G_1 \to E_1 \sqcap \ldots \sqcap G_n \to E_n,$$

where $1 \leq n$. All the guarded expressions $G_i \to E_i$ in which the guard is not *True* "disappear", since they are $\top$, the unit of choice. Of those guarded expressions $G_i \to E_i$ where G_i is *True*, the guards disappear, leaving the E_i. We are left a choice between 'unguarded' expressions. It may be a choice between many expressions, therefore the whole expression is possibly nondetermined. However, it could be the case that none of the guards are true, and then the expression is just $\top$. Therefore it is also possibly infeasible. To make it guaranteed feasible, it must be bracketed by the miracle buster **if** ... **fi**. The general useful form

$$\mathbf{if}\ G_1 \to E_1 \sqcap \ldots \sqcap G_n \to E_n\ \mathbf{fi}$$

is a feasible, nondetermined, many-branched conditional expression, called *alternation* expression [War94]. Alternation expressions are usually refined to many-branched **if then else** expressions.

Guards are useful in generalised choices: If we are looking for any outcome x that satisfies the property G, we write $\sqcap x.G \to x$. All those x that make G not equivalent to *True*, make the guarded expression $G \to x$ become $\top$, and therefore, they are not among the outcomes of $\sqcap x.G \to x$. To specify an outcome of E, with x bound to any outcome satisfying G, we write $\sqcap x.G \to E$. Obviously, that expression could be infeasible, so we enclose it in **if fi** to get

$$\mathbf{if} \sqcap x : T.G \to E\ \mathbf{fi},$$

a very convenient shape for initial, non-algorithmic specifications.

2.2.7 Assertion Expressions

The shape of an assertion expression is $A \succ\!\!- E$, where A is a boolean expression, called the *assertion*, and E is any expression. We write $A \succ\!\!- E$ to express that we intend to use E only when A is *True*, and don't care what happens otherwise. If A is not *True*, the assertion expression is undefined.

$$\mathit{True} \succ\!\!- E \ \equiv\ E$$

$$A \succ\!\!- E \;\equiv\; \bot, \text{ if } A \not\equiv \mathit{True}$$

The type of $A \succ\!\!- E$ is just that of E, and A must have type $\mathbb{B}$. The assertion expression $A \succ\!\!- E$ is equivalent to the one branch alternation **if** $A \rightarrow E$ **fi**, if **feas**E.

Removing an assertion refines an expression:

$$A \succ\!\!- E \sqsubseteq E.$$

Therefore a computer may simply ignore all assertions. In this way assertions are like program comments stating what should be true at this point, for example in $0 < x \succ\!\!- \sqrt{x}$. If the assertion is not fulfilled, we don't care what outcome is produced. The expression could yield a sensible result, for instance for negative x it may yield the expected complex number. But we aren't guaranteed that. The program may not terminate at all, or yield a senseless default value like 0.

Assertions are often used in specifying functions. We write $\lambda x.A \succ\!\!- E$ to express that the function should deliver E for arguments x such that A is true. Otherwise the function may deliver anything.

Ward [War94] calls assertions "assumptions".

2.2.8 Recursion

The expression $\mu x : T.E$ is the least fixpoint of the function $\lambda x : T.E$ with respect to the refinement order $\sqsubseteq$. If $E : T$ under the assumption that $x : T$, then $(\mu x : T.E) : T$. In practice, we will often give self-referential definitions instead of using μ explicitly. We require that E be monotone in x, that is, if $F \sqsubseteq G$, then $E[F/x] \sqsubseteq E[G/x]$. The reason will be detailed in subsection 3.3.4. Furthermore, we make the convention that E should be determined. This convention ensures that $\mu x.E$ is feasible.

Recursion can be a convenient way of defining things. Here are three examples of definitions by recursion:

$$\begin{array}{rcl}
\mathit{fac} & : & \mathbb{N} \rightarrow \mathbb{N} \\
\mathit{fac} & \hat{=} & \mu f.\lambda n.\left(\begin{array}{l} \textbf{if}\, 0 = n \rightarrow 1 \\ \sqcap\, 1 \leq n \rightarrow n * f(n-1) \\ \textbf{fi} \end{array}\right) \\
\mathit{twoThrees} & : & \mathbb{N} \times \mathbb{N} \\
\mathit{twoThrees} & \hat{=} & \mu t.(3, \mathit{fst}\ t) \\
\mathit{primes} & : & [\mathbb{N}] \\
\mathit{primes} & \hat{=} & \mu p.\, 2 : [n \mid n \leftarrow [3, 5..], \wedge/[\neg(a\ \mathit{divides}\ n) \mid a \leftarrow \mathit{takewhile}\ (\leq \frac{n}{2})\ p]]
\end{array}$$

The first defines a function, namely the factorial function, the second a finite data structure, namely a pair of two threes, and the third an infinite data structure, namely the ascending list of prime numbers. Some notation, like list comprehension, has not been explained yet.

2.2.9 Let Expressions

The form of a **let** expression is **let** $x = E$ **in** F where E is a feasible expression, and F is a feasible expression in which x may appear free. **let** expressions are used to tie auxiliary definitions to an expression. The **let** expression **let** $x = E$ **in** F is very similar to the application $(\lambda x.F)\ E$. The difference is in the typing: in the **let** expression polymorphism of E is preserved. So if $F : U$ under the assumption that $x : \forall a.T$, and $E : \forall a.T$, then (**let** $x = E$ **in** F) $: U$. In contrast the typing rule for applying a λ abstraction is: If $F : U$ under the assumption that $x : T$, and $F : T$, then $(\lambda x.F)\ E : U$. There is no $\forall a.$ before T here.

Binding by **let** preserves the polymorphism of an expression, but (as with the β rule) we have to be careful with nondeterminacy. Introducing a **let** expression is a refinement:

$$F[E/x] \sqsubseteq \textbf{let } x = E \textbf{ in } F.$$

It is only an equivalence if E is determined.

As mentioned in subsection 2.1.5, auxiliary definitions and standard constants are technically tied to a program by **let** expressions, although we need not write them explicitly in that way. For calculations it would be most inconvenient if we could make auxiliary definitions (refining the whole), but not unfold them again (worsening the whole). We make therefore the convention that in auxiliary definitions the defining expression (above E) must be determined. Then we can freely fold and unfold them. In practice, this is a very mild restriction. Without it, unfolding a local definition must be preceded by a determinacy proof.

2.2.10 Binding Guards and Assertions

Binding guards are three-place language constructs of the shape $P := A \dashrightarrow E$ where P is a pattern, and A and E are determined expressions. *Patterns* are templates for outcomes. Patterns have free variables in them. The free variables of P may appear free in E, and become bound in $P := A \dashrightarrow E$. The meaning of $P := A \dashrightarrow E$ is constructed by instantiating these variables in such a way that P considered as an expression is equivalent to A. The meaning of $P := A \dashrightarrow E$ will be the meaning of E under the same instantiation.

Finding such an instantiation is called *matching.* A matching can go wrong in two ways. Either A is not defined enough to judge whether it matches the pattern P. In that case we say the matching *diverges.* Or A is defined enough to judge, and the judgement says A has a different shape from P. In that case we say the matching *fails.*

If the matching fails or diverges, the whole expression becomes miraculous.

Some examples will illustrate the different forms of patterns (a grammar is given at the end of the chapter) and how binding guards work. A pattern may be one of the values of the primitive types. For example $3 := A \dashrightarrow$ *"March"* yields the list of characters *"March"* if the outcome of A is 3. If the outcome of A is a different value or undefined, the expression is miraculous.

Generally a pattern may be a constant, a variable, or a tuple of patterns, or a constructor applied to patterns.

Assume the algebraic data type definition

$$\textbf{type } \mathit{Complex} \triangleq \mathit{Cartesian}\ \mathbb{R}\ \mathbb{R} \mid \mathit{Polar}\ \mathbb{R}\ \mathbb{R}.$$

Then $\mathit{Cartesian}\ x\ y := A \twoheadrightarrow \sqrt{x^2+y^2}$ delivers the modulus of the complex number represented by A, if A is built using the constructor *Cartesian*. If it is built using the constructor *Polar*, the match fails, and we have a miracle.

The wildcard pattern _ matches everything and binds nothing. It is like a fresh formal variable that does not occur in its scope. By itself it doesn't seem much use, but it can be used as part of a larger pattern, as in $\mathit{Polar}\ r\ \_ := A \twoheadrightarrow r$. This expression delivers the modulus of complex number A if A is built from the constructor *Polar*. If the number is in that form, we don't need to know the second argument of *Polar*, which is the angle of the complex number.

The type of $P := A \twoheadrightarrow E$ is that of E with the free variables bound so that $P \equiv A$. Binding guards may introduce miracles. Their use is in defining the **case** expression, where their possible infeasibility is tamed.

Binding assertions are similar to binding guards. Their form is $P := A \succ\!\!- E$ for pattern P, and feasible expressions A and E. The variables of P may occur freely in E. Binding assertions differ from binding guards in that when the matching fails or diverges, the whole expression is $\perp$ rather than $\top$. Therefore binding assertions are welcome as complete expressions. They are used in a program when we know that the outcome of an expression is constructed using a certain constructor, and need to extract the data in it. For example we may know that the expression A is a complex number built from the constructor *Polar*. Then we just write $\mathit{Polar}\ \_\ \theta := A \succ\!\!- \theta\ \mathit{mod}\ 360$ to extract its angle. If our belief that the constructor *Polar* has been used to make A was wrong, the expression is $\perp$.

2.2.11 Case Expressions

The syntax of a **case** expression is:

$$\textbf{case } E \textbf{ of } P_1 \twoheadrightarrow A_1 \sqcap \ldots \sqcap P_n \twoheadrightarrow A_n,$$

where E is a feasible expression, the P_i are patterns, the A_i are feasible expressions, and $1 \leq n$. Matching binds the free formal variables of the pattern P_i in the associated expression A_i. The **case** expression is a shorthand for a combination of binding guards, choice, and the miracle buster:

$$\textbf{case } E \textbf{ of } P_1 \twoheadrightarrow A_1 \sqcap \ldots \sqcap P_n \twoheadrightarrow A_n \quad \triangleq \quad \textbf{if } P_1 := E \twoheadrightarrow A_1 \sqcap \ldots \sqcap P_n := E \twoheadrightarrow A_n \textbf{ fi}.$$

Case expressions are 'binding alternations'.

What is the outcome of a **case** expression ? Each of the matchings may succeed, producing a list of formal variable bindings, or it may fail or diverge. Those branches whose matching fails or diverges become miracles, which disappear, since miracle is the unit of choice. The outcome of the **case** expression is the outcome of an A_i such that the associated pattern P_i successfully matches E, binding the free formal variables of P_i in A_i. If there is no such A_i, the **case** expression is undefined.

Since there may be many branches with successful matchings, **case** expressions can introduce nondeterminacy. How can we recognise **case** expressions that do not introduce nondeterminacy ? We say two patterns are *exclusive*, if there is no outcome that they both match. For example $(x, 1)$ and y are not exclusive, since they both match the value $(\mathit{True}, 1)$, but the patterns $(\mathit{Cartesian}\ x\ y, 1)$ and $(\mathit{Polar}\ r\ \theta, z)$ are exclusive.

Case expressions are feasible, but possibly nondetermined. If E and the A_i are determined, and the patterns are mutually exclusive, then the **case** expression is also determined.

Some examples will illustrate the use of **case** expressions. This **case** expression

$$\textbf{case}\ E\ \textbf{of}\ 1 \rightarrow \text{``}\mathit{January}\text{''} \sqcap 2 \rightarrow \text{``}\mathit{February}\text{''} \sqcap \ldots \sqcap 12 \rightarrow \text{``}\mathit{December}\text{''}$$

converts the outcome of E into the name of a month. The outcome of E is matched to the constant patterns $1, 2, \ldots, 12$. If one of those matches succeeds, then the expression will deliver the corresponding name of the month. If all twelve matchings fail, say because the outcome of E is 13, then the **case** expression is $\bot$.

Generally a pattern may be a constant, variable, or a tuple of patterns, or a constructor applied to patterns. So for example

$$\textbf{case}\ E\ \textbf{of}\ \mathit{Cartesian}\ x\ y \rightarrow \sqrt{x^2 + y^2} \sqcap \mathit{Polar}\ r\ \_ \rightarrow r$$

delivers the modulus of the complex number represented by E. If the expression E delivers a value constructed by *Cartesian*, then x and y are bound to the appropriate values, and $\sqrt{x^2 + y^2}$ is the outcome of the **case**, and similarly for the polar case. The wildcard pattern _ is used because if the number is given in polar form, we only need to know its modulus, the angle is irrelevant.

Case expressions supply some syntactical sugar.

- Function abstraction with patterns. The abstraction $\lambda P.E$ where P is a pattern stands for $\lambda x.\textbf{case}\ x\ \textbf{of}\ P \rightarrow E$, where x is a fresh variable.

- Recursion with patterns. The recursion $\mu P.E$ where P is a pattern stands for $\mu x.\textbf{case}\ x\ \textbf{of}\ P \rightarrow E$ where x is a fresh variable.

- Let-expressions with patterns. The expression $\textbf{let}\ P = E\ \textbf{in}\ A$ where P is a pattern stands for $\textbf{let}\ x = E\ \textbf{in}\ (\textbf{case}\ x\ \textbf{of}\ P \rightarrow A)$ where x is fresh.

- Function definitions. A function may be defined by listing applications of it to patterns. These defining equations are combined by **case** and transformed into a function abstraction. These equations defining f:

$$f\ P_1 \triangleq A_1;\ \ldots;\ f\ P_n \triangleq A_n,$$

where $1 \leq n$ stand for the single definition

$$f \mathrel{\hat{=}} \mu f.\lambda x.\mathbf{case}\ x\ \mathbf{of}\ P_1 \to A_1 \sqcap ... \sqcap P_n \to A_n$$

where x is fresh. We have already mentioned the implicit recursion in subsection 2.2.8.

2.3 Set and Lists

This section rounds up some notation for sets and lists.

2.3.1 Sets

Sets are unordered, possibly infinite, collections of elements of the same type. A set may contain an undefined element without being itself undefined. The type of sets containing elements of type T is written $\mathbb{P}\,T$.

A type is acceptable as a set expression, so for example the type $\mathbb{N}$ can also be used as the set of all natural numbers and the undefined natural. The set $\mathbb{N}$ is of type $\mathbb{P}\mathbb{N}$.

For natural n, the set containing the elements $E_1, ..., E_n$ is written $\{E_1, ..., E_n\}$. The order of the elements between the brackets is arbitrary. In particular, the empty set is written $\{\}$.

The usual set-operations are available, for instance, the union of two set expressions S and T is written $S \cup T$, the intersection $S \cap T$, the set difference $S \setminus T$. If E is an element of set S, we write $E \in S$. For integers i and j, the expression $\{i..j\}$ denotes the set of integers from i up to j inclusively. If $i = j$ this is a singleton set, and if $i > j$ this is the empty set. If S is a set of sets, then $\bigcup S$ is the union of its elements.

A function can be *mapped* over a set, that is, we take the image of the set by the function: F^*S is the set of elements $F\ x$ where $x \in S$.

Set *comprehension* are a convenient way of specifying sets. Their general form is $\{E \mid Q_1, ..., Q_n\}$, where $1 \leq n$. The Q_i are qualifiers. They are either of the form $P \leftarrow F$ for pattern P and expression F, or boolean expressions. Set comprehensions can be expressed in terms of mapping, conditionals, and big union:

$$\begin{aligned}
\{E \mid P \leftarrow F\} &\equiv (\lambda P.E)^*F \\
\{E \mid B\} &\equiv \mathbf{if}\ B\ \mathbf{then}\ \{E\}\ \mathbf{else}\ \{\} \\
\{E \mid Q_1, Q_2, ..., Q_k\} &\equiv \bigcup\{\{E \mid Q_2, ..., Q_k\} \mid Q_1\},
\end{aligned}$$

for $2 \leq k$. See [Wad92a] for the monad story behind list comprehensions.

2.3.2 Lists

Lists are linearly ordered collections of elements of the same type. Since it is always easy just to ignore the order of the elements, lists are acceptable wherever a set is expected. The type of lists containing elements of type T is written $[T]$.

The list containing $E_1, ..., E_n$ in this order is written $[E_1, ..., E_n]$, so in particular, the empty list is $[\,]$. The concatenation of two lists E and F is written $E +\!\!+ F$. The infix function (:) 'cons' takes an element E and a list F and delivers the list $[E] +\!\!+ F$.

As in many functional programming languages, lists are modelled by the type

$$\mathbf{type}\ List\ a \quad \hat{=} \quad Nil \mid Cons\ a\ (List\ a).$$

Then $[\,]$ is shorthand for *Nil*, and $a : as$ for $Cons\ a\ as$. This makes : a cheap operation, so our derivations will often favour using : rather than $+\!\!+$. Every defined list is expressible as $[\,]$, or $a : as$ for some a and as, and these two forms will be allowed as patterns, standing for *Nil* and $Cons\ a\ as$.

Since all constructors are lazy, lists are lazy: A list that contains undefined elements is not itself undefined.

For integers i and j, the expression $[i..j]$ denotes the list of integers from i up to j inclusively in order. If $i = j$ this is a singleton list, and if $i > j$ this is the empty list.

Here are a handful of useful list-manipulating functions. Others can be found for example in the Haskell prelude ([PH+96]), or in [BW88].

A function can be *mapped* over a list: *map* is defined by:

$map : (a \to b) \to [a] \to [b]$
$map\ f\ [\,] \mathrel{\hat{=}} [\,]$
$map\ f\ (a : as) \mathrel{\hat{=}} f\ a : map\ f\ as.$

One useful combinator of lists is *foldr*, defined by:

$foldr : (a \to b \to b) \to b \to [a] \to b$
$foldr\ f\ b\ [\,] \mathrel{\hat{=}} b$
$foldr\ f\ b\ (a : as) \mathrel{\hat{=}} f\ a\ (foldr\ f\ b\ as).$

Its sibling *foldl* is defined by

$foldl : (a \to b \to a) \to a \to [b] \to a$
$foldl\ f\ a\ [\,] \mathrel{\hat{=}} a$
$foldl\ f\ a\ (b : bs) \mathrel{\hat{=}} foldl\ f\ (f\ a\ b)\ bs.$

The function *concat* concatenates a list of lists. It is defined by

$concat : [[a]] \to [a]$
$concat \mathrel{\hat{=}} foldr\ (+\!\!+)\ [\,].$

Filtering a list by a boolean valued function p is defined by

$filter : (a \to \mathbb{B}) \to [a] \to [a]$
$filter\ p\ [\,] \mathrel{\hat{=}} [\,]$
$filter\ p\ (a : as) \mathrel{\hat{=}} \mathbf{let}\ xs = filter\ p\ as\ \mathbf{in}\ \mathbf{if}\ p\ a\ \mathbf{then}\ a : xs\ \mathbf{else}\ xs$

The length of a finite list xs is $\#xs$. The function *takewhile* takes a predicate and a list and delivers the longest prefix of the list whose elements satisfy the predicate.

$takewhile : (a \to \mathbb{B}) \to [a] \to [a]$
$takewhile\ p\ [\,] \hat{=} [\,]$
$takewhile\ p\ (a : as) \hat{=} \mathbf{if}\ p\ a\ \mathbf{then}\ a : takewhile\ p\ as\ \mathbf{else}\ [\,]$

The list xs reversed is written $\overleftarrow{xs}$. The function $snoc : [a] \to a \to [a]$ is defined $snoc\ as\ a \hat{=} as +\!\!+ [a]$.

List comprehensions are analogous to set comprehensions. Their general form is $[E \mid Q_1, ..., Q_n]$, where $1 \leq n$. The Q_i are qualifiers. They are either of the form $P \leftarrow F$ for pattern P and expression F, or boolean expressions. List comprehensions are given meaning by translation into expressions using *map*, *concat*, and *filter*:

$$\begin{aligned} [E \mid P \leftarrow F] &\equiv map\ (\lambda P.E)\ F \\ [E \mid B] &\equiv \mathbf{if}\ B\ \mathbf{then}\ [E]\ \mathbf{else}\ [\,] \\ [E \mid Q_1, Q_2, ..., Q_k] &\equiv concat\ [[E \mid Q_2, ..., Q_k] \mid Q_1], \end{aligned}$$

for $2 \leq k$.

2.4 Types and Grammars

The specification language is typed with Hindley-Milner polymorphism [Hin69, Mil78]. The type of an expression depends on the types of the free formal variables in the expression. If expression E has only one free variable x, we write $x : U \vdash E : T$ to express that if x has type U, then the expression E has type T. This form is called a *sequent*. The part before $\vdash$ is the *antecedent*, and the part after it the *consequent*. The antecedent gives the *context*. *Contexts* map variables to types. We use Γ to stand for an arbitrary context. Γ maps the variable x to the type $\Gamma\ x$. The context $\Gamma, x : T$ maps x to T and otherwise agrees with Γ.

The consequent may list a number of typings of expressions, but at least one. So the general form of a sequent is

$$x_1 : U_1, ..., x_n : U_n \vdash E_1 : T_1, ..., E_k : T_k,$$

where $0 \leq n$ and $1 \leq k$. A *type rule* tells us how to derive one sequent called the *conclusion* from a number of others, called the *hypotheses*. We write the hypotheses above a horizontal line, and the conclusion below. So if from the single hypothesis $x : \mathbb{B} \vdash E : \mathbb{N}$ we conclude $x : \mathbb{B} \vdash 6 + E : \mathbb{N}$, we write

$$\frac{x : \mathbb{B} \vdash E : \mathbb{N}}{x : \mathbb{B} \vdash 6 + E : \mathbb{N}}.$$

The types include type variables. To express that the expression E has type T, whatever the type variable a in T stands for, we write $E : \forall a.T$. The type variable a is universally quantified in the type T and we say E is *polymorphic*. A *typescheme* is a type with zero, one, or many universal quantifications. Here we write recursive types using μ, but in a program we will always define a recursive type by an equation. So instead of writing

$$\textbf{type } \mathit{Tree} \;\hat{=}\; \mu\, a.\mathit{Node}\ \mathbb{Z} \mid \mathit{Branch}\ (a \times a)$$

for a binary tree type with integers at the leaves, we would write

$$\textbf{type } \mathit{Tree} \;\hat{=}\; \mathit{Node}\ \mathbb{Z} \mid \mathit{Branch}\ (\mathit{Tree} \times \mathit{Tree}),$$

analogously to the practice of defining recursive functions, thereby removing type variables that are bound by μ. The remaining type variables are bound by $\forall$, which is only allowed to occur at the outside of the typescheme. Since in practice we will only ever use typeschemes in which all free variables are bound by $\forall$ we will often take the $\forall$ as understood without writing it.

Figure 2.1 gives the abstract grammar of types. The state transformer and reference types will be discussed in chapter 4. The abstract grammar of the specification language is given in figure 2.2. The language construct state encapsulation will also be discussed in chapter 4. The abstract grammar of patterns is given in figure 2.3. The unsurprising typing rules are given in figure 2.4. Capital letters with a tilde (like $\tilde{T}$) stand for typeschemes that may have universal quantifications, whereas capital letters without tildes stand for types without universal quantifications.

We will write $\tilde{T}\frac{U}{a}$ for the typescheme that is derived from $\tilde{T}$ by removing the universal quantification over a, and instantiating[3] the then free occurrences of a by the type U.

Typing rules for the language constructs involving patterns have been omitted. They are long to write. The only point to note is that binding to a pattern loses polymorphism. One may miss rules about the constructed types pairs, sums, and sets. They are omitted here because we consider the constructors (for example $\lambda x.\lambda y.(x, y)$) and destructors (for example *fst*) of values of these types as given constants, and each given constant comes with its typing. Later, in chapter 9, figure 9.5 will give the types (and semantics) of some constants.

The specification language consists of the expressions that

- are generated by the grammar for $\mathcal{F}$,
- have a (possibly polymorphic) type according the typing rules,
- and have no free variables.

It can be proven that all those expressions are feasible. Sometimes we'll also consider the typed, possibly infeasible expressions generated from the grammar for $\mathcal{E}$, but they are not acceptable as complete programs.

2.4.1 Determined Expressions

We often may want to find out whether a given expression is determined. We may need this information to satisfy a side condition of a law we want to apply, for instance the β

[3]subject to the normal variable capture restrictions

$\tilde{\mathcal{T}}$	$::=$	$\forall\, a.\tilde{\mathcal{T}}$	typescheme
	$\mid$	$\mathcal{T}$	
$\mathcal{T}$	$::=$	a	type variable
	$\mid$	$\mu\, a.T$	recursive types
	$\mid$	$\mathbb{B} \mid \mathbb{Z} \mid \mathbb{R} \mid ...$	primitive types
	$\mid$	$\mathcal{T} \rightarrow \mathcal{T}$	function
	$\mid$	$\mathcal{T} \times \mathcal{T}$	pair
	$\mid$	$\mathcal{T} + \mathcal{T}$	sum
	$\mid$	$\mathbb{P}\mathcal{T}$	set
	$\mid$	$ST\ \mathcal{T}\ \mathcal{T}$	state transformer
	$\mid$	$Ref\ \mathcal{T}\ \mathcal{T}$	reference

Figure 2.1: Abstract grammar of the types

$\mathcal{F}$	$::=$	$\bot_{\mathcal{T}}$	undefined
	$\mid$	k	constants
	$\mid$	x	variables
	$\mid$	$\mathcal{F}\ \mathcal{F}$	application
	$\mid$	$\lambda\, x.\mathcal{F}$	abstraction
	$\mid$	$\mu\, x.\mathcal{F}$	recursion
	$\mid$	$\heartsuit x : \mathcal{T}.\mathcal{F}$	$\heartsuit$ stands for binders $\forall, \exists$
	$\mid$	**let** $x = \mathcal{F}$ **in** $\mathcal{F}$	local definition
	$\mid$	$\mathcal{T}$	type as set
	$\mid$	$\{\mathcal{F}_1, ..., \mathcal{F}_n\}$	enumerated sets
	$\mid$	**run** $\mathcal{F}$	state encapsulation
	$\mid$	$\mathcal{F} \sqcap \mathcal{F}$	choice
	$\mid$	$\sqcap x : \mathcal{T}.\mathcal{F}$	generalised choice
	$\mid$	$\mathcal{F} \succ\!\!- \mathcal{F}$	assertion
	$\mid$	**if** $\mathcal{E}$ **fi**	miracle buster
	$\mid$	$\mathcal{P} := \mathcal{F} \succ\!\!- \mathcal{F}$	binding assertion
	$\mid$	**case** $\mathcal{F}$ **of** $\mathcal{P}_1 \rightarrow \mathcal{F}_1 \sqcap ... \sqcap \mathcal{P}_n \rightarrow \mathcal{F}_n$	case
	$\mid$	$\mathcal{E} \sqsubseteq \mathcal{E}$	refinement
	$\mid$	$\mathcal{E} \equiv \mathcal{E}$	equivalence
	$\mid$	**def**$\mathcal{E}$	definedness
	$\mid$	**det**$\mathcal{E}$	determinacy
	$\mid$	**feas**$\mathcal{E}$	feasibility
$\mathcal{E}$	$::=$	$\mathcal{F}$	feasible expression
	$\mid$	$\mathcal{E} \sqcap \mathcal{E}$	choice
	$\mid$	$\mathcal{F} \rightarrow \mathcal{F}$	guarded expression
	$\mid$	$\mathcal{P} := \mathcal{F} \rightarrow \mathcal{F}$	binding guard
	$\mid$	$\sqcap x : \mathcal{T}.\mathcal{E}$	generalised choice
	$\mid$	$\top_{\mathcal{T}}$	miracle

Figure 2.2: Abstract grammar of the expressions

$$
\begin{array}{lll}
\mathcal{P} ::= & \_ & \text{wildcard} \\
\quad\mid & x & \text{variable} \\
\quad\mid & k & \text{constant} \\
\quad\mid & (\mathcal{P}_1, ..., \mathcal{P}_n) & \text{tuples} \\
\quad\mid & K\ \mathcal{P}_1 ... \mathcal{P}_n & \text{constructors}
\end{array}
$$

Figure 2.3: Abstract grammar of patterns

$$\frac{}{\Gamma \vdash \bot_T : T, \top_T : T} \qquad \frac{}{\Gamma \vdash x : \Gamma\, x} \qquad \frac{\Gamma \vdash E : T \to U, E : T}{\Gamma \vdash E\ F : U}$$

$$\frac{\Gamma, x : T \vdash E : U}{\Gamma \vdash \lambda x.E : T \to U} \qquad \frac{\Gamma, x : T \vdash E : T}{\Gamma \vdash \mu\, x.E : T} \qquad \frac{\Gamma \vdash E : T, F : T}{\Gamma \vdash E \sqcap F : T}$$

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash \mathbf{def} E : \mathbb{B}, \mathbf{det} E : \mathbb{B}, \mathbf{feas} E : \mathbb{B}} \qquad \frac{\Gamma, x : T \vdash E : U}{\Gamma \vdash \bigsqcap x : T.E : U}$$

$$\frac{\Gamma \vdash A : \mathbb{B}, E : T}{\Gamma \vdash A \succ\!\!- E : T} \qquad \frac{\Gamma \vdash E : T}{\Gamma \vdash \mathbf{if}\, E\, \mathbf{fi} : T} \qquad \frac{\Gamma \vdash G : \mathbb{B}, E : T}{\Gamma \vdash G \to E : T}$$

$$\frac{\Gamma, x : T \vdash E : \mathbb{B}}{\Gamma \vdash \heartsuit x : T.E : \mathbb{B}}, \forall, \exists \qquad \frac{\Gamma \vdash E : \forall a.\tilde{T}}{\Gamma \vdash E : \tilde{T}\left[\frac{U}{a}\right]}, a \text{ not in } U \qquad \frac{\Gamma \vdash E : \tilde{T}}{\Gamma \vdash E : \forall a.\tilde{T}}, a \text{ not in } \Gamma$$

$$\frac{\Gamma \vdash E : \tilde{T} \quad \Gamma, x : \tilde{T} \vdash F : U}{\Gamma \vdash \mathbf{let}\ x = E\ \mathbf{in}\ F : U} \qquad \frac{\Gamma \vdash E : \forall s.ST\ s\ T}{\Gamma \vdash \mathbf{run}\ E : T}, s \text{ not in } T$$

$$\frac{}{\Gamma \vdash T : \mathbb{P}\, T}, \text{ for type } T \qquad \frac{\Gamma \vdash E_1 : T, ..., E_n : T}{\Gamma \vdash \{E_1, ..., E_n\} : \mathbb{P}\, T}$$

$$\frac{\Gamma \vdash E : \tilde{T}, F : \tilde{T}}{\Gamma \vdash (E \sqsubseteq F) : \mathbb{B}} \qquad \frac{\Gamma \vdash E : \tilde{T}, F : \tilde{T}}{\Gamma \vdash (E \equiv F) : \mathbb{B}}$$

Figure 2.4: Typing the expressions

transformation $E[F/x] \equiv (\lambda\, x.E)F$ which is only true if F is determined.

What does determinacy mean exactly in a total correctness calculus ? It is important to realize that an intuitively determined expression can be refined by an obviously nondetermined expression, for example $\perp \sqsubseteq 3 \sqcap 4$, or $\lambda\, x.3 \sqcap 4 \sqsubseteq (\lambda\, x.3) \sqcap (\lambda\, x.4)$. Since we would like to view evaluation of a program simply as a last refining step, it would not be sufficient to say an expression is determined if it has exactly one possible outcome. Rather, an expression is determined if it has a minimum outcome. In the two examples, the minimum outcome of $\perp$ is the undefined outcome, whereas the minimum outcome of $\lambda\, x.3 \sqcap 4$ is just the function that maps any argument to either 3 or 4.

An expression E is determined if it cannot be written as a choice of two expressions[4] that are ***strictly more refined*** than E, that is, if $E \equiv F \sqcap G$, then $E \equiv F$ or $E \equiv G$. For example, $3 \sqcap 4$ is not determined, because it can be written as a choice between two strictly more refined expressions, namely 3 and 4. On the other hand, $\perp$ is determined, because whenever $\perp \equiv F \sqcap G$, at least one of F and G must be $\perp$ itself (which means the other contributes nothing to the choice expression). The example of $\lambda\, x.3 \sqcap 4$, which can also not be written as a choice of two strictly more refined expressions, shows that an expression may be determined and defined, and still have more than one possible outcome under a refining evaluation.

Alternatively, we can express 'E is determined' using existential quantification: $\exists\, x : T.(x \equiv E)$. This works fine for $\perp$, since in our lazy calculus, x may be bound to undefinedness, and the variables in quantifications range over all possible outcomes of its type, including the undefined one.

More simply, since nondeterminacy can only arise from certain language constructs, we also have an easy syntactic check. If the expression E is given by the grammar $\mathcal{D}$ given in figure 2.5 and has no free variables, it is determined. The grammar basically describes a functional programming language. However, if E is not given by this grammar, it may still be determined, for example, **if** $0 = x \rightarrow 1 \sqcap 0 < x \rightarrow 16$ **fi** is obviously determined. Such an expression may always be refined to an equivalent one which is generated from $\mathcal{D}$, such as here **if** $x = 0$ **then** 1 **else** (**if** $0 < x$ **then** 16 **else** $\perp$). All constants are determined, and the functional constants have determined bodies. If an expression is generated by $\mathcal{D}$, but still has free variables, we may not be able to decide whether it is determined. For example, the expression f 3 becomes nondetermined if f is bound to $\lambda\, x.3 \sqcap 4$, which is itself determined, but not drawn from $\mathcal{D}$. Therefore we insist that expressions must have no free variables.

[4]Infeasible expressions need not be excluded here, but they are hopeless, since $E \equiv F \sqcap \top$ means that $F \equiv E$.

$\mathcal{D}$ ::=	$\bot_{\mathcal{T}}$	undefined
$\mid$	k	constants
$\mid$	x	variables
$\mid$	$\mathcal{D}\,\mathcal{D}$	application
$\mid$	$\lambda\, x.\mathcal{D}$	abstraction
$\mid$	$\mu\, x.\mathcal{D}$	recursion
$\mid$	$\heartsuit x : \mathcal{T}.\mathcal{D}$	binders $\forall, \exists$
$\mid$	**let** $x = \mathcal{D}$ **in** $\mathcal{D}$	local definition
$\mid$	$\mathcal{T}$	type as set
$\mid$	$\{\mathcal{D}_1, ..., \mathcal{D}_n\}$	enumerated sets
$\mid$	**run** $\mathcal{D}$	state encapsulation
$\mid$	$\mathcal{D} \succ\!\!- \mathcal{D}$	assertion
$\mid$	**if** $\mathcal{D}$ **then** $\mathcal{D}$ **else** $\mathcal{D}$	conditional
$\mid$	$\mathcal{E} \sqsubseteq \mathcal{E}$	refinement
$\mid$	$\mathcal{E} \equiv \mathcal{E}$	equivalence
$\mid$	**def**$\mathcal{E}$	definedness
$\mid$	**det**$\mathcal{E}$	determinacy
$\mid$	**feas**$\mathcal{E}$	feasibility

Figure 2.5: Determined expressions

Chapter 3

Refinement

This chapter presents a discussion of the meaning and uses of refinement, presents the logical underpinnings of refinement, and then lists the axioms and some theorems.

3.1 General Ideas

This section discusses the meaning and uses of refinement. The subsections treat the differences between specifications and program, the ways refinement is used and their technical implications.

3.1.1 From Specification to Program

The purpose of a refinement calculus is to give the programmer a language in which to specify a program, and a set of refinement rules with which to transform the specification into a program. Both the specification and the program describe sets of desired outcomes, however, they describe them in different ways.

The description given by the specification need not be algorithmic; rather it could be a property that the outcomes should satisfy. Even if a specification, or a part of it, has an algorithmic reading, that algorithm may not be satisfactory as a program if it has an avoidable high time or space complexity.

The program on the other hand describes the outcomes in an algorithmic way. A computer can be used to read and execute the algorithm, yielding one of the described outcomes. Programs are written in the programming language, which has only algorithmic constructs, so all programs describe their outcomes algorithmically. Furthermore the programming language guarantees that all programs have at least one possible outcome: programs are feasible.

A program F implements a specification E, or a specification E is refined by the program F, if the set of outcomes described by the specification is a superset of the set of possible outcomes of the program. We write $E \sqsubseteq F$. Therefore, obviously, refinement reduces nondeterminacy: $3 \sqcap 4 \sqsubseteq 3$. Refinement also increases termination, since all the sets of outcomes of expressions are *upward-closed*, that is, if a set contains an outcome v, it also contains all outcomes better (more terminating) than v. The only way to get an upward-closed subset from an upward-closed set is to remove some elements from its 'lower rim', that is, the least terminating elements in the set. Consequently, refinement increases termination.

Refining a specification into a program may be a difficult problem, so we can't expect to do it at once. We divide the difficult problem into many smaller problems by iterative refinement and stepwise refinement.

3.1.2 Iterative Refinement

Iterative refinement means that we transform a specification E_1 into the program E_n by transforming E_i into E_{i+1} for each $i \in \{1..n-1\}$. Each of the steps $E_i \sqsubseteq E_{i+1}$ is correctness-preserving in that the set of outcomes described by E_i is a superset of the set of outcomes of E_{i+1}. We conclude that E_1 is refined by E_n.

Iterative refinement requires the following two properties of the refinement calculus. First, the specification language and the program language must have a common super-language (the language described in the previous chapter is such a language), for this is the language that the part-specification, part-program E_i with $1 < i < n$ are written in. Since programs must be algorithmic, but specifications need not necessarily be non-algorithmic, we'll consider the program language as a sublanguage of the specification language. So all E_i with $1 \leq i \leq n$ are specifications, and in particular, E_n is also a program.

Secondly, refinement ($\sqsubseteq$) must be a transitive relation between expressions of the unified specification/program language. Indeed our refinement relation is transitive.

3.1.3 Stepwise Refinement

A big transformation can also be factored into smaller transformations by stepwise refinement. Stepwise refinement means that we transform an expression by transforming a subexpression of it. Say the expression is $E[F]$, that is an expression E with subexpression F in a certain place. We refine $E[F]$ into $E[G]$ by showing that F is refined by G. Stepwise refinement requires the language constructs to be monotone with respect to refinement. In the specification language we describe, almost all constructs are monotone. The exceptions are **if**_**fi**, $\equiv$, $\sqsubseteq$, **def**, **det**, **feas**, and the guards and assertions, which are not monotone in their first arguments.

3.2 Logic

This section describes the logical underpinnings of *proving* formulas.

We choose to identify formulas of the logic with boolean expressions. This makes for easy communication between the specification expressions and the formulas. Otherwise, the programmer would have to know two separate languages that closely mirror each other.

The downside of this decision is that since the expression language has undefinedness, nondeterminacy, and miracles, there are strange truth values besides *true* and *false*. We cannot simply use classical logic, but must adapt it in a reasonable way to deal with the strange truth values. There are many possibilities of doing this, and the best choice is not obvious.

The first subsection describes the model theory of our logic, that is, it gives the intended meanings of the formulas. The second subsection describes the proof theory of the logic, given in Hilbert's axiomatic style. The last subsection describes the style of reasoning we'll use in practice – equational reasoning – and how this style relates to a formal logical argument.

3.2.1 Model Theory of Formulas

This subsection describes our intended semantics for the logical formulas. Denotational semantics of the whole language are treated in detail in chapter 9 later.

Boolean Expressions

We take as formulas simply the boolean expressions of the specification language. Therefore, the two truth values *true* and *false* are joined by the undefined truth value, the miraculous truth value, and choices between them. The meaning of an expression is the (upward-closed)[1] set of its possible outcomes. The meaning of the boolean expressions is as follows. The meaning of the expression *True* is the set $\{true\}$, and of *False* it is $\{false\}$. The meaning of the expression $\perp$ is the set containing *true*, *false*, and the special undefined outcome. This is indeed an upward-closed set. Since it contains the undefined outcome, it must also contain all outcomes better than it, which are both *true* and *false*. The expression $\top$ is a miracle. It has no *possible* outcome: therefore its meaning is the empty set.

Feasibility, Definedness, and Determinacy

The feasibility-test **feas**E tells whether the expression E is feasible. Its meaning is simply $\{true\}$ if the meaning of E is not the empty set, and $\{false\}$ otherwise. The meaning of the definedness-test **def**E is $\{false\}$, if the meaning of E contains the undefined outcome (and hence all other outcomes), and $\{true\}$ otherwise. The meaning of **det**E is $\{true\}$, if there is a minimum element in the meaning of E, and $\{false\}$ otherwise.

Equivalence, Refinement, and Choice

The equivalence relation $\equiv$ relates expressions whose sets of possible outcomes are the same. It is sometimes called 'strong equality'. If E and F have the same meaning, then $E \equiv F$ has the meaning $\{true\}$, otherwise $\{false\}$.

The refinement relation $\sqsubseteq$ relates two expressions as follows. If the set of outcomes of E is a superset of the set of outcomes of F, then the meaning of $E \sqsubseteq F$ is $\{true\}$, otherwise it is $\{false\}$.

Binary choice between two expressions is interpreted as the union of their sets of outcomes. Therefore, when the four boolean expressions $\perp$, *True*, *False*, $\top$ are combined by choice, we obtain only one new truth value: *True* $\sqcap$ *False*, whose meaning is $\{true, false\}$.

[1]The definition of the orders for outcomes of each type will be given in chapter 9.

The meaning of generalised choice $\sqcap x : T.E$ is the union of the meanings of E, in environments that bind x to every outcome of type T.

Propositional Calculus

Here we interpret the logical connectives $=_{\mathbb{B}}, \wedge, \vee, \neg, \Rightarrow$. In choosing how the logical connectives should behave, the design aims are:

extend If all expressions used with a logical connective are proper, that is, defined, determined, and feasible, then the connective should behave exactly as its classical version.

laws It is desirable that as many theorems of classical logic as possible are preserved. Some will clearly have to go, for example the theorem of the excluded middle $E \vee \neg E$. The preserved theorems should be sufficient for practical calculation, and (ideally) easily recognisable.

monotone Connectives that are to be used within expressions are desirably monotone with respect to refinement – this allows piecewise refinement and terminating recursion. Furthermore, only monotone constructs can be part of the programming sublanguage.

function For simplicity, it is desirable to treat connectives simply as given constants of functional type rather than as extra language constructs. However, any connective that's a function must behave as all functions do, for instance, distribute over nondeterminacy, and therefore this treatment is not appropriate for all connectives. In particular, we'll treat $\Rightarrow$ as a language construct rather than a function.

The first design aim cannot be compromised if the connectives are to have any use. The second aim, preserving the theorems, requires care. It is not obvious (and somewhat a matter of taste) which theorems should be preserved and which we can do without. The third aim, monotonicity, may be compromised to gain the second, especially since logical connectives are not likely used in recursion, although we may want to refine them stepwise. The fourth aim is stronger than monotonicity. We include it to keep the language simple, with few language constructs.

We make the connectives $=_B, \wedge, \vee, \neg$ functions in the specification language, rather than language constructs. For any function, the meaning of application of a function to an argument is given by mapping the 'interpreted' function over the set of outcomes of the argument.

For example, the meaning of $\neg(\mathit{True} \sqcap \mathit{False})$ is mapping the 'interpreted' function *not* over the set $\{\mathit{true}, \mathit{false}\}$, yielding $\{\mathit{not\ true}, \mathit{not\ false}\}$, which is $\{\mathit{false}, \mathit{true}\}$. If the argument is $\top$, which has no outcomes, the application also has no possible outcomes, for example $\neg\top$ means mapping *not* over the empty set, which is of course still the empty set. We say the function $\neg$ (like all functions) ***preserves miracles.***

From these semantics it follows that all functions, in particular $=_{\mathbb{B}}, \wedge, \vee, \neg$, preserve miracles and distribute over nondeterminacy in all their arguments.

E	F	$E =_{\mathbb{B}} F$	$E \vee F$	$E \wedge F$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	*False*	$\perp$	$\perp$	*False*
$\perp$	*True*	$\perp$	*True*	$\perp$
False	$\perp$	$\perp$	$\perp$	*False*
False	*False*	*True*	*False*	*False*
False	*True*	*False*	*True*	*False*
True	$\perp$	$\perp$	*True*	$\perp$
True	*False*	*False*	*True*	*False*
True	*True*	*True*	*True*	*True*

Figure 3.1: Interpretation of the binary logical connectives

Now we only have to decide how these connectives behave when applied to $\perp$. The undefined boolean expression $\perp$ may result from a not terminating recursion, or more trivially, from a partial function. Intuitively, we can think of $\perp$ as "no information content", or "no answer", where *True* means "yes" and *False* means "no".

Our design aims extension and monotonicity could be easily met by making the connectives preserve $\perp$. Then we would have for example $(\neg\perp \vee E) \equiv \perp$, for feasible E, and *True* $\vee \perp \equiv \perp$ and $\neg\perp \equiv \perp$. It would make the connectives fit in nicely in a language where function application is strict.

But our function application is not strict, and there is good reason to exploit that freedom in designing the logical connectives. For example, it is reasonable to regard a disjunction as true as soon as one of its branches is: $(\frac{x}{x} = 1) \vee (x = 0)$. Similarly, a conjunction may reasonably be regarded as false as soon as one of its branches is false: $(0 < n) \wedge (n! \leq n^n)$. Indeed, if all connectives were strict, we'd have no way of expressing statements about partial functions like the previous formula. This approach is similar to that of LPF ("Logic of Partial Functions"), the logic of VDM, as in [BFL+94].

Equality $=_{\mathbb{B}}$ (sometimes "weak equality") however, is strict: it yields *True* for defined equal arguments, *False* for defined different arguments, and $\perp$ if an argument is undefined. Negation is also strict: $\neg$*True* $\equiv$ *False*, $\neg$*False* $\equiv$ *True*, and $\neg\perp \equiv \perp$.

Summarising, figure 3.1 gives the monotone extensions of the binary connectives $=_{\mathbb{B}}, \vee, \wedge$.

Whenever a connective applied to *True* and to *False* yields the same result, we define it to yield that same result also for $\perp$. Otherwise, the result is $\perp$. In this way we ensure that the extended connective is still monotone with respect to refinement.

$E \Rightarrow F$ is the same as $(E \not\equiv \textit{True}) \vee F$. Therefore, unlike functions, it does not preserve miracles, it does not distribute over nondeterminacy, and it is not monotone in its first argument. However, it does have desirable properties that make up for this: for feasible arguments, it is reflexive and transitive. Furthermore, it satisfies the deduction theorem, that is, if assuming E is a theorem, one can prove F, then one can conclude $E \Rightarrow F$. A complete truth table is given in figure 3.2.

What are the consequences of choosing these extensions ? The most obvious, and inevitable loss, is the excluded third: $E \vee \neg E$. But *False* implies everything feasible, and

$\Rightarrow$	$\bot$	*True* $\sqcap$ *False*	*True*	*False*	$\top$
$\bot$	*True*	*True*	*True*	*True*	$\top$
True $\sqcap$ *False*	*True*	*True*	*True*	*True*	$\top$
True	$\bot$	*True* $\sqcap$ *False*	*True*	*False*	$\top$
False	*True*	*True*	*True*	*True*	$\top$
$\top$	*True*	*True*	*True*	*True*	$\top$

Figure 3.2: Truth table for implication

everything implies *True*. The weakening ($E \Rightarrow E \vee F$) and strengthening ($E \wedge F \Rightarrow F$) laws hold for feasible E and F, but $\wedge$ and $\vee$ do not distribute over each other. The contrapositive law ($E \Rightarrow F \equiv \neg F \Rightarrow \neg E$) does not hold. As mentioned, implication is transitive: $((E \Rightarrow F) \wedge (F \Rightarrow G)) \Rightarrow (E \Rightarrow G)$. Implication distributes over $\vee, \wedge$ and $\Rightarrow$ in its second argument. The shunting law (similar to 'currying') holds: $(E \wedge F \Rightarrow G) \equiv (E \Rightarrow (F \Rightarrow G))$. Almost all $\Rightarrow$-monotonicity laws hold: for example, $(E \Rightarrow F) \Rightarrow (E \wedge G \Rightarrow F \wedge G)$. The symmetric closure of implication is neither equality nor equivalence. The 'consistency law' $(E \wedge F \equiv E) \equiv (E \vee F \equiv F)$ is not a tautology.

Predicate Calculus

Predicate calculus is the layer of logic that introduces variables and the binders $\forall$ and $\exists$ limiting the scope of the variables. In the presence of undefinedness, nondeterminacy, and miracles, two questions are raised. First, what do variables range over ? Secondly, what happens if the body of a quantification turns out to be (equivalent to) a 'strange' (that is, not simply *True* or *False*) boolean expression ?

Our variables range over outcomes, that is, proper outcomes and the special undefined outcome. The undefined outcome is the 'outcome' of a nonterminating expression. The justification is that we are dealing with a lazy language, and it is better to have all variables binding elements of the same set, regardless of whether they are bound by a λ or a quantifier. If one wants to exclude the undefined outcome from the range of the variable x in $\forall x.E$, one simply writes $\forall x.(x \not\equiv \bot) \Rightarrow E$ instead.

For the universal quantifier it means that from $\forall x.E$ we can deduce $E[F/x]$, for determined F, without having to check whether F is defined. However, it also means that from $\exists x.E$ we can deduce that there is an x such that E – but we don't know whether it is defined.

In answer to the second question, we retain universal quantification as generalised conjunction and existential quantification as generalised disjunction.

The meaning of $\forall x : T.E$ is constructed as follows. For each outcome of type T, interpret E with x bound to that outcome, to obtain an upward closed set. Order the obtained upward closed sets by transitive $<_\forall$, defined by

$$\{\} <_\forall \{\mathit{false}\} <_\forall \{\mathit{true}, \mathit{false}, \mathit{undefined}\} <_\forall \{\mathit{true}, \mathit{false}\} <_\forall \{\mathit{true}\}.$$

The least set is the meaning of $\forall x : T.E$.

The meaning of $\exists\, x : T.E$ is constructed similarly, except that we take the least set under the transitive order $<_\exists$, defined by

$$\{\} <_\exists \{\mathit{true}\} <_\exists \{\mathit{true}, \mathit{false}, \mathit{undefined}\} <_\exists \{\mathit{true}, \mathit{false}\} <_\exists \{\mathit{false}\}.$$

Notice that $<_\exists$ is not $>_\forall$, but that $S <_\forall T$ is equal to $\{\mathit{not}\ s \mid s \in S\} <_\exists \{\mathit{not}\ t \mid t \in T\}$ and vice versa. This equality makes De Morgan's laws valid.

What are the consequences of choosing these quantifiers ? Firstly, a quantification can be miraculous, undefined, or nondetermined. If the body does not mention the quantified variable, the quantification is equivalent to its body, that is, $(\forall\, x.E) \equiv E$ and $(\exists\, x.E) \equiv E$, if $x \notin \mathit{fv}(E)$. As mentioned, both De Morgan's laws hold: $\forall\, x.E \equiv \neg \exists\, x.\neg E$ and $\exists\, x.E \equiv \neg \forall\, x.\neg E$. Universal quantification distributes over conjunction, but not over disjunction, and existential quantification distributes over disjunction, but not over conjunction. Neither distributes over choice. Both quantifications are monotone in their bodies with respect to refinement, but not with respect to implication.

Alternatively, one could interpret $\forall\, x.E$ as $\{\mathit{true}\}$ if for every binding of x, the meaning of E is $\{\mathit{true}\}$, and otherwise as $\{\mathit{false}\}$. Similarly, $\exists\, x.E$ is $\{\mathit{true}\}$ if there is a binding of x that makes the meaning of E be $\{\mathit{true}\}$, and $\{\mathit{false}\}$ otherwise. These semantics are simpler, and have the property that quantifications are always defined and determined. But they lack many desirable properties, in particular, discarding quantifiers over unused variables, monotonicity, distribution properties, and De Morgan's laws.

Semantic Consequence

If the meaning of E is $\{\mathit{true}\}$, whatever the free variables in E are bound to, we write $\models E$ and say E is a *tautology*.

3.2.2 Proof Theory

This subsection describes what a formal proof is. We use Hilbert's axiomatic style, where the only inference rules are modus ponens and generalisation. In the next section of this chapter, we will list some boolean expressions as *axioms*. All those boolean expressions are called *theorems* that can be *proven*. A *proof* of expression G consists of a finite list of boolean expressions ending in G. The expressions are usually written one to a line, and numbered for easy reference. Each of the expressions is either an axiom, or is of the form $\forall\, x.E$ and E is an expression earlier in the list, or is of the form F and both expressions E and $E \Rightarrow F$ occur earlier in the list. That is, our only two inference rules are

$$\frac{E}{\forall\, x.E} \qquad \frac{E \qquad E \Rightarrow F}{F},$$

called 'generalisation' and 'modus ponens'. We write $\vdash G$ to express that G is a theorem.

The purpose of such a proof is to show that an expression is true by *syntactic* means rather than by examining its *meaning*.

To achieve this purpose, we must have the guarantee that the logic is *sound.* That means, all theorems are indeed tautologies, or in symbols, $\vdash E$ implies $\models E$. Anything which can be proven is actually true.

Ideally, we would also like the logic to be *complete.* That means, all tautologies are also theorems, or in symbols, $\models E$ implies $\vdash E$. Any true formula can be proven syntactically.

Soundness is relatively easy to show by showing that each of the axioms is a tautology, and that the two inference rules preserve tautologies. We claim that our calculus is sound. For many of the axioms the demonstration is trivial, and will be omitted; for others, a proof will be given later. It is unknown whether the calculus is complete.

3.2.3 Equational Reasoning

In practice, we prove formulas and refine expressions using the calculational style known as *equational reasoning.* It is advocated by many Dutch people (for example [DS90, GS93]), and it is nothing special really: it's just the calculational method taught in any school. In school it is applied to numbers, but we can equally use it for formulas and specifications. To prove that one expression is equivalent to another, the first expression is transformed into the second using as series of 'substitute equals for equals'. To show that one expression is less than another in some sense, we transform the smaller into the larger in a series of increasing steps. In school, the expressions are numbers, and the order is $\leq$, whereas here, the expressions are formulas with order $\Leftarrow$, or specifications with order $\sqsubseteq$.

Equational reasoning uses a linear layout that captures routine proof steps like applications of transitivity and modus ponens.

The layout of an equational proof is like this:

$$\begin{array}{lcl} & E_1 & \\ R_1 & & \mathit{hint}_1 \\ & E_2 & \\ R_2 & & \mathit{hint}_2 \\ & \vdots & \\ R_n & & \mathit{hint}_n \\ & E_{n+1} & \end{array}$$

This sequence proves $E_1 \ R \ E_{n+1}$ where R is $\Pi_{i=1}^{n} R_i$, the relational composition of the R_i.

In practice, R will be $\sqsubseteq, \equiv$, or $\Leftarrow$, and we have a proof of $E_1 \sqsubseteq E_{n+1}$, $E_1 \equiv E_{n+1}$, or $E_1 \Leftarrow E_{n+1}$. Furthermore, if R is $\Leftarrow$ and E_{n+1} is a theorem, then the sequence proves E_1.

Equational reasoning can be seen as a layer of shorthand over axiomatic logic. It is important because the style has a certain calculational dynamic.

We now sketch how equational reasoning is related to axiomatic logic. We'll treat the two cases of using $\sqsubseteq$ and $\Leftarrow$ for R.

Say the relation is $\sqsubseteq$. An equational proof with just one step:

$$\begin{array}{lll} & E_1 & \\ \sqsubseteq & & \textit{hint} \\ & E_2. & \end{array}$$

is translated to the trivial axiomatic proof

1) $E_1 \sqsubseteq E_2$ $--$ *hint.*

The *hint* will be a reference to the theorem of which $E_1 \sqsubseteq E_2$ is an instantiation. Alternatively, if E_1 is syntactically $E[F/x]$, for fresh x, and E_2 is $E[G/x]$, then the hint may also reference a theorem $F \sqsubseteq G$, and a theorem that E is monotone in x. If in fact $F \equiv G$, we can conclude $E_1 \equiv E_2$, whether E is monotone in x or not.

An equational proof with $n > 1$ steps:

$$\begin{array}{lll} & E_1 & \\ \sqsubseteq & & \textit{hint}_1 \\ & \vdots & \\ \sqsubseteq & & \textit{hint}_{j-1} \\ & E_j & \\ \sqsubseteq & & \textit{hint}_j \\ & \vdots & \\ \sqsubseteq & & \textit{hint}_n \\ & E_{n+1}; & \end{array}$$

is translated into an axiomatic proof by combining the axiomatic proofs of

$$\begin{array}{lll} & E_1 & \\ \sqsubseteq & & \textit{hint}_1 \\ \vdots & & \\ \sqsubseteq & & \textit{hint}_{j-1} \\ & E_j & \end{array} \quad \textit{and} \quad \begin{array}{lll} & E_j & \\ \sqsubseteq & & \textit{hint}_j \\ \vdots & & \\ \sqsubseteq & & \textit{hint}_{n-1} \\ & E_n. & \end{array}$$

Assume the axiomatic proof of the first has k lines, and the one of the second l lines. We simply renumber the lines of the second to $k+1..k+l$, and append them to the first axiomatic proof, and add a step at the end, getting:

$$\begin{array}{lll} 1) & \ldots & \\ \vdots & \vdots & \text{first axiomatic proof} \\ k) & E_1 \sqsubseteq E_j & \\ k+1) & \ldots & \\ \vdots & \vdots & \text{second axiomatic proof} \\ k+l) & E_j \sqsubseteq E_n & \\ k+l+1) & E_1 \sqsubseteq E_n & (k), (k+l), \sqsubseteq -\textit{trans.}, \end{array}$$

an axiomatic proof of $E_1 \sqsubseteq E_n$.

A similar translation procedure is applicable for the relation $\equiv$. Obviously, $\equiv$-steps can be mixed freely into $\sqsubseteq$-chains, since $(E \equiv F) \Rightarrow (E \sqsubseteq F)$ is a theorem.

For boolean expressions, we argue similarly, by chains of $\Leftarrow$, using $\Leftarrow$-(anti)monotonicity theorems and transitivity of $\Leftarrow$. Because $(E \equiv F) \Rightarrow (E \Leftarrow F)$ for feasible E, we can mix $\equiv$ steps into the $\Leftarrow$-chain, if – as is usually the case – the expressions are feasible.

3.3 The Axioms

In this section, we'll list the axioms. Where appropriate, we'll also slip in some theorems. Their proofs are left as exercises. The axioms are marked *axm* and the theorems *thm*. Sometimes we'll also give them a short name. As usual, the axioms and theorems are really axiom- and theorem-*schemas*. In them, the identifiers E, F, G, H, and occasionally X, A, B, stand for arbitrary expressions, whereas x, y, z stand for variables, and T, U, V stand for types. We assume that all expressions type-check. We assume the syntactic conveniences $E \Leftarrow F \equiv E \Rightarrow F$ and $(E \not\equiv F) \equiv \neg(E \equiv F)$.

The axioms are all 'simultaneous', of course, but for presentation, we'll group together the axioms concerning refinement, those concerning boolean expressions, those concerning specificational language constructs, and finally, those concerning programming language constructs. Each of the four groups is treated in a subsection.

3.3.1 Refinement

Feasibility, Determinacy, and Definedness

axm	$\alpha(\beta E)$,	where α, β range over **feas**, **det**, **def**
axm	$\neg\mathbf{def}\bot$, $\mathbf{det}\bot$, $\mathbf{feas}\bot$	
axm	$\mathbf{def}E \equiv (E \not\equiv \bot)$	
axm	$\mathbf{def}\top$, $\neg\mathbf{det}\top$, $\neg\mathbf{feas}\top$	
axm	$\mathbf{feas}E \equiv (E \not\equiv \top)$	

Refinement

axm		$\mathbf{def}(E \sqsubseteq F)$, $\mathbf{det}(E \sqsubseteq F)$, $\mathbf{feas}(E \sqsubseteq F)$
axm	$\sqsubseteq$ *–truth*	$((E \sqsubseteq F) \equiv \mathit{True}) \equiv (E \sqsubseteq F)$
axm	$\sqsubseteq$ *–extr.*	$\bot \sqsubseteq E$, $E \sqsubseteq \top$
axm	$\sqsubseteq$ *–refl.*	$E \sqsubseteq E$
axm	$\sqsubseteq$ *–antisym.*	$(E \sqsubseteq F) \wedge (F \sqsubseteq E) \equiv (E \equiv F)$
axm	$\sqsubseteq$ *–trans.*	$(E \sqsubseteq F) \wedge (F \sqsubseteq G) \Rightarrow (E \sqsubseteq G)$

Equivalence

thm		$\mathbf{def}(E \equiv F)$, $\mathbf{det}(E \equiv F)$, $\mathbf{feas}(E \equiv F)$
thm	$\equiv$ *–truth*	$((E \equiv F) \equiv \mathit{True}) \equiv (E \equiv F)$
thm	$\equiv$ *–refl.*	$E \equiv E$
thm	$\equiv$ *–sym.*	$(E \equiv F) \equiv (F \equiv E)$
thm	$\equiv$ *–trans.*	$(E \equiv F) \wedge (F \equiv G) \Rightarrow (E \equiv G)$
axm	*Leibniz*	$(E \equiv F) \Rightarrow (G[E] \equiv G[F])$

$G[E]$ stands for an expression with a syntactic hole, and E placed into that hole.

3.3.2 Boolean Expressions

In principle, the boolean type is just one of a number of primitive types. Each of the primitive types comes with a host of constants and some axioms about those constants. The axioms that every primitive type satisfies will be given later. In particular, they apply to the primitive type boolean. We discuss the boolean type with its associated constants $True, False, \neg, \vee, \wedge, =_{\mathbb{B}}$, and its language construct $\Rightarrow$ here already because it is fundamental in that it is the type of formulas. The functional constants $\neg, \vee, \wedge, =_{\mathbb{B}}$ also satisfy the axioms for all functions. The axioms about functions will be given later.

Two of the five truth values are picked out uniquely by the definedness-test and the feasibility-test. Of the remaining three, *True* $\sqcap$ *False* is identified by the determinacy-test. We could have introduced a truth-test to distinguish the remaining truth values *True* and *False*. Instead of doing that, we test for *True* by giving axioms involving $\equiv$ *True* for each (relevant) connective.

Propositional Logic

The constants that come with the primitive type $\mathbb{B}$ are the following:

$$\begin{array}{lcl} True, False & : & \mathbb{B} \\ \neg & : & \mathbb{B} \to \mathbb{B} \\ \vee, \wedge, =_{\mathbb{B}} & : & \mathbb{B} \to \mathbb{B} \to \mathbb{B}. \end{array}$$

The curried functions in the last line are written between their arguments, as usual. There is also the language construct implication, typed

$$\frac{E, F : \mathbb{B}}{E \Rightarrow F : \mathbb{B}}.$$

Implication and the constants satisfy these axioms:

axm	$\neg - \mathit{strict}$	$\mathbf{def}(\neg E) \equiv \mathbf{def}E$
axm	$\neg - \mathit{inv.}$	$\neg\neg E \equiv E$
axm	$\neg$	$\neg \mathit{True} \equiv \mathit{False}$
axm	$\vee - \mathbf{def}$	$\mathbf{def}(E \vee F) \equiv (\neg\mathbf{def}E \Rightarrow (\mathit{True} \sqsubseteq F)) \wedge (\neg\mathbf{def}F \Rightarrow (\mathit{True} \sqsubseteq E))$
axm	$\vee - \mathit{truth}$	$(E \vee F \equiv \mathit{True}) \equiv (E \equiv \mathit{True}) \vee (F \equiv \mathit{True})$
axm	$\vee - \mathit{idemp.}$	$E \vee E \equiv E$
axm	$\vee - \mathit{com.}$	$E \vee F \equiv F \vee E$
axm	$\vee - \mathit{assoc.}$	$(E \vee F) \vee G \equiv E \vee (F \vee G)$
axm	$\wedge - \mathit{def.}$	$E \wedge F \equiv \neg(\neg E \vee \neg F)$
axm	$\wedge - \mathit{truth}$	$(E \wedge F \equiv \mathit{True}) \equiv (E \equiv \mathit{True}) \wedge (F \equiv \mathit{True})$
axm	$\Rightarrow -\mathit{def.}$	$E \Rightarrow F \equiv (E \not\equiv \mathit{True}) \vee F$
axm	$\Rightarrow / \equiv$	$(E \Rightarrow (F \equiv G)) \equiv ((E \Rightarrow F) \equiv (E \Rightarrow G))$
axm	$\Rightarrow \mathit{weaken}$	$((E \equiv F) \wedge \mathbf{feas}F) \Rightarrow (E \Rightarrow F)$
thm	$\mathbf{def}/ \Rightarrow$	$\mathbf{def}(E \Rightarrow F) \equiv (E \not\equiv \mathit{True}) \vee \mathbf{def}F$
thm	$\mathbf{det}/ \Rightarrow$	$\mathbf{det}(E \Rightarrow F) \equiv (E \not\equiv \mathit{True}) \vee \mathbf{det}F$
thm	$\mathbf{feas}/ \Rightarrow$	$\mathbf{feas}(E \Rightarrow F) \equiv \mathbf{feas}F$
thm	$\Rightarrow -\mathit{truth}$	$((E \Rightarrow F) \equiv \mathit{True}) \equiv ((E \not\equiv \mathit{True}) \vee (F \equiv \mathit{True}))$

The condition **feas**F is necessary in the axiom $\Rightarrow$ *weaken*, since, with the interpretation of implication as discussed previously, $(E \equiv \top) \Rightarrow (E \Rightarrow \top)$ is infeasible rather than true.

Existential and Universal Quantification

Existential and universal quantification generalize disjunction and conjunction. Their axioms follow.

axm	$\exists -\mathbf{def}$	$\mathbf{def}(\exists x.E) \equiv (\exists x.\neg\mathbf{def}E) \Rightarrow (\exists x.(\mathit{True} \sqsubseteq E))$	
axm	$\exists -\mathbf{feas}$	$\mathbf{feas}(\exists x.E) \equiv \neg \exists x.\neg\mathbf{feas}E$	
axm	$\exists -\mathbf{det}$	$\mathbf{det}(\exists x.E) \equiv ((\exists x.\neg\mathbf{det}E) \Rightarrow \exists x.\neg\mathbf{def}E \vee (E \equiv \mathit{True})) \wedge \mathbf{feas}(\exists x.E)$	
axm	$\exists -\mathit{truth}$	$((\exists x.E) \equiv \mathit{True}) \equiv (\exists x.E \equiv \mathit{True}) \wedge \mathbf{feas}(\exists x.E)$	
axm	$\exists -\alpha$	$\exists x.E \equiv \exists y.E[x/y]$,	$y \notin \mathit{fv}(I$
axm	$\exists / \vee$	$\exists x.E \vee F \equiv \exists x.E \vee \exists x.F$	
axm	$\exists -\mathit{mon.}$	$(\forall x.E \sqsubseteq F) \Rightarrow (\exists x.E) \sqsubseteq (\exists x.F)$	
axm	$\exists -\mathit{inst.}$	$(E[F/x] \wedge \mathbf{det}F \Rightarrow \exists x.E) \Leftarrow \mathbf{feas}(\exists x.E)$	
axm	$\forall -\mathit{def.}$	$\forall x.E \equiv \neg \exists x.\neg E$	
axm	$\forall -\mathit{truth}$	$((\forall x.E) \equiv \mathit{True}) \equiv \forall x.(E \equiv \mathit{True})$	
thm	$\forall -\mathit{inst.}$	$((\forall x.E) \wedge \mathbf{det}F \Rightarrow E[F/x]) \Leftarrow \mathbf{feas}(E[F/x])$	
thm	$\forall -\mathit{mon.}$	$(\forall x.E \sqsubseteq F) \Rightarrow (\forall x.E \sqsubseteq \forall x.F)$.	

3.3.3 Specificational Language Constructs

This subsection gives the axioms of the specificational language constructs. Choice and generalized choice bring nondeterminacy into the language. The miracle buster is very simple: it converts miracles into undefined, and leaves everything else untouched. The miracle buster is not monotone with respect to refinement, but almost. Guards are the only language construct that introduce miracles (apart from $\top$ itself).

Choice

axm	$\sqcap - glb.$	$(X \sqsubseteq E) \wedge (X \sqsubseteq F) \equiv (X \sqsubseteq E \sqcap F)$
thm	$\sqcap - lb.$	$E \sqcap F \sqsubseteq E,\ E \sqcap F \sqsubseteq F$
thm	$\sqcap - idem.$	$E \sqcap E \equiv E$
thm	$\sqcap - com.$	$E \sqcap F \equiv F \sqcap E$
thm	$\sqcap - assoc.$	$(E \sqcap F) \sqcap G \equiv E \sqcap (F \sqcap G)$
thm	$\sqcap - dem.$	$E \sqcap \bot \equiv \bot$
thm	$\sqcap - one$	$E \sqcap \top \equiv E$
thm	$\sqcap - mon.$	$(E \sqsubseteq F) \Rightarrow (E \sqcap G \sqsubseteq F \sqcap G)$
thm		$\mathbf{def}(E \sqcap F) \equiv \mathbf{def}E \wedge \mathbf{def}F$
axm		$\mathbf{det}(E \sqcap F) \equiv (\mathbf{det}E \wedge (E \sqsubseteq F)) \vee (\mathbf{det}F \wedge (F \sqsubseteq E))$
thm		$\mathbf{feas}(E \sqcap F) \equiv \mathbf{feas}E \vee \mathbf{feas}F$
thm		$((E \sqcap F) \equiv \mathit{True}) \Leftarrow (E \equiv \mathit{True}) \wedge (\mathit{True} \sqsubseteq F)$

Generalized Choice

axm	$\bigsqcap - glb$	$(\forall x.X \sqsubseteq E) \equiv (X \sqsubseteq \bigsqcap x.E),$	$x \notin fv(X)$
thm	$\bigsqcap - inst.$	$\mathbf{det}F \Rightarrow (\bigsqcap x.E \sqsubseteq E[F/x])$	
thm	$\bigsqcap - drop$	$\bigsqcap x.E \equiv E,$	$x \notin fv(E)$
thm	$\bigsqcap - mon.$	$(\forall x.E \sqsubseteq F) \Rightarrow (\bigsqcap x.E \sqsubseteq \bigsqcap x.F)$	
thm	$\bigsqcap/\sqcap$	$\bigsqcap x.E \sqcap F \equiv \bigsqcap x.E \sqcap \bigsqcap x.F$	
thm	$\mathbf{def}/\bigsqcap$	$\mathbf{def}(\bigsqcap x.E) \equiv \forall x.\mathbf{def}E$	
thm	$\mathbf{feas}/\bigsqcap$	$\mathbf{feas}(\bigsqcap x.E) \equiv \exists x.\mathbf{feas}E$	
axm	$\bigsqcap - \alpha$	$\bigsqcap x.E \equiv \bigsqcap y.E[y/x],$	$y \notin fv(E)$
thm	$\bigsqcap - exch.$	$\bigsqcap x.\bigsqcap y.E \equiv \bigsqcap y.\bigsqcap x.E$	

Miracle Buster

axm	$\mathbf{if}\ \top\ \mathbf{fi} \equiv \bot$
axm	$\mathbf{feas}E \Rightarrow (\mathbf{if}\ E\ \mathbf{fi} \equiv E)$
thm	$\mathbf{feas}(\mathbf{if}\ E\ \mathbf{fi})$
thm	$\mathbf{def}(\mathbf{if}\ E\ \mathbf{fi}) \equiv \mathbf{def}E \wedge \mathbf{feas}E$
thm	$\mathbf{det}(\mathbf{if}\ E\ \mathbf{fi}) \equiv \mathbf{det}E \vee (E \equiv \top)$
thm	$\mathbf{if}\ E\ \mathbf{fi} \sqsubseteq E$
thm	$(E \sqsubseteq F) \wedge \mathbf{feas}F \Rightarrow (E \sqsubseteq \mathbf{if}\ F\ \mathbf{fi})$
thm	$(E \sqsubseteq F) \wedge \mathbf{feas}F \Rightarrow (\mathbf{if}\ E\ \mathbf{fi} \sqsubseteq \mathbf{if}\ F\ \mathbf{fi})$

Guards

axm		$True \rightarrow E \equiv E$	
axm		$\mathbf{feas}(G \rightarrow E) \equiv (G \equiv True) \wedge \mathbf{feas}E$	
thm	$\mathbf{det}/\rightarrow$	$\mathbf{det}(G \rightarrow E) \equiv (G \equiv True) \wedge \mathbf{det}E$	
thm	$\mathbf{def}/\rightarrow$	$\mathbf{def}(G \rightarrow E) \equiv G \Rightarrow \mathbf{def}E$	
thm	*use* $- \rightarrow$	$(G \Rightarrow (E \sqsubseteq F)) \Rightarrow (E \sqsubseteq G \rightarrow F)$	
thm	$\rightarrow -$ *mon.*	$(E \sqsubseteq F) \Rightarrow (G \rightarrow E \sqsubseteq G \rightarrow F)$	
thm	*strengthen* $- \rightarrow$	$(G \Rightarrow H) \Rightarrow (H \rightarrow E \sqsubseteq G \rightarrow E)$	
thm	$\rightarrow/\sqcap$	$G \rightarrow (E \sqcap F) \equiv G \rightarrow E \sqcap G \rightarrow F$	
thm	$\rightarrow/\bigsqcap$	$G \rightarrow \bigsqcap x.E \equiv \bigsqcap x.G \rightarrow E,$	$x \notin fv(G)$
thm	*add* $- \rightarrow$	$E \sqsubseteq G \rightarrow E.$	

Assertions

axm		$True \succ\!\!- E \equiv E$
axm		$\mathbf{def}(A \succ\!\!- E) \equiv (A \equiv True) \wedge \mathbf{def}E$
thm		$(A \succ\!\!- E \equiv \mathbf{if}\ A \rightarrow E\ \mathbf{fi}) \Leftarrow \mathbf{feas}E$
thm	*use* $- \succ\!\!-$	$(A \Rightarrow (E \sqsubseteq F)) \Rightarrow (A \succ\!\!- E \sqsubseteq F)$
thm	$\succ\!\!- -$ *mon.*	$(E \sqsubseteq F) \Rightarrow (A \succ\!\!- E \sqsubseteq A \succ\!\!- F)$
thm	*weaken* $- \succ\!\!-$	$(A \Rightarrow B) \Rightarrow (A \succ\!\!- E \sqsubseteq B \succ\!\!- E)$
thm	*rem.* $- \succ\!\!-$	$A \succ\!\!- E \sqsubseteq E.$

3.3.4 Programming Language Constructs

Primitive Types

Each of the primitive types $\mathbb{B}, \mathbb{Z}, \mathbb{R}, ...$ comes with a host of constants. Some are the elements of the type (for example *True*, 17, 7.659), others are functions on elements of the type (for example $\wedge, +, ln$). Each constant comes with a type rule, and possibly some axioms. Here we'll just list the axioms that apply to *every* primitive type. In particular, every primitive type T comes with a strict equality function, written $=_T$. We may drop the subscript if it is obvious from the context. Below, k and l range over distinct constants, whereas e, f, g range over *determined, defined, feasible* expressions of the primitive type T. In three axioms, the side condition $k : U \rightarrow V$ simply indicates that k should be of some functional type.

axm	*const.*	$\mathbf{def}k$, $\mathbf{det}k$, $\mathbf{feas}k$	
thm	*dist. const.*	$k \not\equiv l$	
axm	*dist. const.*	$k \not\sqsubseteq l$	
axm	*func. const.*	$\mathbf{det}(k\ E) \equiv \mathbf{det}E$,	*if* $k : U \to V$
axm	*func. const.*	$\mathbf{feas}(k\ E) \equiv \mathbf{feas}E$,	*if* $k : U \to V$
axm	*func. mon.*	$(E \sqsubseteq F) \Rightarrow (k\ E \sqsubseteq k\ F)$,	*if* $k : U \to V$
axm	$=$ *−strict*	$\mathbf{def}(E =_T F) \equiv \mathbf{def}E \wedge \mathbf{def}F$	
axm	$=$ *−refl.*	$e = e$	
axm	$=$ *−sym.*	$(e = f) \equiv (f = e)$	
axm	$=$ *−trans.*	$(e = f) \wedge (f = g) \Rightarrow (e = g)$	
axm	$= / \equiv$	$(e = f) \equiv (e \equiv f)$	
thm	$=$ *−Leibniz*	$(e = f) \Rightarrow G[e] \equiv G[f]$	

Tuples

As an example, we treat pairs. For other tuples, the appropriate generalizations apply. The last four lines should be generalized to all positions in a tuple, and all projections.

axm		$\mathbf{def}(E, F)$	
axm		$\mathbf{det}(E, F) \equiv \mathbf{det}E \wedge \mathbf{det}F$	
axm		$\mathbf{feas}(E, F) \equiv \mathbf{feas}E \wedge \mathbf{feas}F$	
axm	$\sqsubseteq$ *−tuple*	$(E \sqsubseteq F) \wedge (G \sqsubseteq H) \equiv ((E, G) \sqsubseteq (F, H))$	
thm	*tuple*/$\sqcap$	$(E, F \sqcap G) \equiv (E, F) \sqcap (E, G)$	
thm	*tuple*/$\bigsqcap$	$(E, \bigsqcap x.F) \equiv \bigsqcap x.(E, F)$,	$x \notin fv(E)$
axm	*proj. strict*	$fst\bot \equiv \bot$	
axm	*proj.*	$fst(E, F) \equiv E$	

Sums

As an example, we treat the binary sum type. The appropriate generalizations of the given axioms and theorems to other sum types apply. We also use **Inl** here as representative of all constructors. However, the **Inr** stands for any constructor different from **Inl**. The expression $(E \bigtriangledown F)G$ stands for the (very simple) **case** expression **case** G **of** $\mathbf{Inl}x \to E\ x \sqcap \mathbf{Inr}y \to F\ y$, where x and y are fresh.

axm		$\mathbf{def}(\mathbf{Inl}E)$
axm		$\mathbf{det}(\mathbf{Inl}E) \equiv \mathbf{det}E$
axm		$\mathbf{feas}(\mathbf{Inl}E) \equiv \mathbf{feas}E$
axm	$\sqsubseteq$ *−sum*	$(E \sqsubseteq F) \equiv (\mathbf{Inl}E \sqsubseteq \mathbf{Inl}F)$
axm		$(\mathbf{Inl}E \not\sqsubseteq \mathbf{Inr}F) \Leftarrow \mathbf{feas}F$
thm	*constr.*/$\sqcap$	$\mathbf{Inl}(E \sqcap F) \equiv \mathbf{Inl}E \sqcap \mathbf{Inl}F$
thm	*constr.*/$\bigsqcap$	$\mathbf{Inl}(\bigsqcap x.E) \equiv \bigsqcap x.\mathbf{Inl}E$
axm	**case** *strict*	$(E \triangledown F)\bot \equiv \bot$
axm	**case**	$(E \triangledown F)(\mathbf{Inl}G) \equiv E\ G$
axm	**case**	$(E \triangledown F)(\mathbf{Inr}G) \equiv F\ G$

Functions

Functions are formed from λ abstractions. We impose the restriction that the body of an abstraction must be monotone (with respect to refinement) in the variable. That is, for abstraction $\lambda x.E$ to be well formed, we require

$$(F \sqsubseteq G) \Rightarrow E[F/x] \sqsubseteq E[G/x].$$

The restriction is necessary to ensure consistency of the language, and it is used in proving some of the theorems below. If, in future work, the typing rules and axioms were joined in a unified deductive calculus, the restriction could be added to the typing rule for abstraction, along the lines of

$$\frac{x : T \vdash E : U \qquad (F \sqsubseteq G) \Rightarrow (E[F/x] \sqsubseteq E[G/x])}{\lambda x.E : T \to U}.$$

If the bodies of λ-abstractions were allowed to be not monotone in the variable, the language would become inconsistent. For example, take $f \mathrel{\hat{=}} \lambda x.\mathbf{if}\ (x \equiv \bot) \to \mathit{False} \sqcap (x \equiv 3) \to \mathit{True}\ \mathbf{fi}$. Then we would have

$$\begin{array}{ll} & \mathit{False} \\ \equiv & \qquad \text{axm } \beta \equiv\text{, since } \mathbf{det}\bot \\ & f\ \bot \\ \equiv & \qquad \text{thm } E \sqcap \bot \equiv \bot\text{, axm} \equiv \text{congr.} \\ & f(\bot \sqcap 3) \\ \equiv & \qquad \text{axm } \lambda/\sqcap \\ & f\ \bot \sqcap f\ 3 \\ \equiv & \qquad \text{axm } \beta \equiv\text{, since } \mathbf{det}\bot \text{ and } \mathbf{det}3 \\ & \mathit{False} \sqcap \mathit{True}. \end{array}$$

By generalizing this argument, we could show that in fact all expressions are equivalent! Therefore we require abstraction bodies to be monotone in their arguments.

The axioms and some theorems concerning functions are:

axm		$\mathbf{def}(\lambda x.E)$, $\mathbf{det}(\lambda x.E)$, $\mathbf{feas}(\lambda x.E)$	
axm	λ *−mon.*	$(\forall x.E \sqsubseteq F) \equiv (\lambda x.E \sqsubseteq \lambda x.F)$	
axm	*app.*/$\sqcap$	$E(F \sqcap G) \equiv E\ F \sqcap E\ G$	
axm	*app.*/$\bigsqcap$	$E(\bigsqcap x.F) \equiv \bigsqcap x.E\ F$,	$x \notin fv(E)$
axm	*app.*/$\sqcap$	$(E \sqcap F)G \equiv E\ F \sqcap E\ G$	
axm	*app.*/$\bigsqcap$	$(\bigsqcap x.E)F \equiv \bigsqcap x.E\ F$,	$x \notin fv(F)$
axm		$\bot\ E \equiv \bot \Leftarrow \mathbf{feas}E$	
axm	$\beta- \equiv$	$(E[F/x] \equiv (\lambda x.E)F) \Leftarrow \mathbf{det}F$	
thm	$\beta- \sqsubseteq$	$E[F/x] \sqsubseteq (\lambda x.E)F$	
thm	$\lambda - \alpha$	$\lambda x.E \equiv \lambda y.E[y/x]$,	$y \notin fv(E)$
thm	η	$\mathbf{def}E \Rightarrow (E \equiv \lambda x.E\ x)$,	$x \notin E,\ E : T \to U$
thm	$\lambda/\sqcap$	$\lambda x.E \sqcap F \sqsubseteq (\lambda x.E) \sqcap (\lambda x.F)$	
thm	$\lambda/\bigsqcap$	$\lambda x.\bigsqcap y.E \sqsubseteq \bigsqcap y.\lambda x.E$,	$x \notin fv(x)$
thm	*app. mon*	$(E \sqsubseteq F) \wedge (G \sqsubseteq H) \Rightarrow (E\ G \sqsubseteq F\ H)$.	

As theorems $\lambda/\sqcap$ and $\lambda/\bigsqcap$ show, function abstraction with a choice in the body is refined by a choice between function abstractions, and not necessarily equivalent. Therefore the language is 'truly nondeterministic' rather than 'underdetermined' in the senses of [War94]. For example, $\lambda x.x - 1 \sqcap x + 1 \sqsubseteq (\lambda x.x - 1) \sqcap (\lambda x.x + 1)$. The two expressions can produce different results when they are themselves arguments to a function, for example to $f \mathrel{\hat=} \lambda g.(g\ 0, g\ 0)$. We get

$$f\ (\lambda x.x - 1 \sqcap x + 1) \ \equiv\ (1,1) \sqcap (-1,1) \sqcap (1,-1) \sqcap (-1,-1)$$

and

$$f\ ((\lambda x.x - 1) \sqcap (\lambda x.x + 1)) \ \equiv\ (1,1) \sqcap (-1,-1).$$

Recursion

We impose the same restriction on the bodies of recursive expressions as on those of λ abstractions, that is, they must be monotone (with respect to refinement) in the variable. For recursive expression $\mu x.E$ to be well formed, we require

$$(F \sqsubseteq G) \Rightarrow E[F/x] \sqsubseteq E[G/x].$$

We require this because a recursive expression is defined as the fixpoint of the corresponding abstraction. We must ensure that that corresponding abstraction is well-formed.

The axioms say recursive expressions are fixpoints, and least prefixpoints. A recursive expression with a determined body is itself determined.

axm	μ *−fixp.*	$(\lambda x.E)(\mu x.E) \equiv \mu x.E$
axm	μ *−least prefixp.*	$((\lambda x.E)\ F \sqsubseteq F) \Rightarrow (\mu x.E \sqsubseteq F)$
axm	μ −**det**	$(\forall x.\mathbf{det}E) \Rightarrow \mathbf{det}(\mu x.E)$.

Introducing Recursion by Circular Refinement

Recursion is central to computation, and therefore introducing recursion is central to program derivation. For this reason, and because this thesis provides only incomplete theorems for introducing recursion, we'll discuss the topic in more depth.

We would like a law something like

$$(E \sqsubseteq F[E]) \Rightarrow (E \sqsubseteq \mu\, x.F[x]),$$

for fresh x, maybe with some conditions. This is similar to co-induction $(x \sqsubseteq f\ x) \Rightarrow (x \sqsubseteq Mf)$. But unfortunately the fixpoints are not the same: Mf is the *greatest* fixpoint of f, whereas recursion is modelled by the *least* fixpoint.

What if, in our proposed law above, the recursion doesn't terminate, that is, $\mu\, x.F[x] \equiv \bot$? This would be the case for $F[x] \equiv x$. From $E \sqsubseteq E$, which is true whatever E is, we would conclude $E \sqsubseteq \bot$, which makes the system inconsistent. Intuitively we must make sure that F is making progress, and that after a finite number of applications of F all the work is done. But how much work is there to be done ? It depends on the destructors of the type of E. We must do enough unfoldings to be able to apply any of the type's destructors.

Functions are an easy special case: the only function destructor (in the programming language) is function application. If we refine a function specification to a recursion, we must guarantee that for every possible argument, we can unfold the recursion often enough to apply the function to that argument. In other words, each unfolding must make progress. We can do that by requiring that on each recursive call of the function, the argument must decrease in size, in a well-founded order. Eventually the size of the argument will reach the bottom, it cannot be decreased any further and no further unfoldings are necessary. Usually the well-founded order is $<$ on the natural numbers.

thm *Intro. rec. func.* $(E \sqsubseteq F[\lambda\, y.y < x \succ\!\!- E[y/x]]) \Rightarrow (\lambda\, x.E \sqsubseteq \mu f.\lambda\, x.F[f])$,

where $x, y : T$, for which $<$ is a well-founded order, and $F[f]$ is monotone in f.

Law 29 in [War94] is almost the same, but it is only true for at most one occurrence of the hole in expression $F[\,]$. This form of the theorem is essentially due to J. Morris [unpublished]. It follows from well-founded induction

$$\frac{\forall\, x.(\forall\, y.y < x \Rightarrow E[y]) \Rightarrow E[x]}{\forall\, x.E[x]}.$$

Every programmer knows that to make a well-defined recursive function, the argument of the recursive call must decrease. The theorem captures this strategy formally.

When the x reaches a minimal value, there is no y such that the assertion $y < x$ is true. Therefore the assumption becomes $E \sqsubseteq F[\lambda\, y.\bot]$. Intuitively, F must satisfy E without using a recursive call.

The theorem makes no restriction of the shape of E, that is, it covers all kinds of recursion: one or many recursive calls, tail-recursion or non-tail-recursion, linear or nonlinear recursive calls. It could be specialised to any of these kinds of recursion. The concise practical form of the law however makes this unnecessary.

Just for comparison, here is one specialisation: the invariance theorem of the imperative refinement calculus (for instance in chapter 5 of [Mor94]). It is used to refine a specification statement by a *while* loop. The desired postcondition is factored into an invariant and a termination condition. Rewritten in functional notation, the law reads

$$\begin{array}{ll} \lambda x. & \mathit{inv}\ x \succ\!\!- \\ & \mathbf{if}\ \sqcap z.\mathit{inv}\ z \wedge \mathit{done}\ z \rightarrow z\ \mathbf{fi} \end{array} \sqsubseteq \begin{array}{lll} \mu f.\lambda x.\mathit{inv}\ x \succ\!\!- & \mathbf{if} & \mathit{done}\ x \rightarrow x \\ & \sqcap & \neg\mathit{done}\ x \rightarrow f(\mathit{body}\ x) \\ & \mathbf{fi} & \end{array}$$

if

1. The argument size is in a well-founded set, for example $\mathbb{N}$.
2. The arguments decrease, that is, $\mathit{inv}\ x \wedge \neg\mathit{done}\ x \Rightarrow \mathit{body}\ x < x$.
3. The body of the loop preserves the invariant, that is $\mathit{inv}\ x \wedge \neg\mathit{done}\ x \Rightarrow \mathit{inv}(\mathit{body}\ x)$.

The imperative *while* loop corresponds to single tail recursion in a two-branch conditional.

For types other than functions we can introduce recursion by a refinement that is not valid in general, but is valid in a particular surrounding context. If we can show that in one particular context j unfoldings are enough, for some natural j, we can introduce recursion into that context.

$$\mathit{thm}\quad \mu/\mathit{context}\quad (E \sqsubseteq F[E]) \wedge (\exists j : \mathbb{N}.P[F^j[E]] \sqsubseteq P[F^j[\bot]]) \Rightarrow (P[E] \sqsubseteq P[\mu x.F[x]]),$$

if $P[\,]$ and $F[\,]$ are monotone. The proof use induction on the natural numbers. The notation $F^i[E]$ is defined by $F^0[E]$ is E, and $F^{n+1}[E]$ is $F[F^n[E]]$.

An Operational Law about Recursion

Say we have an expression E that is a program, or a subexpression of a program. We transform E into $F[E]$ where F is an expression with zero or more occurrences of a syntactic hole in it, and $F[E]$ is F with E put into the hole. In this transformation we use beta reductions $\xrightarrow{\beta}$ and **let** reductions $\xrightarrow{let}$ and μ unfoldings $\xrightarrow{\mu}$. We denote such a transformation by $\longrightarrow$ and call it 'reduction'.

$$\begin{array}{lll} \mathit{axm} & (\lambda x.E)\ F \xrightarrow{\beta} E[F/x], & \mathit{if}\ \mathbf{det}F \\ \mathit{axm} & \mathbf{let}\ x \hat{=} F\ \mathbf{in}\ E \xrightarrow{let} E[F/x], & \mathit{if}\ \mathbf{det}F \\ \mathit{axm} & \mu x.E \xrightarrow{\mu} (\lambda x.E)(\mu x.E) & \end{array}$$

Reduction is the smallest relation between expressions that includes β reductions, **let** reductions, and μ unfoldings, is transitive, and is a congruence.

$$\begin{array}{ll}
\mathit{axm} & (E_1 \xrightarrow{\beta} E_2) \Rightarrow (E_1 \longrightarrow E_2) \\
\mathit{axm} & (E_1 \xrightarrow{\textbf{let}} E_2) \Rightarrow (E_1 \longrightarrow E_2) \\
\mathit{axm} & (E_1 \xrightarrow{\mu} E_2) \Rightarrow (E_1 \longrightarrow E_2 \\
\mathit{axm} & (E_1 \longrightarrow E_2) \wedge (E_2 \longrightarrow E_3) \Rightarrow (E_1 \longrightarrow E_3) \\
\mathit{axm} & (E_1 \longrightarrow E_2) \Rightarrow (F[E_1] \longrightarrow F[E_2])
\end{array}$$

Obviously $E \longrightarrow F$ implies $E \equiv F$. Therefore, if for some E and F we know $E \longrightarrow F[E]$ we can conclude $E \equiv F[E] \equiv F[F[E]] \equiv ... \equiv F^j[E]$ for any natural j. E is a fixpoint of F.

Reduction captures progress in an operational semantics, namely progress in the evaluation of a program. We postulate

$$\mathit{axm} \quad \mathit{operational\ recursion} \quad (E \longrightarrow F[E]) \Rightarrow (E \equiv \mu\, x.F[x]), \quad x \notin \mathit{fv}(F).$$

E must be a program or a subexpression of a program, since operational semantics only make sense for an executable language, not for a specification language.

The hypothesis $E \longrightarrow F[E]$ can only be fulfilled if the transformation includes at least one μ unfolding $\xrightarrow{\mu}$. This is because $E \longrightarrow F[E]$ means there is an infinite reduction sequence starting at E. But in a typed λ-calculus-like language with recursion, only unfolding recursion can lead to such sequences.

Refining Recursion

We know how to unfold a recursive expression. More generally we can partially unfold a recursive expression that has a function composition as its body. This law is called the 'rolling rule'. It follows from the characterisation of recursion.

$$\mathit{thm} \quad \mathit{Rolling\ rule} \quad \mu\, x.(f \circ g)x \equiv f(\mu\, y.(g \circ f)y)$$

A doubly recursive expression is equivalent to a singly recursive one. This law is called the 'diagonal rule'. It follows from the characterisation of recursion.

$$\mathit{thm} \quad \mathit{Diagonal\ rule} \quad \mu\, x.\mu\, y.x \oplus y \equiv \mu\, x.x \oplus x,$$

for binary function $\oplus$.

Here's an example application of the diagonal rule. Define $mg : [a] \rightarrow [a] \rightarrow [a]$ by $mg(a : as)(b : bs) \mathrel{\hat{=}} a : b : mg\ as\ bs$. The two expressions $\mu\, t.1 : 2 : mg\ t\ t$ and $\mu\, t.\mu\, x.1 : 2 : mg\ t\ x$ are equivalent. They both yield an infinite list starting $1:2:1:1:2:2:1:1:1:1:2:2:2:2:....$

The recursive expression $\mu\, x.E$ is monotone in E. Therefore we may refine it stepwise. This theorem follows from the characterisation of recursion.

thm μ*–mon.* $(\forall x.E \sqsubseteq F) \Rightarrow (\mu x.E \sqsubseteq \mu x.F)$

Let Expressions

For the **let** expression **let** $x = E$ **in** F to be well-formed, we require that F be monotone in x. Without this restriction, the language would become inconsistent, as a similar argument to the one given in the subsubsection about functions shows.

The axioms for **let** expressions follow.

axm	*fold* let	$(E[F/x] \equiv \textbf{let } x = F \textbf{ in } E) \Leftarrow \textbf{det} F$	
thm	*fold* let	$E[F/x] \sqsubseteq \textbf{let } x = F \textbf{ in } E$	
axm	*func.*/let	$E(\textbf{let } x = F \textbf{ in } G) \equiv \textbf{let } x = F \textbf{ in } E\ G,$	$x \notin fv(E)$
axm	let /*mon.*	$(\forall x.E \sqsubseteq F) \land (G \sqsubseteq H) \Rightarrow (\textbf{let } x = E \textbf{ in } G \sqsubseteq \textbf{let } x = F \textbf{ in } H)$	
axm	let /$\sqcap$	$\textbf{let } x = (E \sqcap F) \textbf{ in } G \equiv (\textbf{let } x = E \textbf{ in } G) \sqcap (\textbf{let } x = F \textbf{ in } G)$	
axm	let /$\sqcap$	$\textbf{let } x = E \textbf{ in } (F \sqcap G) \equiv (\textbf{let } x = E \textbf{ in } F) \sqcap (\textbf{let } x = E \textbf{ in } G)$	
axm	let /$\sqcap$	$\textbf{let } x = \sqcap y.E \textbf{ in } F \equiv \sqcap y.(\textbf{let } x = E \textbf{ in } F),$	$y \notin fv(F)$
axm	let /$\sqcap$	$\textbf{let } x = E \textbf{ in } \sqcap y.F \equiv \sqcap y.(\textbf{let } x = E \textbf{ in } F),$	$y \notin fv(E)$
axm	let /$\bot$	$\textbf{let } x = E \textbf{ in } \bot \equiv \bot$.	
axm	let /α	$\textbf{let } x = E \textbf{ in } F \equiv \textbf{let } y = E \textbf{ in } F[y/x],$	$y \notin fv(F).$

3.3.5 Discussion

Recursion

The characterisation of recursion as a fixpoint and the least prefixpoint is the Knaster-Tarski theorem. The Knaster-Tarski theorem applies to monotone functions between complete lattices. In the specification language the functions are indeed monotone (ensured syntactically), and there is a least expression $\bot$, a greatest expression $\top$, and a binary greatest lower bound operation, namely choice. However, for a complete lattice we would need expressions that are least upper bounds and greatest lower bounds for any set of expressions. If desired, a least upper bound language construct can be added to the language. It would model angelic choice.

The paper [oPCG95] by the Eindhoven group gives a calculational treatment of fixpoints. The names 'rolling rule' and 'diagonal rule' are taken from it.

Operational recursion

This axiom is quite different. All the other axioms can be shown sound using the denotational semantics of chapter 9. But the operational recursion law deals with expressions as snapshots in the evaluation of a program. Evaluating an expression is rewriting it, using a set of rewrite rules that include $\longrightarrow$. During such an evaluation the denotation never changes. So the denotational semantics of chapter 9 can't be used to prove this law sound. It would have to be proven sound with respect to a rewriting operational semantics. This semantics would only cover the programming sublanguage of the specification language, since evaluation of an un-algorithmic specification is meaningless.

As defined $\longrightarrow$ implies equivalence. If its definition included $E \sqcap F \longrightarrow E$, the $\longrightarrow$ would imply only $\sqsubseteq$. Such operational semantics are treated for example in [dP92].

3.4 Comparison to Previous Work on 3-Valued Logics

Classical logic knows two truth values: *True* and *False*. In order to reason about partial functions such as in 3/0 or recursively defined computations, many researchers have extended classical 2-valued logic to 3-valed logics, the new extra value being $\perp$, a representation for undefined or nonterminating expressions. This section presents a brief selective survey of such work. However, in order to reason about the nondetermined and infeasible expressions occurring in specifications, we have found it necessary to add the further truth values *True* $\sqcap$ *False* and $\top$, producing a 5-valued logic. In this survey, we draw from [CJ91].

A third truth value beside *True* and *False* is motivated by apparently meaningless expressions. These expressions arise from partial functions such as in 3/0, *ln* 0, *head* [ ]. In program development, such expressions are quite natural and the theory must deal with them. Of course they are 'detectable at run-time' and therefore could be treated by some exception mechanism, for example the exception monad (for example in [Wad92c]). Whereas such a treament may be desirable in a program, it is too cumbersome for development of programs, and furthermore is not usable for the other source of undefined expressions, nonterminating recursive programs. Recursion (in some form) is the essence of any programming language, but sadly, it introduces programs that don't terminate and – as expressions – are undefined. This kind of undefinedness cannot be caught, and so we must be able to reason about nonterminating programs.

One way of extending the logical operators to cover three values was proposed in [McC67]. It suggested conditional versions of conjunction (*cand*), disjunction (*cor*), and implication (*cimp*) that may deliver a defined outcome although their second argument is not defined. These operators are based on a lazy conditional operator **if then else** with the following properties

$$\begin{aligned}
\mathbf{if}\ \perp\ \mathbf{then}\ E\ \mathbf{else}\ F &\equiv \perp \\
\mathbf{if}\ \mathit{True}\ \mathbf{then}\ E\ \mathbf{else}\ F &\equiv E \\
\mathbf{if}\ \mathit{False}\ \mathbf{then}\ E\ \mathbf{else}\ F &\equiv F.
\end{aligned}$$

The conditional logical operators are given these definitions:

$$\begin{aligned}
E\ \mathit{cand}\ F &\mathrel{\hat{=}} \mathbf{if}\ E\ \mathbf{then}\ F\ \mathbf{else}\ \mathit{False} \\
E\ \mathit{cor}\ F &\mathrel{\hat{=}} \mathbf{if}\ E\ \mathbf{then}\ \mathit{True}\ \mathbf{else}\ F \\
E\ \mathit{cimp}\ F &\mathrel{\hat{=}} \mathbf{if}\ E\ \mathbf{then}\ F\ \mathbf{else}\ \mathit{True}.
\end{aligned}$$

These operators are monotone, and are convenient in a programming language. They are provided in several versions of imperative languages, and ML [HMT88, MTH90], and are usual in lazy functional languages such as Haskell [PH$^+$96]. They are used alongside the classical operators in [Jon72] and [Dij76]. In calculations they are cumbersome, for although they have some familiar properties like forms of De Morgan's law and left-distribution of *cand* over *cor*, other familiar properties like commutativity and right-distribution of *cand* over *cor* are missing. The operators have an implicit left-ro-right evaluation order – which is of little guidance when it comes to interpreting universal and existential quantification, usually the generalization of conjunction and disjunction.

A different approach is taken by [Luk20] and many later researchers. It is to use the maximal monotone extensions of classical conjunction and disjunction. The truth tables are as in figure 3.1. These operators are pleasingly commutative, and imply no evaluation order. Furthermore they can be generalized naturally to universal and existential quantification. These operators are taken up by [Kle52]. Their proof theory is studied in [Kol76, Kol81], where it is shown that they can express all monotone functions $\{\mathit{True}, \mathit{False}, \bot\}^n \to \{\mathit{True}, \mathit{False}, \bot\}$ for any natural n. They are taken up in [BCJ84] which describes LPF, the "Logic of Partial Functions", that underlies the program development method VDM [Jon86, Jon90, JS90a].

In LPF, which is presented in natural deduction style, there are typically two introduction and two elimination rules for each operator. For instance the disjunction-introduction rules are

$$\vee - I\frac{E_i}{E_1 \vee ... \vee E_n} \qquad \neg\vee - I\frac{\neg E_1, ..., \neg E_n}{\neg(E_1 \vee ... \vee E_n)},$$

where $1 \leq i \leq n$. This approach is necessary because – as in every 3-valued logic – the "middle" is not excluded: $E \vee \neg E$ is not a tautology.

LPF is formalised as a first order predicate calculus with equality with a set-theoretical semantics in the thesis [Che86]. A typed version of it is proposed in [Jon90] for which [JM94] provides the formal basis.

The implication of LPF retains its usual classical connection to negation and disjunction:

$$E \Rightarrow F \ \equiv\ \neg E \vee F.$$

However this implication does not support the deduction theorem; a definedness hypothesis must be added:

$$\frac{E \vdash F \qquad \delta E}{E \Rightarrow F},$$

where $\delta E \mathrel{\hat{=}} E \vee \neg E$. Furthermore, implication is not reflexive or transitive.

These weaknesses are not acceptable for equational reasoning; therefore [Mor96a] proposes a logic based on LPF, with a different implication. It is the same implication as in this thesis, and it is indeed reflexive and transitive, and does support the deduction theorem.

[Mor96b] develops this logic further by showing that a nondetermined truth value (like our *True* $\sqcap$ *False*) can be added at almost no extra cost, thereby making the logic useful for settings in which nondeterminacy can occur, for instance in program derivation.

The present logic is a further extension adding $\top$ as a fifth truth value. The miraculous truth value has much less intuitive meaning and brings the number of truth values up to an unpleasant 5. However, miracles do arise fairly naturally as overspecified expressions – and they can be contained by simple syntactic measures. In practice, a proof of **feas**E is a trivial syntactic check.

Chapter 4

Imperative Expressions

4.1 Imperative Expressions

Imperative programming techniques are made possible in the specification language by the imperative expressions of the state monad. This section describes the state monad informally.

4.1.1 What is State ?

We regard a *state* as a mapping from references to values. A *reference* to a value is the address of, or a pointer to, a value. We can change the value to which a reference points, that is, change what the state maps that reference to, and still the reference itself stays unchanged. If a reference has type $Ref\ s\ a$, then the value it references has type a. Here the type s identifies which state the reference belongs to. There may be doubt since there may be more than one state in a program.

A state may be changed by special expressions called *state transformers.* Apart from delivering a value, like all expressions, these special expressions also take a state and deliver a state. Their type is $ST\ s\ a$, where a is the type of the value. We say a state transformer of type $ST\ s\ a$ *returns* a value of type a. Again, s identifies which state is transformed. It plays a purely technical role in forbidding certain forms of erroneous programs. There's only one interesting instantiation of it: the abstract type *RealWorld*, whose values represent the real world. It is convenient to use type synonyms:

$$\begin{aligned}\textbf{type}\ IO\ a &\mathrel{\hat{=}} ST\ RealWorld\ a\\ \textbf{type}\ RefIO\ a &\mathrel{\hat{=}} Ref\ RealWorld\ a.\end{aligned}$$

A state transformer of type $IO\ a$ is an *IO-transformer.* It can perform reference accesses like other state transformers, and in addition, it can perform changes to the real world, that is, it can perform IO operations. This specialisation of state transformers to IO transformers is presented in [LJ94].

In a program the state may only be manipulated via the state transformer primitives. It can never be bound to a variable. This ensures that the state cannot be duplicated, and thus it may be implemented directly using the memory of the machine.

Technically, state is treated like an abstract data type. The state transformer primitives themselves are available in the programming language, but not their definitions. However, for explaining the state transformer primitives it is useful to give them 'informal definitions'. They are informal in that their purpose is to give an operational understanding of the state transformer primitives. The state transformer primitives are of course formally defined by their axioms. In fact, the informal definitions given here do not capture garbage collection, as discussed later, and therefore, strictly, are not a model for the axioms. In these informal definitions (and in calculations with state later, but never in programs), we do allow the state to be bound to a variable (we use σ, τ, v). We make an exception to the requirement that every expression has a type: the state itself has no type. If such a type *State* existed, we could define $ST\ s\ a$ in terms of it:

$$ST\ s\ a \quad \hat{=} \quad State \rightarrow a \times State.$$

The type variable s does not occur on the right. Its role will be explained in subsection 4.1.3.

4.1.2 Primitive State Transformers

Here are five primitives that make state transformers. The state transformers create a *new* reference, *put* a value into the place to which the reference points, *get* the value to which a reference points, and *return* a value without using the state, or while *breaking* the state:

Definition 1 (Primitive state transformers)

$$\begin{array}{rcl}
new & : & a \rightarrow ST\ s\ (Ref\ s\ a) \\
new\ a & \hat{=} & \lambda\sigma.\sqcap v.(v \in Ref\ s\ a \setminus \mathrm{dom}\ \sigma) \rightarrow (v, \sigma[v \mapsto a]) \\
put & : & Ref\ s\ a \rightarrow a \rightarrow ST\ s\ () \\
put\ v\ a & \hat{=} & \lambda\sigma.((), \sigma[v \mapsto a]) \\
get & : & Ref\ s\ a \rightarrow ST\ s\ a \\
get\ v & \hat{=} & \lambda\sigma.(\sigma\ v, \sigma) \\
return & : & a \rightarrow ST\ s\ a \\
return\ a & \hat{=} & \lambda\sigma.(a, \sigma) \\
break & : & a \rightarrow ST\ s\ a \\
break\ a & \hat{=} & \lambda\sigma.(a, \perp_{State}).
\end{array}$$

In all types the type variables a and s are (implicitly) universally quantified. As usual in functional programming literature, we use a in two roles: In the types, it is a type variable, and in the definitions, it is a variable (of that type). The symbol $\perp_{State}$ stands for the undefined state.

The state transformer *new a* creates a new references in the state, and stores a at the location it points to. It returns this new reference. The new reference is of type *Ref s a*. Any reference of that type will do, as long as it was not already used in the state. Therefore *new* $a\ \sigma$ is nondetermined. We make the reasonable assumptions that the set of available references *Ref* $s\ a$ is infinite, whereas the set of allocated references in any state is finite. In practical terms, this is assuming that we never try to allocate more memory than is available. Therefore *new* $a\ \sigma$ is feasible.

The state transformer *put* $v\ a$ changes the state by storing a at the location that reference v pointed to. Otherwise the state is left the same. The state transformer *put* $v\ a$ has nothing interesting to return. So we'll just let it return the empty tuple ().

The state transformer *get* v returns the value stored under the reference v. It leaves the state unchanged. The state transformer *return* a simply returns a and leaves the state unchanged.

The state transformer *break* a returns its argument a and destroys the state. This is not useful in programs, but in program calculation. We have *break* $\sqsubseteq$ *return* .

4.1.3 Composing and Encapsulating State Transformers

Two state transformers may be composed in sequence by the infix primitive semicolon. Semicolon is a primitive, but for explanation, we give an 'informal definition':

Definition 2 (Semicolon)

$$\begin{aligned} ; \quad &: \quad ST\ s\ a \to (a \to ST\ s\ b) \to ST\ s\ b \\ m;\ k \quad &\hat{=} \quad \lambda\ \sigma.\mathbf{let}\ (a, \sigma') = m\ \sigma\ \mathbf{in}\ k\ a\ \sigma'. \end{aligned}$$

If m is a state transformer and k is a state-transformer valued function, then $m;\ k$ is a state transformer. It takes a state and passes the state to m. The resulting value and state are passed to k.

The triple $(ST\ \ s, \mathit{return}, ;\ )$ is the state monad of [Wad92b], and it satisfies the monad laws:

axm	*left return*	$\mathit{return}\ E;\ F \equiv F\ E$
axm	*right return*	$E;\ \mathit{return}\ \equiv E$
axm	; *assoc.*	$E;\ (\lambda x.F;\ G) \equiv (E;\ \lambda x.F);\ G,\ \ x$ not free in G.

As an example of composing state transformers, the definition of the combinator *modify*, which applies a function to the contents of a reference, is:

$$\begin{aligned} \mathit{modify} \quad &: \quad (a \to a) \to \mathit{Ref}\ s\ a \to ST\ s\ () \\ \mathit{modify}\ f\ v \quad &\hat{=} \quad \mathit{get}\ v;\ \lambda\, a.\mathit{put}\ v\ (f\ a). \end{aligned}$$

A state transformer may be made into a state-less expression by applying **run** to it.

Definition 3 (Run)

$$\mathbf{run}\ k \quad \hat{=} \quad \sqcap\sigma.(\bot_{State} \not\equiv \sigma) \to \mathit{fst}(k\ \sigma).$$

The expression **run** k applies the state transformer k to an arbitrary proper state, thereby obtaining a value-state pair. The state is discarded, and the value is the outcome of **run** k. We say that **run** *encapsulates* the imperative program k. Since proper states exist, the expression **run**k is feasible.

If $k : \forall s.ST\ s\ a$, then **run** $k : a$. The $\forall$ captures that k must be a state transformer for an arbitrary state. For instance *get* v would not do, since its type is $ST\ s\ a$ for one particular s, not for any s. The state transformer *new* 13 would also not do. It can take any state, but it returns a reference created in that state. The type of what it returns is $Ref\ s\ \mathbb{N}$, which contains s and therefore cannot be matched to a. A reference may be imported into the state thread encapsulated by **run**, but such an imported reference cannot be accessed. So

$$\mathbf{run}(new\ 13;\ \lambda v.return\ (\mathbf{run}(new\ v;\ get\ );\ \lambda v.get\ ))$$

has type $\mathbb{N}$, but

$$\mathbf{run}(new\ 13;\ \lambda v.return\ (\mathbf{run}(get\ v)))$$

cannot be typed. No reference may be exported from the thread it was created in. The typing rule for **run** also prohibits encapsulating an IO transformer, since an IO transformer has type $IO\ a$, which is $ST\ RealWorld\ a$.

The typing rule of **run** means that **run** is a language construct and not a primitive constant. As constant its type would be $\forall a.(\forall s.ST\ s\ a) \to a$. But we are using the Hindley-Milner type system, in which all quantifications must be at the outside.

4.1.4 State Readers

[Wad92a] presents a monad closely related to the state transformer monad: the *state reader monad.* An expression of type $SR\ s\ a$ takes a state (indexed by the type s) and delivers a value of type a. The value a state reader delivers may depend on the state, but a state reader cannot change the state.

We won't define any combinators for the type, except for the combinator *ro* (for "read-only"). Its typing and its informal definition are

Definition 4 (ro)

$$\begin{aligned} ro &\ :\ SR\ s\ a \to ST\ s\ a \\ ro\ r &\ \hat{=}\ \lambda\sigma.(r\ \sigma, \sigma). \end{aligned}$$

4.1.5 Assertions and Guards with State

Sometimes we would like to put an assertion or a guard before a state transformer. In writing the assertion or guard, sometimes we would like to refer to the state. For that purpose we introduce two new language constructs. In effect they are just asserting and guarding 'lifted'. Here are their informal definitions.

Definition 5 (Assertions and guards with state)

$$(A \mathrel{\hat{\succ\!\!-}} E)\ \sigma \ \hat{=}\ A\ \sigma \succ\!\!- E\ \sigma$$

$$(G \hat{\rightarrow} E)\,\sigma \;\hat{=}\; G\,\sigma \dashrightarrow E\,\sigma$$

The typing rules are as expected. If $A : SR\ s\ \mathbb{B}$ and $E : ST\ s\ T$, then $A \hat{\succ\!\!-} E : ST\ s\ T$, and similar for the guards.

4.1.6 Discussion

Notation

Good notation is an issue here. The paper [Wad92a] presents monads as a generalisation of list comprehensions, and uses monad comprehensions. Monad comprehensions are adapted in [Lau93, KL93] as the do-notation, which was adopted in Gofer [Jon93] and Haskell 1.3 [PH+96]. Specialised to the state monad (a monad without a *zero* : $ST\ s$ ()) the do-notation works like this. We have the following additional form of expressions:

$$\begin{array}{rcl} \textit{expression} & ::= & \mathbf{do}\{\textit{qualifier};\ ...;\ \textit{qualifier};\ \textit{expression}\} \\ \textit{qualifier} & ::= & \textit{pattern} \leftarrow \textit{expression} \\ & | & \textit{expression}. \end{array}$$

The do-notation is translated into the *bind*, *return* notation by repeatedly applying the following rules. We'll use "*bind*" where we'd normally use "; " to avoid confusion with the different ; in the do-notation.

$$\begin{array}{rcl} \mathbf{do}\{E\} & \equiv & E \\ \mathbf{do}\{P \leftarrow E;\ F\} & \equiv & E\ \textit{bind}\ \lambda P.\mathbf{do}\{Q\} \\ \mathbf{do}\{E;\ Q\} & \equiv & E\ \textit{bind}\ \lambda\_.\mathbf{do}\{Q\} \end{array}$$

In the do-notation a composition of a state transformer with another that ignores the result of the first is written very lightly. In our notation we often have $\lambda\_$.. But that is not a great failing of our notation, and following Haskell's $\gg\_ : ST\ s\ a \dashrightarrow ST\ s\ b \dashrightarrow ST\ s\ b$, we could easily define such a composition.

It is claimed in [MJ95] that the do-notation makes programs look more comprehensible. This repeats the pun of the title of [Wad92a], "Comprehending Monads". A program written in do-notation is more familiar to readers used to imperative languages like Pascal. But for calculations the notation is inappropriate because it tears apart a bound variable and its scope, just like assignments do. The ";" in the do-notation splits the abstract syntax entity λ abstraction in two. Calculations become convoluted because of constant translation in and out of the do-notation. For comparison, here are the monad laws written in the do-notation.

$$\begin{array}{lcll} \mathbf{do}\{x \leftarrow \textit{return}\ E;\ F\ x\} & \equiv & F\ E, & x \text{ not free in } F \\ \mathbf{do}\{x \leftarrow E;\ \textit{return}\ x\} & \equiv & E & \\ \mathbf{do}\{y \leftarrow \mathbf{do}\{x \leftarrow E;\ F\ x\};\ G\ y\} & \equiv & \mathbf{do}\{x \leftarrow E;\ y \leftarrow F\ x;\ G\ y\}, & x \text{ not free in } G. \end{array}$$

The third line in particular is convoluted and doesn't display the essential property of *bind*.

How to Calculate with State ?

In the program language, state is manipulated by the state monad operations. Using only the state monad primitives guarantees that the state is never duplicated and therefore it can be implemented by the real state of the memory of a computer, and the real state of the world surrounding the computer. In specifications outside of the program language this guarantee is not needed: After all, non-program specifications only exist on paper anyway. In some calculations, for example in chapter 7, always calculating at the level of state transformers is enough. We never mention the state explicitly. But for more involved use of state, for instance the dynamic use of state in chapter 8, it is necessary to manipulate the state directly. Therefore, in non-program specifications, we allow the state to be bound to a variable. Then it becomes possible to write the state transformer $\lambda\,\sigma.(\sigma,\sigma)$ that returns a copy of the state. In calculations we treat the state as a function from references to stored values.

Here we meet a technical problem. One of the requirements of expressions is that they must have a type. But what is the type of a state ? We can define

$$\textbf{type}\ ST\ s\ a\ \hat{=}\ State\ s \to a \times State\ s,$$

but what is the definition of *State s* ? The problem is that the state is a mapping from references of type $Ref\ s\ \tau$ to values of type τ, for every type τ (except possibly the type of the state itself). Such a polymorphism is not available in our type language. In a more generous type language it could be expressed as $State\ s \mathrel{\hat{=}} \Pi_\tau Ref\ s\ \tau \to \tau$, an infinite tuple with an element for every type τ. Applying the state to a reference $(\sigma\ v)$ is read as short for applying the appropriate projection of the state: $(\pi_\tau\ \sigma)\ v$, where $v : Ref\ s\ \tau$.

In principle, the state must be able to accommodate references to any type. However, in any given expression, references for only a *finite* number of types are allocated, for example only integers and booleans. So in any given expression the infinite tuple $\Pi_\tau Ref\ s\ \tau \to \tau$ can be approximated by a finite tuple, for example $(Ref\ s\ \mathbb{Z} \to \mathbb{Z}) \times (Ref\ s\ \mathbb{B} \to \mathbb{B})$, without any loss.

Either one treats the type of the state as an infinite tuple and takes this as a special case, or one approximates the infinite tuple with a finite tuple appropriate for the given expression. In the following we'll simply use states as if they were ordinary functions, and not be concerned with the definition of the type *State s*.

If it were generally allowed to bind the state to a variable, we could take the 'informal definitions' of the state transformer primitives as their definitions. We won't do so because they don't capture garbage collection: stored values that have become inaccessible may be deleted from the state. More generally the inaccessible part of the state may change at any time. This assumption is justified since the inaccessible part of the state can never be observed by a program anyway.

In the axioms given in this chapter, garbage collection is captured by the axiom 7, which is: if v is not free in K, and K is strict in the state, then $new\ E;\ \lambda\,v.K \equiv K$.

This law is vital for introducing references into a derivation (or removing unused ones), but it is not true taking the informal definitions for *new* and semicolon.

Definitions of the state transformer primitives that capture garbage collection cannot be given within the language because the notion of accessibility depends on variables currently in scope, the environment. In chapter 9 we sketch denotational semantics for the state transformer primitives that capture garbage collection.

4.2 Imperative Axioms and Theorems

This section lists the axioms and some theorems of the state monad primitives. The first subsection shows how imperative threads can be introduced, and combined, that is it lists the axioms about **run**, semicolon, and return. The second subsection shows how the primitive state transformers that create and access references can be introduced. The third subsection briefly mentions some examples of state transformer combinators. The fourth subsection deals with commutativity axioms and theorems. The final subsection summarises and discusses related work.

4.2.1 Imperative Threads

This subsection gives axioms and theorems that let us manipulate **run**, semicolon, and *return* . They let us introduce imperative threads into a program derivation, and move calculational work into the thread.

The monad laws are true for the state monad, and are repeated here.

Axiom 1 (Monad axioms)

axm	*left return*	$return\ E;\ F \equiv F\ E$
axm	*right return*	$E;\ return\ \equiv E$
axm	*; assoc.*	$E;\ (\lambda x.F;\ G) \equiv (E;\ \lambda x.F);\ G,\quad x \notin fv(G).$

The third we'll refer to as '; assoc.', using the word 'associative' rather loosely.

Since **run** discards the final state, using *break* to destroy the final state changes nothing.

Axiom 2 (final break) **run** $E \equiv$ **run**$(E;\ break)$

Making an expression into a state transformer by applying *return* to it and then running that state transformer is the identity function:

Axiom 3 (run intro) $E \equiv$ **run**$(return\ \ E)$.

By the previous axiom, we can also use *break* in place of *return* . This law is the first step in making a functional program imperative.

Combining *return* and **run** the other way around is identity with a side condition.

Axiom 4 (flattening nested threads) *return* (**run**E) $\equiv E$, *for* $E : \forall s.ST\ s\ a$ *that doesn't break the state, that is* $\forall \sigma.$**def**$\sigma \Rightarrow$ **def**$(snd(E\ \sigma))$.

E must be a 'runnable' state transformer. That is captured by its type. E must also not break the state, that is, if E is given a proper state, it will produce a proper state.

A function F applied to the result of a **run** can be pushed into the **run**:

Axiom 5 (function into run) $F(\mathbf{run}E) \equiv \mathbf{run}(E;\ \lambda x.\mathit{return}\ (F\ x))$ *for fresh* x.

Again by axiom 2 we can replace *return* by *break*. This axiom is used to move algorithmic work from the functional to the imperative part of the program.

The following axiom is also used for that purpose, this time to move **let** into the thread.

Axiom 6 (let/return) **let** $x = E$ **in** $F \equiv \mathit{return}\ E;\ \lambda x.F$ *for state transformer* F *and* E *not polymorphic.*

As an easy consequence we can apply **run** to both sides to get:

Theorem 1 **let** $x = E$ **in** $\mathbf{run}F \equiv \mathbf{run}(\mathit{return}\ E;\ \lambda x.F)$ *for state transformer* F *and* E *not polymorphic.*

4.2.2 Reference Accessors

This subsection lists axioms that show how the primitive state transformers *new* , *get* , *put* can be introduced into a program derivation.

Applying a state transformer to a proper state is the same as creating a new reference in the state and then applying the state transformer.

Axiom 7 (new-intro) $\mathit{new}\ E;\ \lambda v.F \equiv F$ *if* v *is not free in* F*, and* F *is strict in the state, that is,* $\mathbf{def}(F\ \sigma) \Rightarrow \mathbf{def}\sigma$.

If F is not strict (for example *return* 14), we can still use the axiom, as long as the initial state for both sides is proper. This axiom can be read in two directions. From right to left it is the axiom that we use to introduce references into a program derivation. It may seem futile, since v is not free in F. But after using the axiom, we are free to refine F to an expression in which v does appear free. From left to right the axiom is used to remove references that have become redundant, captured by the condition v not in F. The axiom captures garbage collection of inaccessible references.

We can introduce *get* and *put* behind a *new* by the following laws.

Axiom 8 (new generates put) $\mathit{new}\ F;\ \lambda v.G \equiv \mathit{new}\ E;\ \lambda v.\mathit{put}\ v\ F;\ \lambda\_.G$

Initialising a new reference to E and then overwriting it by F is the same as initialising it to F immediately. The state transformer G may contain v.

Axiom 9 (new generates get) $(\mathit{new}\ E;\ \lambda v.F\ E \equiv \mathit{new}\ E;\ \lambda v.\mathit{get}\ v;\ F) \Leftarrow \mathbf{det}E$

Putting a value into a new reference and retrieving it is the same as just putting it into a new reference and returning it. The expression F may contain v.

4.2.3 State Transformer Combinators

A convenient combinator of state transformers is *seq*. It takes a list of state transformers, composes them in sequence, and delivers a single state transformer. That single state transformer gathers the values returned by the composed state transformers into a list and itself returns that list.

The definition of *seq* is:

Definition 6

$$\begin{array}{ll} seq & : [ST\ s\ a] \to ST\ s\ [a] \\ seq\ [] & \hat{=}\ return\ [] \\ seq(k : ks) & \hat{=}\ k;\ \lambda\, a.seq\ ks;\ \lambda\, as.return\ (a : as). \end{array}$$

Almost immediate from the definition is this property of *seq*:

Theorem 2 $seq(ks +\!\!+ hs) \equiv seq\ ks;\ \lambda\, as.seq\ hs;\ \lambda\, bs.return\ (as +\!\!+ bs)$, *where ks and hs are lists of state transformers.*

4.2.4 Commuting State Transformers

This subsection defines commutativity for state transformers, and lists axioms about commuting state transformers.

If composing the two state transformers M and K in either order yields the same state transformer we say they *commute* and write $M \mid K$, defined as

Definition 7 (Commutative state transformers) $K \mid M$ *stands for*

$$\begin{array}{ll} & K;\ \lambda\, x.M;\ \lambda\, y.return\ (x, y) \\ \equiv & M;\ \lambda\, y.K;\ \lambda\, x.return\ (x, y), \end{array}$$

for fresh x *and* y.

A bunch of axioms are concerned with commuting state transformers:

Axiom 10 $get\ v \mid get\ w$,

Axiom 11 $get\ v \mid put\ w\ E$,

Axiom 12 $put\ v\ E \mid put\ w\ F$

where v and w are distinct proper references.

The trivial state transformer *return* A and the allocating state transformer *new* E commute with every state transformer K:

Axiom 13 (return commutes) $return\ A \mid K$,

Axiom 14 (new commutes) $new\ E \mid K$.

This axiom captures the intuition that the reference allocated and returned by *new* is an arbitrary new one. If *new* had a determined semantics (say it always allocates the least available reference) then the axiom would not be sound. For example, assume references are modelled by natural numbers, and nothing has been allocated so far. Then $\mathit{new}\ 56;\ \lambda v.\mathit{new}\ 47;\ \lambda w.\mathit{return}\ (v, w)$ would return $(0, 1)$, whereas $\mathit{new}\ 47;\ \lambda w.\mathit{new}\ 56;\ \lambda v.\mathit{return}\ (v, w)$ would return $(1, 0)$. With a nondetermined semantics both expressions return an arbitrary pair of distinct natural numbers, and are therefore equivalent. Of course the implementation of *new* may still be determined. The above axiom simply guarantees that when we are reasoning about expressions, that determinacy is invisible to us. We only know that each call of *new* returns a fresh reference, but not which fresh reference. We calculate with the specification of *new* , not with its implementation.

If a state transformer K commutes with every state transformer in a list gs, then it also commutes with the application of *seq* to the list.

Theorem 3 $\forall g \in gs.\ g \mid K \quad \Rightarrow \quad \mathit{seq}\ gs \mid K$

The proof is straightforward by induction on gs.

Finally, here is a theorem that allows us to replace two composed *seq* expressions by one *seq* expression. The list of results returned by the first *seq* expression is bound to a formal variable, xs say. The list of state transformers in the second *seq* expression is generated by mapping a function h over xs.

Theorem 4 (Merging *seq*)

$$\mathit{seq}\ gs;\ \lambda xs.\mathit{seq}(h^{*}xs) \equiv \mathit{seq}[g;\ h \mid g \leftarrow gs]$$

$$\Leftarrow$$

$$\forall g \in gs.\ \forall x.\ g \mid h\ x$$

The proof is by induction on gs, using the previous theorem.

4.2.5 Discussion

Using a monad to model imperative programming naturally provides the three monad axioms. In addition we have provided some axioms about the reference accessing state monad primitives.

Wadler [Wad92a] uses the monad comprehension notation for the state monad. This early version of the state monad operates on a fixed state type S. The operations given are

$$\begin{array}{ll}
\textbf{type}\ ST\ a & \hat{=}\ S \to S \times a \\
\mathit{init} & :\ S \to ST\ a \to a \\
\mathit{init}\ \sigma\ k & \hat{=}\ \textbf{let}\ (a, \_) = k\ \sigma\ \textbf{in}\ a \\
\mathit{assign} & :\ S \to ST\ () \\
\mathit{assign}\ \sigma & \hat{=}\ \lambda \sigma'.((), \sigma)
\end{array}$$

$$\begin{array}{ll} \mathit{fetch} & : ST\ S \\ \mathit{fetch} & \hat{=} \lambda\sigma.(\sigma,\sigma). \end{array}$$

In addition to the monad axioms, he lists the following three axioms on *qualifiers* (that is, comma-separated lists of $P \leftarrow E$ for pattern P and expression E). The notation $[x]^{ST}$ is our *return x*.

$$\begin{array}{rcl} x \leftarrow \mathit{fetch}, y \leftarrow \mathit{fetch} & = & x \leftarrow \mathit{fetch}, y \leftarrow [x]^{ST} \\ () \leftarrow \mathit{assign}\ u, y \leftarrow \mathit{fetch} & = & () \leftarrow \mathit{assign}\ u, y \leftarrow [u]^{ST} \\ () \leftarrow \mathit{assign}\ u, () \leftarrow \mathit{assign}\ v & = & () \leftarrow \mathit{assign}\ v \end{array}$$

These axioms rewritten as equivalences of state transformers (acting on single references rather than the whole state) are true in our system. They could be called 'get absorbs get', 'put absorbs get', and 'put absorbs put'. We did not find that they are required in deriving imperative programs, although they may be convenient if one wants to show two given state transformers are equivalent. Since any proper reference that *get* and *put* can be applied to must have been allocated by *new* previously, it is not hard to see that these three axioms follow from our axioms 8 and 9.

In [Wad92a] the axioms are presented as axioms on qualifiers. Qualifiers are not expressions. They have no type, and they bind variables, but the scope of the bindings is left open. We can replace equal qualifiers by one another, using axioms like the three above, but that means we are reasoning equationally in two separate worlds: the world of expressions, and the world of qualifiers. In effect this is the same as reasoning about the world of expressions, and the world of assignments in languages with assignments.

Wadler also gives three axioms on expressions. The first, $\mathit{init}\ u\ [t]^{ST} = t$ is closely related to our axiom 3. The other two are $\mathit{init}\ u\ [t \mid () \leftarrow \mathit{assign}\ v, q]^{ST} = \mathit{init}\ v\ [t \mid q]^{ST}$ and $\mathit{init}\ u\ [t \mid q, () \leftarrow \mathit{assign}\ v]^{ST} = \mathit{init}\ u\ [t \mid q]^{ST}$. They are not comparable to laws in our system.

Johnsson [Joh95] describes a known optimisation of the G-machine calculationally. First the calculation is done on an implementation without sharing, and then on one with sharing, that is, one using graphs, and the state monad to capture graph manipulation in a functional language. To do the calculation, Johnsson postulates only four axioms of state transformers (in addition to the monad axioms). (He calls the primitives *store*, *fetch*, *update* instead of our *new* , *get* , *put* .)

$$\begin{array}{lcll} \mathit{store}\ v;\ \lambda p.\mathit{fetch}\ p;\ \lambda v'.m & = & \mathit{store}\ v;\ \lambda p.m[v/v'] & \text{store/fetch} \\ \mathit{store}\ v;\ \lambda p.\mathit{update}\ v'\ p;\ \lambda().m & = & \mathit{store}\ v';\ \lambda p.m & \text{store/update} \\ \mathit{store}\ v;\ \lambda p.m & \Rightarrow & m & \text{garbage store, } p \notin \mathit{fv}(m) \\ \mathit{store}\ e;\ \lambda p.m;\ \lambda x.k & = & m;\ \lambda x.\mathit{store}\ e;\ \lambda p.k & \text{move store, } p \notin \mathit{fv}(m), x \notin \mathit{fv}(e) \end{array}$$

The store/fetch axiom is my axiom 'new generates get', law 9. The store/update axiom is my axiom 'new generates put', law 8. In the garbage store axiom, $\Rightarrow$ stands for a kind of refinement. Johnsson says the garbage store axiom is 'hard to prove'. He doesn't give detailed semantics of the state monad primitives. The axiom can only be proven sound if the semantics captures garbage collection of inaccessible references. The left hand side allocates a new reference in the state, and the right does not. If we had some simple model of state where the state is completely observable (such as a list of stored values) the two sides are incomparable. Johnsson uses the axiom in one direction (the direction of $\Rightarrow$) to eliminate unused references, but it is also useful in the other direction to introduce references into a derivation.

The move store axiom depends on nondetermined semantics of *store*, that is, *store v* really must return an arbitrary new reference that was previously free. Johnsson defines *store* by $\lambda x.\lambda g.(\mathit{newPointer}\ g, \mathit{updateGraph}\ g\ (\mathit{newPointer}\ g, x))$. For this definition to work, *newPointer g* must be determined, since if the two uses of *newPointer g* had different outcomes, it would be nonsense. That means *store* is overspecified and the axiom doesn't hold. With our nondetermined semantics of *new* (denotational semantics given in chapter 9) it holds, and is the axiom 14.

The good tutorial [MJ95] on folds and monads uses a variation of Johnsson's calculation as an example. It gives four axioms about the state monad. They are 'new generates get' (axiom 9), 'new-generates-put' (axiom 8), 'new commutes' (axiom 14), and an erroneous 'get-store' axiom. The authors present the 'new commutes' axiom using $\cong$ rather than $=$ and remark that the two sides are not equal (since they affect the state differently), but that no surrounding program can distinguish them (since the state itself and the values of references are not observable).

Chapter 5

Data Refinement

5.1 What is Data Refinement of Expressions ?

In the early stages of a program development we may express the data in the program in an abstract way that is easy to understand and easy to calculate with, but not easily or efficiently implementable. In the later stages we may want to express the same data in a concrete way that is implementable, but may be more difficult to understand. For instance, we may want to represent sets by boolean arrays, or finite maps by lists of pairs, or complex numbers by pairs of reals. Converting the program from abstract to concrete data is called *data refinement.*

A program expression has some subexpressions whose types depend on a particular type A, called the *abstract type.* Some, but not necessarily all, of these expressions are troublesome: they are impossible or expensive to execute. These are called the *abstract expressions*, and the aim of *data refinement* is to replace them by *concrete expressions*, that is, expressions that have essentially the same meaning, but use a certain *concrete type* C instead of A. The concrete expressions should be cheaper to execute than the abstract ones. The abstract type and the concrete type are related by the *representation relation* $I : A \leftrightarrow C$ (called "abstraction invariant" in [Mor89] and "coupling invariant" in [Mor94]).

The types A and C may be primitive or constructed, and they needn't be distinct. The representation relation $I : A \leftrightarrow C$ may be total, surjective, functional, or injective[1], but it needn't be.

It needn't be total: we may only want to represent those elements of the abstract type that are actually used. We may for instance implement sets, but only have an implementation for sets below a certain size. I needn't be surjective: There may be concrete

[1]in the sense of van Gasteren[vG88]: relation R is

total:	$\forall\, a . \exists\, b.$	$a\ R\ b$
surjective:	$\forall\, b . \exists\, a.$	$a\ R\ b$
functional:	$\forall\, a, b, b'.$	$a\ R\ b \wedge a\ R\ b' \Rightarrow b = b'$
injective:	$\forall\, a, a', b.$	$a\ R\ b \wedge a'\ R\ b \Rightarrow a = a'.$

outcomes that don't represent any abstract outcome. We may for instance implement complex numbers by pairs of reals representing the magnitude and the angle. No complex number has a negative magnitude, so $(-17, \pi/2)$ represents nothing. I needn't be functional: an abstract outcome may be represented by many concrete outcomes. We may for instance implement lists by pairs of integers and arrays representing the length of the list and its elements in order. Arrays representing short lists will contain some arbitrary elements. I needn't be injective: We may only be interested in some aspect of the abstract outcome, so the data refinement loses (unwanted) information. For instance we may represent a long list over a small type by a histogram, if we are only interested in the frequencies.

The programmer will write the representation relation as a function of type $A \to C \to \mathbb{B}$ (abbreviate this to $A \leftrightarrow C$). This allows a great flexibility. However, in practice sensible representation relations deliver *True* or *False* if applied to defined determined arguments. For example we may represent booleans by the parity of integers:

$$\begin{aligned} I &: \mathbb{B} \leftrightarrow \mathbb{Z} \\ b\ I\ z &\mathrel{\hat{=}} b =_{\mathbb{B}} odd\ z. \end{aligned}$$

The types A and C, and the relation $I : A \leftrightarrow C$ are supplied by the programmer. The programmer is also expected to know which subexpressions are to be replaced to obtain a typed and efficient program. This decision is captured by a mapping from a type to a type. An example is T defined by $T[\,] \mathrel{\hat{=}} \mathbb{Z} \times [\,]$. Then $T[\mathbb{Z}]$ would be $\mathbb{Z} \times \mathbb{Z}$.

To data-refine a given expression E the programmer has to identify the abstract type A, the desired concrete type C, the representation relation I between them, and the mapping T identifying which uses of type A should be data-refined. The expression E must be of type $T[A]$, and the data refinement will produce an expression, say F, of type $T[C]$. So data refinement has four parameters: A, C, I, T. It is clear that generally data refinement is not a super-relation of equivalence, since even the types $T[A]$ and $T[C]$ may be different. Therefore we will use the symbol $\ll$ that has no horizontal bar in it, and subscript it with the T and I, thereby implicitly also identifying A and C. We write $E \ll_{T,I} F$. If the subscripts are obvious from the context, we may omit them. As usual, we always assume that expressions have the appropriate type.

The typing rule for the data-refinement connective $\ll$ is:

$$\frac{I : A \leftrightarrow C \qquad \Gamma \vdash E : T[A] \qquad \Gamma \vdash F : T[C]}{\Gamma \vdash (E \ll_{T,I} F) : \mathbb{B}}.$$

Given A, C, and I, the axioms of data refinement enable us to prove formulas of the form $E \ll_{T,I} F$ for any mapping T. In the degenerate case where the body of T doesn't mention the argument, data refinement specialises to refinement. In this case the axioms of data refinement become refinement axioms. By this specialisation of axioms about data refinement to refinement, many of the axioms listed in chapter 3 become theorems, provable from the axioms given here. In order to keep chapter 3 self-contained however, we keep them as axioms there. It is not hard to see which axioms become theorems: in particular, the axioms stating that refinement is reflexive and antisymmetric are still necessary.

5.2 Axioms of Data Refinement

In the following, we'll list the axioms and some theorems of data refinement. For most of the shapes of an expression, there is a data refinement theorem that allows data-refining the whole by data-refining its parts. Specialised to refinement, these theorems state that almost any language construct is monotone. We are prevented from gathering them into one general theorem by the non-monotone constructs, such as **if fi**. The monotonicity theorems allow us to distribute data refinement into the subexpressions of an expression. The base cases of data refinement are given by refinement, and by the representation relation I.

Fundamental Axioms

The base cases of data refinement are given by specialising data refinement to refinement, and by the representation relation itself. In the second axiom, the type-to-type mapping $[\,]$ maps an type U to the type U itself. In addition, we give similar axioms about data refinement as those about refinement. In the axiom $\ll$ */trans.*, the relation $I \circ J$ is defined by $a(I \circ J)c \mathrel{\hat{=}} \exists\, b.a\ I\ b \wedge b\ J\ c$.

axm	$\ll / \sqsubseteq$	$(E \ll_{T,I} F) \equiv (E \sqsubseteq F)$,	if T is constant
axm	$\ll / I$	$(E\ I\ F) \Rightarrow (E \ll_{[\,],I} F)$	
axm		$\mathbf{def}(E \ll F), \mathbf{det}(E \ll F), \mathbf{feas}(E \ll F)$	
axm	$\ll$ *truth*	$((E \ll F) \equiv \mathit{True}) \equiv (E \ll F)$	
axm	$\ll$ *extr.*	$\bot \ll E, E \ll \top$	
axm	$\ll$ *trans.*	$(E \ll_{T,I} F) \wedge (F \ll_{T,J} G) \Rightarrow (E \ll_{T,I \circ J} G)$	

Language Constructs

Most language constructs are monotone with respect to data refinement.

axm	$\sqcap$	$(X \ll_{T,I} E) \wedge (X \ll_{T,I} F) \equiv (X \ll_{T,I} E \sqcap F)$	
thm	$\sqcap -$ *mon.*	$(E \ll_{T,I} F) \wedge (G \ll_{T,I} H) \Rightarrow (E \sqcap G \ll_{T,I} F \sqcap H)$	
axm	$\bigsqcap$	$(\forall x.X \ll_{T,I} E) \equiv (X \ll_{T,I} \bigsqcap x.E)$,	$x \notin \mathit{fv}(X)$
thm	$\rightarrow$*mon.*	$(E \ll_{T,I} F) \Rightarrow (G \rightarrow E \ll_{T,I} G \rightarrow F)$	
thm	$>\!\!-$*mon.*	$(E \ll_{T,I} F) \Rightarrow (G >\!\!- E \ll_{T,I} G >\!\!- F)$	
thm	*intro.* **if fi**	$(E \ll_{T,I} F) \wedge \mathbf{feas}F \Rightarrow (E \ll_{T,I} \mathbf{if}\ F\ \mathbf{fi})$	
thm	*use*$\rightarrow$	$(G \Rightarrow (E \ll_{T,I} F)) \Rightarrow (E \ll_{T,I} G \rightarrow F)$	
thm	*use*$>\!\!-$	$(G \Rightarrow (E \ll_{T,I} F)) \Rightarrow (G >\!\!- E \ll_{T,I} F)$	

The second last theorem is called 'data refinement by miracle' by [Mor88b], and analogously, the last could be called 'data refinement by failure'.

Higher Types

Data refinement on higher types (tuples, sums, functions) is given in terms of data refinement on their constituent types. As before, we take the binary case as example of the general case.

axm	$\ll$ *tuple*	$(E \ll_{T,I} F) \wedge (G \ll_{U,I} H) \equiv ((E,G) \ll_{T\times U,I} (F,H))$
axm	$\ll$ *sum*	$(E \ll_{T,I} F) \equiv (\mathbf{Inl}E \ll_{T+U,I} \mathbf{Inl}F)$
axm		$\neg(\mathbf{Inl}E \ll_{T+U,I} \mathbf{Inr}F) \Leftarrow \mathbf{feas}F$
thm	*app.mon.*	$(E \ll_{T\to U,I} F) \wedge (G \ll_{T,I} H) \Rightarrow (E\ G \ll_{U,I} F\ H)$

Variables

For the language constructs that bind variables, we must also introduce axioms that allow data refinement of the bound variable. The axioms about generalised choice and the quantifications are weaker than the theorems saying that generalised choice and quantifications are monotone with respect to refinement in their bodies: They also require that the bodies be monotone in the bound variable. For λ abstractions, recursive expressions, and **let** expressions, we already make that assumption. Therefore, specialising the last three axioms here to refinement yields exactly the theorems that (well-formed) λ abstractions, recursive expressions, and **let** expressions are monotone in their bodies. In all six axioms below, we assume $x \notin fv(F)$ and $y \notin fv(E)$.

axm	$\sqcap$*var.*	$(\forall x : T[A].\forall y : T[C].(x \ll_{T,I} y) \Rightarrow (E \ll_{U,I} F)) \Rightarrow$ $(\sqcap x : T[A].E \ll_{U,I} \sqcap y : T[C].F)$
axm	$\forall$ *var.*	$(\forall x : T[A].\forall y : T[C].(x \ll_{T,I} y) \Rightarrow (E \sqsubseteq F)) \Rightarrow$ $(\forall x : T[A].E \sqsubseteq \forall y : T[C].F)$
axm	$\exists$ *var.*	$(\forall x : T[A].\forall y : T[C].(x \ll_{T,I} y) \Rightarrow (E \sqsubseteq F)) \Rightarrow$ $(\exists x : T[A].E \sqsubseteq \exists y : T[C].F)$
axm	λ *var.*	$(\forall x : T[A].\forall y : T[C].(x \ll_{T,I} y) \Rightarrow (E \ll_{U,I} F)) \Rightarrow$ $(\lambda x : T[A].E \ll_{T\to U,I} \lambda y : T[C].F)$
axm	let *var.*	$(\forall x : T[A].\forall y : T[C].(x \ll_{T,I} y) \Rightarrow (E \ll_{U,I} F)) \wedge (G \ll_{T,I} H) \Rightarrow$ $(\mathbf{let}\ x = G\ \mathbf{in}\ E \ll_{U,I} \mathbf{let}\ y = H\ \mathbf{in}\ F)$
axm	μ *var.*	$(\forall x : T[A].\forall y : T[C].(x \ll_{T,I} y) \Rightarrow (E \ll_{T,I} F)) \Rightarrow$ $(\mu x : T[A].E \ll_{T,I} \mu y : T[C].F)$

5.3 Containing a Data Refinement in a Refinement

Usually in a derivation, the final program refines the initial specification (rather than *data*-refining it), and data refinement is only applied to subexpressions of the whole specification. For this method to work, certain combinations of data-refined expressions must lead to refined expressions. These combinations arise from the axioms in which the type mapping changes. In particular those include the axiom saying that function application is monotone with respect to data refinement, and the axioms dealing with data refinement of variables. Whenever a mapping K is reached that maps any type to a constant type, that is, $K[T] = K[U]$ for any types T and U, the trivial data refinement $\ll_{K,I}$ is actually just ordinary refinement $\sqsubseteq$.

We briefly outline (somewhat abstractly) one usual case of containing a data refinement in a refinement in this way. It is based on function application and λ-abstraction. Similar situations based on, for example, function composition or the other variable binders exist.

Assume we have some specification E of type $\mathbb{Z}$, say, that we want to implement. Some mathematical thought shows the problem can be factored into producing a set of naturals, and applying some function to that set. That is,

$$E \sqsubseteq (\lambda x.F)G,$$

for some $G : \mathbb{P}\mathbb{N}$ and $\lambda x.F : \mathbb{P}\mathbb{N} \to \mathbb{Z}$. We decide to use this shape as a basis for an implementation after checking that we have an implementable representation for the range of sets that can occur. This representation could be an array together with an index. Let's call this concrete type C, and the representation relation

$$I : \mathbb{P}\mathbb{N} \leftrightarrow C.$$

We produce a data refinement of G:

$$G \ll_{Id,I} G',$$

where Id is the type mapping defined by $Id[T] \mathrel{\hat{=}} T$ for every type T. We specialise the axiom about data-refining a λ-abstraction by letting U in it be constant mapping $U[T] \mathrel{\hat{=}} \mathbb{Z}$, so that $\ll_{U,I}$ is in fact just $\sqsubseteq$. Then we look for a data-refinement of F. Call it $F' : \mathbb{Z}$ with a free variable $y : C$, say, such that $x \ll_{Id,I} y$ implies $F \sqsubseteq F'$. We conclude that

$$\lambda x : Id[\mathbb{P}\mathbb{N}].F \ll_{Id \to U,I} \lambda y : Id[C].F'.$$

We have found data refinements for both the function and the argument in our specification. Furthermore, both are real data refinements in that the types really change. By the application-axiom we can combine them to conclude:

$$(\lambda x.F)G \sqsubseteq (\lambda y.F')G',$$

and this is just an ordinary refinement. A data refinement has been contained in a refinement.

The corresponding situation in data refinement of imperative programs (see for instance [Mor88a, Mor89]) is encapsulating data refinement in a block. So command s may be data-refined to command t under representation relation I, in symbols $s \ll_I t$, but when the abstract, respectively, concrete variables are bound in blocks, we return to ordinary refinement:

$$[\![\mathbf{var}\ a : A;\ s]\!] \ \sqsubseteq\ [\![\mathbf{var}\ c : C;\ t]\!],$$

where $[\![\ ]\!]$ delimits the scope of the variables a and c respectively.

Chapter 6

Simple Example Programs

This chapter gives four (partial) program derivations illustrating the use of the specification language. Three programs are formulated imperatively because they perform simple IO. In the first example, data refinement is used to derive an implementation for the abstract type $\mathbb{Z}$. In the second example, state is used to record a history of the program's successive inputs, and data refinement is used to store that information in a more compact way. The third example illustrates a common use of state: a reference is added, storing a kind of index, calculated at little extra cost in one run of the program, and making the next run of the program more efficient, or (as here) fairer. In the fourth example state is used to precompute and store in an array a set of values which otherwise may be recomputed many times over in the program.

6.1 What's the Time ?

We specify and implement a small imperative program that reads the time from the system clock, and displays it on the screen in an agreeable format. The program is easily specified using integers. By a data refinement, the integers are implemented as signed 32 bit integers and unsigned 32 bit integers.

This program derivation mainly illustrates data refinement in practice. The specification is very short and clear, but formulated in terms of types and operations not available in real programming languages. The task is to substitute suitable concrete types and operations for the abstract ones. The derivation uses piecewise data refinement. Two different concrete types are used to implement the same abstract type. Both representation relations are not total, so there are abstract values that can't be represented. Assertions are used to make sure abstract values lie in the representable ranges. The program is a small state transformer because it performs IO.

6.1.1 The Problem

Assume an *IO*-transformer *getTime* : *IO* $\mathbb{Z}$ delivers the number of seconds since the beginning of time. Time began at 0:00:00 of 1 Jan 1904 and it will end at 6:28:15 of 6

Feb 2040[1]. Therefore the integer delivered by *getTime* will be in the range $0..2^{32}-1$. Write a program that displays the time in an agreeable format.

```
getTime; λ ts.
0 ≤ ts ≤ 2^32 − 1>—
let
    seconds = ts mod 60
    tm      = ts div 60
    minutes= tm mod 60
    th      = tm div 60
    hours   = th mod 24
in
    write(hours, minutes, seconds)
```

Unfortunately the programming language lacks the type $\mathbb{Z}$. It does however provide 32-bit, two's complement 'integers' as a type called *Signed*[2]. This type can represent integers in the range $-2^{31}..2^{31}-1$. It also has equivalents of the integer operations *mod*, *div*, $*, +, -$. We'll use that type to represent most numbers in the program. But since the beginning of time more than 91 years, that is $91 * 365 * 24 * 60 * 60$ seconds, have expired, and that's more than $2^{31}-1$ seconds, the largest integer that can be represented by a *Signed*. However, there was no time before the beginning of time, so negative numbers need not be represented. The *IO* transformer *getTime* doesn't exist in the programming languages, only its sibling $\underline{getTime}$: *IO Unsigned*, which delivers an unsigned 32 bit 'integer', of type *Unsigned*.

We data-refine the specification to use *Unsigned* and *Signed* instead of $\mathbb{Z}$. In the following we'll abstract from 32 by declaring a constant: $N \mathrel{\hat{=}} 31$.

6.1.2 The Representation Relations

Signed $N+1$ Bit Integers

We represent $N+1$ bits mathematically by finite functions, so we write:

$$\textbf{type } Signed \;\hat{=}\; \{0..N\} \to \{0,1\}$$

It follows from the representation relation below that *Signed* values can represent integers in the range $-2^N..2^N-1$.

The representation relation $S : \mathbb{Z} \leftrightarrow Signed$ expressed numerically is: $z\ S\ s$ is defined

$$\begin{aligned} z \;=\; & \textbf{if}\ s\ N = 0 \to sum[2^i * s\ i \mid i \leftarrow 0..N-1] \\ & \square s\ N = 1 \to sum[2^i * s\ i \mid i \leftarrow 0..N-1] - 2^N \\ & \textbf{fi}, \end{aligned}$$

[1]We ignore leap-seconds.

[2]Most real programming languages call something like that "*integer*".

where $sum \mathrel{\hat{=}} foldr\ (+)\ 0$. A small proof shows that the same expressed in terms of bits and bit-inversion is:

$$\begin{array}{rcl} z & = & \mathbf{if}\, s\ N = 0 \to sum[2^i * s\ i \mid i \leftarrow 0..N-1] \\ & & \sqcap s\ N = 1 \to -(sum[2^i * \overline{s\ i} \mid i \leftarrow 0..N-1] + 1) \\ & & \mathbf{fi}, \end{array}$$

where the bar over denotes bit-inverting: $\overline{0} \mathrel{\hat{=}} 1$ and $\overline{1} \mathrel{\hat{=}} 0$.

In our programming language there are many convenient functions and constants on the type *Signed*, for example:

$$\begin{array}{rcl} \underline{mod}, \underline{div}, \underline{+}, \underline{*} & : & Signed \to Signed \to Signed \\ \underline{0}, \underline{1}, \underline{24}, \underline{60}, \underline{35791394} & : & Signed \\ \underline{write} & : & Signed \times Signed \times Signed \to IO\ (). \end{array}$$

We use the underline to distinguish them from their siblings on the type $\mathbb{Z}$:

$$\begin{array}{rcl} mod, div, +, * & : & \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \\ 0, 1, 24, 60, 35791394 & : & \mathbb{Z} \\ write & : & \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to IO\ (). \end{array}$$

Since *Signed* is a function type, we can apply a *Signed* to an integer in the range $0..N$, or use function override. For example, the third bit of $s[3 \mapsto 0]$ is 0, and otherwise it is the same as s.

Unsigned $N+1$ Bit Integers

$$\mathbf{type}\ Unsigned \ \mathrel{\hat{=}}\ \{0..N\} \to \{0, 1\}$$

It follows from the representation relation below that *Unsigned* values can represent integers in the range $0..2^{N+1} - 1$.

The representation relation $U : \mathbb{Z} \leftrightarrow Unsigned$ is:

$$z\ U\ u \ \mathrel{\hat{=}}\ sum[2^i * u\ i \mid i \leftarrow 0..N] = z.$$

In our programming language there are no predefined functions for the type *Unsigned* other than as already mentioned $getTime : IO\ Unsigned$. We must therefore be content with function application and function override.

Since *Signed* and *Unsigned* are represented in the same way, namely as $\{0..N\} \to \{0, 1\}$, a value of one type is also a value of the other. As functions $\{0..N\} \to \{0, 1\}$ they are equal, but unless the most significant bit is 0, they *represent* different integers.

6.1.3 Data Refinement

We'll apply two data refinements to the program:

- We'll data-refine uses of $\mathbb{Z}$ to uses of *Unsigned* where necessary because of large numbers.
- We'll data-refine the remaining uses of $\mathbb{Z}$ to uses of *Signed*, taking advantage of the convenient predefined functions and constants.

Making the Large Integers Unsigned

We'll data-refine *compositionally*, that is to say, we data-refine a composite expression by data-refining each of its subexpressions, and putting them together to get a data-refinement of the whole expression.

Our program, for example, is an application of the function '; ' to the two arguments *getTime* and the abstraction starting $\lambda\, ts$. We data-refine the application compositionally. The imaginary *getTime* : *IO* $\mathbb{Z}$ is data-refined by the existing $\underline{getTime}$: *IO Unsigned*. We data-refine the abstraction simply by changing $ts : \mathbb{Z}$ to $ts' : Unsigned$ and data-refining its body.

Before data-refining the body of the abstraction starting $\lambda\, ts$, for convenience we push the assertion into the local definitions of the **let** expression:

$$
\begin{array}{l}
0 \le ts \le 2^{N+1} - 1 \succ\!\!- \\
\textbf{let} \\
\quad \begin{array}{ll}
seconds & = ts\ mod\ 60 \\
tm & = ts\ div\ 60 \\
minutes & = tm\ mod\ 60 \\
th & = tm\ div\ 60 \\
hours & = th\ mod\ 24
\end{array} \\
\textbf{in} \\
\quad write(hours, minutes, seconds) \\
\sqsubseteq \\
\textbf{let} \\
\quad \begin{array}{ll}
seconds & = (0 \le ts \le 2^{N+1} - 1 \succ\!\!- ts)\ mod\ 60 \\
tm & = (0 \le ts \le 2^{N+1} - 1 \succ\!\!- ts)\ div\ 60 \\
minutes & = tm\ mod\ 60 \\
th & = tm\ div\ 60 \\
hours & = th\ mod\ 24
\end{array} \\
\textbf{in} \\
\quad write(hours, minutes, seconds)
\end{array}
$$

Now we data-refine the **let** expression compositionally. We'll only change the local definitions of *seconds* and *tm*. Since we're not changing the types or values of any of the five local variables, the body of the **let** expression, namely *write*(*hours*, *minutes*, *seconds*), can stay as it is.

Furthermore, we data-refine the applications of *mod* and *div* compositionally: the functions and the integer 60 are trivially data-refined by themselves. For the two bracketed expressions we'll use the theorem *use*$\succ\!\!-$. The information in the assertion guarantees that a data refinement is indeed possible. For the integer *ts* to be represented by

$ts' : Unsigned$, ts must be in the range $0..2^{N+1}-1$, and that's just what the assertion guarantees. For all $ts \ll_{[\,],U} ts'$ we have

$$0 \le ts \le 2^{N+1}-1 \succ\!\!- ts \quad \ll_{\mathbb{Z},U} \quad sum[2^i * ts'\ i \mid i \leftarrow 0..N].$$

After the first data refinement, the complete program reads:

$\underline{getTime}$; $\lambda\, ts'$.
let
  $seconds = sum[2^i * ts'\ i \mid i \leftarrow 0..N]\ mod\ 60$
  $tm\quad = sum[2^i * ts'\ i \mid i \leftarrow 0..N]\ div\ 60$
  $minutes = tm\ mod\ 60$
  $th\quad = tm\ mod\ 60$
  $hours\ = th\ mod\ 24$
in
  $write(hours, minutes, seconds)$.

Let's do a quick mental renaming of the variable $ts' : Unsigned$ to $ts : Unsigned$, to lose the apostrophe.

Making the Remaining Integers Signed

Now we will replace all integer arithmetic by *Signed* arithmetic. This is done compositionally, and is largely boring: just underline *mod*, *div*, *write*, and the numbers 60, 24. However, there is a problem in the local definitions of *seconds* and *tm*: the integer $sum[2^i * ts\ i \mid i \leftarrow 0..N]$ is not guaranteed to be in the representable range of *Signed*, so we must data-refine the whole expression $sum[2^i * ts\ i \mid i \leftarrow 0..N]\ mod\ 60$ rather than data-refine it compositionally, and similarly for *tm*. We calculate:

$$\begin{aligned} & sum[2^i * ts\ i \mid i \leftarrow 0..N]\ mod\ 60 \\ \equiv &\qquad \text{list and } sum \\ & (2^N * ts\ N + sum[2^i * ts\ i \mid i \leftarrow 0..N-1])\ mod\ 60 \\ \equiv &\qquad \text{properties of } mod \\ & ((2^N\ mod\ 60) * ts\ N + sum[2^i * ts\ i \mid i \leftarrow 0..N-1]\ mod\ 60)\ mod\ 60 \end{aligned}$$

Now $2^N\ mod\ 60$ is a fixed integer within the range of *Signed*. The highest value $sum[2^i * ts\ i \mid i \leftarrow 0..N-1]$ can take is 2^N-1, which is also the highest integer that can be represented by *Signed*. In fact the integer $sum[2^i * ts\ i \mid i \leftarrow 0..N-1]$ is data-refined by the *Signed* expression $ts[N \mapsto 0]$. We data-refine compositionally:

$$\begin{aligned} & ((2^N\ mod\ 60) * ts\ N + sum[2^i * ts\ i \mid i \leftarrow 0..N-1]\ mod\ 60)\ mod\ 60 \\ \ll_{[\,],S} & \\ & (\underline{k}\ \underline{*}\ f(ts\ N)\ \underline{+}\ ts[N \mapsto 0]\ \underline{mod}\ \underline{60})\ \underline{mod}\ \underline{60}, \end{aligned}$$

where the f converts a bit into a *Signed*:

$f\ \ : \{0,1\} \rightarrow Signed$
$f\ 0 \mathrel{\hat=} \underline{0}$
$f\ 1 \mathrel{\hat=} \underline{1}$,

and $\underline{k} : Signed$ satisfies $2^N \; mod \; 60 \ll_{[\,],S} \underline{k}$. The value of $\underline{k}$ depends on the constant N. If N is 31, then $\underline{k}$ will be $\underline{8}$.

We treat the right side of the local definition of tm similarly:

$$\begin{array}{ll}
 & sum[2^i * ts\ i \mid i \leftarrow 0..N] \; div \; 60 \\
\equiv & \qquad \text{list and } sum \\
 & (2^N * ts\ N + sum[2^i * ts\ i \mid i \leftarrow 0..N-1]) \; div \; 60 \\
\equiv & \qquad \text{properties of } div, mod \\
 & (2^N \; div \; 60) * ts\ N + sum[2^i * ts\ i \mid i \leftarrow 0..N-1] \; div \; 60 \\
 & +((2^N \; mod \; 60) * ts\ N + sum[2^i * ts\ i \mid i \leftarrow 0..N-1] \; mod \; 60) \; div \; 60 \\
\ll_{[\,],S} & \qquad \text{everything in range} \\
 & \underline{m} \underline{*} f(ts\ N) \underline{+} ts[N \mapsto 0] \; \underline{div} \; \underline{60} \\
 & \underline{+}(\underline{k} \underline{*} f(ts\ N) \underline{+} ts[N \mapsto 0] \, \underline{mod}\, \underline{60}) \; \underline{div} \; 60,
\end{array}$$

where $\underline{k} : Signed$ is as before, and $\underline{m} : Signed$ satisfies $2^N \; div \; 60 \ll_{[\,],S} m$. It is easy to show that an $\underline{m}$ exists. If N is 31, then $\underline{m}$ will be $\underline{35,791,394}$.

The complete program reads:

```
$\underline{getTime}$; $\lambda\, ts$.
let
    $seconds = (\underline{k} \underline{*} f(ts\ N) \underline{+} ts[N \mapsto 0]\, \underline{mod}\, \underline{60})\, \underline{mod}\, \underline{60}$
    $tm \quad\;\; = \underline{m} \underline{*} f(ts\ N) \underline{+} ts[N \mapsto 0]\, \underline{div}\, \underline{60}$
              $+(\underline{k} \underline{*} f(ts\ N) \underline{+} ts[N \mapsto 0]\, \underline{mod}\, \underline{60})\, \underline{div}\, \underline{60}$
    $minutes = tm\, \underline{mod}\, \underline{60}$
    $th \quad\;\;\; = tm\, \underline{div}\, \underline{60}$
    $hours \;\; = th\, \underline{mod}\, \underline{24}$
in
    $write(hours, minutes, seconds)$
where
    $f : \{0, 1\} \to Signed$
    $f\ 0 \mathrel{\hat=} \underline{0}$
    $f\ 1 \mathrel{\hat=} \underline{1},$
```

and $\underline{k}, \underline{m} : Signed$ are constants depending on constant $N : \mathbb{N}$. $N + 1$ is the number of bits used for *Signed* and *Unsigned*, $\underline{k}$ is the *Signed* representing $2^N \; mod \; 60$, and $\underline{m}$ is the *Signed* representing $2^N \; div \; 60$.

6.2 Pure Tuning

In this section, we specify and implement a small imperative program that repeatedly receives a single input from the user and responds with an output. Each output depends on all the preceding inputs, so the program has to maintain a history of inputs. In the application discussed here, keyboard control of a music synthesiser, the relevant information can be stored in a compact form.

The specification is factored into the imperative structure (essentially a *while*-loop) and the calculational part. The imperative structure given in the specification is not changed, but the type of references that are manipulated is changed. By a data refinement we replace storing an arbitrary long list by a fixed-length array (represented by a finite function). This more compact representation is found calculationally. The representation relation is not functional from abstract to concrete values, contradicting the sometimes held view that data refinements always *add* detail.

6.2.1 Introduction

Imagine a program that holds a dialog with the user. The program starts up and does nothing visible until the user gives it one input, whatever that may be. Moreover, the program responds with exactly one output, which depends on the input just received, and then waits for the next input. Whenever the user supplies a single input, the program responds with a single output, which depends on all previous inputs up to and including the one just received. To express such a program, we must specify the algorithm that is used to calculate outputs from the inputs. We must also somehow specify the desired IO behaviour of the program: the input actions and output actions should occur in the interleaved order described above. To specify the calculational part of the program we use a function

$$h : [\mathit{Input}] \to \mathit{Output}.$$

The function h applied to the list of inputs so far produces the next output. To specify the IO behaviour of the program we will use state transformers. The state is used to string up the IO actions in the right temporal order, like pearls on a thread. The program would have a shape like this:

$$\mathit{while}\ \mathit{more}\ (\mathit{in};\ \mathit{body};\ \mathit{out}),$$

where the state transformer combinator *while* is defined recursively

$$\begin{array}{l} \mathit{while} : ST\ s\ B \to ST\ s\ a \to ST\ s\ [a] \\ \mathit{while}\ \mathit{more}\ \mathit{body} \mathrel{\hat{=}} \mathit{more};\ \lambda\, b.\mathbf{if}\ b \to \mathit{body};\ \lambda\, a.\mathit{while}\ \mathit{more}\ \mathit{body};\ \lambda\, \mathit{as}.\mathit{return}\ (a : \mathit{as}) \\ \qquad\qquad\qquad\qquad\qquad\quad \sqcap \neg b \to \mathit{return}\ [] \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{fi}. \end{array}$$

This is a recursive definition like any other. Of course we don't know whether a program written in terms of *while* will terminate; This depends on the IO that *more* and *body* may perform. However, the unpredictability of this IO is fixed in the arguments *more* and *body* as state transformers, that is, mappings from states. For every fixed *more* and *body*, *while* is defined as a certain fixpoint.

We assume that the state transformers $\mathit{more} : \mathit{IO}\ \mathbb{B}$, $\mathit{in} : \mathit{IO}\ \mathit{Input}$, and $\mathit{out} : \mathit{Output} \to \mathit{IO}\ ()$, are given as part of the specification. They find out whether there is some more input available, get the next input from the user, and give an output to the user. It is the state transformer *body* that we have to find. It should receive the next input and calculate the next output, without doing any IO. We connect this specification of the IO behaviour with the specification of the calculation given in h in the following way: We use a reference to store the list of inputs so far. This list will be initialised to the empty list. On each iteration of the *while* loop, the next input is received, and appended to the stored list. Then *h* applied to the list is outputted.

```
init; λ v.
while more (in; body v; out)
where
init     : ST s (Ref s [Input])
init     ≙new []
body     : Ref s [Input] → Input → ST s Output
body v n ≙get v; λ ns.put  v (ns ++ [n]); λ_.return(h(ns ++ [n])).
```

This specifies an arbitrary dialog program: to apply it to a particular case just plug in the particular definitions of *Input* and *Output*, and *in* and *out*, and the calculational specification captured in the function *h*. In this specification, state is used for two purposes: firstly, it orders the IO actions in time, and secondly, it stores the list of inputs so far.

6.2.2 Specification

The program will control the tuning of a musical synthesiser. The tuning depends on the musical key of the piece. As the key may change during the piece, the program is to monitor the notes played, deduce the current key, and if there is a change of key, adapt the tuning to the new key. We'll just do the program that always keeps track of the current key. Adapting the current tuning to the current key is an easy extension.

The first subsubsection motivates the program. The second specifies the calculational part of the program formally, and the third gives the complete specification, covering calculation and IO behaviour.

The Maths of Music

This subsubsection briefly explains pure tuning and well-tempered tuning. It motivates this particular program, but contains no definitions or calculations.

It is amazing (and disappointing) that to a large extent, mathematics determines what humans find pleasing. In music we find the sound of two notes played together (an 'interval') pleasing if their frequencies have a simple ratio. The simplest (non-trivial) ratio is 1:2 and it determines the simplest (non-trivial) interval, the octave. Notes one octave apart sound so closely related that we consider them essentially the same. We are therefore free to transpose any note up or down an octave by doubling or halving its frequency. The next simplest ratios are 1:3 (essentially the same as 2:3) giving the interval 'fifth', 1:4 (the same as 1:2) giving another octave, and 1:5 (the same as 4:5) giving the interval 'major third'. Together these notes are a major chord. We can determine frequencies for all the seven notes of a given key by a mathematical procedure like this. For the remaining five chromatic notes there is not one natural choice of frequency, but for example [Bar72] gives several reasonable methods. With such a tuning, the main chords and intervals of the chosen key will sound nice. This is called 'pure tuning', and it depends on which key is chosen.

If a piano is in pure tuning, it will sound nice in one key and wrong in all the other keys. Therefore 'equal-tempered' tuning was invented. In it, an octave still is the ratio 1:2, but all the other intervals are determined by chopping the octave up into twelve equal parts. Each single step upwards increases the frequency by a factor of $\sqrt[12]{2} \approx 1.05946$. Equal-tempered tuning is a compromise and makes every key sound equally out of tune.

The ideal intelligent synthesiser could start in equal-tempered tuning, and follow the notes that are played. As soon as the key, or later a change to a new key, is detected, the instrument retunes itself. It will always be in pure tuning!

Specification

We'll define some values capturing notes and keys, then we'll specify $h : [Note] \to Key$, and finally put together the whole specification of calculational and IO behaviour. For that, we need a type to represent the twelve notes, which we'll call $c, c\sharp, \ldots$ with suitable synonyms like $a\flat = g\sharp$. The names of the notes should not imply any key.

$$\textbf{type}\ Note \ \hat{=}\ \{c, c\sharp, d, d\sharp, e, f, f\sharp, g, g\sharp, a, a\sharp, b\}$$

Each of the twelve (major) keys will be represented by its root note.

$$\textbf{type}\ Key \ \hat{=}\ Note$$

Each key can also be seen as the set of notes. We assume key membership is given:

$$\begin{aligned} \in &: Note \to Key \to \mathbb{B} \\ n \in k &\ \hat{=}\ \text{`}n\text{ is a note in the key }k\text{'} \end{aligned}$$

Key membership can easily but tediously be defined formally by enumerating the notes of each key. We miss out the details here, since there is nothing interesting in it. Now we'll specify h. For each list of notes played so far, we want a key that covers a longest suffix of it.

$$\begin{aligned} h &: [Note] \to Key \\ h\ ns &\ \hat{=}\ \text{`any key that covers a longest suffix of }ns\text{'} \end{aligned}$$

To express the part in quotation marks, we define an auxiliary function that given a list of notes and a key, returns the length of the longest suffix of the list that is covered by the key.

$$\begin{aligned} cover &: [Note] \to Key \to \mathbb{N} \\ cover\ ns\ k &\ \hat{=}\ \#(takewhile(\in k)\ \overleftarrow{ns}) \end{aligned}$$

We note two properties of cover:

$$\begin{aligned} cover\ [\,] &\equiv \lambda k.0 \\ cover\ (ns +\!\!+ [n]) &\equiv \lambda k.\textbf{if}\ n \in k\ \textbf{then}\ (cover\ ns\ k) + 1\ \textbf{else}\ 0. \end{aligned}$$

They are both easily proved. Together they say that *cover* is a homomorphism from *snoc*[3] to some function. In other words, cover may be written *foldl f a* for some *f* and *a*. A key *k* that covers a longest suffix of *ns* will satisfy $cover\ ns\ k' \leq cover\ ns\ k$ for any key k'. We use that to specify *h*:

$$\begin{array}{rcl} h & : & [Note] \to Key \\ h\ ns & \hat{=} & \mathbf{if} \sqcap k : Key.(\forall k' : Key.cover\ ns\ k' \leq cover\ ns\ k) \to k\ \mathbf{fi}. \end{array}$$

The **if** **fi** can be omitted, since they enclose a feasible expression. This function *h* specifies the calculational part of the program. So we'll plug it into our general dialog-program shape of subsection 6.2.1 to get:

$init;\ \lambda v.$
$while\ more\ (in;\ body\ v;\ out)$
where
$init : ST\ s(Ref\ s\ [Note])$
$init \mathrel{\hat{=}} new\ [\,]$
$body : Ref\ s\ [Note] \to Note \to ST\ s\ Key$
$body\ v\ n \mathrel{\hat{=}} get\ v;\ \lambda ns.put\ v\ (ns +\!\!+ [n]);\ \lambda\_.return\ (h(ns +\!\!+ [n])).$

This is the complete specification, with given IO state transformers *more*, *in*, and *out*.

6.2.3 Calculation

The specification is not acceptable as an implementation, since it stores a list of every input so far, and that list may become very long. We aim for a program with a constant small storage. We will later replace the stored list by a stored small array, by data refinement. First, we'll determine what information needs to be stored.

Let's assume the stored information has type X. We need an initial value $start : X$, and a function that adds the information contained in a new input note to the storage: $\hookleftarrow : X \to Note \to X$. We also need to extract the keys from the storage somehow: $use : X \to Key$. The conditions on $start, \hookleftarrow$, and *use* are that they should be cheaply implementable (in particular the type X should use little storage space), and that they yield the same results as the specification. So, extracting a key from the initial store should be the same as the current key of no notes:

$$use\ start \equiv h\,[\,].$$

For every list of notes, adding them to the store and extracting a key should yield the same key as applying *h* to the list:

$$\begin{array}{rcl} use(start \hookleftarrow n_1) & \equiv & h[n_1] \\ use(start \hookleftarrow n_1 \hookleftarrow n_2) & \equiv & h[n_1, n_2] \\ use(start \hookleftarrow \ldots \hookleftarrow n_k) & \equiv & h[n_1, \ldots, n_k] \end{array}$$

[3]an ugly name for a function that is a kind of reverse of *cons*: $snoc\ as\ a \mathrel{\hat{=}} as +\!\!+ [a]$.

In short, we demand

$$use \circ (foldl \hookleftarrow start) \;\equiv\; h.$$

In words, we are seeking a homomorphism from *snoc* to some function $\hookleftarrow$, that composed with *use* yields h. Here are two trivial attempts at a solution: First,

$$\begin{aligned}
&\mathbf{type}\ X && \hat{=}\ [Note]\\
&\hookleftarrow && \hat{=}\ snoc\\
&start && \hat{=}\ [\,]\\
&use && \hat{=}\ h.
\end{aligned}$$

This 'solution' is correct, but useless. It is just the specification, and suffers from unlimited storage demand. Second attempt:

$$\begin{aligned}
&\mathbf{type}\ X && \hat{=}\ Key\\
&\hookleftarrow && \hat{=}\ ...?\\
&start && \hat{=}\ h[\,]\\
&use && \hat{=}\ id.
\end{aligned}$$

This would be ideal. The only thing stored is the previous key. We are looking for $\hookleftarrow$: $Key \rightarrow Note \rightarrow Key$, such that $h(ns +\!\!+ [n]) = h\ ns \hookleftarrow n$, for all n and ns. In words, given current key k, and next note n, $k \hookleftarrow n$ should be the next key. Unfortunately no such $\hookleftarrow$ exists. This can be seen by contradiction: Assume such a $\hookleftarrow$ exists. Consider $ns \hat{=} [c, d, e, f, g, a, b] +\!\!+ [c, d, f, g]$ and $n \hat{=} a\flat$. We then have $h\ ns \equiv c$, and $h(ns +\!\!+ [n]) \equiv e\flat$. Therefore we conclude $c \hookleftarrow a\flat \equiv e\flat$. On the other hand, consider $ns \hat{=} [c, d, e, f, g, a, b] +\!\!+ [d, e, a, b]$ and $n \hat{=} g\sharp$. We have $h\ ns \equiv c$ and $h(ns +\!\!+ [n]) \equiv a$. So we conclude $c \hookleftarrow g\sharp \equiv a$. But $a\flat \equiv g\sharp$, so $c \hookleftarrow a\flat$ should equal $c \hookleftarrow g\sharp$. By contradiction there is no function such $\hookleftarrow$. The problem lies not in the underdetermined specification of h: all four applications of h above are determined. The problem is a musical one. The note $a\flat$ played after a list of notes in the key of c does not uniquely determine the new key. This contradiction also shows that the particular sets and function, $Note, Key, \in$, have no hidden property that allows a trivial implementation of the program.

Let's search by calculation. What do we need to know about ns to calculate $h(ns +\!\!+ [n])$ easily ? Let's calculate:

$$
\begin{aligned}
& h(ns +\!\!+ [n]) \\
\equiv \quad & \quad \text{def. } h\text{, omitting } \mathbf{if}\ \mathbf{fi} \\
& \sqcap k.(\forall k'.cover(ns +\!\!+ [n])k' \leq cover(ns +\!\!+ [n])k) \rightarrow k \\
\equiv \quad & \quad \text{property of } cover \\
& \sqcap k.\forall k'. \\
& \mathbf{if}\ n \in k \cup k' \rightarrow cover\ ns\ k' + 1 \leq cover\ ns\ k + 1 \\
& \sqcap n \in k \setminus k' \rightarrow 0 \leq cover\ ns\ k + 1 \\
& \sqcap n \in k' \setminus k \rightarrow cover\ ns\ k' + 1 \leq 0 \\
& \sqcap n \in k \cap k' \rightarrow 0 \leq 0 \\
& \ \mathbf{fi} \rightarrow k \\
\equiv \quad & \quad \text{maths} \\
& \sqcap k.\forall k'. \\
& \mathbf{if}\ n \in k \cup k' \rightarrow cover\ ns\ k' \leq cover\ ns\ k \\
& \sqcap n \in k \setminus k' \rightarrow True \\
& \sqcap n \in k' \setminus k \rightarrow False \\
& \sqcap n \in k \cap k' \rightarrow True \\
& \ \mathbf{fi} \rightarrow k.
\end{aligned}
$$

The information needed is *cover ns*, a function from keys to naturals. In words, for each key k, we need to know what length suffix of the inputs so far k covers. We let X be the function type from keys to naturals. We already know that *cover* is a homomorphism from *snoc*.

$$
\begin{aligned}
&\mathbf{type}\ X && \hat{=} && Key \rightarrow \mathbb{N} \\
&s \hookleftarrow n && \hat{=} && \lambda k.\mathbf{if}\ n\ k\ \mathbf{then}\ s\ k + 1\ \mathbf{else}\ 0 \\
&start && \hat{=} && \lambda k.0
\end{aligned}
$$

The selection of a best key is left to the function *use* (again, we can omit **if fi**):

$$
\begin{aligned}
&use && : && (Key \rightarrow N) \rightarrow Key \\
&use\ s && \hat{=} && \mathbf{if} \sqcap k : Key.(\forall k' : Key.s\ k' \leq s\ k) \rightarrow k\ \mathbf{fi}
\end{aligned}
$$

The type *Key* contains only twelve elements, so $Key \rightarrow \mathbb{N}$ can be implemented by an array with twelve cells indexed by notes and each containing a natural. We still have to prove $use \circ (foldl \hookleftarrow start) \equiv h$, but this is easily done by *snoc*-induction on the argument list, and using the two properties of cover.

6.2.4 Data Refinement

The complete specification was:

$init;\ \lambda v.$
$while\ more\ (in;\ body\ v;\ out)$
where
$init : ST\ s\ (Ref\ s\ [Note])$
$init \mathrel{\hat{=}} new\ [\,]$
$body : Ref\ s\ [Note] \rightarrow Note \rightarrow ST\ s\ Key$
$body\ v\ n \mathrel{\hat{=}} get\ v;\ \lambda ns.put\ v\ (ns +\!\!+ [n]);\ \lambda v\_.return(h(ns +\!\!+ [n]))$
$h : [Note] \rightarrow Key$
$h\ ns \mathrel{\hat{=}} \sqcap k.(\forall k'.cover\ ns\ k' \leq cover\ ns\ k) \rightarrow k$

$cover : [Note] \to Key \to N$
$cover\ ns\ k \mathrel{\hat{=}} \#(takewhile(\in k)\ \overleftarrow{ns})$

We want to represent the abstract type $[Note]$ by the concrete type $Key \to \mathbb{N}$. The representation relation is

$$\begin{array}{rcl} I & : & [Note] \leftrightarrow (Key \to \mathbb{N}) \\ ns\ I\ s & \hat{=} & cover\ ns =_{Key \to \mathbb{N}} s, \end{array}$$

where $f =_{Key \to \mathbb{N}} g \mathrel{\hat{=}} \forall k : Key. f\ k =_{\mathbb{N}} g\ k$. We data-refine the program be decomposing it into subexpressions and replacing subexpressions of types depending on $[Note]$.

The first such subexpression is $[\,]$ in the definition of *init*. It will be replaced by $\lambda k.0$. The proof for $[\,] \ll_{[\,],I} \lambda k.0$ is simply the first of the two properties of cover.

The second subexpression of type dependent on $[Note]$ is

$$\lambda v.while\ more\ (in;\ body\ v;\ out)$$

of type $RefIO\ [Note] \to IO\ [()]$. We deal with it using the rule for data refining λ expressions, which means in effect, we simply change $v : RefIO\ [Note]$ to $v' : RefIO\ (Key \to \mathbb{N})$, and continue data-refining subexpressions under the assumption $v \ll_{RefIO\ [\,],I} v'$. In fact, let's do this trivial change mentally, and keep the identifier v.

We similarly have to data-refine the λ expression

$$\lambda ns.put\ v(ns +\!\!+ [n]);\ \lambda\_.return(h(ns +\!\!+ [n])).$$

This time we will change the identifier from $ns : [Note]$ to $s : Key \to \mathbb{N}$. We continue data-refining the body of the λ expression, under the assumption that $ns \ll_{[\,],I} s$.

We'll replace the subexpression $ns +\!\!+ [n]$ by $\lambda k.\mathbf{if}\ n \in k\ \mathbf{then}\ s\ k + 1\ \mathbf{else}\ 0$. The proof uses the second property of cover:

$$\begin{array}{ll} & (ns +\!\!+ [n]) \ll_{[\,],I} (\lambda k.\mathbf{if}\ n \in k\ \mathbf{then}\ s\ k + 1\ \mathbf{else}\ 0) \\ \equiv & \qquad \text{def. } I \\ & cover(ns +\!\!+ [n])k \equiv \mathbf{if}\ n \in k\ \mathbf{then}\ s\ k + 1\ \mathbf{else}\ 0 \\ \Leftarrow & \qquad \text{property of } cover \\ & cover\ ns =_{Key \to \mathbb{N}} s \\ \equiv & \qquad \text{def. } I \\ & ns\ I\ s. \end{array}$$

Finally, we have to data-refine the expression h, appearing at the end of the definition of *body*. We'll replace it by *use*, defined in subsection 6.2.3. For the proof we use the compositional laws of data refinement.

$$
\begin{array}{ll}
& h \ll_{[\,]\to Key, I} use \\
\Leftarrow & \qquad \forall\, ns : [Note], s : Key \to \mathbb{N} \\
& (ns \ll_{[\,], I} s) \Rightarrow (h\ ns \ll_{Key, I} use\ s) \\
\equiv & \\
& (cover\ ns =_{Key \to \mathbb{N}} s) \Rightarrow (h\ ns \equiv use\ s) \\
\equiv & \\
& use(cover\ ns) \equiv h
\end{array}
$$

...and that's an equivalence we noted in subsection 6.2.3. The complete data-refined program reads:

$$
\begin{array}{ll}
\multicolumn{2}{l}{init;\ \lambda\, v.} \\
\multicolumn{2}{l}{while\ more\ (in;\ body\ v;\ out)} \\
\multicolumn{2}{l}{\textbf{where}} \\
init & : ST\ s\ (Ref\ s\ (Key \to \mathbb{N})) \\
init & \hat{=}\ new(\lambda\, k.0) \\
body & : Ref\ s\ (Key \to \mathbb{N}) \to Note \to ST\ s\ Key \\
body\ v\ n & \hat{=}\ get\ v;\ \lambda\, s. \\
& \quad put\ v\ (\lambda\, k.\textbf{if}\ n \in k\ \textbf{then}\ s\ k + 1\ \textbf{else}\ 0);\ \lambda\_. \\
& \quad return(use(\lambda\, k.\textbf{if}\ n \in k\ \textbf{then}\ s\ k + 1\ \textbf{else}\ 0)) \\
use & : (Key \to \mathbb{N}) \to Key \\
use\ s & \hat{=}\ \sqcap k.(\forall\, k'.s\ k' \leq s\ k) \to k.
\end{array}
$$

6.3 Choosing a Free Printer

This example program illustrates the use of state to add more detail to a program that is already imperative. Here the program is imperative because it controls a part of the real world: a bench of printers.

This derivation illustrates the use of generalized choice in specifications to generate arbitrary values. It is also an example of a program-improvement that is not reflected in the technical notion of refinement: improving the fairness of program implementing a nondetermined specification.

6.3.1 Specification

There is a bench of N printers, indexed by $0..N-1$. They are controlled by a system that offers the users only one operation: to send a document to an arbitrary printer. If no printer is free the operation fails. The system is initialised by *init*.

$$
\begin{array}{l}
init : IO\ (Doc \to IO\ ()) \\
init\ \hat{=}\ return\ \ print \\
\qquad \textbf{where}\ print\ d \mathrel{\hat{=}} \lambda\, \sigma.\textbf{if}\ \sqcap i : \{0..N-1\}.free\ i\ \sigma \to pp\ d\ i\ \sigma\ \textbf{fi}
\end{array}
$$

The state transformer *print* is specified and will be implemented in terms of two low level operations. One of them is the state reader $free : \mathbb{N} \to IO\ \mathbb{B}$ which tests whether an indexed printer is free. The other is $pp : Doc \to \mathbb{N} \to IO\ ()$ which prints a document on an indexed printer. That printer must be free, otherwise *pp* fails.

6.3.2 Implementing the Search for a Free Printer by a Loop

We will need to calculate modulo N. Let underlining denote taking a number modulo N, that is $\underline{n} \hat{=} n \; mod \; N$. The prescription in the specification is refined by a loop that starts at an arbitrary index, and steps through the indexes until it finds the index of a free printer. If there is a free printer, the loop will find one and terminate. If there is no free printer, the loop will keep going until there is. Since choosing an arbitrary index from a non-empty range is feasible, we can omit the **if fi** below.

$$\begin{array}{l} \quad \mathit{print} \; d \\ \sqsubseteq \\ \quad \mathit{rec}(\mathbf{if} \sqcap i : \mathbb{N}.0 \leq i < N \rightarrow i \; \mathbf{fi}); \; \lambda\, i.\mathit{pp} \; d \; i \\ \quad \mathbf{where} \; \mathit{rec} \; i \; \hat{=} \; \mathit{ro}(\mathit{free} \; i); \; \lambda\, b.\mathbf{if} \; b \; \mathbf{then} \; \mathit{return} \; i \; \mathbf{else} \; \mathit{rec} \; \underline{i+1} \end{array}$$

6.3.3 A Simple Refinement

The expression still contains a subexpression not in the programming language: the choice of the first index to be tested is left open. We can easily refine that to any fixed integer modulo N, for example 0.

However, this would lead to a very uneven use of the printers. The lower the index of a printer, the faster it would wear out. Printer 0 would have to be replaced soon. Still, the program does implement the specification. Asking for an implementation that uses the printers evenly is adding a requirement that is not captured in the specification. It doesn't seem easy to add that requirement to the specification.

But asking for it is not unreasonable – even without changing the specification. The specification only captures some of the desired properties. Others it leaves out, like speed, or program size, or in this case, fairness of printer selection.

6.3.4 A Better Refinement

A fairer implementation would not start the search at the same index each time. Instead, it could store the index of the last printer used in an extra reference and start the next search from that index.

The initialisation will allocate a reference, store an arbitrary element of $\{\underline{n} \mid n \in \mathbb{N}\}$ in it, and bind the reference to a variable, v say. The new reference will have the invariant that the stored integer will always be a valid printer index, that is, an integer in $\{0..N-1\}$. With this invariant the reference can always be used to get a starting index for a search, and storing the index of the last used printer obviously preserves the invariance.

$$\begin{array}{l} \mathit{init} \; \hat{=} \; \mathit{new} \; (\sqcap i : \mathbb{N}.0 \leq i < N \rightarrow i); \; \lambda\, v.\mathit{return} \; \mathit{print} \\ \qquad \mathbf{where} \; \mathit{print} \; d \; \hat{=} \; \mathit{get} \; v; \; \lambda\, i.\mathit{rec} \; i; \; \lambda\, i.\mathit{put} \; v \; i, \end{array}$$

and *rec* is as before.

6.4 A Substring Searching Algorithm

The task is to search for occurrences of a string w (called the *pattern*) as substring of t (the *text*). A naive substring searching algorithm has time complexity $O(mn)$, where m is the length of w, and n is the length of t. A complexity of $O(m^2 + n)$ can be achieved by some precomputation: before the pattern is compared to the text, it is compared to prefixes of itself. The results of the precomputation are stored in an array, and are looked up later in the program. The complexity $O(m^2 + n)$ is a satisfactory improvement over $O(mn)$ if we assume the pattern to be short and the text long. In the Knuth-Morris-Pratt algorithm the complexity is further reduced to $O(m + n)$ by exploiting the order of dependence of the precomputed values.

In the following, we use the naive $O(mn)$ algorithm as specification, and illustrate the general method of precomputation by deriving the $O(m^2 + n)$ algorithm. We don't take the step to the Knuth-Morris-Pratt algorithm with complexity $O(m + n)$ since it relies on a particular property of the precomputed values.

The Knuth-Morris-Pratt algorithm was first presented in [KMP77]. It has been derived in [Dij76, Dro82], and [BGJ89]. The last derivation by Bird, Gibbons, and Jones, is particularly interesting to us, because it is presented in a functional language. The program is specified as an inefficient functional program, which is refined in two phases: First, a more efficient functional program is derived. This program does not have the desired complexity, because it keeps recomputing matches of the pattern against itself. In the second phase, a functional table is introduced to avoid this recomputation. The method of *tabulation* is taken from [Bir80], and is not easy to understand.

In this section, we'll summarise the first phase of the program development from [BGJ89], but then introduce an array to store the table, rather than use Bird's tabulation technique. This means we'll have to add state to the program. We don't consider that a disadvantage, since in a practical implementation (some of) the strings will likely be implemented by indices into a long array of characters held in the state, so the program will be imperative anyway. This intended representation of strings also justifies our use of expressions of the form $xs +\!\!+ [x]$ in the program: With index manipulation, this is as cheap as $x : xs$.

6.4.1 Specification

We are to find occurrences of the (short) not empty string w called the *pattern* in the (long) string t (called the *text*), that is, we need an efficient implementation of

$$\begin{array}{ll} \mathit{match} & : \mathit{String} \to \mathit{String} \to [\mathbb{B}] \\ \mathit{match}\ w\ t & \hat{=} (\lambda\, xs.w \in \mathit{tails}\ xs)^*(\mathit{inits}\ t). \end{array}$$

The ith boolean in the list *matches* w t tells whether w is identical to positions $i - \#w$ to i of the string t. The above specification is already an executable program of complexity $O(mn)$, if we assume that $w \in \mathit{tails}\ xs$ is implemented with complexity $O(m)$.

The auxiliary functions *inits* and *tails* are defined:

$$\begin{array}{ll} \mathit{inits}\ [\,] & \hat{=}\ [[\,]] \\ \mathit{inits}\ (x : xs) & \hat{=}\ [\,] : (x\,:)^* \mathit{inits}\ xs \\ \mathit{tails}\ [\,] & \hat{=}\ [[\,]] \\ \mathit{tails}\ (x : xs) & \hat{=}\ (x : xs) : \mathit{tails}\ xs. \end{array}$$

6.4.2 Calculation

We aim to apply the equivalence (lemma 5 in [Bir87])

$$(\mathit{foldl}\ \oplus\ e)^*(\mathit{inits}\ t) \quad \equiv \quad \mathit{scanl}\ \oplus\ e\ t.$$

This is a complexity-reducing transformation, since the left side involves $O(n^2)$ applications of $\oplus$, whereas the right has only $O(n)$.

We can use this transformation if we can express $\lambda\, xs.w \in \mathit{tails}\ xs$ using *foldl*. However, the function does not produce enough information for this to be done. The inventive step of the derivation is therefore to generalise this function to

$$\begin{array}{ll} f & : \mathit{String} \rightarrow \mathit{String} \rightarrow \mathit{String} \\ f\ w\ xs & \hat{=}\ \uparrow_{\#}/(\mathit{inits}\ w \cap \mathit{tails}\ xs). \end{array}$$

Here $\uparrow_{\#}/$ is some function that returns a longest string from a not-empty set of strings. We have $\lambda\, xs.w \in \mathit{tails}\ xs \equiv (w\ =) \circ (f\ w)$. The function f can be expressed by *foldl*: [BGJ89] shows that $f\ w \equiv \mathit{foldl}\ \oplus\ [\,]$, where

$$\begin{array}{lll} \oplus & : & \mathit{String} \rightarrow \mathit{Char} \rightarrow \mathit{String} \\ xs \oplus x & \hat{=} & \textbf{if}\ \mathit{delta}\ w\ x\ xs \rightarrow \quad xs +\!\!+ [x] \\ & & \sqcap\ \neg \mathit{delta}\ w\ x\ xs \wedge x = [\,] \rightarrow \quad [\,] \\ & & \sqcap\ \neg \mathit{delta}\ w\ x\ xs \wedge x \neq [\,] \rightarrow \quad f\ w\ (\mathit{tl}\ xs) \oplus a \\ & & \textbf{fi} \\ \mathit{delta}\ w\ x\ xs & \hat{=} & xs \neq w \wedge x = \mathit{hd}(xs^{-1} +\!\!+ w). \end{array}$$

By $xs^{-1} +\!\!+ w$ we mean the string ys such that $xs +\!\!+ ys = w$, or $\perp$ if none such string exists. The complexity of $xs \oplus x$ is not $O(1)$ yet – the last branch prevents it – but later, after precomputing the necessary values of $f\ w\ (\mathit{tl}\ xs)$ it will be.

The program becomes

$$\begin{array}{ll} & \mathit{match}\ w\ t \\ \equiv & \qquad \text{def. } \mathit{match} \\ & (\lambda\, xs.w \in \mathit{tails}\ xs)^*(\mathit{inits}\ t) \\ \equiv & \qquad \text{def. } f \\ & (w\ =)^*((f\ w)^*(\mathit{inits}\ t)) \\ \equiv & \qquad f \text{ using } \mathit{foldl} \\ & (w\ =)^*((\mathit{foldl}\ \oplus\ [\,])^*(\mathit{inits}\ t)) \\ \equiv & \qquad \mathit{foldl}/\mathit{scanl} \\ & (w\ =)^*(\mathit{scanl}\ \oplus\ [\,]\ t). \end{array}$$

Assuming the complexity of $\oplus$ were $O(1)$, the complexity of this program is still $O(mn)$, since each of the n elements of the list produced by *scanl* is compared to w – an $O(m)$ operation. We optimise these final comparisons by extending f to a function h that returns a pair of strings. The first string is the same as f's result, whereas the second is its complement with respect to w. That way, instead of comparing the first string to w (an $O(m)$ operation), we simply compare the second to $[]$ (an $O(1)$ operation). The definition of h is:

$$\begin{array}{ll} h & : \mathit{String} \to \mathit{String} \to \mathit{String}^2 \\ h\ w\ xs & \hat{=}\ (f\ w\ xs, (f\ w\ xs)^{-1} + \!\!+\ w). \end{array}$$

Like f, h can be expressed using *foldl*. We have $h\ w \equiv \mathit{foldl}\ \otimes\ ([], w)$, where

$$\begin{array}{lcll} \otimes & : & \mathit{String}^2 \to \mathit{Char} \to \mathit{String}^2 & \\ (\mathit{fs}, \mathit{gs}) \otimes x & \hat{=} & \textbf{if}\ \mathit{delta}\ w\ x\ \mathit{fs}\ \mathit{gs} \to & (\mathit{fs} + \!\!+ [a], \mathit{tl}\ \mathit{gs}) \\ & & \sqcap \neg \mathit{delta}\ w\ x\ \mathit{fs}\ \mathit{gs} \wedge \mathit{fs} = [] \to & ([], w) \\ & & \sqcap \neg \mathit{delta}\ w\ x\ \mathit{fs}\ \mathit{gs} \wedge \mathit{fs} \neq [] \to & h\ w\ (\mathit{tl}\ \mathit{fs}) \otimes x) \\ & & \textbf{fi} & \\ \mathit{delta}\ w\ x\ \mathit{fs}\ \mathit{gs} & \hat{=} & \mathit{gs} \neq [] \wedge x = \mathit{hd}\ \mathit{gs}. & \end{array}$$

The second string gs in the pair is also used to optimise the decider function *delta*. We have $(\lambda\, xs.w \in \mathit{tails}\ xs) \equiv ([] =) \circ \mathit{snd} \circ (h\ w)$, as desired. The program becomes

$$\begin{array}{ll} & \mathit{match}\ w\ t \\ \equiv & \quad \text{def. } \mathit{match} \\ & (\lambda\, xs.w \in \mathit{tails}\ xs)^*(\mathit{inits}\ t) \\ \equiv & \quad \text{def. } h \\ & (([] =) \circ \mathit{snd})^*((h\ w)^*(\mathit{inits}\ t))) \\ \equiv & \quad h \text{ using } \mathit{foldl} \\ & (([] =) \circ \mathit{snd})^*((\mathit{foldl}\ \otimes\ ([], w))^*(\mathit{inits}\ t)) \\ \equiv & \quad \mathit{foldl}/\mathit{scanl} \\ & (([] =) \circ \mathit{snd})^*(\mathit{scanl}\ \otimes\ ([], w)\ t). \end{array}$$

The complexity of the program would be $O(n)$, if we can implement $\otimes$ in constant time. And indeed, all the operations in the definition of $\otimes$ are constant-time, except for the $h\ w\ (\mathit{tl}\ \mathit{fs})$ occurring in the last branch. Its value may be required for each $\mathit{fs} \in \mathit{tl}(\mathit{inits}\ w)$. We'll precompute those m values (at $O(m^2)$ cost), and store them in an array for later use. The overall complexity of the program will then be $O(m^2 + n)$ as desired.

6.4.3 Preparing for Precomputation

First, we introduce state to those parts of the program that will access the array. We use a state-carrying version of *scanl* called *scanlST*, defined by

$$\begin{array}{ll} \mathit{scanlST} & : (a \to b \to ST\ s\ a) \to a \to [b] \to ST\ s\ [a] \\ \mathit{scanlST}\ \odot\ a\ [] & \hat{=}\ \mathit{return}\ [a] \\ \mathit{scanlST}\ \odot\ a\ (b : bs) & \hat{=}\ a \odot b;\ \lambda\, a'. \\ & \quad \mathit{scanlST}\ \odot\ a'\ bs;\ \lambda\, as. \\ & \quad \mathit{return}\ (a : as). \end{array}$$

It can easily be shown (by induction on bs) that

$$\mathit{scanlST}\ (\lambda\, a.\lambda\, b.\mathit{return}\ (a \oplus b))\ a\ bs \equiv \mathit{return}\ (\mathit{scanl}\ \oplus\ a\ bs).$$

The program becomes

$((\,[] =) \circ snd)^*(scanl\ \otimes\ ([], w)\ t)$

$\equiv$ **run** intro, law 3

$((\,[] =) \circ snd)^*\mathbf{run}(return\ (scanl\ \otimes\ ([], w)\ t))$

$\equiv$ $scanl/scanlST$

$((\,[] =) \circ snd)^*\mathbf{run}(scanlST\ (\lambda\, a.\lambda\, b.return\ (a \otimes b))\ ([], w)\ t)$

We can push the *return* into the definition of $\otimes$:

$return\ ((fs, gs) \otimes x)$

$\equiv$ def. $\otimes$, function alternation

```
if $delta\ w\ x\ fs\ gs \rightarrow$                 $return\ (fs ++ [x], tl\ gs)$
⊓ $\neg delta\ w\ x\ fs\ gs \wedge fs = [] \rightarrow$   $return\ ([], w)$
⊓ $\neg delta\ w\ x\ fs\ gs \wedge fs \neq [] \rightarrow$   $return\ (h\ w\ (tl\ fs) \otimes x))$
fi
```

$\equiv$ left *return* , law 1; β equivalence

```
if $delta\ w\ x\ fs\ gs \rightarrow$                 $return\ (fs ++ [x], tl\ gs)$
⊓ $\neg delta\ w\ x\ fs\ gs \wedge fs = [] \rightarrow$   $return\ ([], w)$
⊓ $\neg delta\ w\ x\ fs\ gs \wedge fs \neq [] \rightarrow$   $(\lambda\, xs.return\ (h\ w\ (tl\ xs)))\ fs;\ \lambda\, p.return\ (p \oplus x))$
fi
```

We pull the function $\lambda\, xs.return\ (h\ w\ (tl\ xs))$ out to the front of the program:

```
$((\,[] =) \circ snd)^*$
$\mathbf{run}(return\ (\lambda\, xs.return\ (h\ w\ (tl\ xs)));\ \lambda\, tf.$
  $scanlST\ \odot\ ([], w)\ t$
   where $(fs, gs) \odot x \doteq$ if $delta\ w\ x\ fs\ gs \rightarrow$                 $return\ (fs ++ [x], tl\ gs)$
                          ⊓ $\neg delta\ w\ x\ fs\ gs \wedge fs = [] \rightarrow$   $return\ ([], w)$
                          ⊓ $\neg delta\ w\ x\ fs\ gs \wedge fs \neq [] \rightarrow$   $tf\ fs;\ \lambda\, p. \oplus x)$
                          fi
  ).
```

6.4.4 Precomputation

We have prepared the program for precomputation. Now all we need to do is replace the first state transformer in the above program by one which allocates and fills up an array that holds the precomputed results of $h\ w\ (tl\ xs)$ for $xs \in tl(inits\ w)$, and returns an array-indexing state transformer.

```
$precompute$           : $(a \rightarrow b) \rightarrow [a] \rightarrow ST\ s\ (a \rightarrow ST\ s\ b)$
$precompute\ f\ range \doteq arrayST\ range\ (f^*range);\ \lambda\, m.$
                        $return\ (\lambda\, a.getArray\ m\ a).$
```

Here *arrayST* takes two lists, and allocates and returns a reference to a mutable array indexed by the elements of the first list and containing the elements of the second list. The function *getArray* takes an array reference, and an index a, and returns the contents of the ath cell of the array. For indexing to be determined, the range should be a list without duplicates. For it to be efficient, the indices should be represented by an initial part of the naturals. We'll indeed use a duplicate-free range, but won't make the integer representation explicit.

We use the equivalence

$$precompute\ f\ range \equiv return\ (\lambda\, a.a \in range \succ\!\!- return\ (f\ a))$$

to put precomputation into the program. A formal proof of this equivalence would require more investigation of state in combination with invariants, scope, and garbage collection. Here, the function to be precomputed is $\lambda\, xs.h\ w\ (tl\ xs)$ and the range is $tl(inits\ w)$. For array indexing, the elements of the range can be represented by their lengths.

The program with precomputation reads:

```
$(([] =) \circ snd)^*$
$\textbf{run}(precompute\ (\lambda\, xs.h\ w\ (tl\ xs)\ (tl(inits\ w)));\ \lambda\, tf.$
    $scanlST\ \odot\ ([], w)\ t$
     $\textbf{where}\ (fs, gs) \odot x \mathrel{\hat=}$  $\textbf{if}\ delta\ w\ x\ fs\ gs \rightarrow$                $return\ (fs +\!\!+ [x], tl\ gs)$
                           $\sqcap \neg delta\ w\ x\ fs\ gs \wedge fs = [] \rightarrow$  $return\ ([], w)$
                           $\sqcap \neg delta\ w\ x\ fs\ gs \wedge fs \neq [] \rightarrow$  $tf\ fs;\ \lambda\, p. \oplus x)$
                           $\textbf{fi}$
   ).
```

The complexity of this program is $O(m^2 + n)$. After precomputation, the program takes $O(n)$ steps as desired, and precomputing the m values at up to m steps each is $O(m^2)$. The Knuth-Morris-Pratt algorithm reduces that to $O(m)$ by precomputing the values in ascending order of arguments, and re-using the results so far. But since m is assumed small, we'll be satisfied with overall complexity $O(m^2 + n)$ instead of $O(m + n)$ as in the KMP algorithm.

Under a lazy evaluation strategy, the program above will calculate the values of $h\ w\ (tl\ xs)$ at most once, and never if they are not required. Under an eager evaluation strategy, it would precompute all the values and store them in the array, before the *scanlST* is evaluated. We can prevent computation of values that will never be needed by refining *precompute* to *memo*:

```
$memo$            : $(a \rightarrow b) \rightarrow [a] \rightarrow a \rightarrow ST\ s\ b$
$memo\ f\ range \mathrel{\hat=}$ $arrayST\ range\ [No \mid r \leftarrow range];\ \lambda\, m.$
                 $return\ (\lambda\, a.getArray\ m\ a;\ \lambda\, c.$
                      $\textbf{case}\ c\ \textbf{of}$
                        $No \rightarrow$  $\textbf{let}\ b \mathrel{\hat=} f\ a\ \textbf{in}$
                               $putArray\ m\ a\ (Yes\ b);\ \lambda\_.$
                               $return\ b$
                      $\sqcap\ Yes\ b \rightarrow return\ b).$
```

Precomputing a function lazily is memoizing it.

Chapter 7

Lines, Circles, Spheres

7.1 Introduction

This chapter gives derivations of three related programs: Bresenham's line drawing algorithm [Bre65], Bresenham's circle drawing algorithm [Bre77], and the fast sphere drawing algorithm [FGS$^+$85, Pat93]. They calculate the pixels that best approximate a line, a circle, or a sphere, and display them on a screen.

Calculating the pixels can be written without using state. We will still express it with state for two reasons.

The first is that an imperative program seems – to many people – a natural description of the algorithms. In each algorithm, a bunch of variables is initialised, and repeatedly modified, each time producing one pixel. It is natural to express this repeated modification by in-place updates.

The second reason for an imperative formulations is that displaying pixels on a screen is an IO operation, and therefore necessarily imperative. If calculating the pixels is expressed imperatively, displaying and calculating can be merged, so that – even without a lazy evaluation strategy – each pixel is displayed as soon as it's calculated.

All three algorithms make only static use of state, that is, the number of updatable variables is fixed. No pointers are required, just in-place updates.

These derivations show examples of calculating with state transformers, in particular, introducing state into a state-free program, and moving more and more of the calculational work into the state. The program is first formulated in terms of higher-order functions. A theorem is used that transforms programs written in such a way into equivalent imperative ones. The proof of the theorem illustrates most of the state monad axioms. The final stages illustrate the techniques of flattening nested state threads, and of commuting state transformers.

Mathematically, lines, circles, and spheres are infinite sets of real number pairs or triples. These infinite sets are mapped to finite sets of integer tuples by rounding. We will use these sets as specifications.

The task is to derive algorithms that enumerate the elements of the sets, preferably using integer operations rather than real number operations.

We will use recurrence relations of the set elements to express the sets in terms of the standard higher order combinators *take*, *map*, and *iterate* (for example found in

Haskell [HPW92]). We give an imperative implementation of a combination of these combinators. Using this implementation in the three programs yields three imperative programs. Each calculates a list of pixels which are then fed to the displaying IO transformer.

Finally, we push the IO transformer that displays the pixels forward so that each pixel is displayed as soon as it's calculated.

The three algorithms have the same structure; we will derive them in parallel. The sphere algorithm re-uses part of the circle algorithm.

In [Spr82] Bresenham's line drawing algorithm is used to demonstrate program derivation by transformation, using a form of Pascal extended with real numbers (as opposed to floating point numbers). The transformation steps are justified informally.

In [Wir90] the line and circle drawing algorithms are presented in the imperative guarded command language. The initial specification there is $x := 0$; **do** $x < a \rightarrow y := (b * x)$ *DIV* a; $Mark(x, y)$; $x := x + 1$ **od**.

In [Sne93] Bresenham's line drawing algorithm is presented in the imperative guarded command-language. Its semantics is given by Hoare-triples.

7.2 Specifications

We will give the specifications in terms of set comprehensions and real numbers. The derivation will have to substitute the real numbers by integers, and give an algorithm to enumerate the sets. We'll do some initial transformations to get expressions of the form *map f* $[i..j]$ for some function f and some integer range $[i..j]$.

Before we start, let us define some types for two- and three- dimensional points and pixels.

type $Point2 \mathrel{\hat{=}} \mathbb{R} \times \mathbb{R}$
type $Point3 \mathrel{\hat{=}} \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
type $Pixel2 \mathrel{\hat{=}} \mathbb{Z} \times \mathbb{Z}$
type $Pixel3 \mathrel{\hat{=}} \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$

7.2.1 Lines

The task is to find a set of pixels L that represent well the points of the straight line segment from the point (x_1, y_1) to (x_2, y_2). For simplicity we assume that $x_1, x_2, y_1, y_2 \in \mathbb{Z}$, and that the slope of the line is between 0 and 1, so $0 \leq \frac{dy}{dx} \leq 1$, where $dy \mathrel{\hat{=}} y_2 - y_1$ and $dx \mathrel{\hat{=}} x_2 - x_1$. The points of the line segment are

$$\{(x, y) \mid x, y \in \mathbb{R}, y - y_1 = \frac{dy}{dx}(x - x_1), x_1 \leq x \leq x_2\}.$$

This is an infinite set of points. We round to get a finite set of pixels. We express the vertical coordinate as a function of the horizontal coordinate:

$$\{(round\ x, round(l\ x)) \mid x \in \mathbb{R}, x_1 \leq x \leq x_2\}.$$

The gradient of a curve given by function f is the derivative of f, written f'. We can represent the points of a curve with a 'shallow' gradient well by choosing one pixel in each column of pixels. That means we can eliminate real numbers from the horizontal coordinate.

Theorem 5 (Shallow gradient) *If* $|f'x| \leq 1$ *for* x *such that* $a \leq x \leq b$, *then*

$$\{(\mathit{round}\ x, \mathit{round}(f\ x)) \mid x \in \mathbb{R}, a \leq x \leq b\}$$

$$=$$

$$\{(i, \mathit{round}(f\ i)) \mid i \in \mathbb{Z}, \mathit{round}\ a \leq i \leq \mathit{round}\ b\}.$$

The gradient of l is $\frac{dy}{dx}$, which is indeed shallow by assumption. We can apply the theorem to get:

$$\{(i, \mathit{round}(l\ i)) \mid i \in \mathbb{Z}, x_1 \leq i \leq x_2\}.$$

Let's abbreviate $\mathit{round} \circ l$ by rl, and implement the unordered data structure set by the ordered data structure list:

$$L \mathrel{\hat{=}} \mathit{map}\ (\lambda\, i.(i, \mathit{rl}\ i))\ [x_1..x_2].$$

7.2.2 Circles

The task here is to select a set of pixels C that represent the points of the circle around the origin of radius $r \in \mathbb{N} \setminus \{0\}$. The circle is given by

$$\{(x, y) \mid x, y \in \mathbb{R}, x^2 + y^2 = r^2\}.$$

A circle around the origin can be generated trivially from the arc in one of its octants by symmetry. We will calculate the pixels of the second octant because in it, the gradient of the curve is shallow.

$$\{(x, y) \mid x, y \in \mathbb{R}, x^2 + y^2 = r^2, 0 \leq x \leq \frac{r}{\sqrt{2}}\}$$

The limit $\mathit{round}(r/\sqrt{2})$ for the range of i is derived from geometry. See figure 7.1, in which $\sqrt{2}$ is written $\mathit{sqrt}(2)$.

Once again, we round to get a finite set of pixels, and express the vertical coordinate as a function of the horizontal coordinate:

$$\{(\mathit{round}\ x, \mathit{round}(c\ x)) \mid x \in \mathbb{R}, 0 \leq x \leq \frac{r}{\sqrt{2}}\},$$

where

$$\begin{aligned} &c\ \ : \mathbb{R} \rightarrow \mathbb{R} \\ &c\ x \mathrel{\hat{=}} \sqrt{r^2 - x^2}. \end{aligned}$$

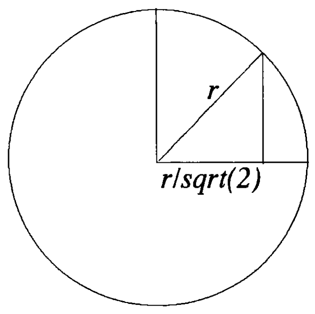

Figure 7.1: Horizontal limits of the arc of the second octant

We observe that indeed for $0 \le x \le \frac{r}{\sqrt{2}}$ the gradient is shallow ($|c'\ x| \le 1$), and apply theorem 5. We implement the set by a list, and abbreviate *round* $\circ$ *c* to *rc*:

$$C \mathrel{\hat{=}} map\ (\lambda\, i.(i, rc\ i))\ [0..round \frac{r}{\sqrt{2}}].$$

7.2.3 Spheres

A sphere of radius $r \in \mathbb{N} \setminus \{0\}$ around the origin is given by this set comprehension:

$$\{(x, y, z) \mid x, y, z \in \mathbb{R}, x^2 + y^2 + z^2 = r^2\}.$$

We'll only calculate the points (x, y, z) of the second octant (where $0 \le x \le y$) of the front hemisphere (where $0 \le z$), since the other points can then easily be generated by symmetry.

$$\{(x, y, z) \mid x, y, z \in \mathbb{R}, x^2 + y^2 + x^2 = r^2, 0 \le x \le y, 0 \le z\}$$

We round to get a finite set of three dimensional pixels:

$$\{(round\ x, round\ y, round\ z) \mid x, y, z \in \mathbb{R}, x^2 + y^2 + z^2 = r^2, 0 \le x \le y, 0 \le z\}.$$

The pixels will be displayed on a screen by projecting (i, j, k) to the two dimensional pixel (i, j) and representing k by giving (i, j) an appropriate colour. Clearly for fixed i and j, of the set $\{(i, j, round\ z) \mid x, y, z \in \mathbb{R}, x^2+y^2+z^2 = r^2, i = round\ x, j = round\ y\}$, only one pixel can be displayed. We choose $(i, j, round\sqrt{r^2 - i^2 - j^2})$.

The program is to enumerate the set

$$\begin{aligned} S \ \mathrel{\hat{=}}\ & \{(i, j, round(s(i, j))) \mid x, y \in \mathbb{R}, 0 \le x \le y, x^2 + y^2 \le r^2, \\ & i, j \in \mathbb{Z}, i = round\ x, j = round\ y\}, \end{aligned}$$

where

$$\begin{aligned} &s && : Pixel2 \to \mathbb{R} \\ &s(i, j) && \mathrel{\hat{=}} \sqrt{r^2 - i^2 - j^2}. \end{aligned}$$

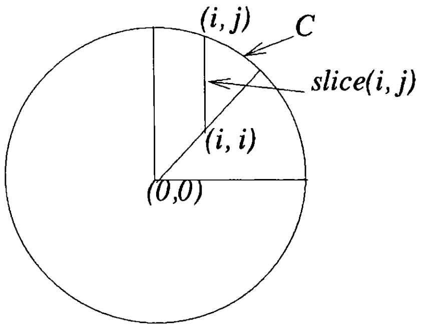

Figure 7.2: A slice

We will calculate S by partitioning it into subsets with constant first coordinate, and taking the union of the subsets. Geometrically a subset with constant first coordinate is a slice of the shape we aim at. See figure 7.2.

The set C is the set of pixels representing the points on the arc of the second octant of the circle. It will be implemented by Bresenham's circle drawing algorithm. We will later re-use its implementation in the implementation of sphere drawing.

Given a pixel near the arc, the slice is delivered by the function

$$\begin{aligned} &\mathit{slice} && : \mathit{Pixel2} \rightarrow \mathbb{P}\,\mathit{Pixel3} \\ &\mathit{slice}(i,j) && \hat{=} \{(i,k,\mathit{rs}(i,k)) \mid k \in \mathbb{Z}, i \leq k \leq j\}, \end{aligned}$$

where rs abbreviates $\mathit{round} \circ s$.

We have $S \equiv (\cup/ \circ \mathit{slice}^*)\ C$. We'll continue working on *slice*. We implement the set by a list and express it using *map*.

$$\mathit{slice}(i,j) \equiv \mathit{map}\ (\lambda k.(i,k,\mathit{rs}(i,k)))\ [i..j].$$

7.3 Functional Programs

In this section, we'll list the definitions of some standard higher-order combinators, and a couple of lemmas about them. We'll combine them to get a theorem that transforms an expression of the form $\mathit{map}\ f\ [i..j]$ to an expression that is iterative in style. The theorem will be applied to each of the three program derivations.

7.3.1 Some Higher-Order Combinators

Our aim is to transform programs of the shape $\mathit{map}\ f\ [i..k]$ into iterative programs, where calculation of $f(j+1)$ can re-use some of the work that went into calculating $f\ j$, for integers j such that $i \leq j < k$. Say calculating $f(j+1)$ from $f\ j$ is performed by applying a (simple) function called *next* to $f\ j$, that is formally,

$$\forall j.(i \leq j < k) \Rightarrow (f(j+1) = \mathit{next}(f\ j)).$$

Then we would only have to apply f once, namely to i, the first integer in the range $[i..k]$. After that, we could simply keep applying *next*. We stop if we have a result for each integer in the range $[i..k]$. We express this formally using the functions *iterate* and *take*, which are found in the standard Haskell prelude [HPW92], and can be defined in any lazy functional language. They are defined as:

$$\begin{array}{ll}
\mathit{take} & : \mathbb{N} \to [a] \to [a] \\
\mathit{take}\ 0\ \mathit{as} & \hat{=} [\,] \\
\mathit{take}\ n\ [\,] & \hat{=} [\,] \\
\mathit{take}\ (n+1)\ (a : \mathit{as}) & \hat{=} a : \mathit{take}\ n\ \mathit{as} \\
\mathit{iterate} & : (a \to a) \to a \to [a] \\
\mathit{iterate}\ f\ a & \hat{=} a : \mathit{iterate}\ f\ (f\ a)
\end{array}$$

The function *take* receives a natural number n, say, and a list, and delivers a list of the first n elements of the list. If the list has fewer than n elements, the whole list is returned. The function *iterate* receives a function f with equal domain and range type, and a value a of that type. It delivers the infinite list $[a, fa, f(fa), f(f(fa)), ...]$. Typically one applies *take* n to the infinite result of *iterate* f a, and receives a finite result. In the expression *take* n (*iterate* f a) laziness allows us to separate the loop, represented by *iterate* f, from the halting condition of the loop, represented by *take* n.

One more bit of notation: $\#xs$ stands for the length of the list xs. For an integer range it is of course trivial to calculate its length: $\#[x..y] \equiv max(0, y - x + 1)$.

We are now ready to formalise the transformation described above:

Lemma 6 (Map to iterate)

$$\mathit{map}\ f\ [x..y] \equiv (\mathit{take}\ \#[x..y] \circ \mathit{iterate}\ \mathit{next})(f\ x)$$

if $x \le i < y$ *implies* $f(i+1) \equiv \mathit{next}(f\ i)$.

The proof is by induction on the length of the list.

To make this a little more general, we need another lemma:

Lemma 7 (Map and take commute) $\mathit{map}\ f \circ \mathit{take}\ n \equiv \mathit{take}\ n \circ \mathit{map}\ f$

These two functions on lists are the same. The proof is by induction on their argument list.

We also know that mapping a composition is the same as composing maps:

Lemma 8 (Composing map) $\mathit{map}(f \circ g) \equiv \mathit{map}\ f \circ \mathit{map}\ g$

Now we generalise lemma 'map to iterate' by precomposing both sides with *map use*, and using the two previous lemmas:

Theorem 9 (Map to iterate)

$$\mathit{map}(\mathit{use} \circ \mathit{make})[x..y] \equiv (\mathit{take}\ \#[x..y] \circ \mathit{map}\ \mathit{use} \circ \mathit{iterate}\ \mathit{next})(\mathit{make}\ x)$$

if $x \le i < y$ *implies* $\mathit{make}(i+1) \equiv \mathit{next}(\mathit{make}\ i)$.

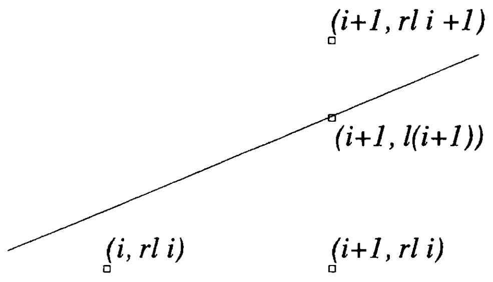

Figure 7.3: Choosing the next pixel to represent the line.

This theorem says that to map a function f, say, over an integer range, all we have to do is find three functions, here called *make*, *use*, and *next*, such that *use* composed with *make* is our original function f, and *next* captures a recurrence relation on *make*. Typically *make* will do what f does, and some extra work, whereas *use* is typically a simple function like a projection from a tuple. Naturally this theorem only reduces work if the function *next* is simpler than f.

7.3.2 Lines

The last formulation was

$$L \equiv map\ (\lambda\, i.(i, rl\ i))\ [x_1..x_2].$$

To apply theorem 9 we have to find a recurrence relation for rl.

For integer i such that $x_1 \le i < x_2$, can we calculate $rl(i+1)$ from $rl\ i$? Since the gradient $\frac{dy}{dx}$ is between 0 and 1 inclusively, $rl(i+1)$ is either one more, or the same as $rl\ i$. See figure 7.3.

We compare distances to choose the next pixel:

$rl(i+1)$

$\equiv$

if $|rl\ i+1-l(i+1)| < |l(i+1)-rl\ i|$ **then** $rl\ i+1$ **else** $rl\ i$

$\equiv$ maths; def. *err* below

if $err\ i < 0$ **then** $rl\ i+1$ **else** $rl\ i$,

where *err* is defined:

$err\ : \mathbb{Z} \times \mathbb{Z}$

$err\ i \hat{=} 2dx(rl\ i - y_1) - 2dy(i+1-x_1) + dx.$

We have a recurrence relation for rl, but it involves another function, namely *err*. Fortunately *err* also satisfies a recurrence relation:

$err(i+1)$

$\equiv$ maths

$err\ i + 2dx(rl(i+1) - rl\ i) - 2dy$

$\equiv$ introduce **if** ; recurrence rl

if $err\ i < 0$ **then** $err\ i + 2(dx - dy)$ **else** $err\ i - 2dy$,

for any integer i.

By defining

$$\begin{array}{ll}
\textit{make} & : \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \\
\textit{make } i & \hat{=} (i, \textit{rl } i, \textit{err } i) \\
\textit{next} & : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \\
\textit{next } (i, r, e) & \hat{=} \textbf{if } e < 0 \textbf{ then } (i+1, r+1, e+2(dx-dy)) \textbf{ else } (i+1, r, e-2dy) \\
\textit{use} & : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \\
\textit{use } (i, j, \_) & \hat{=} (i, j)
\end{array}$$

we satisfy the condition of theorem 9 and conclude

$$L \equiv (\textit{take } \#[x_1..y_1] \circ \textit{map use} \circ \textit{iterate next})\ (\textit{make } x_1).$$

The initial values *make* x_1 are easily calculated:

$$\begin{array}{l}
\quad \textit{make } x_1 \\
\equiv \\
\quad (x_1, \textit{rl } x_1, \textit{err } x_1) \\
\equiv \\
\quad (x_1, y_1, -2dy + dx).
\end{array}$$

7.3.3 Circles

The last formulation was

$$C \equiv \textit{map } (\lambda\, i.(i, \textit{rc } i))\ [0..\textit{round}\frac{r}{\sqrt{2}}].$$

We would like to use theorem 9, so we aim for a recurrence relation on function *rc*. We know that the gradient on the arc in the second octant is between -1 and 0 inclusive, so *rc* $(i+1)$ is either *rc* i, or one less.

But which of the pixels $(i+1, \textit{rc } i - 1)$ and $(i+1, \textit{rc } i)$ is closer to the circle point $(i+1, c(i+1))$? We could simply compare the distances:

$$\begin{array}{l}
\quad \textit{rc}(i+1) \\
\equiv \\
\quad \textbf{if } |\textit{rc } i - f(i+1)| < |f(i+1) - (\textit{rc } i - 1)| \textbf{ then } \textit{rc } i \textbf{ else } \textit{rc } i - 1.
\end{array}$$

There is, however, an equivalent way of deciding: Consider the midpoint $(i+1, \textit{rc } i - \frac{1}{2})$ of the two candidate pixels. If it lies inside the circle, then we choose the upper candidate. See figure 7.4.

Whether a point lies inside, on, or outside the circle can be determined from this function *in*:

$$\begin{array}{ll}
\textit{in} & : \textit{Point} \to \mathbb{R} \\
\textit{in}(x, y) & \hat{=} 4(x^2 + y^2 - r^2)
\end{array}$$

We know

$$\begin{array}{rcl}
\textit{in}(x, y) < 0 & \equiv & (x, y) \text{ inside circle}, \\
\textit{in}(x, y) = 0 & \equiv & (x, y) \text{ on circle},
\end{array}$$

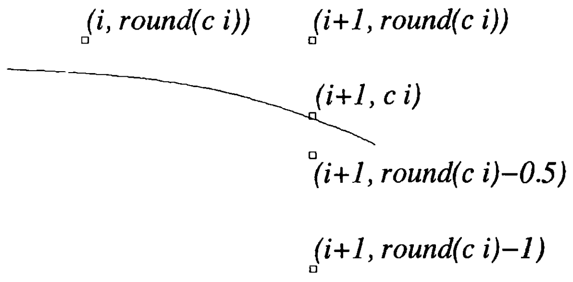

Figure 7.4: Choosing the next pixel to represent the circle.

$$0 < in(x, y) \equiv (x, y) \text{ outside circle.}$$

The factor 4 is not critical at this stage. It doesn't affect the comparisons to 0. Later the 4 will cancel a $\frac{1}{4}$ and thereby ensure the final program uses only integers.

So we have:

$rc(i + 1)$

$\equiv$

if $in(i + 1, rc\ i - \frac{1}{2}) < 0$ **then** $rc\ i$ **else** $rc\ i - 1$

$\equiv$ def. q below

if $q\ i < 0$ **then** $rc\ i$ **else** $rc\ i - 1$.

The function q is defined

$q\ : \mathbb{Z} \to \mathbb{Z}$
$q\ i \hat{=} in(i + 1, rc\ i - \frac{1}{2})$.

The two ways of deciding between the candidate pixels, they are comparing distances and using the midpoint, are in fact equivalent. The proof is school algebra.

We now have a recurrence relation for the function rc. But it uses another function q. Therefore, to apply theorem 9, we must also find a recurrence relation for q. Can we express $q(i+1)$ in terms of $q\ i$ (and $rc\ i$) for integer i such that $0 \leq i < round\frac{r}{\sqrt{2}}$? The answer is yes, by simple mathematical calculations:

$q(i + 1)$

$\equiv$ def. q

$in(i + 2, rc\ (i + 1) - \frac{1}{2})$

$\equiv$ recurrence for rc; function into **if**

if $q\ i < 0$ **then** $in(i + 2, rc\ i - \frac{1}{2})$ **else** $in(i + 2, rc\ i - 1\frac{1}{2})$

$\equiv$ def. in, maths

if $q\ i < 0$ **then** $in(i + 1, rc\ i - \frac{1}{2}) + 4i + 6$
else $in(i + 1, rc\ i - \frac{1}{2}) + 4(i - rc\ i) + 10$

$\equiv$ def. q

if $q\ i < 0$ **then** $q\ i + 4i + 6$ **else** $q\ i + 4(i - rc\ i) + 10$.

By defining

$make \quad : \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
$make\ i \quad \hat{=} (i, rc\ i, q\ i)$
$next \quad : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
$next(i, rci, qi) \hat{=} \textbf{if } qi < 0 \textbf{ then } (i+1, rci, qi + 4i + 6) \textbf{ else } (i+1, rci - 1, qi + 4(i - rci) + 10)$
$use \quad : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to Pixel2$
$use(i, rci, \_) \quad \hat{=} (i, rci)$

we satisfy the condition of theorem 9 and conclude

$$C \equiv (take\ \#[0..round\frac{r}{\sqrt{2}}] \circ map\ use \circ iterate\ next)\ (make\ 0).$$

The initial values *make* 0 are as follows.

We have $rc\ 0 \equiv round(c\ 0) \equiv r$. The first midpoint to be tested is $(1, r - \frac{1}{2})$, and we get

$q\ 0$
$\equiv$ def. q
$in(1, r - \frac{1}{2})$
$\equiv$ def. in
$4(1^2 + (r - \frac{1}{2})^2 - r^2)$
$\equiv$ maths
$4(1\frac{1}{4} - r)$
$\equiv$ The 4 fulfils its purpose.
$5 - 4r$.

Therefore $make\ 0 \equiv (0, rc\ 0, q\ 0) \equiv (0, r, 5 - 4r)$.

7.3.4 Spheres

The last formulation of $slice(i, j)$ was

$$map\ (\lambda k.(i, k, rs(i, k)))\ [i..j],$$

where $rs \hat{=} round \circ s$. To use theorem 9 we search for a recurrence relation for rs. Let's try for a recurrence for s.

$s(x, y + 1)$
$\equiv$
$s(x, y) - s(x, y) + s(x, y + 1)$
$\equiv$ def. s
$s(x, y) - \sqrt{r^2 - x^2 - y^2} + \sqrt{r^2 - x^2 - (y+1)^2}$

Unfortunately there is no easy way to proceed from these square roots! We will borrow an idea from [FGS+85] and use a parabola instead of a sphere. Instead of s, we will use function p given by

$p \quad : Point2 \to \mathbb{R}$
$p(x, y) \hat{=} \frac{r^2 - x^2 - y^2}{r}.$

The function p receives (x,y) and delivers z, such that (x,y,z) is a point on the three-dimensional parabola given by $x^2+y^2+zr \equiv r^2$. For (x,y) within the circle of radius r around $(0,0)$, $p(x,y)$ is close to $s(x,y)$, and it allows us to find a recurrence.

How wrong is this approximation ? Within the circle, we always have $p(x,y) \leq s(x,y)$. The error is minimal at the origin and at the circumference, where it is 0. The error is maximal at $\frac{\sqrt{3}}{2}r \approx 0.87r$ distance from the origin, where it is $\frac{1}{4}r$.

We were stuck looking for a recurrence for s, but for p it is easy:

$$p(x,y+1) \equiv p(x,y) + f\ y$$

where auxiliary function f is defined:

$f\ \ : \mathbb{R} \rightarrow \mathbb{R}$
$f\ y \mathrel{\hat{=}} -\frac{2y+1}{r}$

It also allows a recurrence:

$$f(y+1) \equiv f\ y - 2/r$$

We redefine *slice* to use p in place of s.

$\mathit{slice}\ \ : \mathit{Pixel2} \rightarrow [\mathit{Pixel3}]$
$\mathit{slice}(i,j) \mathrel{\hat{=}} \mathit{map}\ (\lambda\, k.(i,k,\mathit{round}(p(i,k))))\ [i..j]$

We define

$\mathit{make}\ \ : \mathbb{Z} \rightarrow \mathbb{Z} \times \mathbb{R} \times \mathbb{R}$
$\mathit{make}\ k \mathrel{\hat{=}} (k, p(i,k), f\ k)$
$\mathit{next}\ \ : \mathbb{Z} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{Z} \times \mathbb{R} \times \mathbb{R}$
$\mathit{next}(k,\mathit{pik},\mathit{fk}) \mathrel{\hat{=}} (k+1, \mathit{pik}+\mathit{fk}, \mathit{fk} - \frac{2}{r})$
$\mathit{use}\ \ : \mathbb{Z} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathit{Pixel3}$
$\mathit{use}(k,\mathit{pik},\_) \mathrel{\hat{=}} (i,k,\mathit{round}\ \mathit{pik})$.

The definition of *make*, *next*, *use* are local to the body of *slice*. The arguments i and j of *slice* appear free in them.

The condition of 9 is satisfied, and we conclude

$$\mathit{slice}(i,j) \equiv (\mathit{take}\ \#[i..j] \circ \mathit{map}\ \mathit{use} \circ \mathit{iterate}\ \mathit{next})\ (\mathit{make}\ i).$$

But where do the initial values

$$\mathit{make}\ i \equiv (i, p(i,i), f\ i) \equiv (i, \frac{r^2-i^2-i^2}{r}, \frac{2i+r}{r})$$

come from ? We will extend the definition of *slice*, and make it receive the second and third elements of above triple as extra arguments:

$\mathit{slice2}\ \ : \mathit{Pixel} \times \mathbb{R} \times \mathbb{R} \rightarrow [\mathit{Pixel3}]$
$\mathit{slice2}((i,j),\mathit{pii},\mathit{fi}) \mathrel{\hat{=}} (\mathit{pii} = p(i,i) \wedge \mathit{fi} = f\ i) \succ\!\!- \mathit{slice}(i,j)$.

We conclude

$$slice2((i,j),pii,fi) \equiv (take\ \#[i..j] \circ map\ use \circ iterate\ next)\ (i,pii,fi).$$

The whole program used to be

$$S \equiv (\cup/ \circ slice^*)\ C$$

but now *slice*2 requires a larger argument, so we get

$$S \equiv (\cup/ \circ slice2^*)\ D,$$

where

$D : [Pixel2 \times \mathbb{R} \times \mathbb{R}]$
$D \mathrel{\hat{=}} map\ (\lambda(i,j).((i,j),p(i,i),f\ i))\ C$

We calculate:

D

$\equiv$ def. D and previous development of C

$map\ (\lambda(i,j).((i,j),p(i,i),f\ i))\ (map\ (\lambda\, i.(i,rc\ i))\ [0..round\frac{0}{\sqrt{2}}])$

$\equiv$

$map\ (\lambda\, i.((i,rc\ i),p(i,i),f\ i))\ [0..round\frac{r}{\sqrt{2}}]$

We re-use the recurrence for rc.

A recurrence relation for p is also easy to find:

$$p(x+1,y+1) \equiv p(x,y) + g(x,y),$$

where the auxiliary function g defined by:

$g \quad : Point2 \to \mathbb{R}$
$g(x,y) \mathrel{\hat{=}} -(2x+2y+2)/r.$

Again, we must also find a recurrence relation for it:

$$g(x+1,y+1) \equiv g(x,y) - 4/r.$$

And finally we re-use the recurrence relation for f that we already used in deriving the body of slice:

$$f(i+1) \equiv f\ i - 2/r$$

Putting the recurrences for the functions rc, q, p, g, and f together, we define

$make \quad : \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
$make\ i \quad \hat{=}(i, rc\ i, q\ i, p(i,i), g(i,i), f\ i)$
$next \quad : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
$next(i,rci,qi,pii,gii,fi) \mathrel{\hat{=}} \mathbf{if}\ qi < 0\ \mathbf{then}(i+1, rci, qi+4i+6, pii+gii, gii-4/r, fi-2/r)$
$\qquad \mathbf{else}(i+1, c-1, q+4(i-c)+10, p+g, f-2/r)$

```
use                      : $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow Pixel2 \times \mathbb{R} \times \mathbb{R}$
use(i, rci, _, pii, _, fi)   $\hat{=}((i, rci), pii, fi)$
```

and as required we find $use \circ make \equiv \lambda i.((i, rc\ i), p(i, i), f\ i)$ and $make(i + 1) \equiv next(make\ i)$ for i such that $0 \leq i < round(r/\sqrt{2})$. The initial application of *make* to the first integer in the list $[0..round(r/\sqrt{2})]$ is straightforward:

```
   make 0
≡        def. make
   (0, rc 0, q 0, p(0, 0), g(0, 0), f 0)
≡        definitions
   (0, r, 5 − 4r, r, −2/r, −1/r)
```

The whole program put together is:

```
let
   slice : Pixel2 × ℝ × ℝ → [Pixel3]
   slice((x, y), z, k) ≙ (take n ∘ map use ∘ iterate next) init
   where
         n ≙ #[x..y]
         init ≙ (x, z, k)
         use : ℤ × ℝ × ℝ → Pixel3
         use(j, p, _) ≙ (x, j, p)
         next : ℤ × ℝ × ℝ → ℤ × ℝ × ℝ
         next(j, p, f) ≙ (j + 1, p + f, f − 2/r)
   d : [Pixel2 × ℝ × ℝ]
   d ≙ (take n ∘ map use ∘ iterate next) init
   where
         n ≙ #[0..round(r/√2)]
         init ≙ (0, r, 5 − 4r, r, −2/r, −1/r)
         next                       : ℤ × ℤ × ℤ × ℝ × ℝ × ℝ → ℤ × ℤ × ℤ × ℝ × ℝ × ℝ
         next(i, rci, qi, pii, gii, fi) ≙ if qi < 0
                                      then(i + 1, rci, qi + 4i + 6, pii + gii, gii − 4/r, fi − 2/r)
                                      else (i + 1, c − 1, q + 4(i − c) + 10, p + g, f − 2/r)
         use                        : ℤ × ℤ × ℤ × ℝ × ℝ × ℝ → Pixel3 × ℝ × ℝ
         use(i, rci, _, pii, _, fi)    ≙((i, rci), pii, fi)
in
   (flatten ∘ map slice) d
```

The function *flatten* takes a list of lists, and concatenates the lists.

7.4 Imperative Programs

In this section we'll give an imperative implementation for expressions of the shape $(take\ n \circ map\ use \circ iterate\ next)\ init$. We then use that implementation to derive imperative implementations of the three graphics algorithms.

7.4.1 Some Lemmas about the Imperative Combinator *for*

The flexibility of imperative expressions in a rich expression language allows us to define many kinds of imperative control structures within the language as higher-order functions. We use this freedom to define *for* loops:

```
for           : $\mathbb{N} \to ST\ s\ a \to ST\ s\ [a]$
for 0 k       $\hat{=}$return [ ]
for (n + 1) k $\hat{=}$k; $\lambda\, a.$
                 for n k; $\lambda\, as.$
                 return (a : as)
```

The state transformer *for n k* is the state transformer that sequentially composes n copies of k, and returns the list of their results. We can express *for* in terms of *seq*.

Lemma 10 (for/seq) $\mathit{for}\ n\ k \equiv \mathit{seq}[k \mid i \leftarrow [1..n]]$

A function mapped over the result of a *for* loop can be moved into the loop body.

Lemma 11 (function into *for*) *For proper* $n : \mathbb{N}$ *we have*

$$\mathit{for}\ n\ k;\ \lambda\, as.\mathit{return}\ (\mathit{map}\ f\ as) \equiv \mathit{for}\ n\ (k;\ \lambda\, a.\mathit{return}\ (f\ a))$$

The proof uses the merging *seq* lemma 4. It appears in section 7.5.

In a similar lemma the results of a *for* loop are fed to a state transformer valued function in order.

Lemma 12

$$\mathit{for}\ n\ k;\ \lambda\, as.\mathit{seq}(\mathit{map}\ l\ as) \equiv \mathit{for}\ n\ (k;\ l)$$

if $k \mid l\ a$.

The lemma is a consequence of merging *seq*, lemma 4 in chapter 3. Its proof is in section 7.5.

We use *for* to give an imperative implementation of a finite iteration.

Lemma 13 (Imperative *iterate*)

```
  take n (iterate next init)
$\equiv$
  run(new init; $\lambda\, v.$
      for n (get v; $\lambda\, x.$
             put v (next x); $\lambda\, \_.$
             return x))
```

The proof is by induction on proper $n : \mathbb{N}$. Both the base case and the step case appeal to lots of state transformers laws. A detailed proof is given in section 7.5.

Theorem 14 (take,map,iterate)

```
  (take n ∘ map use ∘ iterate f) init
≡
  run(new init; λv.
        for n (get v; λx.
              put v (f x); λ_.
              return (use x)))
```

The theorem is a combination of the previous lemmas. Its proof is given in section 7.5.

7.4.2 Lines

The last formulation was: $L \equiv$ (*take n* $\circ$ *map use* $\circ$ *iterate next*) *init*, where n, *init*, *next*, *use* are defined as previously. We may apply the theorem 14 to derive the imperative program

run(*new init*; λ *v.for n* (*get v*; λ *x.put v* (*next x*); λ_.*return* (*use x*))).

The program evaluates to the list of pixels approximating the line. The natural use for this list is to display the pixels on a screen. Let's assume the IO transformer *out*2 : *Pixel*2 $\rightarrow$ *IO* () receives a pixel and displays it. We make the reasonable assumption that displaying a pixel commutes with reference accesses: *out*2 *p* | *get v* and *out*2 *p* | *put v x*.

Our aim is to move *out*2 forward in the program so that each pixel is displayed as soon as it is calculated.

```
  seq(map out2 L)
≡               left return, law 1
  return L; λps.
  seq(map out2 ps)
≡               theorem 14
  return (run(new init; λv.for n (...))); λps.
  seq(map out2 ps)
≡               law 4, return/run
  (new init; λv.for n (...)); λps.
  seq(map out2 ps)
≡               ; assoc., law 1
  new init; λv.
  for n (get v; λx.put v (next x); λ_.return (use x))); λps.
  seq(map out2 ps)
≡               function into for, lemma 12, out2 commutes
  new init; λv.
  for n ((get v; λx.put v (next x); λ_.return (use x)); λp.
        out2 p)
≡               ; assoc., law 1, and left return, law 1
  new init; λv.
  for n (get v; λx.
        put v (next x); λ_.
        out2(use x))
```

We have derived an imperative program for Bresenham's line drawing algorithm. Each pixel is displayed on the screen as soon as it is calculated.

7.4.3 Circles

The last formulation was: $C \equiv (\mathit{take}\ n \circ \mathit{map}\ \mathit{use} \circ \mathit{iterate}\ \mathit{next})\ \mathit{init}$, where n, *init*, *next*, *use* are defined as previously. We use the theorem 14 to derive an imperative program that displays the pixels on the screen, just as for the line drawing program.

```
  seq(map out2 C)
≡               theorem 14, etc.
  return (run(new init; λv.for n (...))); λps.
  seq(map out2 ps)
≡               return/run, law 4, etc.
  new init; λv.
  for n (get v; λx.
        put v (next x); λ_.
        out2(use x))
```

We have derived an imperative program for Bresenham's circle drawing algorithm. Each pixel is displayed on the screen as soon as it is calculated.

7.4.4 Spheres

In the last formulation of the sphere drawing program, the body of function *slice* and *d* both can be made into imperative programs by theorem 14. With *n*, *init*, *make*, *use*, *next* as defined before, we have:

```
  flatten(map slice d)
≡                 theorem 14 for d
  flatten(map slice
  (run(new init; λv.for n (get v; λx.put v (next x); λ_.return (use x)))))
≡                 twice function into run, law 5
  run(((new init; λv.
      for n (get v; λx.
            put v (next x); λ_.
            return (use x))); λts.
      return (map slice ts)); λcs.
      return (flatten cs))
≡                 ; assoc., 1
  run((new init; λv.
      for n (get v; λx.
            put v (next x); λ_.
```

```
              return (use x)); λ ts.
         return (map slice ts)); λ cs.
         return (flatten cs))
≡                       for/seq/map, 12
   run((new init; λ v.
         for n (get v; λ x.
              put v (next x); λ_.
              return (use x); λ t.
              return (slice t))); λ cs.
         return (flatten cs))
≡                       theorem 14 for slice, return/run, law 4
   run((new init; λ v.
         for n (get v; λ x.
              put v (next x); λ_.
              return (use x); λ((x, y), z, k).
              (let (n, init, next, use) = (...as given in development of slice...) in
              new init; λ w.
              for n (get w; λ t.
                   put w (next w); λ_.
                   return (use t)))); λ cs.
         return (flatten cs))
```

The points calculated by this program are to be displayed on a screen. Let's assume there is a IO transformer valued function *out*3 : *Pixel*3 → *IO* () that takes a three dimensional pixel and displays it on the screen. We append displaying the pixels to the above program, and then move displaying forward in the program so that each pixel is displayed as soon as it is calculated. It is reasonable to assume that displaying a pixel commutes with reference accesses: *out*3 *p* | *get v* and *out*3 *p* | *put v x*.

```
   new init; λ v.
   for n (get v; λ x.
        put v (next x); λ_.
        return (use x); λ((x, y), z, k).
        (let (n, init, next, use) = (...as given in development of slice...) in
        new init; λ w.
        for n (get w; λ t.
             put w (next w); λ_.
             return (use t))); λ cs.
   return (flatten cs); λ ps.
   seq(map out3 ps)
≡
   new init; λ v.
   for n (get v; λ x.
        put v (next x); λ_.
```

```
          return (use x); λ((x, y), z, k).
          (let (n, init, next, use) = (...as given in development of slice...) in
          new init; λ w.
          for n (get w; λ t.
                put w (next w); λ_.
                return (use t))); λ css.
      seq(map (seq ∘ (map out3)) ps); λ rss.
      return (flatten rss)
≡               for/seq, lemma 12
      new init; λ v.
      for n (get v; λ x.
          put v (next x); λ_.
          return (use x); λ((x, y), z, k).
          (let (n, init, next, use) = (...as given in development of slice...) in
          new init; λ w.
          for n (get w; λ t.
                put w (next w); λ_.
                return (use t)); λ as.
          seq(map out3 as)); λ rss.
      return (flatten rss)
≡               for/seq, lemma 12
      new init; λ v.
      for n (get v; λ x.
          put v (next x); λ_.
          return (use x); λ((x, y), z, k).
          (let (n, init, next, use) = (...as given in development of slice...) in
          new init; λ w.
          for n (get w; λ t.
                put w (next w); λ_.
                out3(use t))); λ rss.
      return (flatten rss)
```

Each pixel is drawn as soon as it is calculated. The last line of the program collects the results of applying *out*3 and returns them in a list. Since they are only empty tuples, we may want to discard them.

Postscript: Real Numbers

We intended to rid the program of real number arithmetic, but there are quite a few real number operations left in the auxiliary functions of the above program. Fortunately all real number expressions are of the form E/r, where E is an integer expression, and r is the radius of the sphere.

So we will apply an old trick: instead of dividing by r, we'll multiply by an integer that represents the first few significant digits of $1/r$. Let us say, for example, that $r \equiv 17$, and therefore $1/r \approx 0.05882352941$. We are satisfied with a precision of five digits, so we define the integer $\rho \hat{=} 58823$. We do the calculations, multiplying by ρ instead of dividing by r. In the end we simply divide by 10^6 in this example. That of course corresponds to the easy operation of shifting the decimal point. In the program, we will use the same trick, with base 2 instead of 10 of course.

The real division operations become integer multiplications, followed by a shift in the inner *use*. We just give a specification of *man*(tissa) and *exp*(ponent), since how to calculate them is a separate problem. The one remaining real number calculation of $round(r/\sqrt{2})$ is not a problem, since it occurs only once. The radius r is the argument of the final program. The accuracy is here set to 8, but any suitable natural would do.

```
$\lambda r : \mathbb{N}.$
**let**
    $accuracy \hat{=} 8$
    $(man, exp) \hat{=} \sqcap man : \mathbb{R}.\sqcap exp : \mathbb{Z}.(1/r = man * 2^{exp} \wedge 0.5 \leq |man| \leq 1) \rightarrow (man, exp)$
    $\rho \hat{=} round(man * 2^{accuracy})$
    $n \hat{=} \#[0..round(r/\sqrt{2})]$
    $init \hat{=} (0, r, 5 - 4 * r, r, -2 * \rho, -\rho)$
    $next\ (i, c, q, p, g, f) \hat{=}$ **if** $q < 0$ **then** $(i + 1, c, q + 4 * i + 6, p + g, g - 4 * \rho, f - 2 * \rho)$
                                   **else**$(i + 1, c - 1, q + 4 * (i - c) + 10, p + g, f - 2 * \rho)$
    $use\ (i, c, \_, p, \_, f) \hat{=} ((i, c), p, f)$
**in**
$new\ init;\ \lambda v.$
$for\ n\ (get\ v;\ \lambda x.$
    $put\ v\ (next\ x);\ \lambda\_.$
    $return\ (use\ x);\ \lambda((x, y), z, k).$
    (**let**
        $n \hat{=} \#[x..y]$
        $init \hat{=} (x, z, k)$
        $next\ (j, p, f) \hat{=} (j + 1, p + f, f - 2 * \rho)$
        $use\ (j, p, \_) \hat{=} (x, j, p * 2^{exp-accuracy})$
    **in**
    $new\ init;\ \lambda w.$
    $for\ n\ (get\ w;\ \lambda t.$
        $put\ w\ (next\ w);\ \lambda\_.$
        $out3(use\ t)))$
```

7.5 Proofs

This section gives detailed proofs of some lemmas that were used in the three derivations, but are of general use. The lemmas are fairly believable. We still give the proofs in such detail as a test of mechanical state transformer calculations. The most frequently used laws are the monad laws, in particular the left return law and 'associativity' of semicolon.

Lemma 15 (function into *for*) *For proper $n : \mathbb{N}$ we have*

$$for\ n\ k;\ \lambda\, as.return\ (map\ f\ as) \equiv for\ n\ (k;\ \lambda\, a.return\ (f\ a))$$

The proof uses the merging *seq* lemma 4.

$for\ n\ k;\ \lambda\, as.return\ (map\ f\ as)$

$\equiv$

$seq[k \mid i \leftarrow [1..n]];\ \lambda\, as.seq[return\ (f\ a) \mid a \leftarrow as]$

$\equiv$ merging seq, lemma 4, return commutes, law 13

$seq[k;\ \lambda\, a.return\ (f\ a) \mid i \leftarrow [1..n]]$

$\equiv$

$for\ n\ (k;\ \lambda\, a.return\ (f\ a)).$

Lemma proven.

Lemma 16

$$for\ n\ k;\ \lambda\, as.seq(map\ l\ as) \equiv for\ n\ (k;\ l)$$

if $k \mid l\ a.$

Proof. A consequence of merging *seq*, lemma 4 in chapter 3.

$for\ n\ k;\ \lambda\, as.seq(map\ l\ as)$

$\equiv$

$seq[k \mid i \leftarrow [1..n]];\ \lambda\, as.seq(map\ l\ as)$

$\equiv$ merging seq, lemma 4, assumption

$seq[k;\ l \mid i \leftarrow [1..n]]$

$\equiv$

$for\ n(k;\ l).$

Lemma proven.

Lemma 17 (Imperative *iterate*)

$take\ n\ (iterate\ next\ init)$

$\equiv$

```
run(new init; λv.
    for n (get v; λx.
            put v (next x); λ_.
            return x))
```

Proof. For $n \equiv \bot$ the lemma holds trivially. For proper $n : \mathbb{N}$ we use induction. The base case $n \equiv 0$ is proven:

$\textbf{run}(\mathit{new}\ \mathit{init};\ \lambda\, v.\mathit{for}\ 0\ (\ldots))$
$\equiv$ $\quad\mathit{for}$
$\textbf{run}(\mathit{new}\ \mathit{init};\ \lambda\, v.\mathit{return}\ [\,])$
$\equiv$ $\quad\mathit{new}$ intro, law 7
$\textbf{run}(\mathit{return}\ [\,])$
$\equiv$ $\quad\textbf{run}$ intro, law 3
$[\,]$
$\equiv$ $\quad\mathit{take}$
$\mathit{take}\ 0\ (\mathit{iterate}\ \mathit{next}\ \mathit{init}).$

The induction hypothesis is: for arbitrary I and fixed n we have $\mathit{take}\ n\ (\mathit{iterate}\ \mathit{next}\ I) \equiv \textbf{run}(\mathit{new}\ I;\ \lambda\, v.\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x))$. The inductive case is proven:

$\textbf{run}(\mathit{new}\ \mathit{init};\ \lambda\, v.$
$\quad\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x))$
$\equiv$ $\quad\mathit{for}$
$\textbf{run}(\mathit{new}\ \mathit{init};\ \lambda\, v.$
$\quad(\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x);\ \lambda\, a.$
$\quad\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x);\ \lambda\, \mathit{as}.$
$\quad\mathit{return}\ (a : \mathit{as}))$
$\equiv$ $\quad$; assoc. twice, law 1
$\textbf{run}(\mathit{new}\ \mathit{init};\ \lambda\, v.$
$\quad\mathit{get}\ v;\ \lambda\, x.$
$\quad\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.$
$\quad\mathit{return}\ x;\ \lambda\, a.$
$\quad\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x);\ \lambda\, \mathit{as}.$
$\quad\mathit{return}\ (a : \mathit{as}))$
$\equiv$ $\quad$left return , law 1
$\textbf{run}(\mathit{new}\ \mathit{init};\ \lambda\, v.$
$\quad\mathit{get}\ v;\ \lambda\, x.$
$\quad\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.$
$\quad\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x);\ \lambda\, \mathit{as}.$
$\quad\mathit{return}\ (x : \mathit{as}))$
$\equiv$ $\quad\mathit{new}$ generates get , law 9
$\textbf{run}(\mathit{new}\ \mathit{init};\ \lambda\, v.$
$\quad\mathit{put}\ v\ (\mathit{next}\ \mathit{init});\ \lambda\_.$
$\quad\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x);\ \lambda\, \mathit{as}.$
$\quad\mathit{return}\ (\mathit{init} : \mathit{as}))$
$\equiv$ $\quad\mathit{new}$ generates put , law 8
$\textbf{run}(\mathit{new}\ (\mathit{next}\ \mathit{init});\ \lambda\, v.$
$\quad\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x);\ \lambda\, \mathit{as}.$
$\quad\mathit{return}\ (\mathit{init} : \mathit{as}))$
$\equiv$ $\quad$; assoc. , law 1
$\textbf{run}((\mathit{new}\ (\mathit{next}\ \mathit{init});\ \lambda\, v.\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x));\ \lambda\, \mathit{as}.$

$\quad\quad \mathit{return}\ (\mathit{init} : \mathit{as}))$

$\equiv \qquad$ function into **run** , law 5

$\quad \mathit{init} : \mathbf{run}(\mathit{new}\ (\mathit{next}\ \mathit{init});\ \lambda\, v.\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ x);\ \lambda\_.\mathit{return}\ x))$

$\equiv \qquad$ IH with *next init* for I

$\quad \mathit{init} : \mathit{take}\ n\ (\mathit{iterate}\ \mathit{next}\ (\mathit{next}\ \mathit{init}))$

$\equiv \qquad$ *take*, *iterate*

$\quad \mathit{take}\ (n+1)\ (\mathit{iterate}\ \mathit{next}\ \mathit{init}).$

Lemma proven by induction.

Theorem 18 (take,map,iterate)
$(\mathit{take}\ n\ \circ \mathit{map}\ \mathit{use} \circ \mathit{iterate}\ f)\ \mathit{init}$

$\equiv$

```
run(new init; λ v.
    for n (get v; λ x.
        put v (f x); λ_.
        return (use x)))
```

The theorem is a combination of the previous lemmas.

$\quad (\mathit{take}\ n \circ\ \mathit{map}\ \mathit{use}\ \mathit{iterate}\ \mathit{next})\ \mathit{init}$

$\equiv \qquad$ take/map, lemma 7

$\quad \mathit{map}\ \mathit{use}(\mathit{take}\ n\ (\mathit{iterate}\ \mathit{next}\ \mathit{init}))$

$\equiv \qquad$ imperative iterate, theorem 13

$\quad \mathit{map}\ \mathit{use}$
$\quad (\mathbf{run}(\mathit{new}\ \mathit{init};\ \lambda\, v.\mathit{for}\ n\ (\mathit{get}\ v;\ \lambda\, x.\mathit{put}\ v\ (\mathit{next}\ \mathit{init});\ \lambda\_.\mathit{return}\ x)))$

$\equiv \qquad$ function into run, law 5

```
run((new init; λ v.
    for n (get v; λ x.put v (next x); λ_.return x)); λ cs.
    return (map use cs))
```

$\equiv \qquad$; assoc., law 1

```
run(new init; λ v.
    for n (get v; λ x.put v (next x); λ_.return x); λ cs.
    return (map use cs))
```

$\equiv \qquad$ function into *for*, lemma 11

```
run(new init; λ v.
    for n ((get v; λ x.put v (next x); λ_.return x); λ a.
        return (use a))
```

$\equiv \qquad$ twice ; assoc., law 1

```
run(new init; λ v.
    for n (get v; λ x.
        put v (next x); λ_.
        return x; λ a.
        return (use a))
```

$\equiv \qquad$ left return, law 1

```
run(new init; λ v.
for n (get v; λx.
    put v (next x); λ_.
    return (use x)).
```

Theorem proven.

Chapter 8

A Simple Combinator Graph Reducer

In this chapter, we'll specify and derive a program for an imperative algorithm (one that necessarily requires state).

It implements graph reduction for a simple language with the combinators S, K, I, Y, integers, and binary integer operations. The algorithm requires state because it repeatedly modifies a graph. Each change in one part of the graph is potentially visible from any other part of the graph.

The program could be written in a functional language without state, simply by simulating the state by a functional value that is passed around. However, without a guarantee that this simulated state is passed around single-threadedly, it could not be changed in-place, but would have to be copied. Although the program would be correct, it would not have the desired complexity. Instead, by using state in the state monad we have that guarantee.

This derivation is the most involved of this thesis. It illustrates the use of many of the language features (generalized choice, case-expressions, recursion, assertions, etc) but most importantly it is an example of a derivation in which state is manipulated directly rather than only through the state monad primitives. In the derivation, variables (usually σ) are used to stand for individual states. There is nothing wrong with that (apart from the technical point of the state's type), as long as no state appears in the final program, which must be constructed in terms of the state monad primitives only. The program comes from [KL93], where it demonstrates the expressiveness gained by adding state to Haskell.

The first section describes the SKIY language, and gives the specification of the program. The second section gives the derivation of the graph reducing program in detail, preceded by a road map in its first subsection. The third section discusses the derivation. The fourth section contains the details of some proofs that were skipped in section 2.

8.1 Specification

We describe a simple combinator language. Then we formulate the specification of the program. Before the derivation, we examine some properties of the specification.

8.1.1 The SKIY Language

A term of the SKIY language we will use is either a number, or an operator, or one of the combinators S, K, I, and Y, or an application of one SKIY term to another.

$$\begin{array}{lcl} \textbf{type}\ \mathit{SKIY} & \hat{=} & \mathit{Num}\ \mathbb{Z} \\ & \mid & \mathit{Opr}\ (\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}) \\ & \mid & S \mid K \mid I \mid Y \\ & \mid & \mathit{App}\ \mathit{SKIY}\ \mathit{SKIY} \end{array}$$

Definition 8 (*rewrite*) *We define $\overset{*}{\to}$ as the smallest binary relation satisfying these eight axioms:*

1. $x \overset{*}{\to} x$
2. $x \overset{*}{\to} y \wedge y \overset{*}{\to} z \Rightarrow x \overset{*}{\to} z$
3. $\mathit{App}\ I\ x \overset{*}{\to} x$
4. $\mathit{App}\ (\mathit{App}\ K\ x)\ y \overset{*}{\to} x$
5. $\mathit{App}\ (\mathit{App}\ (\mathit{App}\ S\ f)\ g)\ x \overset{*}{\to} \mathit{App}\ (\mathit{App}\ f\ x)\ (\mathit{App}\ g\ x)$
6. $\mathit{App}\ Y\ f \overset{*}{\to} \mathit{App}\ f\ (\mathit{App}\ Y\ f)$
7. $\mathit{App}\ (\mathit{App}\ (\mathit{Opr}\ \oplus)\ (\mathit{Num}\ m))\ (\mathit{Num}\ n) \overset{*}{\to} \mathit{Num}\ (m \oplus n)$
8. $f \overset{*}{\to} f' \wedge x \overset{*}{\to} x' \Rightarrow \mathit{App}\ f\ x \overset{*}{\to} \mathit{App}\ f'\ x'$

Axioms 1 and 2 say $\overset{*}{\to}$ is reflexive and transitive, axioms 3-6 capture combinator reduction, axiom 7 captures operator reduction, and axiom 8 says $\overset{*}{\to}$ distributes through the term structure. If $x \overset{*}{\to} y$, we say 'x can be rewritten to y'.

Strictly speaking, in the SKIY terms $\mathit{Num}\ z$ and $\mathit{Opr}\ \oplus$, z is only some representation of an integer, and $\oplus$ is only the representation of some function. On the right hand side of axiom 7 there should be some interpretation map applied to n, m, and to $\oplus$. But we'll ignore that to aid clarity.

Clearly we can keep applying the Y axiom forever:

$$\mathit{App}\ Y\ f \overset{*}{\to} \mathit{App}\ f\ (\mathit{App}\ Y\ f) \overset{*}{\to} \mathit{App}\ f\ (\mathit{App}\ f\ (\mathit{App}\ Y\ f)) \overset{*}{\to} \ldots$$

The limit of this sequence is the recursively defined infinite expression $\mu\, x.\mathit{App}\ f\ x$ of type *SKIY*. By appealing to the axioms 6 and 8 'infinitely many times', we deduce

$$\mathit{App}\ Y\ f \quad \overset{*}{\to} \quad \mu\, x.\mathit{App}\ f\ x.$$

8.1.2 Specification of the Program

We are seeking a program, that given a (finite) SKIY term x, delivers an integer z such that $x \overset{*}{\to} \mathit{Num}\ z$, if such an integer exists. Formally that is

Specification 1 (eval)

$$\begin{aligned} \mathit{eval} &: \mathit{SKIY} \to \mathbb{Z} \\ \mathit{eval}\ x &\mathrel{\hat{=}} \textbf{if} \sqcap z : \mathbb{Z}.(x \overset{*}{\to} \mathit{Num}\ z) \to z\ \textbf{fi}. \end{aligned}$$

8.1.3 Properties of $\overset{*}{\to}$ and consequently of *eval*

Here are some fairly obvious properties of $\overset{*}{\to}$. A number cannot be rewritten any further, that is $\mathit{Num}\ z \overset{*}{\to} x$ implies $\mathit{Num}\ z = x$. We'll call a term s ***hopeful*** if $\exists\, z : \mathbb{Z}.s \overset{*}{\to} \mathit{Num}\ z$. Otherwise it is ***stuck***. A number applied to an argument is stuck, that is the term $\mathit{App}\ (\mathit{Num}\ z)\ x$ is stuck. The combinator S needs at least three arguments. The terms S, $\mathit{App}\ S\ f$, and $\mathit{App}\ (\mathit{App}\ S\ f)\ g$ are stuck. The combinator K needs at least two arguments. The terms K and $\mathit{App}\ K\ x$ are stuck. The combinators I and Y need at least one argument, so the term I and Y are stuck.

An operator needs exactly two arguments, which must themselves not be stuck. The terms $\mathit{Opr}\ \oplus$ and $\mathit{App}\ (\mathit{Opr}\ \oplus)\ x$ are stuck. If an operator has two arguments that are not stuck, then the arguments can be rewritten to numbers (axiom 8), and then the operator can be reduced (axiom 7). With the result being a number, if there were any further arguments, the term would be stuck.

It can be shown that, given a hopeful SKIY term, there is an algorithm for finding the number it can be rewritten to. Our program will be based on this algorithm. The algorithm is called *normal order reduction* and is captured in this subrelation of $\overset{*}{\to}$:

Definition 9 (Normal order reduction) *Let $\overset{n}{\to}$ be the weakest transitive relation satisfying these axioms:*

1. $\mathit{App}\ I\ x \overset{n}{\to} x$
2. $\mathit{App}\ (\mathit{App}\ K\ x)\ y \overset{n}{\to} x$
3. $\mathit{App}\ (\mathit{App}\ (\mathit{App}\ S\ f)\ g)\ x \overset{n}{\to} \mathit{App}(\mathit{App}\ f\ x)(\mathit{App}\ g\ x)$
4. $\mathit{App}\ Y\ f \overset{n}{\to} \mathit{App}\ f(\mathit{App}\ Y\ f)$
5. $f \overset{n}{\to} f'$ *implies* $\mathit{App}\ f\ x \overset{n}{\to} \mathit{App}\ f'\ x$
6. $m \overset{n}{\to} m'$ *implies* $\mathit{App}(\mathit{App}(\mathit{Opr}\ \oplus)\ m)\ n \overset{n}{\to} \mathit{App}(\mathit{App}(\mathit{Opr}\ \oplus)\ m')\ n$
7. $n \overset{n}{\to} n'$ *implies* $\mathit{App}(\mathit{App}(\mathit{Opr}\ \oplus)\ m)\ n \overset{n}{\to} \mathit{App}(\mathit{App}(\mathit{Opr}\ \oplus)\ m)\ n'$
8. $\mathit{App}(\mathit{App}(\mathit{Opr}\ \oplus)\ (\mathit{Num}\ m))(\mathit{Num}\ n) \overset{n}{\to} \mathit{Num}(m \oplus n)$

It can be shown that $\overset{*}{\to}$ has the 'diamond' property, that is

$$n \overset{*}{\to} w \wedge n \overset{*}{\to} e \Rightarrow \exists\, s.\ w \overset{*}{\to} s \wedge e \overset{*}{\to} s.$$

From it we can conclude that for any x, there is at most one z fulfilling the specification. Therefore applications of *eval* are determined. We can also conclude that the normal-order sequence of reduction steps may safely be interrupted by (finitely many) reduction steps that are not in the normal order without affecting termination or the result. In other words backtracking is never necessary. A useful special case of the diamond property is $x \overset{*}{\dashrightarrow} \mathit{Num}\ z \wedge x \overset{*}{\dashrightarrow} y \Rightarrow y \overset{*}{\dashrightarrow} \mathit{Num}\ z$.

8.2 Implementation

This section gives a detailed derivation of the graph reducer. The first subsection gives a road map of the derivation. The leisurely reader may skip this subsection, and the hurried reader may read it and skip the rest of this section.

The following subsections show how to represent terms as graphs, and how to model term rewriting by graph reduction. The last two subsections discuss termination of the program (which we have failed to prove) and list the final program.

8.2.1 Road Map

The function *eval* will be implemented by graph reduction, which is essentially an imperative program. The graph will be represented as a function from vertex references to vertex contents, so it's a kind of adjacency list. The state is just this function.

First, we'll give some types appropriate for storing a SKIY expression as a graph in the state. Then we'll define a function that takes a SKIY expression and stores it, returning a reference to the root of the graph. From it, a function can be derived that, given this reference, extracts the SKIY expression from the state.

These two functions will be used to refine *eval* into a program that has state, but doesn't yet use it for graph reduction:

$$\begin{array}{ll} & \mathit{eval}\ x \\ \sqsubseteq & \quad \text{defs., run intro} \\ & \mathbf{run}(\mathit{store}\ x;\ \mathit{extract};\ \lambda x.\mathit{break}(\mathit{eval}\ x)). \end{array}$$

The graph reducing state transformer we are looking for does work equivalent to the last two state transformers in the program above, but the work of *eval* is pushed into the extracting. We specify

$$\begin{array}{rcl} \mathit{red} & : & \mathit{Ref}\ s\ (\mathit{Vert}\ s) \to \mathit{ST}\ s\ \mathbb{Z} \\ \mathit{red}\ v & \hat{=} & \mathit{extract}\ v;\ \lambda x.\mathit{break}(\mathit{eval}\ x), \end{array}$$

and the program becomes

$$\begin{array}{ll} \equiv & \quad \text{spec. } \mathit{red} \\ & \mathbf{run}(\mathit{store}\ x;\ \mathit{red}). \end{array}$$

In trying to implement *red* we find we have to keep track of the *spine* of the SKIY expression, that is, a stack of the application vertices from the leftmost vertex to the root of the graph. We specify

Specification 2 (red)

$$\begin{aligned} \mathit{redS} \quad &: \quad \mathit{Ref}\ s\ (\mathit{Vert}\ s) \to [\mathit{Ref}\ s\ (\mathit{Vert}\ s)] \to \mathit{ST}\ s\ \mathbb{Z} \\ \mathit{redS}\ v\ xs \quad &\hat{=} \quad \mathit{spine}\ v\ xs \hat{\succ} \mathit{red}(\mathit{last}(v : xs)), \end{aligned}$$

introducing auxiliaries *spine* and *extS*. The program becomes

$\sqsubseteq$ spec. *redS*
run(*store x*; $\lambda v.\mathit{redS}\ v\ [\,]$).

In implementing *redS* the need for indirections becomes obvious. Since we are heading for a recursive implementation, we notice that the specification of *redS* (and *red* on which it is based) is too weak. The specifications say what result these state transformers should return, but leave open what final state they produce. Formally they produce the worst final state $\bot_{State}$. We are forced the 'strengthen the induction hypothesis' by requiring *red* to change the state in a restricted way: we add an unspecified binary relation ϕ between initial and final state to the specification of *red*. We continue refining *redS*, collecting requirements about ϕ. Gathered together these requirements give a definition of ϕ. On the way, we also collect lemmas required of the auxiliaries *ext*, *spine*, *extS* that were introduced. These lemmas are proved separately.

Finally we take the recursive refinement of $\mathit{redS} \sqsubseteq F[\mathit{redS}]$ and conclude that *redS* is refined by the least fixpoint, $\mathit{redS} \sqsubseteq \mu F$. In a total correctness calculus like ours, this conclusion is of course only valid if a termination argument were provided. We have no termination argument, but discuss the search for one in the second last subsection.

8.2.2 Representing SKIY Terms as Graphs

This is a straightforward implementation of the SKIY datatype using references to the vertices of the graph. The type of the vertices is *Vert s*, and therefore the type of the references is *Ref s* (*Vert s*). In these types, the type variable *s* indexes the state thread a reference was created in. Since the sumtype *Vert s* includes references, it also must be parametrised over *s*.

The only unexpected thing in the definition of the type *Vert s* below are the indirections *Ind* (*Ref s* (*Vert s*)). Their use will become apparent later.

$$\begin{aligned} \textbf{type}\ \mathit{Vert}\ s \ \hat{=}\ & \mathit{Num}\ \mathbb{Z} \\ |\ & \mathit{Opr}\ (\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}) \\ |\ & S \mid K \mid I \mid Y \\ |\ & \mathit{App}\ (\mathit{Ref}\ s\ (\mathit{Vert}\ s))\ (\mathit{Ref}\ s\ (\mathit{Vert}\ s)) \\ |\ & \mathit{Ind}\ (\mathit{Ref}\ s\ (\mathit{Vert}\ s)) \end{aligned}$$

The program will be imperative. During the derivation, the state will be simulated by a function from references to elements of *Vert s*.

$$\textbf{type}\ \mathit{State}\ s \ \hat{=}\ \mathit{Ref}\ s\ (\mathit{Vert}\ s) \to \mathit{Vert}\ s$$

However, in the final program it will only be manipulated by the state transformer primitives, and therefore the real state can be used.

The state transformer *store* puts a SKIY term into the state.

Definition 10 (store)

$$\begin{array}{ll}
\multicolumn{2}{l}{store : SKIY \to ST\ s\ (Ref\ s\ (Vert\ s))} \\
store\ (Num\ z) & \hat{=}\ new\ (Num\ z) \\
store\ (Opr\ \oplus) & \hat{=}\ new\ (Opr\ \oplus) \\
store\ S & \hat{=}\ new\ S \\
store\ K & \hat{=}\ new\ K \\
store\ I & \hat{=}\ new\ I \\
store\ Y & \hat{=}\ new\ Y \\
store\ (App\ f\ x) & \hat{=}\ store\ f;\ \lambda\, l. \\
 & \quad store\ x;\ \lambda\, r. \\
 & \quad new\ (App\ l\ r)
\end{array}$$

A SKIY term is represented by a reference $v : Ref\ s\ (Vert\ s)$ and a state $\sigma : State$ such that $extract\ v\ \sigma = x$, where *extract* recursively replaces the references by the things they point at in the state. We'll derive the definition of *extract* from the specification

Specification 3 (extract)

$$\begin{array}{rcl}
extract & : & Ref\ s\ (Vert\ s) \to ST\ s\ SKIY \\
store\ s;\ extract & \equiv & return\ s.
\end{array}$$

Since *extract* takes a reference as argument, it is reasonable to expect that it will look up the reference, and do a case distinction on the result. The derivation of *extract* will be done by case distinction on $s : SKIY$. The cases $S, K, I, Y, Num\ z, Opr\ \oplus$ are very similar. Here's the case for S.

$$\begin{array}{ll}
 & store\ S;\ extract \\
\equiv & \qquad \text{def. } store \\
 & new\ S;\ extract \\
\equiv & \qquad \text{shape of } ext \\
 & new\ S;\ get;\ \lambda\, e.\mathbf{case}\ e\ \mathbf{of} \ldots \sqcap S \to X \sqcap \ldots \\
\equiv & \qquad \text{law get/new} \\
 & return\ S;\ \lambda\, e.\mathbf{case}\ e\ \mathbf{of} \ldots \sqcap S \to X \sqcap \ldots \\
\equiv & \qquad \text{law return(1)},\ \beta,\ \mathbf{case} \\
 & X \\
\equiv & \qquad \text{defining } X \\
 & return\ S
\end{array}$$

For the recursive case, namely $App\ f\ x$, we get:

$$
\begin{array}{ll}
 & store(App\ f\ x);\ extract \\
\equiv & \quad \text{def. } store \\
 & store\ f;\ \lambda\, u.store\ x;\ \lambda\, v.new\ (App\ u\ v);\ extract \\
\equiv & \quad \text{shape of } extract \\
 & store\ f;\ \lambda\, u.store\ x;\ \lambda\, v.new\ (App\ u\ v);\ get\ ;\ \lambda\, e.\mathbf{case}\ e\ \mathbf{of} \ldots \sqcap\ App\ u\ v \rightarrow X \sqcap \ldots \\
\equiv & \quad \text{new/get} \\
 & store\ f;\ \lambda\, u.store\ x;\ \lambda\, v.return\ (App\ u\ v);\ \lambda\, e.\mathbf{case}\ e\ \mathbf{of} \ldots \sqcap\ App\ u\ v \rightarrow X \sqcap \ldots \\
\equiv & \quad \text{return, } \beta \text{, case} \\
 & store\ f;\ \lambda\, u.store\ x;\ \lambda\, v.X \\
\equiv & \quad \text{defining } X \\
 & store\ f;\ \lambda\, u.store\ x;\ \lambda\, v.extract\ v;\ \lambda\, x'.extract\ u;\ \lambda f'.return\ (App\ f'\ x') \\
\equiv & \quad \text{bind, } \eta \\
 & store\ f;\ \lambda\, u.(store\ x;\ extract);\ \lambda\, x'.extract\ u;\ \lambda f'.return\ (App\ f'\ x') \\
\equiv & \quad \text{IH} \\
 & store\ f;\ \lambda\, u.return\ x;\ \lambda\, x'.extract\ u;\ \lambda f'.return\ (App\ f'\ x') \\
\equiv & \quad \text{return(1), } \beta \text{, bind, } \eta \\
 & (store\ f;\ ext);\ \lambda f'.return\ (App\ f'\ x) \\
\equiv & \quad \text{IH} \\
 & return\ f;\ \lambda f'.return\ (App\ f'\ x) \\
\equiv & \quad \text{return(1), } \beta \\
 & return\ (App\ f\ x)
\end{array}
$$

We collect the definition of *extract*. In it, the *Ind* case can be defined in any way. One could choose $\bot$ and then later on refine.

Definition 11 (extract)
$extract : Ref\ s\ (Vert\ s) \rightarrow ST\ s\ SKIY$
$extract\ v \hat{=}\ get\ v.\, \lambda\, e.\mathbf{case}\ e\ \mathbf{of}$

$$
\begin{array}{ll}
Num\ z & \rightarrow return\ (Num\ z) \\
\sqcap\ Opr\ \oplus & \rightarrow return\ (Opr\ \oplus) \\
\sqcap\ S & \rightarrow return\ S \\
\sqcap\ K & \rightarrow return\ K \\
\sqcap\ I & \rightarrow return\ I \\
\sqcap\ Y & \rightarrow return\ Y \\
\sqcap\ App\ f\ x & \rightarrow extract\ x;\ \lambda\, u.extract\ f;\ \lambda\, v.return\ (App\ u\ v) \\
\sqcap\ Ind\ u & \rightarrow extract\ u
\end{array}
$$

For convenience, we also define a state reader *ext v* that does the same thing as the state transformer *extract v*. The specification is simply

Specification 4 (ext)

$$
\begin{array}{rcl}
ext & : & Ref\ s\ (Vert\ s) \rightarrow SR\ s\ SKIY \\
ro(ext) & \equiv & extract,
\end{array}
$$

where $ro : SR\ s\ a \rightarrow ST\ s\ a$ makes a state reader into a state transformer that delivers the same result and leaves state unchanged[Wad92a].

8.2.3 Introducing State into the Specification

We introduce state:

$$
\begin{array}{ll}
 & eval\ s \\
\equiv & \qquad \text{futile run} \\
 & eval(\mathbf{run}(return\ s)) \\
\equiv & \qquad \text{spec. } extract \\
 & eval(\mathbf{run}(store\ s;\ extract)) \\
\equiv & \qquad \text{function into run} \\
 & \mathbf{run}(store\ s;\ extract;\ \lambda s.break(eval\ s))
\end{array}
$$

We pick out the last two state transformers in the run to continue refining. For convenience, give them a name. Specify function *red* by

Specification 5 (red)

$$
\begin{array}{rcl}
red & : & Ref\ s\ (Vert\ s) \to ST\ s\ \mathbb{Z} \\
red\ v & \hat{=} & extract\ v;\ \lambda s.break(eval\ s)
\end{array}
$$

with the aim of implementing *red* by graph reduction. The program so far becomes

$$
\begin{array}{ll}
\equiv & \qquad \text{spec. } red \\
 & \mathbf{run}(store\ s;\ red).
\end{array}
$$

Now we try to derive an implementation for *red*. Its argument is a reference, so like for *extract*, it suggests itself that the reference is looked up and then a case analysis is made on what it stored. Thus, the shape of *red* will be: $red\ v \equiv get\ v;\ \lambda e.\mathbf{case}\ e\ \mathbf{of} \ldots \sqcap$ $App\ u\ v \to X \sqcap \ldots$. First, we move *eval* into *extract*:

$$
\begin{array}{ll}
 & red\ v \\
\equiv & \qquad \text{spec. } red \\
 & extract\ v;\ (break \circ eval) \\
\equiv & \qquad \text{def. extract} \\
 & (get\ v;\ \lambda e.\mathbf{case}\ e\ \mathbf{of}\ Num\ z \to return\ (Num\ z) \sqcap \ldots);\ (break \circ eval) \\
\equiv & \qquad \text{bind, function into case} \\
 & get\ v;\ \lambda e. \\
 & \mathbf{case}\ e\ \mathbf{of} \\
 & \quad \begin{array}{lll}
 & Num\ z \to & return\ (Num\ z);\ (break \circ eval) \\
\sqcap & Ind\ u \to & extract\ u;\ (break \circ eval) \\
\sqcap & Opr \oplus \to & return\ (Opr\oplus);\ (break \circ eval) \\
\sqcap & S, K, I \to & return\ (\ldots);\ (break \circ eval) \\
\sqcap & App\ u\ v \to & extract\ v;\ \lambda x.extract\ u;\ \lambda f.return\ (App\ f\ x);\ (break \circ eval)
\end{array}
\end{array}
$$

We'll continue refining each of the case branches. The first is trivial:

$$\begin{array}{ll} & \mathit{return}\ (\mathit{Num}\ z);\ \mathit{break} \circ \mathit{eval} \\ \equiv & \qquad \text{left unit} \\ & \mathit{break}(\mathit{eval}(\mathit{Num}\ z)) \\ \equiv & \qquad \text{property } \mathit{eval} \\ & \mathit{break}\ z. \end{array}$$

The indirection case is easy too:

$$\begin{array}{ll} & \mathit{extract}\ u;\ (\mathit{break} \circ \mathit{eval}) \\ \equiv & \qquad \text{spec. } \mathit{red} \\ & \mathit{red}\ u. \end{array}$$

For the cases *Opr* $\oplus, S, K, I$ there's no proper result because they need arguments. Example K:

$$\begin{array}{ll} & \mathit{return}\ K;\ (\mathit{break} \circ \mathit{eval}) \\ \equiv & \qquad \text{left unit} \\ & \mathit{break}(\mathit{eval}\ K) \\ \equiv & \qquad \text{property } \mathit{eval} \\ & \mathit{break}\ \bot. \end{array}$$

For applications, nothing much is gained:

$$\begin{array}{ll} & \mathit{extract}\ v;\ \lambda x.\mathit{extract}\ u;\ \lambda f.\mathit{return}\ (\mathit{App}\ f\ x);\ (\mathit{break} \circ \mathit{eval}) \\ \equiv & \qquad \text{left unit} \\ & \mathit{extract}\ v;\ \lambda x.\mathit{extract}\ u;\ \lambda f.\mathit{break}(\mathit{eval}(\mathit{App}\ f\ x)) \end{array}$$

Now if $f = I$, we could continue one step. Unless f is a function that needs exactly one argument, the next step is $\mathit{break}\bot$. Therefore the cases of f being *Opr* $\oplus, S, K$, *Num* z can be dealt with. But what if f is *App* $g\ y$? We would have to do some more case analysis on g! Clearly this path of the derivation has reached a dead end.

We can only reduce a vertex containing a combinator or an operator if we have collected enough arguments for the combinator or operator. To find the arguments, we need to descend application vertices on the left, keeping track of the application vertices we've seen in a stack. Call this stack the *spine*.

8.2.4 Simple Cases, with *spine*

Define

Definition 12 (spine)

$$\begin{array}{rcl} \mathit{spine} & : & \mathit{Ref}\ s\ (\mathit{Vert}\ s) \to [\mathit{Ref}\ s\ (\mathit{Vert}\ s)] \to \mathit{SR}\ s\ \mathbb{B} \\ \mathit{spine}\ v\ [\,]\ \sigma & \hat{=} & \mathit{True} \\ \mathit{spine}\ f\ (v : \mathit{vs})\ \sigma & \hat{=} & \exists x.\sigma\ v = \mathit{App}\ f\ x \wedge \mathit{spine}\ v\ \mathit{vs}\ \sigma. \end{array}$$

A reference f and a list of references vs satisfies the *spine* property in a given state σ if each reference v in the list points to an application vertex *App* u _, and u is directly before v in the list. The first reference in the list must point to a vertex $App\ f\ \_$. We write $spine\ f\ vs\ \sigma$.

Let's extend the definition of *spine*, and make it take three arguments rather than just two. In an alternative version with just two arguments the first argument would correspond to the first argument in the version above cons'd onto the second. We choose this formulation to emphasise that the first reference has a different role from the other references. One can read $spine\ f\ vs\ \sigma$ as 'in state σ the references vs point to a chain of application vertices ending in what f points to'. In the alternative representation $spine : [Ref\ s\ (Vert\ s)] \rightarrow SR\ s\ \mathbb{B}$, one could write $spine\ [\,]\ \sigma$, although such an 'empty' spine leading nowhere does not make practical sense.

However, we will find out later that the definition above is not enough: it doesn't account for indirections.

Given just the root reference r of a graph, we have an easy initial spine, since $spine\ r\ [\,]\ \sigma$ will hold.

Specify the function *redS* as a version of *red* using spines.

Specification 6 (redS)

$$\begin{aligned} redS &: Ref\ s\ (Vert\ s) \rightarrow [Ref\ s\ (Vert\ s)] \rightarrow ST\ s\ \mathbb{Z} \\ redS\ v\ xs &\mathrel{\hat{=}} spine\ v\ xs \mathrel{\hat{\succ\!\!-}} red(last(v : xs)). \end{aligned}$$

As for *extract*, since *redS* takes a reference, it is reasonable to look for an implementation that looks up the reference and then does a case analysis: $redS\ v\ xs \equiv get\ v;\ \lambda\, e.\mathbf{case}\ e\ \mathbf{of}\ \ldots \sqcap Num\ z \rightarrow redS\ v\ xs \sqcap \ldots$. We'll refine $redS\ v\ xs\ \sigma$ by case analysis of $\sigma\ v$. The case of an application vertex is easy. The fact that *redS* has encountered an *App* vertex is recorded in the spine, and *redS* descends to the left, that is to the function. Assume $\sigma\ v = App\ f\ x$. Then

$$\begin{aligned} & redS\ v\ xs\ \sigma \\ \equiv\ & \qquad \text{spec. } redS \\ & spine\ v\ xs\ \sigma \succ\!\!- red(last(v : xs))\ \sigma \\ \equiv\ & \qquad \text{def. } spine \text{ and assumption} \\ & spine\ f\ (v : xs)\ \sigma \succ\!\!- red(last(v : xs))\ \sigma \\ \equiv\ & \qquad \text{def. } last \\ & spine\ f\ (v : xs)\ \sigma \succ\!\!- red(last(f : v : xs)))\ \sigma \\ \equiv\ & \qquad \text{spec. } redS \\ & redS\ f\ (v : xs)\ \sigma. \end{aligned}$$

The case of a number vertex is easy too. The number is just retrieved from the state and returned. We assume the spine is empty since a number applied to an argument is a stuck term anyway. Assume $\sigma\ v = Num\ z$. Then

$$\begin{array}{ll}
 & spine\ v\ [\,]\ \sigma \succ\!\!- red(last[v])\ \sigma \\
\equiv & \qquad \text{def. } spine, last \\
 & True \succ\!\!- red\ v\ \sigma \\
\equiv & \qquad \text{previous refinement of } red \\
 & break\ z.
\end{array}$$

What should happen when *redS* finds an indirection vertex ? The function *ext* simply follows an indirection, so the function *redS* which is defined in terms of *ext* must do the same. Assume $spine\ v\ xs\ \sigma$ and $\sigma\ v = Ind\ u$. Then

$$\begin{array}{ll}
 & redS\ v\ xs\ \sigma \\
\equiv & \qquad \text{spec. } redS \\
 & spine\ v\ xs\ \sigma \succ\!\!- red\ (last(v : xs))\ \sigma \\
\equiv & \qquad \text{def. } last,\ red, \text{ and } ext \\
 & spine\ v\ xs\ \sigma \succ\!\!- red\ (last(u : xs))\ \sigma \\
\not\equiv & \qquad \text{a desired step contradicting the present definition of } spine \\
 & spine\ u\ xs\ \sigma \succ\!\!- red\ (last(u : xs))\ \sigma \\
\equiv & \qquad \text{spec. } redS \\
 & redS\ u\ xs\ \sigma
\end{array}$$

The desired step above is not true since our present formulation of *spine* does not allow spines with some indirections in them. We widen the definition of *spine* to allow indirections. With the new definition the above $\not\equiv$ becomes $\equiv$.

Definition 13 (spine, updated) *Let spine be the weakest predicate satisfying the these three axioms.*

1. *Empty spine.* $spine\ v\ [\,]\ \sigma$
2. *Application.* $spine\ f\ (v : vs)\ \sigma \Leftarrow \exists x.\sigma\ v = App\ f\ x \wedge spine\ v\ vs\ \sigma$
3. *Indirections.* $spine\ u\ vs\ \sigma \Leftarrow \sigma\ v = Ind\ u \wedge spine\ v\ vs\ \sigma$

For developments of the other branches of *redS*, namely the operator and combinator branches, we have to unfold the definitions right down to the rewriting relation $\xrightarrow{*}$. We'll unfold $redS\ v\ xs\ \sigma$ under the assumption that indeed $spine\ v\ xs\ \sigma$ is true:

$$\begin{array}{ll}
 & redS\ v\ xs\ \sigma \\
\equiv & \qquad \text{spec. } redS \\
 & spine\ v\ xs\ \sigma \succ\!\!- red(last(v : xs))\ \sigma \\
\equiv & \qquad \text{true assertion} \\
 & red(last(v : xs))\ \sigma \\
\equiv & \qquad \text{def. } red \\
 & (extract(last(v : xs));\ \lambda\, s.break(eval\ s))\ \sigma \\
\equiv & \qquad \text{defs.} \\
 & \mathbf{if} \sqcap (z, \tau).(ext\ (last(v : xs))\ \sigma \xrightarrow{*} Num\ z) \rightarrow (z, \tau)\ \mathbf{fi} \\
\equiv & \qquad \text{new auxiliary } extS \\
 & \mathbf{if} \sqcap (z, \tau).(extS\ v\ xs\ \sigma \xrightarrow{*} Num\ z) \rightarrow (z, \tau)\ \mathbf{fi}.
\end{array}$$

The new auxiliary *extS* extracts a SKIY term from a state, given the spine of the term. Its definition is

Definition 14 (extS)

$$\begin{array}{rcl} \mathit{extS} & : & \mathit{Ref}\ s\ (\mathit{Vert}\ s) \to [\mathit{Ref}\ s\ (\mathit{Vert}\ s)] \to \mathit{SR}\ s\ \mathit{SKIY} \\ \mathit{extS}\ v\ xs & \hat{=} & \mathit{ext}\ (\mathit{last}(v : xs)). \end{array}$$

The purpose of this auxiliary is just to break up big proofs into smaller proofs.

8.2.5 Meeting an Operator

Let's try the case of an operator. An operator $\mathit{Opr}\ \oplus$ needs exactly two arguments, otherwise the term is stuck. Let us therefore assume that there are two references to application vertices on the spine. To give some names, say $\sigma\ v = \mathit{Opr}\ \oplus$ and $\mathit{spine}\ v\ [b, c]\ \sigma$. For this whole subsection let x and y be such that $\sigma\ b = \mathit{App}\ \_\ x$ and $\sigma\ c = \mathit{App}\ \_\ y$. We are guaranteed existence of these values by the existential quantifier in the definition of *spine*. Then

$\mathit{redS}\ v\ [b, c]\ \sigma$

$\equiv$ spec. *redS*, ass., spec. *red*

if $\sqcap(z, \tau).(\mathit{ext}\ c\ \sigma \overset{*}{\to} \mathit{Num}\ z) \to (z, \mathit{tau})$ **fi**

$\equiv$ ass.

if $\sqcap(z, \tau).(\mathit{App}(\mathit{App}(\mathit{Opr}\ \oplus)(\mathit{eval}(\mathit{ext}\ x\ \sigma)))(\mathit{eval}(\mathit{ext}\ y\ \sigma)) \overset{*}{\to} \mathit{Num}\ z) \to (z, \tau)$ **fi**

$\equiv$ def. $\overset{*}{\to}$, successful rewriting is determined

if $\sqcap(z, \tau).(\exists! m, n.\mathit{ext}\ x\ \sigma \overset{*}{\to} \mathit{Num}\ m \wedge \mathit{ext}\ x\ \sigma \overset{*}{\to} \mathit{Num}\ n \wedge z = m \oplus n) \to (z, \tau)$ **fi**

$\sqsubseteq$ Heading for two recursive calls. law: $\exists!$ out of $\sqcap$

if $\sqcap m.\mathit{ext}\ x\ \sigma \overset{*}{\to} \mathit{Num}\ n \to$

$\sqcap n.\mathit{ext}\ y\ \sigma \overset{*}{\to} \mathit{Num}\ m \to$

$\sqcap(z, \tau).(z = m \oplus n) \to (z, \tau)$ **fi**

$\sqsubseteq$ last generalised choice is trivial

Have to make the preceding $\sqcap$ into state transformers, so add state.

if $\sqcap(m, \sigma_1).\mathit{ext}\ x\ \sigma \overset{*}{\to} \mathit{Num}\ m \to$

$\sqcap(n, \sigma_2).\mathit{ext}\ y\ \sigma \overset{*}{\to} \mathit{Num}\ n \to$

$\mathit{break}(m \oplus n)\bot_{\mathit{State}}$ **fi**

$\sqsubseteq$ trying to sequentialise. For *break* any state will do, so σ_2

if $\sqcap(m, \sigma_1).\mathit{ext}\ x\ \sigma \overset{*}{\to} \mathit{Num}\ m \to$

$\sqcap(n, \sigma_2).\mathit{ext}\ y\ \sigma \overset{*}{\to} \mathit{Num}\ n \to$

$\mathit{break}(m \oplus n)\sigma_2$ **fi**

$\sqsubseteq$ spec. *red*

let $(m, \sigma_1) \hat{=} \mathit{red}\ x\ \sigma$ **in**

let $(n, \sigma_2) \hat{=} \mathit{red}\ y\ \sigma$ **in**

$\mathit{break}(m \oplus n)\sigma_2$

The two **let** use recursive calls of *red*, but both on the same input state σ. We have to sequentialise them, that is, make the second recursive call use as input state the output state σ_1 of the first recursive call. The specification of *red* leaves open what happens to the state, so we are free to add a binary relation on states that describes what should happen to the state. Let's call the relation ϕ. What are the requirements on ϕ ?

1. Total. $\forall \sigma. \exists \tau. \sigma\phi\tau$ This requirement is necessary for the step 'refining binding assertions' above, to keep feasibility of the prescriptions.

2. Reflexive. $\forall \sigma. \sigma\phi\sigma$. This requirement includes totality. It stems from our previous refinements of *red* v σ for the case of $\sigma\ v = Num\ z$. We implemented it by ***break*** z, which is trivially refined by ***return*** z. The state is not changed at all. If this is to refine the new version of *red* with the relation ϕ, then ϕ must allow that, that is, it must be reflexive.

3. Transitive. There are two consecutive recursive calls, so if together they are to satisfy ϕ by each satisfying it, ϕ must be transitive.

4. Rewriting. If $\sigma\phi\sigma_1$, then $ext\ y\ \sigma \xrightarrow{*} Num\ n$ implies $ext\ y\ \sigma_1 \xrightarrow{*} Num\ n$. This is to ensure that the second recursive call can safely use σ_1 instead of σ without affecting the result. The formal variable y ranges over all accessible references.

In short, we are looking for a preorder on states that rewrites (or leaves unchanged) the extractions of every accessible vertex. The simplest ϕ would be $=_{State}$, but that would rule out any change to the state. For now, let ϕ be any relation that satisfies these requirements.

Update the specification of *red* and *redS* to:

Specification 7 (*red*, *redS*, **second try**)

$$\begin{aligned} red\ v\ \sigma &\mathrel{\hat{=}} \textbf{if} \textstyle\bigsqcap(z,\tau).(ext\ v\ \sigma \xrightarrow{*} Num\ z \wedge \sigma\phi\tau) \rightarrow (z,\tau)\ \textbf{fi} \\ redS\ v\ xs\ \sigma &\mathrel{\hat{=}} spine\ v\ xs\ \sigma \succ\!\!- \textbf{if} \textstyle\bigsqcap(z,\tau).(extS\ v\ xs \xrightarrow{*} Num\ z \wedge \sigma\phi\tau) \rightarrow (z,\tau)\ \textbf{fi}. \end{aligned}$$

The above derivation goes through with the new versions of *red* and *redS*, and ***break*** refined to ***return*** , and is continued by:

$$\begin{aligned} \sqsubseteq &\quad \text{sequentialise} \\ &(red\ x;\ \lambda m.red\ y;\ \lambda n.\mathbf{return}\ (m \oplus n))\ \sigma \\ \equiv &\quad \text{spec. } redS \\ &(redS\ x\ [\,];\ \lambda m.redS\ y\ [\,];\ \lambda n.\mathbf{return}\ (m \oplus n))\ \sigma. \end{aligned}$$

The previous developments of *redS* for the case *Num* z and *App* u v still go through with the new versions of *red* and *redS*.

8.2.6 The case of I

Now the cases of three combinators. We'll start with the simplest: the I combinator. The I combinator is stuck unless it has at least one argument. So we'll assume *spine* v $(b : xs)$ σ and $\sigma\ v = I$. Throughout this subsection let x be such that $\sigma\ b = App\ \_\ x$. Then

$$\begin{array}{ll} & redS\ v\ (b : xs)\ \sigma \\ \equiv & \qquad \text{spec. } redS \\ & \textbf{if} \sqcap(z,\tau).(extS\ v\ (b : xs)\ \sigma \overset{*}{\to} Num\ z \wedge \sigma\phi\tau) \to (z,\tau)\ \textbf{fi} \end{array}$$

At this point we would like to make use of the local knowledge of the graph to apply the I transformation. First we isolate a lemma.

Lemma 19 (extS/I) .

$$\begin{array}{ll} & I = \sigma\ v \wedge spine\ v\ (b : xs)\ \sigma \\ \Rightarrow & \\ & extS\ v\ (b : xs)\ \sigma \overset{*}{\to} extS\ x\ xs\ \sigma[b \mapsto Ind\ x] \end{array}$$

It is easy to see that lemma *extS/I* follows from this lemma:

Lemma 20 (ext/I) .

$$\begin{array}{ll} & I = \sigma\ v \wedge spine\ v\ (b : xs)\ \sigma \\ \Rightarrow & \\ & ext\ u\ \sigma \overset{*}{\to} ext\ u\ \sigma[b \mapsto Ind\ x] \end{array}$$

Here, we formulate both lemmas assuming $spine\ v\ (b : xs)\ \sigma$ rather than just assuming $\sigma\ b = App\ v\ x$ in order to account for indirections. Furthermore, we formulate the lemma for arbitrary reference u rather than only b because the extra generality is not much more work to prove and useful later on.

Proof. The proof is not as straightforward as it may at first appear since we can make no assumptions about absence of cycles in the graph. In particular the node b may be accessible from node x. In that case, extracting from b would yield an infinite SKIY expression containing infinitely many occurrences of I, all of which are removed in one graph transformation. We first prove the special case $u = b$, and then show it for any u. We use the recursion law based on operational semantics.

We generate the expression with a syntactic hole $X[\,]$ by extracting from x as far as possible, but stop as soon as we reach b.

$$ext\ x\ \sigma \quad \longrightarrow \quad X[ext\ b\ \sigma]$$

Doing this is just a series of μ unfoldings, **let** reductions, and β reductions. If b is not reachable from x, then there is no occurrence of the hole in X. From

$$\begin{array}{ll} & ext\ b\ \sigma \\ \longrightarrow & \qquad \text{unfold } ext \\ & App\ I\ (ext\ x\ \sigma) \\ \longrightarrow & \qquad \text{constr of } X \\ & App\ I\ (X[ext\ b\ \sigma]) \end{array}$$

we conclude

$$ext\ b\ \sigma \quad \equiv \quad \mu\, t.App\ I\ (X[t]).$$

From

$$\begin{array}{ll} & \mathit{ext}\ b\ \sigma[b \mapsto \mathit{Ind}\ x] \\ \longrightarrow & \qquad \text{unfold } \mathit{ext} \\ & \mathit{ext}\ x\ \sigma[b \mapsto \mathit{Ind}\ x] \\ \longrightarrow & \qquad \text{construction of } X \\ & X[\mathit{ext}\ b\ \sigma[b \mapsto \mathit{Ind}\ x]] \end{array}$$

we conclude

$$\mathit{ext}\ b\ \sigma[b \mapsto \mathit{Ind}\ x] \ \equiv\ \mu\, t.X[t].$$

Therefore

$$\begin{array}{ll} & \mathit{ext}\ b\ \sigma \\ \equiv & \\ & \mu\, t.\mathit{App}\ I(X[t]) \\ \overset{*}{\rightarrow} & \qquad \text{rewriting} \\ & \mu\, t.X[t] \\ \equiv & \\ & \mathit{ext}\ b\ \sigma[b \mapsto \mathit{Ind}\ x]. \end{array}$$

The case for $u = b$ is proven. Further, for arbitrary u, let $U[\ ]$ be the expression with a syntactic hole generated by extracting from u until b is reached. Then

$$\begin{array}{ll} & \mathit{ext}\ u\ \sigma \\ \equiv & \qquad \text{construction of } U \\ & U[\mathit{ext}\ b\ \sigma] \\ \overset{*}{\rightarrow} & \qquad \text{just proved; compositional } \overset{*}{\rightarrow} \\ & U[\mathit{ext}\ b\ \sigma[b \mapsto \mathit{Ind}\ x]] \\ \equiv & \qquad \text{construction of } U \\ & \mathit{ext}\ u\ \sigma[b \mapsto \mathit{Ind}\ x], \end{array}$$

which we set out to show.

We continue the implementation. The assumptions were *spine* $v\ (b : xs)\ \sigma$ and $\sigma\ v = I$. Then

$\mathit{redS}\ v\ (b : \mathit{xs})\ \sigma$

$\equiv$ spec. *redS*

$\mathbf{if} \bigsqcap(z,\tau).(\mathit{extS}\ v\ (b : \mathit{xs})\ \sigma \overset{*}{\rightarrow} \mathit{Num}\ z \wedge \sigma\phi\tau) \rightarrow (z,\tau)\ \mathbf{fi}$

$\sqsubseteq$ lemma *extS/I*, diamond property of $\overset{*}{\rightarrow}$

$\mathbf{if} \bigsqcap(z,\tau).(\mathit{extS}\ x\ \mathit{xs}\ \sigma[b \mapsto \mathit{Ind}\ x]\ \sigma \overset{*}{\rightarrow} \mathit{Num}\ z \wedge \sigma\phi\tau) \rightarrow (z,\tau)\ \mathbf{fi}$

$\sqsubseteq$ ϕ trans., collect req.: $\sigma\phi\sigma[b \mapsto \mathit{Ind}\ x]$

$\mathbf{if} \bigsqcap(z,\tau).(\mathit{extS}\ x\ \mathit{xs}\ \sigma[b \mapsto \mathit{Ind}\ x] \overset{*}{\rightarrow} \mathit{Num}\ z \wedge \sigma[b \mapsto \mathit{Ind}\ x]\phi\tau) \rightarrow (z,\tau)\ \mathbf{fi}$

$\sqsubseteq$ defs. *put* , *red*

$(\mathit{put}\ b\ (\mathit{Ind}\ x);\ \lambda\_.\mathit{red}\ (\mathit{last}(x : \mathit{xs})))\ \sigma$

$\equiv$ lemma *spine/I*

$(\mathit{put}\ b\ (\mathit{Ind}\ x);\ \lambda\_.\mathit{spine}\ x\ \mathit{xs} \hat{\succ} \mathit{red}\ (\mathit{last}(x : \mathit{xs})))\ \sigma$

$\equiv$ spec. *redS*

$(\mathit{put}\ b\ (\mathit{Ind}\ x);\ \lambda\_.\mathit{redS}\ x\ \mathit{xs})\ \sigma$

We have assumed one lemma about the auxiliary *spine* and another requirement for ϕ. The requirements for ϕ will be collected later. The lemma is:

Lemma 21 (spine/I) .

$I = \sigma\ v\ \wedge\ \mathit{spine}\ v\ (b : \mathit{xs})\ \sigma$

$\Rightarrow$

$\mathit{spine}\ x\ \mathit{xs}\ \sigma[b \mapsto \mathit{Ind}\ x]$

Informal justification. From *spine* $v\ (b : \mathit{xs})\ \sigma$ we get *spine* $b\ \mathit{xs}\ \sigma$ by the axioms of the definition of *spine*. We know that b does not occur in xs since it points to an application node whose left reference points to I. So *spine* $b\ \mathit{xs}\ \sigma$ is true regardless of what $\sigma\ b$ is. We overwrite $\sigma\ b$ with *Ind* x, to get *spine* $b\ \mathit{xs}\ \sigma[b \mapsto \mathit{Ind}\ x]$ and appeal to the indirection axiom of the definition of *spine*.

8.2.7 K is met

Now the case of combinator K. We recognise that K needs at least two arguments to yield a proper result, so we assume there are at least two pairs on the spine. Assume $\sigma\ v = K$ and *spine* $v\ (b : c : \mathit{xs})\ \sigma$. In this subsection let x and y be such that $\sigma\ b = \mathit{App}\ \_\ x$ and $\sigma\ c = \mathit{App}\ \_\ y$. Then

$\mathit{redS}\ v\ (b : c : \mathit{xs})\ \sigma$

$\equiv$ spec. *redS*

$\mathbf{if} \bigsqcap(z,\tau).(\mathit{extS}\ v\ (b : c : \mathit{xs})\ \sigma \overset{*}{\rightarrow} \mathit{Num}\ z \wedge \sigma\phi\tau) \rightarrow (z,\tau)\ \mathbf{fi}$

$\sqsubseteq$ lemma *extS/K*

$\mathbf{if} \bigsqcap(z,\tau).(\mathit{extS}\ x\ \mathit{xs}\ \sigma[c \mapsto \mathit{Ind}\ x] \overset{*}{\rightarrow} \mathit{Num}\ z \wedge \sigma\phi\tau) \rightarrow (z,\tau)\ \mathbf{fi}$

$\sqsubseteq$ ϕ trans., collect req. $\sigma\phi\sigma[c \mapsto \mathit{Ind}\ x]$

$\mathbf{if} \bigsqcap(z,\tau).(\mathit{extS}\ x\ \mathit{xs}\ \sigma[c \mapsto \mathit{Ind}\ x] \overset{*}{\rightarrow} \mathit{Num}\ z \wedge \sigma[c \mapsto \mathit{Ind}\ x]\phi\tau) \rightarrow (z,\tau)\ \mathbf{fi}$

$\sqsubseteq$ lemma *spine/K*

$(\mathit{put}\ c\ (\mathit{Ind}\ x);\ \lambda\_.\mathit{redS}\ x\ \mathit{xs})\ \sigma.$

Again we have used two lemmas. The first one is:

Lemma 22 (extS/K) .

$$K = \sigma\ v \wedge spine\ v\ (b : c : xs)\ \sigma$$
$$\Rightarrow$$
$$extS\ v\ (b : c : xs)\ \sigma \overset{*}{\rightarrow} extS\ x\ xs\ \sigma[c \mapsto Ind\ x]$$

It is a consequence of:

Lemma 23 (ext/K) .

$$K = \sigma\ v \wedge spine\ v\ (b : c : xs)\ \sigma$$
$$\Rightarrow$$
$$ext\ u\ \sigma \overset{*}{\rightarrow} ext\ u\ \sigma[c \mapsto Ind\ x]$$

The proof follows the same strategy as for combinator I, and is moved to the last section.

8.2.8 S encountered

The case of combinator S is more interesting, because it introduces sharing. Since S can only yield a proper result if it is applied to three arguments, we assume there are at least three pairs on the spine. Assume $\sigma\ v = S$ and $spine\ v\ (b : c : d : xs)\ \sigma$. Throughout this subsection let f, g, and x be such that $\sigma\ b = App\ \_\ f$, and $\sigma\ c = App\ \_\ g$, and $\sigma\ d = App\ \_\ x$. Then

$$
\begin{array}{ll}
 & redS\ v\ (b : c : d : xs)\ \sigma \\
\equiv & \qquad \text{spec. } redS \\
 & \textbf{if} \sqcap (z, \tau).(extS\ v\ (b : c : d : xs)\ \sigma \overset{*}{\rightarrow} Num\ z \wedge \sigma\phi\tau) \rightarrow (z, \tau)\ \textbf{fi} \\
\sqsubseteq & \qquad \text{lemma } extS/S\text{, abbr. } \sigma_3 \text{ given below} \\
 & \textbf{if} \sqcap (z, \tau).(extS\ d\ xs\ \sigma_3 \overset{*}{\rightarrow} Num\ z \wedge \sigma\phi\tau) \rightarrow (z, \tau)\ \textbf{fi} \\
\sqsubseteq & \qquad \phi \text{ trans., collect req. } \sigma\phi\sigma_3 \\
 & \textbf{if} \sqcap (z, \tau).(extS\ d\ xs\ \sigma_3 \overset{*}{\rightarrow} Num\ z \wedge \sigma_3\phi\tau) \rightarrow (z, \tau)\ \textbf{fi} \\
\sqsubseteq & \qquad \text{expand } \sigma_3\text{, lemma } spine/S \\
 & (new\ (App\ f\ x);\ \lambda fx.new\ (App\ g\ x);\ \lambda gx.put\ d\ (App\ fx\ gx);\ \lambda\_.redS\ d\ xs)\ \sigma.
\end{array}
$$

The abbreviation σ_3 in this whole subsection stands for

let $(fx, \sigma_1) = new\ (App\ f\ x)\ \sigma$ **in**
let $(gx, \sigma_2) = new\ (App\ g\ x)\ \sigma_2$ **in**
$\sigma_2[d \mapsto App\ fx\ gx]$.

We have appealed to

Lemma 24 (extS/S) .

$$S = \sigma\ v \wedge spine\ v\ (b : c : d : xs)\ \sigma$$
$$\Rightarrow$$
$$extS\ d\ xs\ \sigma \overset{*}{\rightarrow} extS\ d\ xs\ \sigma_3$$

which is a consequence of

Lemma 25 (ext/S) .

$S =\sigma\ v \wedge spine\ v\ (b : c : d : xs)\ \sigma$

$\Rightarrow$

$ext\ u\ \sigma \overset{*}{\rightarrow} ext\ u\ \sigma_3.$

The proof follows the same strategy as the proof of lemma ext/I, and is therefore moved to the last section.

8.2.9 Reference points to Y

For Y not to be stuck, at least one argument is required. Let us assume $\sigma\ v = Y$ and *spine* $v\ (b : xs)\ \sigma$. In this subsection let f be such that $\sigma\ b = App\ \_\ f$. Then

$redS\ v\ (b : xs)\ \sigma$

$\equiv$ spec. *redS*

if $\sqcap(z,\tau).(extS\ v\ (b : xs)\ \sigma \overset{*}{\rightarrow} Num\ z \wedge \sigma\phi\tau) \rightarrow (z,\tau)$ **fi**

$\sqsubseteq$ lemma *extS/Y*

if $\sqcap(z,\tau).(extS\ b\ xs\ \sigma[b \mapsto App\ f\ b] \overset{*}{\rightarrow} Num\ z \wedge \sigma\phi\tau) \rightarrow (z,\tau)$ **fi**

$\sqsubseteq$ ϕ transitive, collect req. $\sigma\ \phi\ \sigma[b \mapsto App\ f\ b]$

if $\sqcap(z,\tau).(extS\ b\ xs\ \sigma[b \mapsto App\ f\ b] \overset{*}{\rightarrow} Num\ z \wedge \sigma[b \mapsto App\ f\ b]\phi\tau) \rightarrow (z,\tau)$ **fi**

$\sqsubseteq$ defs. *put* , *red*

$(put\ b\ (App\ f\ b);\ \lambda\_.red(last(b : xs)))\ \sigma$

$\equiv$ lemma *spine/S*

$(put\ b\ (App\ f\ b);\ \lambda\_.spine\ b\ xs \hat{>\!\!-} red(last(b : xs)))\ \sigma$

$\equiv$ spec. *redS*

$(put\ b\ (App\ f\ b);\ \lambda\_.redS\ b\ xs)\ \sigma$

We used

Lemma 26 (extS/Y)

$\sigma\ v = Y \wedge spine\ v\ (b : xs)\ \sigma$

$\Rightarrow$

$extS\ v\ (b : xs)\ \sigma \overset{*}{\rightarrow} extS\ b\ xs\ \sigma[b \mapsto App\ f\ b]$

which is a consequence of the following lemma.

8.2.10 So what's ϕ ?

We collect the requirements for ϕ. The first three have already been discussed. Then there are four new ones.

1. Reflexive. $\forall\sigma.\sigma\phi\sigma$.
2. Transitive. $\sigma\phi\tau$ and $\tau\phi v$ imply $\sigma\phi v$.
3. Rewriting. If $\sigma\phi\tau$, then $ext\ v\ \sigma \overset{*}{\rightarrow} Num\ n$ implies $ext\ v\ \tau \overset{*}{\rightarrow} Num\ n$. The formal variable v ranges over all accessible references.
4. Reducing I. $\sigma\ v = I$ and $spine\ v\ (b : xs)\ \sigma$ imply $\sigma\phi\sigma[b \mapsto Ind\ x]$.

5. Reducing K. $\sigma\ v = K$ and *spine* $v\ (b : c : xs)\ \sigma$ imply $\sigma\phi\sigma[c \mapsto Ind\ x]$.
6. Reducing S. $\sigma\ v = S$ and *spine* $v\ (b : c : d : xs)\ \sigma$ imply $\sigma\phi\sigma_3$ where σ_3 is as before.
7. Reducing Y. $\sigma\ v = K$ and *spine* $v\ (b : xs)\ \sigma$ imply $\sigma\phi\sigma[b \mapsto App\ f\ b]$.

We can strengthen the third requirement simply to $\sigma\phi\tau$ implies $ext\ v\ \sigma \overset{*}{\to} ext\ v\ \tau$ for all accessible references v. By the lemmas *ext/I*, *ext/K*, *ext/S*, and *ext/Y*, the four requirements about combinators are then satisfied. We have gathered and analysed the requirements for the relation ϕ, now we'll define the relation ϕ.

Definition 15 (ϕ)

$$\begin{array}{rcl} \phi & : & \mathit{State} \to \mathit{State} \to \mathbb{B} \\ \phi\ \sigma\ \tau & \hat{=} & \textit{for all accessible references } v\textit{, ext } v\ \sigma \overset{*}{\to} \textit{ext } v\ \tau. \end{array}$$

8.2.11 Does it Terminate ?

We have found a recursive refinement for *redS*, that is an expression with holes F such that $redS \sqsubseteq F[redS]$. We would like to conclude $redS \sqsubseteq reduce$, where $reduce \,\hat{=}\, \mu f.F[f]$. For that we aim to use the law introducing a recursive function. We repeat it here:

Theorem 27 (Introducing recursive function) *If $x, y : T$, $<$ is a well-founded order on T, and expression $F[f]$ is monotone in f then*

$$(\forall x : T.E \sqsubseteq F[\lambda y.y < x \succ E[y/x]]) \Rightarrow (\lambda x.E \sqsubseteq \mu f.\lambda x.F[f]).$$

To ensure termination of the recursion, we have to find a well-founded order on the argument of the function. The law is formulated for a function of one argument. We can apply it to the function *redS* by considering its three arguments as a triple $Ref\ s\ (Vert\ s) \times [Ref\ s\ (Vert\ s)] \times State$. We need a well-founded order on that. In the following the unsuccessful search for such an order will be described.

Before the start of the search we note that some *SKIY* terms simply cannot be rewritten to a number. This may be due to 'recursion' expressed by the combinator Y, but there are also terms without Y that can't be rewritten to a number.

So our program is not obliged to terminate for any input, but only for members of this set: $\{s : SKIY \mid \exists z : \mathbb{Z}.s \overset{*}{\to} Num\ z\}$, the *hopeful* terms. Normal order reduction $\overset{n}{\to}$ as defined before is indeed a well-founded order for this set. That is, there are no infinite chains $s_0 \overset{n}{\to} s_1 \overset{n}{\to} s_2 \overset{n}{\to} \ldots$.

Thus, if we can show that each unfolding of *reduce* does the equivalent of one normal order step, we are fine.

The first problem is that because of sharing, the S, K, I branches of *reduce* do the work equivalent of one normal-order-reduction step, and maybe more. The Y definitely does more: it does the work of infinitely many steps, the first of which is a normal order step. We can solve that problem. The order $\overset{n*}{\to} \,\hat{=}\, (\overset{*}{\to})^* \circ (\overset{n}{\to}) \circ (\overset{*}{\to})^*$, that is, any number of rewriting steps, and one normal order reduction step, is still a well-founded order on the set of hopeful SKIY terms. If each unfolding of *reduce* does the equivalent of one $\overset{n*}{\rightarrow}$ step, termination necessarily follows.

The second problem is that two branches of *reduce* don't correspond to $\overset{n*}{\rightarrow}$ steps: the indirection following branch and the application vertex branch. A finite number of unfoldings that don't make progress are acceptable, but how can we be sure there is never an infinite succession of calls of these branches in the reduction of a hopeful SKIY term ? It seems that this is a hard question. Since the graph is always finite, the problem would occur if there was a cycle of indirections and application vertices on the spine. We would have to show that if a SKIY term is hopeful, then the graph generated from it never develops these harmful cyclic spines. This second problem remains unsolved.

The third problem is that the branch of *reduce* that deals with operator vertices does not pass (a representation of) the whole term to its two recursive calls, but (representations of) subterms. The argument and the arguments of its recursive calls don't satisfy the relation $\overset{n*}{\rightarrow}$, but the 'has-subterm' relation. It seems we should construct a termination proof by induction on the number of *Opr* vertices in a SKIY term. However, a SKIY term represented by a graph with a cycle may have infinitely many *Opr* vertices! This third problem is also unsolved.

In summary, we have no proof that unfolding the recursive definition of *reduce* makes progress. The program given is thus a 'partially correct' implementation of the specification: it will never return an incorrect answer, however, we have no guarantee it ever returns an answer. To conclude, we have implemented $eval \sqcap \bot$.

However, informally we may have some confidence in termination of the program. It is based on normal-order-reduction, which is guaranteed to terminate for hopeful terms. The second problem mentioned above stems from cycles on the spine; we suppose that graphs with these harmful cycles can only appear from SKIY terms with Y that are hopeless anyway.

8.2.12 The Program

If we had a termination proof for *reduce*, we'd conclude:

$$eval\ s \ \sqsubseteq\ \mathbf{run}(store\ s;\ \lambda\, v.reduce\ v\ [\,])$$

where

```
$reduce : Ref\ s\ (Vert\ s) \rightarrow [Ref\ s\ (Vert\ s)] \rightarrow STs\ \mathbb{Z}$
$reduce\ v\ xs \mathrel{\hat{=}} get\ v;\ \lambda\, node.$
          case $node$ of
            $Num\ z \rightarrow return\ z$
          $\sqcap\ Ind\ u \rightarrow reduce\ u\ xs$
          $\sqcap\ App\ f\ x \rightarrow reduce\ f\ (v : xs)$
          $\sqcap\ I \rightarrow b : xs' := xs \succ\!\!-$
                    $get\ b;\ \lambda\, App\ \_\ x.$
                    $put\ b\ (Ind\ x);\ \lambda\_.$
                    $reduce\ x\ xs'$
          $\sqcap\ K \rightarrow b : c : xs' := xs \succ\!\!-$
```

```
                  get b; λ App _ b.
                  put c (Ind x); λ_.
                  reduce x xs'
      ⊓ S       → b : c : d : xs' := xs>—
                  get b; λ App _ f.
                  get c; λ App _ g.
                  get d; λ App _ x.
                  new (App f x); λ fx.
                  new (App g x); λ gx.
                  put d (App fx gx); λ_.
                  reduce d xs'
      ⊓ Y       → (b,f) : xs' := xs>—
                  get b; λ App _ f.
                  put b (App f b); λ_.
                  reduce b xs'
      ⊓ Opr ⊕   → [b, c] := xs>—
                  get b; λ App _ x.
                  get c; λ App _ y.
                  reduce x [ ]; λ m.
                  reduce y [ ]; λ n.
                  put c (Num (m ⊕ n)); λ_.
                  return (m ⊕ n).
```

In any case, this program remains rather clumsy: all the *get* are superfluous, except the first one. They can be avoided by storing more information in the spine. The branch dealing with application vertex loses information: the x in *App f x* is lost. If we implement the spine as a list of pairs of references (each pointing to one application vertex, and to its argument child), the program becomes:

```
reduce : Ref s (Vert s) → [Ref s (Vert s) × Ref s (Vert s)] → STs ℤ
reduce v xs ≙ get v; λ node.
              case node of
                Num z   → return z
              ⊓ Ind u   → reduce u xs
              ⊓ App f x → reduce f ((v, x) : xs)
              ⊓ I       → (b, x) : xs' := xs>—
                          put b (Ind x); λ_.
                          reduce x xs'
              ⊓ K       → (b, x) : (c, y) : xs' := xs>—
                          put c (Ind x); λ_.
                          reduce x xs'
              ⊓ S       → (b, f) : (c, g) : (d, x) : xs' := xs>—
                          new (App f x); λ fx.
                          new (App g x); λ gx.
                          put d (App fx gx); λ_.
```

```
                 reduce d xs'
  ⊓ Y        → (b, f) : xs' := xs>—
                 put b (App f b); λ_.
                 reduce b xs'
  ⊓ Opr ⊕   → [(b, x), (c, y)] := xs>—
                 reduce x [ ]; λm.
                 reduce y [ ]; λn.
                 put c (Num (m ⊕ n)); λ_.
                 return (m ⊕ n).
```

8.3 Proofs

This sections lists proofs that were omitted from the main text.

8.3.1 Lemma ext/K

Lemma 28 (ext/K) .

$K = \sigma\ v \wedge spine\ v\ (b : c : xs)\ \sigma$

$\Rightarrow$

$ext\ u\ \sigma \overset{*}{\to} ext\ u\ \sigma[c \mapsto Ind\ x]$

Proof. We first show the special case $u = c$, and then generalise. Let $X[\ ]$ be the expression with a syntactic hole generated by extracting from x until c is reached. From

$$\begin{array}{ll} & ext\ c\ \sigma \\ \longrightarrow & \qquad \text{unfold } ext \\ & App(App\ K\ (ext\ x\ \sigma))(ext\ y\ \sigma) \\ \longrightarrow & \qquad \text{construction } X \\ & App(App\ K\ (X[ext\ c\ \sigma]))(ext\ y\ \sigma) \end{array}$$

we conclude

$$ext\ c\ \sigma \ \equiv\ \mu\, t.App(App\ K(X[t]))(ext\ y\ \sigma).$$

From

$$\begin{array}{ll} & ext\ c\ \sigma[c \mapsto Ind\ x] \\ \longrightarrow & \qquad \text{unfold } ext \\ & ext\ x\ \sigma[c \mapsto Ind\ x] \\ \longrightarrow & \qquad \text{construction } X \\ & X[ext\ c\ \sigma[c \mapsto Ind\ x]] \end{array}$$

we conclude

$$ext\ c\ \sigma[c \mapsto Ind\ x] \ \equiv\ \mu\, t.X[t].$$

Therefore

$$\begin{array}{ll} & \mathit{ext}\ c\ \sigma \\ \equiv & \\ & \mu\, t.\mathit{App}\ (\mathit{App}\ K\ (X[t]))\ (\mathit{ext}\ y\ \sigma) \\ \overset{*}{\dashrightarrow} & \qquad \text{SKIY rewrite rules} \\ & \mu\, t.X[t] \\ \equiv & \\ & \mathit{ext}\ c\ \sigma[c \mapsto \mathit{Ind}\ x], \end{array}$$

which we set out to show. Now for the general case. Let u be any reference. Let U ne the expression with a syntactic hole generated by extracting from u until c is reached. Then

$$\begin{array}{ll} & \mathit{ext}\ u\ \sigma \\ \equiv & \qquad \text{construction } U \\ & U[\mathit{ext}\ c\ \sigma] \\ \overset{*}{\dashrightarrow} & \qquad \text{just proved; compositional } \overset{*}{\dashrightarrow} \\ & U[\mathit{ext}\ c\ \sigma[c \mapsto \mathit{Ind}\ x]] \\ \equiv & \qquad \text{construction } U \\ & \mathit{ext}\ u\ \sigma[c \mapsto \mathit{Ind}\ x], \end{array}$$

which we set out to show.

Lemma 29 (spine/K)

$$\begin{array}{ll} & K =_\sigma v\ \wedge\ \mathit{spine}\ v\ (b : c : \mathit{xs})\ \sigma \\ \Rightarrow & \\ & \mathit{spine}\ x\ \mathit{xs}\ \sigma[c \mapsto \mathit{Ind}\ x] \end{array}$$

Informal justification. Similar to the lemma *spine/I*. From $\mathit{spine}\ v\ (b : c : \mathit{xs})\ \sigma$ we deduce $\mathit{spine}\ c\ \mathit{xs}\ \sigma$. Since c is the end of this spine, and has obviously not occurred in xs, it can be overwritten by anything. Therefore $\mathit{spine}\ c\ \mathit{xs}\ \sigma[c \mapsto \mathit{Ind}\ x]$. We use the indirections axiom of the definition of *spine*.

8.3.2 Lemma ext/S

Lemma 30 (ext/S) .

$$\begin{array}{ll} & S =_\sigma v \wedge \mathit{spine}\ v\ (b : c : d : \mathit{xs})\ \sigma \\ \Rightarrow & \\ & \mathit{ext}\ u\ \sigma \overset{*}{\dashrightarrow} \mathit{ext}\ u\ \sigma_3. \end{array}$$

Proof. We'll first show the special case $u = d$. Let F, G, X be the expressions with a syntactic hole generated by extracting from f, g, x respectively until d is reached. From

$$\begin{array}{ll} & \mathit{ext}\ d\ \sigma \\ \longrightarrow & \qquad \text{unfold } \mathit{ext} \\ & \mathit{App}(\mathit{App}(\mathit{App}\ S\ (\mathit{ext}\ f\ \sigma))\ (\mathit{ext}\ g\sigma))\ (\mathit{ext}\ x\ \sigma) \\ \longrightarrow & \qquad \text{constructions } F, G, X \\ & \mathit{App}(\mathit{App}(\mathit{App}\ S\ F[\mathit{ext}\ d\ \sigma])\ G[\mathit{ext}\ g\ \sigma])\ X[\mathit{ext}\ x\ \sigma] \end{array}$$

we conclude

$$ext\ c\ \sigma \ \equiv\ \mu\, t.App(App(App\ S\ F[t])\ G[t])\ X[t].$$

From

$ext\ d\ \sigma_3$
$\longrightarrow$ unfold ext
$App(App(ext\ f\ \sigma_3)(ext\ x\ \sigma_3))(App(ext\ g\ \sigma_3)(ext\ x\ \sigma_3))$
$\longrightarrow$ construction F, G, X
$App(App\ F[ext\ c\ \sigma_3]\ X[ext\ d\ \sigma_3])\ (App\ G[ext\ d\ \sigma_3]\ X[ext\ d\ \sigma_3])$

we conclude

$$ext\ d\ \sigma_3 \ \equiv\ \mu\, t.App(App\ F[t]\ X[t])\ (App\ G[t]\ X[t]).$$

Therefore
$ext\ c\ \sigma$
$\equiv$
$\mu\, t.App(App(App\ SF[t])\ G[t])\ X[t]$
$\overset{*}{\to}$ $\overset{*}{\to}$ rule
$\mu\, t.App(App(App\ SF[t])\ G[t])\ X[t] \quad t])$
$\overset{*}{\to}$ $\overset{*}{\to}$ rule
$\mu\, t.App(App\ F[t]\ X[t])\ (App\ G[t]\ X[t])$

Now to prove the general lemma, let u be any reference. Let U be the expression with a syntactic hole generated by extracting from u until d is reached. Then
$ext\ u\ \sigma$
$\equiv$ construction of U
$U[ext\ d\ \sigma]$
$\overset{*}{\to}$ just proven; compositional $\overset{*}{\to}$
$U[ext\ d\ \sigma_3]$
$\equiv$ construction U
$ext\ u\ \sigma_3$,
which is what we set out to show.

Lemma 31 (spine/S)
$S =_\sigma v\ \wedge\ spine\ v\ (b : c : d : xs)\ \sigma$
$\Rightarrow$
$spine\ d\ xs\ \sigma_3$.

Informal justification. From $spine\ v\ (b : c : d : xs)\ \sigma$ we deduce $spine\ d\ xs\ \sigma$. The reference d cannot appear in xs, so it can be overwritten arbitrarily. Therefore $spine\ d\ xs\ \sigma_3$.

8.3.3 Lemma ext/Y

Lemma 32 (ext/Y)

$$\begin{array}{l} \sigma\ v = Y \wedge spine\ v\ (b : xs)\ \sigma \\ \Rightarrow \\ ext\ u\ \sigma \overset{*}{\longrightarrow} ext\ u\ \sigma[b \mapsto App\ f\ b] \end{array}$$

The proof follows the same strategy as the case of combinator I.

Proof. Once again we first show the case $u = b$, and then the general case. Let F be the expression with a syntactic hole generated by extracting from f until b is reached.

$$ext\ f\ \sigma \quad \longrightarrow \quad F[ext\ b\ \sigma]$$

From

$$\begin{array}{ll} & ext\ b\ \sigma \\ \longrightarrow & \qquad \text{unfold } ext \\ & App\ Y\ (ext\ f\ \sigma) \\ \longrightarrow & \qquad \text{constr. } F \\ & App\ Y\ F[ext\ b\ \sigma] \end{array}$$

we conclude

$$ext\ b\ \sigma \ \equiv\ \mu\, t.App\ Y\ F[t].$$

From

$$\begin{array}{ll} & ext\ b\ \sigma[b \mapsto App\ f\ b] \\ \longrightarrow & \qquad \text{unfold } ext \\ & App\ (ext\ f\ \sigma[b \mapsto App\ f\ b])\ (ext\ b\ \sigma[b \mapsto App\ f\ b]) \\ \longrightarrow & \qquad \text{constr. } F \\ & App\ F[ext\ b\ \sigma[b \mapsto App\ f\ b]]\ (ext\ b\ \sigma[b \mapsto App\ f\ b]) \end{array}$$

we conclude

$$ext\ b\ \sigma[b \mapsto App\ f\ b] \ \equiv\ \mu\, t.App\ F[t]\ t.$$

Therefore

$$
\begin{array}{ll}
 & \mathit{ext}\ b\ \sigma \\
\equiv & \\
 & \mu\, t.\mathit{App}\ Y\ F[t] \\
\xrightarrow{*} & \qquad Y \text{ rewrite} \\
 & \mu\, t.\mu\, x.\mathit{App}\ F[t]\ x \\
\equiv & \qquad \text{merging recursion} \\
 & \mu\, t.\mathit{App}\ F[t]\ t \\
\equiv & \\
 & \mathit{ext}\ b\ \sigma[b \mapsto \mathit{App}\ f\ b].
\end{array}
$$

Now let u be any reference. Let U be the expression with a syntactic hole generated by extracting from u until b is reached. Then

$$
\begin{array}{ll}
 & \mathit{ext}\ u\ \sigma \\
\equiv & \\
 & U[\mathit{ext}\ b\ \sigma] \\
\xrightarrow{*} & \qquad \text{compositional } \xrightarrow{*} \\
 & U[\mathit{ext}\ b\ \sigma[b \mapsto \mathit{App}\ f\ b]] \\
\equiv & \qquad \text{constr. } U \\
 & \mathit{ext}\ u\ \sigma[b \mapsto \mathit{App}\ f\ b],
\end{array}
$$

which is what we set out to prove.

Lemma 33 (spine/Y)

$$
\begin{array}{ll}
 & \sigma\ v = Y \wedge \mathit{spine}\ v\ (b : \mathit{xs})\ \sigma \\
\Rightarrow & \\
 & \mathit{spine}\ b\ \mathit{xs}\ \sigma[b \mapsto \mathit{App}\ f\ b]
\end{array}
$$

Informal justification. From $\mathit{spine}\ v\ (b : \mathit{xs})\ \sigma$ we deduce $\mathit{spine}\ b\ \mathit{xs}\ \sigma$. Since b does not occur in xs, we can overwrite it.

Chapter 9

Denotational Semantics

This chapter gives denotational semantics for the specification language. The semantics are a (non-trivial) model of the calculus, and therefore the calculus is consistent. Each axiom can be proven sound with respect to the semantics by verifying that it is indeed a tautology.

A programmer needs the semantics to write the initial specification, and to decide what shape of program to aim for, but not in the derivation itself. The semantics are needed to decide what to prove, but not to do the proof. However, for the programmer, the informal semantics given in chapter 2 should suffice.

In giving the semantics, we use some algebra of complete partial orders and upward closed sets that is introduced briefly before the semantics are given. Finally, we give soundness proofs of some axioms.

For this chapter, the language is slightly simplified. We present only binary products and sums. The generalisations are trivial. We omit semantics of data refinement.

9.1 Complete Partial Orders and Upward Closed Sets

We'll interpret each type of the specification language by a complete partial order (cpo), and each expression as an upward closed subset of the interpretation of its type. We must order the interpretations of the types to give meaning to refinement and recursion. Expressions are interpreted as sets to capture nondeterminacy. This section describes the set and order algebra that will be used. First, we'll introduce some set algebra that will be used to construct the interpretations. Then, we'll present partial orders to go with the sets. In the third subsection, we'll construct some complete partial orders. These will later be used to interpret expressions.

9.1.1 The Sets

We start by building up an algebra of partially ordered sets from some primitive sets. Let's consider the sets first, and the associated orders in subsection 9.1.3. The primitive sets contain primitive, uniquely named values. An example is $\{\mathit{true}, \mathit{false}\}$. Other primitive sets are the naturals, integers, reals, and characters. The elements of each primitive set P come with a primitive equality, denoted $=_P$.

When giving interpretations to the types later on, we will use these six ways of constructing sets from other sets. They are Cartesian product, disjoint sum, monotone function space, fraction, upward closed sets, and convex sets.

$$\begin{array}{lcl}
A \times B & \hat{=} & \{(a,b) \mid a \in A, b \in B\} \\
A + B & \hat{=} & \{\mathbf{Inl}a, \mathbf{Inr}b \mid a \in A, b \in B\} \\
A \to B & \hat{=} & \{f \mid a \in A, f\ a \in B, f \text{ monotone}\} \\
\frac{A}{B} & \hat{=} & \{\mathbf{Hi}a, \mathbf{Lo}b \mid a \in A, b \in B\} \\
\mathbb{U}A & \hat{=} & \{\uparrow_A S \mid S \subseteq A\} \\
\mathbb{C}A & \hat{=} & \{\mathit{close}_A\ S \mid S \subseteq A\}
\end{array}$$

The first three constructions are standard. The functions are monotone with respect to the orders on A and B that will be presented later. Rather than representing functions as sets of pairs, we'll be content with taking functions as basic mathematical objects. The Cartesian product set $A \times B$ may also be used as the full relation between the sets A and B:

$$a\ A{\times}B\ b \ \hat{=}\ a \in A \wedge b \in B.$$

As sets, fractions are isomorphic to disjoint sums, but will later ordered differently. The fraction constructor will be used to add bottom elements to sets. For convenience, here is an abbreviation:

$$A_\perp \ \hat{=}\ \frac{A}{\{\perp\}}.$$

We say "A is lifted". In the definitions of disjoint sums and fractions, **Inl**, **Inr**, **Hi**, **Lo** are four distinct tags. We'll also extend the four tags **Inl**, **Inr**, **Hi**, **Lo** with sets and binary relations. For set S and relation $R : A \leftrightarrow B$, define

$$\begin{array}{lcl}
\mathbf{Inl}^* S & \hat{=} & \{\mathbf{Inl}s \mid s \in S\} \\
x\ \mathbf{Inl}^\bullet R\ y & \hat{=} & x = \mathbf{Inl}a \wedge y = \mathbf{Inl}b \wedge aRb.
\end{array}$$

The set $\mathbb{U}A$ of upward closed sets and the set $\mathbb{C}A$ of closed sets are different subsets of the powerset of A. They are defined in terms of the order associated with A. $\mathbb{U}A$ contains all *upward closed sets*, that is, sets that must contain y whenever $x \sqsubseteq y$ and they already contain x. Upward-closing a set (within universe A) is written as a prefix $\uparrow_A$ defined by

$$\uparrow_A S \,\hat{=}\, \{x \in A \mid \exists s \in S. s \sqsubseteq x\}.$$

$\mathbb{C}A$ contains all closed sets, that is, sets that contain the limits of all chains they contain. Closing a set (within universe A) is written by a prefix close_A defined by

$$\mathit{close}_A\ S \ \hat{=}\ \{x \in A \mid (\exists s \in S. s \sqsubseteq x) \wedge (\forall z. z \sqsubseteq x \Rightarrow \exists s \in S. z \sqsubseteq s)\}.$$

The mappings $\uparrow_A$ and close_A produce the smallest superset of a set that is upward closed, respectively closed. They produce the smallest such superset within the universe A, which may be omitted when obvious. An upward closed set is also closed, but not vice-versa.

Binary and general union and intersection of sets are as usual, and will also be used.

9.1.2 Relation Algebra

We define relation-constructors $\times, +, \rightarrow, -, \mathcal{S}, \mathcal{P}$ corresponding to the set-constructors $\times, +, \rightarrow, -, \mathbb{U}, \mathbb{C}$. From $R : A \leftrightarrow B$ and $S : C \leftrightarrow D$ we construct

$$\begin{array}{lcl}
R \times S & : & A \times C \leftrightarrow B \times D \\
(a, c)\; R \times S\; (b, d) & \hat{=} & aRb \wedge cSd \\
\\
R + S & : & A + C \leftrightarrow B + D \\
R + S & \hat{=} & \mathbf{Inl}^{\bullet} R \cup \mathbf{Inr}^{\bullet} S \\
\\
R \rightarrow S & : & A \rightarrow C \;\leftrightarrow\; B \rightarrow D \\
f\; R \rightarrow S\; g & \hat{=} & \forall\, a \in A, b \in B.\; aRb \Rightarrow f\; a\; S\; g\; b \\
\\
\frac{R}{S} & : & \frac{A}{B} \leftrightarrow \frac{C}{D} \\
\frac{R}{S} & \hat{=} & \mathbf{Hi}^{\bullet} R \cup \mathbf{Lo}^{\bullet} S \\
\\
\mathcal{S}R & : & \mathbb{U}A \leftrightarrow \mathbb{U}B \\
\alpha \mathcal{S}R\, \beta & \hat{=} & \forall\, b \in \beta. \exists\, a \in \alpha. aRb \\
\\
\mathcal{P}R & : & \mathbb{C}A \leftrightarrow \mathbb{C}B \\
\alpha \mathcal{P}R\, \beta & \hat{=} & \forall\, b \in \beta. \exists\, a \in \alpha. aRb \\
 & \wedge & \forall\, a \in \alpha. \exists\, b \in \beta. aRb
\end{array}$$

$R \times S$ is called "parallel composition", and in Ruby [JS93] is written $[R, S]$. $R \times S$ is not the Cartesian product of the two relations R and S seen as sets. The $\rightarrow$ combinator of relations can be used to express properties like monotonicity or anti-monotonicity. For example, "f is anti-monotone" is $f \leq\rightarrow\geq f$. Its calculational properties are explored in [Bac90], which gives "pointless" definitions for it.

$\mathcal{P}$ and $\mathcal{S}$ are ways of distributing a relation into a set. The symbols are chosen because, when applied to the order of the set elements, they give rise to the Plotkin and Smyth order respectively. The relation $\mathcal{P}R$ is the conjunction of $\mathcal{S}R$ and $\mathcal{H}R$, where $\mathcal{H}$ would give rise to the Hoare order. We won't use $\mathcal{H}$ however. We will also use the conjunction and disjunction of a set of relations. For S a set of relations between A and B, we have

$$\begin{array}{lcl}
\bigcap S, \bigcup S & : & A \leftrightarrow B \\
a \bigcap S\; b & \hat{=} & \forall\, R \in S. aRb \\
a \bigcup S\; b & \hat{=} & \exists\, R \in S. aRb.
\end{array}$$

Binary conjunction and disjunction of relations $R, R' : A \leftrightarrow B$ is as expected:

$$\begin{array}{lcl}
R \cup R', R \cap R' & : & A \leftrightarrow B \\
a\; R \cup R'\; b & \hat{=} & aRb \vee aR'b \\
a\; R \cap R'\; b & \hat{=} & aRb \wedge aR'b.
\end{array}$$

9.1.3 Partial Orders

We define *partial order* and then construct a partial order for each of the set constructions given in the first subsection.

A set S is *partially ordered* by the relation $\sqsubseteq: S \times S$ if $\sqsubseteq$ is reflexive, antisymmetric, and transitive. We say "S is a partial order".

For each set S constructed from primitive sets using $\times, +, -, \rightarrow, \mathbb{U}, \mathbb{C}$, we define a partial order $\sqsubseteq_S$ on its elements. We re-use the symbol $\sqsubseteq$, but this is not refinement of specification expressions, it's a relation between the mathematical values used to model the specification expressions. One can think of the symbol $\sqsubseteq$ as a mapping from a set expression to a relation. The argument is the index. We define $\sqsubseteq_S$ inductively on the structure of S seen as an expression. For primitive sets P and partial orders A and B, we have

$$\begin{array}{lcl}
\sqsubseteq_P & \hat{=} & =_P \\
\sqsubseteq_{A\times B} & \hat{=} & \sqsubseteq_A \times \sqsubseteq_B \\
\sqsubseteq_{A+B} & \hat{=} & \sqsubseteq_A + \sqsubseteq_B \\
\sqsubseteq_{A\rightarrow B} & \hat{=} & \sqsubseteq_A \rightarrow \sqsubseteq_B
\end{array}$$

Primitive sets P are mapped to primitive equality $=_P$, and the set constructors $\times, +, \rightarrow$ to the relation constructors $\times, +, \rightarrow$.

For arbitrary functions, our function 'order' $\sqsubseteq_{A\rightarrow B}$ is not a partial order. However, we only use monotone functions, and for them, $\sqsubseteq_A \rightarrow \sqsubseteq_B$ is indeed a partial order, and it is in fact the same as the usual order on functions $=_A \rightarrow \sqsubseteq_B$.

Fractions $\frac{A}{B}$ are treated specially: In addition to distributing $\sqsubseteq$ through the fraction constructor $-$, anything from B is below anything from A. It follows that $\sqsubseteq_{A_\perp} = \mathbf{Hi}^\bullet \sqsubseteq_A \cup (\{\mathbf{Lo}\perp\} \times A_\perp)$, that is, the order of $A_\perp$ is essentially the order of A, and in addition, $\mathbf{Lo}\perp$ is below everything.

$$\sqsubseteq_{\frac{A}{B}} \;\hat{=}\; \frac{\sqsubseteq_A}{\sqsubseteq_B} \cup (\mathbf{Lo}^* B \times \mathbf{Hi}^* A)$$

Upward closed sets are ordered using the Smyth-order (the order of the "weak powerdomain" in [Smy78]), whereas closed sets are ordered using the Plotkin order.

$$\begin{array}{lcl}
\sqsubseteq_{\mathbb{U}A} & \hat{=} & \mathcal{S} \sqsubseteq_A \\
\sqsubseteq_{\mathbb{C}A} & \hat{=} & \mathcal{P} \sqsubseteq_A
\end{array}$$

It doesn't conflict with the previous definitions to let order distribute into intersection and union of sets.

$$\begin{array}{lcl}
\sqsubseteq_{\cap S} & \hat{=} & \bigcap_{s\in S} \sqsubseteq_s \\
\sqsubseteq_{\cup S} & \hat{=} & \bigcup_{s\in S} \sqsubseteq_s .
\end{array}$$

Theorem 34 *For every set S built from primitive sets using $\times, +, \rightarrow, -, \mathbb{U}, \mathbb{C}$, and $\bigcap$, the relation $\sqsubseteq_S$ is indeed a partial order.*

For the case of $S = A+B$, $\sqsubseteq_S$ is defined as the relational disjunction $\mathbf{Inl}^\bullet \sqsubseteq_A \cup \mathbf{Inr}^\bullet \sqsubseteq_B$. In general, the relational disjunction of partial orders need not be a partial order, but in this case it is because $\mathbf{Inl}^\bullet \sqsubseteq_A$ and $\mathbf{Inr}^\bullet \sqsubseteq_B$ are disjoint.

As mentioned, the relation $\sqsubseteq_A \rightarrow \sqsubseteq_B$ is not a partial order for arbitrary functions: it is not reflexive, although it is antisymmetric and transitive. However, for *monotone* functions, $\sqsubseteq_{A\to B} = \sqsubseteq_A \to \sqsubseteq_B$ is indeed a partial order, and on monotone functions, it coincides with $=_A \rightarrow \sqsubseteq_B$.

Neither $\mathcal{S}\sqsubseteq_A$ nor $\mathcal{P}\sqsubseteq_A$ are antisymmetric for arbitrary subsets of A; but for upward closed, respectively closed, sets, they are antisymmetric.

When there is no confusion we will drop the subscript and write $\sqsubseteq$ for the order on the appropriate set.

9.1.4 Complete Partial Orders

We define *complete partial order* and then present some ways of constructing complete partial orders.

A set S is a *complete partial order* (cpo) ordered by $\sqsubseteq$ if S is partially ordered by $\sqsubseteq$, S contains a least element, and every increasing chain of elements of S has a least upper bound in S. We will write $\bot_S$ for the least element. It is least if $\bot \sqsubseteq s$ for every element of S. We will write $\bigsqcup_i s_i$ for the least upper bound of the chain $s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq ....$ It satisfies, for arbitrary x,

$$\forall i . s_i \sqsubseteq x \quad = \quad \bigsqcup_i s_i \sqsubseteq x.$$

The index i ranges over some unspecified index-set.

We call a function $f : A \to B$ for complete partial orders A and B *continuous* if it preserves least upper bounds. In symbols, that is,

$$f(\bigsqcup_i s_i) \quad = \quad \bigsqcup_i f\ s_i.$$

For primitive sets P and cpos A and B, these constructions are also cpos:

$P_\bot$
$(A \times B)_\bot$
$(A + B)_\bot$
$(A \to \mathbb{U}B)_\bot$
$(\mathbb{C}A)_\bot$.

There are others (for example $A \times B$ and $A \to B$), but here we only discuss those that will later be used as type-interpretations. A primitive set P is almost a cpo: it only lacks a least element. Therefore, we simply add one and form $P_\bot$. The product $A \times B$ is already a cpo, with least element $(\bot_A, \bot_B)$. However, since we aim to model lazy pair-formation in the specification language, we add a new least element. The least element of $(A \times B)_\bot$ is **Lo**$\bot$, which is strictly less than **Hi**$(\bot_A, \bot_B)$. The sum $A + B$ is not already a cpo, since it lacks a least element. The candidates **Inl**$\bot_A$ and **Inr**$\bot_B$ are incomparable. We must add a new bottom, and form $(A + B)_\bot$. As will be shown in the next subsection, for cpo B, the set $\mathbb{U}B$ is also a cpo. Therefore the monotone function space $A \to \mathbb{U}B$ is also a cpo. We use $\mathbb{U}B$ as the domain of the functions rather than B because we aim to model functions with nondetermined bodies. Because in the specification language we aim to distinguish the undefined function $\perp$ from the "everywhere undefined function" $\lambda x.\perp$ (which is itself defined), we lift the monotone function space, and form $(A \rightarrow \mathbb{U}B)_\perp$.

Finally we construct the cpo $(\mathbb{C}A)_\perp$ which will later be used to model set-values. In fact, $\mathbb{C}A \setminus \{\{\}\}$ is already a cpo, namely a version of the Plotkin powerdomain [Plo76]. Unlike [Plo76], we don't restrict ourselves to finitely-generable sets since we aim to model unbounded non-computational sets like $\mathbb{N}$ as well as computational sets. This matter is further discussed in section 9.3.1. Recall the definitions

$$\begin{aligned} \mathbb{C}A &\mathrel{\hat{=}} \{close_A\ S \mid S \subseteq A\} \\ close_A\ S &\mathrel{\hat{=}} \{x \in A \mid (\exists s \in S.s \sqsubseteq x) \wedge (\forall z.z \sqsubseteq x \Rightarrow \exists s \in S.z \sqsubseteq s)\}. \end{aligned}$$

By using closed subsets of A rather than just arbitrary subsets, we ensure that the order is in fact antisymmetric, that singleton formation and the union operator are as desired continuous, that and $\mathbb{C}A \setminus \{\{\}\}$ is in fact a cpo. A more detailed presentation can be found in [Plo76, Plo83, Sch86]. The least element of the cpo $\mathbb{C}A \setminus \{\{\}\}$ is $\{\perp_A\}$. The empty set has to be excluded because it is incomparable to any other set, in particular to $\{\perp_A\}$. However, we need to model the empty set too, and furthermore we aim to model *lazy* set formation (a set with undefined elements is not itself undefined) and therefore add a new least element below everything, including $\{\perp_A\}$ and $\{\}$, and form the cpo $(\mathbb{C}A)_\perp$.

Finally, here is one last cpo-construction: separated sum. It will not be used directly in interpreting the types of the specification language, but it will be necessary to define the universe of all outcomes later. For cpos A and B, the cpo $A \oplus B$ is defined as $((A \setminus \{\perp_A\}) + (B \setminus \{\perp_B\}))_\perp$ with the same order as $(A + B)_\perp$. This binary construction can be generalized to any finite number of arguments.

Summarising, we have the following theorem.

Theorem 35 *For primitive sets P and cpos A and B, the following are also cpos: $P_\perp, (A \times B)_\perp, (A + B)_\perp, (A \rightarrow \mathbb{U}B)_\perp, (\mathbb{C}A)_\perp, A \oplus B$, with notation and associated order as described.*

9.1.5 Upward Closed Sets

We will model expressions by upward closed sets. This subsection defines upward closed sets, and presents some lemmas about upward-closure. Finally it relates the order properties of a set A to the order properties of its upward closed subsets $\mathbb{U}A$ with a view to using least upper bounds and fixpoints to model recursion later.

The upward closure of set S within universe A is defined as

$$\uparrow_A S \mathrel{\hat{=}} \{b \in A \mid \exists a \in S.a \sqsubseteq b\}.$$

$\uparrow_A S$ is the set of elements that can be reached from elements of S by ascending. When the universe is obvious, we'll omit the subscript. It is easy to show that

Lemma 36 *Upward closing is monotone with respect to $\subseteq$: $A \subseteq B$ implies $\uparrow_U A \subseteq \uparrow_U B$.*

Upward closing by the relation $\sqsubseteq$ has these properties:

Lemma 37 *Upward closing increases a set:* $S \subseteq \uparrow S$.

This is an easy consequence of $\sqsubseteq$ being reflexive.

Lemma 38 *Upward closing is idempotent:* $\uparrow_A \uparrow_A S = \uparrow_A S$.

This lemma follows from reflexivity and transitivity of $\sqsubseteq$.

An easy proof will show

Lemma 39 *Arbitrary intersections and unions of upward closed sets (within the same universe) are upward closed:* $\bigcap A = \uparrow \bigcap A$ *and* $\bigcup A = \uparrow \bigcup A$ *for any set of upward closed sets* A.

Theorem 40 *For upward closed sets (within the same universe), the order* $\mathcal{S}\sqsubseteq$ *is simply supersetting:* $\uparrow A \;\mathcal{S}\sqsubseteq\; \uparrow B$ *iff* $\uparrow A \supseteq \uparrow B$.

This theorem is the motivation for rather than the consequence of using upward closed sets. If we interpreted specification language expressions simply as the set of their outcomes, then the order that captures the desired meaning of refinement would be the Smyth order $\mathcal{S}\sqsubseteq$ where $\sqsubseteq$ is the order on outcomes. But using upward closed sets of outcomes to interpret specification language expressions, the order interpreting refinement is the simpler superset order. The proof of the theorem is:

$$
\begin{array}{ll}
 & \uparrow A \supseteq \uparrow B \\
= & \qquad \text{superset} \\
 & \forall\, b \in \uparrow B . b \in \uparrow A \\
= & \qquad \text{reflexive } \sqsubseteq \\
 & \forall\, b \in \uparrow B . \exists\, a \in \uparrow A . a \sqsubseteq b \wedge b \in \uparrow A \\
= & \qquad \uparrow A \\
 & \forall\, b \in \uparrow B . \exists\, a \in \uparrow A . a \sqsubseteq b \\
= & \qquad \mathcal{S}\sqsubseteq \\
 & \uparrow A \;\mathcal{S}\sqsubseteq\; \uparrow B.
\end{array}
$$

Lemma 41 *In particular, for upward closed sets (within the same universe)* $\mathcal{S}\sqsubseteq$ *is antisymmetric:* $\uparrow A \;\mathcal{S}\sqsubseteq\; \uparrow B$ *and* $\uparrow B \;\mathcal{S}\sqsubseteq\; \uparrow A$ *imply* $\uparrow A = \uparrow B$.

This lemma gives a simple interpretation of specification equivalence, namely set equality.

A *complete lattice* is a set S ordered by some relation $\sqsubseteq$ such that $\sqsubseteq$ is a partial order, and least upper bounds and greatest lower bounds of arbitrary subsets of S exist. In particular, there are a least and a greatest element, necessary as the least upper and the greatest lower bound of the empty subset of S. Obviously, any complete lattice is also a cpo.

We conclude:

Theorem 42 *For each partial order S, upward closed subsets of S, $\mathbb{U}S$, are a complete lattice. The order is supersetting, the least element is S itself, the greatest element is the empty set, least upper bounds and greatest lower bounds are given by intersections and unions.*

The least upper bound of a chain of upward closed sets (that is, its intersection) obviously exists, and is itself upward closed, but it may be empty. For example, let $S_i \mathrel{\hat{=}} \{x \mid 0 < x \leq \frac{1}{i}\}$. The S_i are upward closed with respect to $\sqsubseteq_{\mathbb{R}}$ (which is simply $=_{\mathbb{R}}$) and moreover, since $S_i \supseteq S_{i+1}$ for each i, they form a chain, but $\bigcap\{S_i \mid i \in \mathbb{N}\}$ is empty. Now upward closed sets will be used to denote the expressions of the specification language, and in particular, the empty set will denote an infeasible expression. We will use least upper bounds to give semantics to recursive expressions. Clearly, there always is an upper bound, namely the empty set. However, one would expect a recursive expression (with feasible body) to be itself feasible. How can we ensure that empty sets cannot arise as the denotations of recursive expressions ?

If we demand, firstly, that S be not merely a partial order, but in fact a *complete* partial order, and secondly, restrict our attention to chains of (upward closed) *singleton* sets, that is $S_i = \uparrow\{s_i\}$ for each i, then we can guarantee *non-empty* least upper bounds. Since $S_i \sqsubseteq_{\mathbb{U}S} S_j$ implies $s_i \sqsubseteq_S s_j$, we obtain a chain $s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq \ldots$ in S. S is a cpo, and therefore this chain has a least upper bound $\bigsqcup_i s_i$. Since singleton formation $s \mapsto \uparrow\{s\}$ is continuous, we have $\bigsqcup_i \uparrow\{s_i\} = \uparrow\{\bigsqcup_i s_i\}$, our desired non-empty least upper bound. We have

Theorem 43 *For each complete partial order S, chains of upward closed singleton subsets of S have a non-empty least upper bound in the complete lattice $\mathbb{U}S$.*

In the semantics that follow this subsection, we often make sets upward-closed by applying $\uparrow$. In calculations, we can remove the $\uparrow$ under certain conditions, making use of monotonicity. For instance, when calculating the semantics of a function application, we will find sets given by $\{y \mid f \leftarrow \uparrow F, x \leftarrow \uparrow X, y \leftarrow f\ x\}$[1]. Given that all functions in F are monotone, we can remove both occurrences of $\uparrow$ without changing the set. The result is a combination of the following two lemmas.

Lemma 44 $\{y \mid f \leftarrow \uparrow F, x \leftarrow X, y \leftarrow f\ x\} = \{y \mid f \leftarrow F, x \leftarrow X, x \leftarrow f\ x\}$, *where F is a set of functions to upward closed sets.*

Proof.

$$
\begin{array}{ll}
 & \{y \mid f \leftarrow \uparrow F, x \leftarrow X, y \leftarrow f\ x\} \\
= & \qquad \text{set comprehension} \\
 & \bigcup\, (\lambda f.\{y \mid x \leftarrow X, y \leftarrow f\ x\})^{*}\ \uparrow F \\
= & \qquad \lambda f.\{y \mid x \leftarrow X, y \leftarrow f\ x\} \text{ is } \sqsubseteq\!\to\!\supseteq \text{ monotone, lemma 46} \\
 & \bigcup\, (\lambda f.\{y \mid x \leftarrow X, y \leftarrow f\ x\})^{*}\ F \\
= & \qquad \text{set comprehension} \\
 & \{y \mid f \leftarrow F, x \leftarrow X, y \leftarrow f\ x\}.
\end{array}
$$

[1]We use set-comprehension notation with $\leftarrow$. It may be more familiar to some readers when $\in$ is substituted for $\leftarrow$. Others may want to remove the set comprehension notation completely using these rules: $\{e \mid x \leftarrow s\} = \{e \mid x \in s\}$, and $\{e \mid q_1, \ldots, q_n\} = \bigcup\{\{e \mid q_2, \ldots, q_n\} \mid q_1\}$ where each q_i is of the form $x \leftarrow s$ for some variable x and set s

The function $\lambda f.\{y \mid x \leftarrow X, y \leftarrow f\ x\}$ is indeed $\sqsubseteq\!\rightarrow\!\supseteq$ monotone, since its body is $\supseteq\!\rightarrow\!\supseteq$ monotone in $f\ x$, and the order on upward-closed-set-valued functions (like f) gives us $f \sqsubseteq f'$ is $\forall x.f\ x \supseteq f'\ x$. Lemma proven, for arbitrary functions $f : A \rightarrow \mathbb{U}B$.

Lemma 45 *$\{y \mid f \leftarrow F, x \leftarrow \uparrow X, y \leftarrow f\ x\} = \{y \mid f \leftarrow F, x \leftarrow X, x \leftarrow f\ x\}$, where F is a set of $\sqsubseteq\!\rightarrow\!\supseteq$ monotone functions.*

Proof.

$$\begin{array}{ll} & \{y \mid f \leftarrow F, x \leftarrow \uparrow X, y \leftarrow f\ x\} \\ = & \qquad \text{set comprehension} \\ & \bigcup \{\bigcup f^* \uparrow X \mid f \leftarrow F\} \\ = & \qquad f \text{ is } \sqsubseteq\!\rightarrow\!\supseteq \text{ monotone, lemma 46} \\ & \bigcup \{\bigcup f^* X \mid f \leftarrow F\} \\ = & \qquad \text{set comprehension} \\ & \{y \mid f \leftarrow F, x \leftarrow X, y \leftarrow f\ x\}. \end{array}$$

Lemma proven, using monotonicity of f.

We used the lemma:

Lemma 46 *$\bigcup f^* \uparrow S = \bigcup f^* S$, if f is $\sqsubseteq\!\rightarrow\!\supseteq$ monotone.*

Proof.

$$\begin{array}{ll} & a \in \bigcup f^* \uparrow S \\ = & \qquad \uparrow \\ & \exists x. \exists s \in S.s \sqsubseteq x \wedge a \in f\ x \\ = & \qquad f \text{ is } \sqsubseteq\!\rightarrow\!\supseteq \text{ monotone} \\ & \exists s \in S.a \in f\ s \\ = & \\ & a \in \bigcup f^* S. \end{array}$$

Lemma proven.

Another place where $\uparrow$ can be dropped is in calculating the semantics of enumerated sets. Here we meet sets of the form $\uparrow\{close\{v_1, ..., v_n\} \mid v_i \leftarrow \uparrow S_i\}$, which are equal to $\uparrow\{close\{v_1, ..., v_n\} \mid v_i \leftarrow S_i\}$.

Lemma 47 $\uparrow\{close\{v_1, ..., v_n\} \mid v_i \leftarrow \uparrow S_i\} = \uparrow\{close\{v_1, ..., v_n\} \mid v_i \leftarrow S_i\}$

Proof.

$$\begin{array}{ll} & s \in LHS \\ = & \qquad \uparrow \\ & \exists v_i. \exists s_i \in S_i.s_i \sqsubseteq v_i \wedge close\{v_1, ..., v_n\}\ \mathcal{P}\!\sqsubseteq\ s \\ = & \qquad close\{a_1, ..., a_n\} \text{ is } \sqsubseteq\!\rightarrow \mathcal{P}\!\sqsubseteq \text{ monotone in } a_i,\ \mathcal{P}\!\sqsubseteq \text{ transitive} \\ & \exists s_i \in S_i.close\{s_1, ..., s_n\}\ \mathcal{P}\!\sqsubseteq\ s \\ = & \\ & s \in RHS. \end{array}$$

9.2 Semantics

We'll now give semantics to the specification language, but first some preliminaries. The semantics of state transformers are given in a separate subsection. We'll use three

$\mathcal{T}\llbracket \mathbb{B} \rrbracket\iota$	$\hat{=}$	$\{\mathit{true}, \mathit{false}\}_\bot$	example of a primitive type
$\mathcal{T}\llbracket A \times B \rrbracket\iota$	$\hat{=}$	$(\mathcal{T}\llbracket A \rrbracket\iota \times \mathcal{T}\llbracket B \rrbracket\iota)_\bot$	pairs
$\mathcal{T}\llbracket A + B \rrbracket\iota$	$\hat{=}$	$(\mathcal{T}\llbracket A \rrbracket\iota + \mathcal{T}\llbracket B \rrbracket\iota)_\bot$	sums
$\mathcal{T}\llbracket A \rightarrow B \rrbracket\iota$	$\hat{=}$	$\{\text{monotone } f : D \rightarrow D$ $\mid \forall\, d \in \mathcal{T}\llbracket A \rrbracket\iota. f\ d \in \mathbb{U}\mathcal{T}\llbracket B \rrbracket\iota\}_\bot$	functions
$\mathcal{T}\llbracket \mathbb{P}\, A \rrbracket\iota$	$\hat{=}$	$(\mathbb{C}\mathcal{T}\llbracket A \rrbracket\iota)_\bot$	sets
$\mathcal{T}\llbracket \forall\, a. T \rrbracket\iota$	$\hat{=}$	$\bigcap_\tau \mathcal{T}\llbracket T \rrbracket\iota[a \mapsto \tau]$	typeschemes
$\mathcal{T}\llbracket a \rrbracket\iota$	$\hat{=}$	$\iota\ a$	type variables
$\mathcal{T}\llbracket \mu\, a. T \rrbracket\iota$	$\hat{=}$	$\mu\, F$ where $F\ S \mathrel{\hat{=}} \mathcal{T}\llbracket T \rrbracket\iota[a \mapsto S]$	recursive types

Figure 9.1: Interpretations of the types

meaning functions. $\mathcal{T}\llbracket\ \rrbracket$ gives the meaning of a type, which is a set, $\mathcal{E}\llbracket\ \rrbracket\rho$ gives the meaning of an expression in environment ρ, which is a set, and $\mathcal{C}\llbracket\ \rrbracket$ gives the meaning of a constant, which is a single element.

9.2.1 Types and Environments

Every type T is interpreted by a set of outcomes $\mathcal{T}\llbracket T \rrbracket\iota$, where ι is a mapping from type variables to sets. $\mathcal{T}\llbracket T \rrbracket\iota$ will be a subset of the universe of outcomes D, to be described shortly. Figure 9.1 gives the interpretations of the types.

The primitive types are interpreted by the appropriate primitive sets, lifted to add a least element. Pair types and sum types are interpreted by lifted cartesian products and lifted disjoint sets. This means that pair and sum expressions may contain undefinedness without being themselves undefined. Function types are interpreted by lifted monotone function spaces, with range upward closed sets. This makes the undefined function $\bot$ different from the "everywhere-undefined" function $\lambda x.\bot$. The range of the interpretation are sets to capture the nondeterminacy contained in function-expression bodies. Monotone functions are sufficient to model the functions in the language since (for reasons explained in subsection 3.3.4, under "Functions") we restricted functions in the language to be monotone. The set type is interpreted by closed sets lifted. This means a set expression can contain undefinedness without being itself undefined.

The interpretation of a typescheme $\forall\, a. T$ is the intersection of all its interpreted instantiations T where the type variable a is replaced by a (monomorphic) type τ. For example, the interpretation of $\forall\, a.[a]$ will be the intersection of all monomorphic list-types. This intersection is not empty. It will contain representations of the undefined list $\bot$, the empty list $[\ ]$, and all lists obtained by "cons"-ing $\bot$ onto these lists arbitrarily many times. The interpretation of the type $\forall\, a, b. a \times b \rightarrow b \times a$ is a subset of the intersection of, for instance, the interpretations of $\mathbb{N} \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{N}$ and $\mathbb{B} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{B}$. This intersection is not empty. It contains the (representation of the) polymorphic pair-swapping function and various lower functions. This becomes obvious when the interpretation of $\mathbb{N} \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{N}$ is considered: it is $\{\text{monotone } f : D \rightarrow D \mid \forall\, d \in \mathcal{T}\llbracket \mathbb{N} \times \mathbb{B} \rrbracket\iota.\ f\ d \in \mathbb{U}\mathcal{T}\llbracket \mathbb{B} \times \mathbb{N} \rrbracket\iota\}_\bot$. A non-bottom element of it, f say, maps values of

$\mathbb{N} \times \mathbb{B}$ to (upward closed sets of) values of $\mathbb{B} \times \mathbb{N}$; but it also maps other elements of D to arbitrary elements of D. What these elements are doesn't matter, because type-checking ensures that such applications will never be of interest. However, the interpretation of the polymorphic pair-swapping functions is indeed an element of $\mathcal{T}[\![\mathbb{N} \times \mathbb{B} \to \mathbb{B} \times \mathbb{N}]\!]\iota$, and of every other type of shape $T \times U \to U \times T$, and therefore also of $\mathcal{T}[\![\forall\, a, b.a \times b \to b \times a]\!]\iota$.

A type variable is interpreted simply by looking it up in the mapping ι. The interpretation of $\mu\, a.T$ under type-variable mapping ι is the least fixpoint (with respect to $\subseteq$) of the set-to-set function F, which maps a set S to $\mathcal{T}[\![T]\!]\iota[a \mapsto S]$. The fixpoint exists if F is monotone with respect to $\subseteq$. We will only use recursive types for which this is indeed the case. This is quite easy to ensure since apart from function types, which are not monotone in their first argument, all type constructions are monotone in their arguments. However, to make types that are easily understood, all the recursive types we use have the form $\mu\, a.T$ where T is a sumtype, one of whose summands is a product containing a. Types are equal under unfolding of recursion, that is $\mu\, a.T = T[(\mu\, a.T)/a]$.

By theorem 35 every type interpretation is a complete partial order.

The universe of all outcomes D should be a set that contains as subsets $\mathcal{T}[\![T]\!]\iota$ for every type T. Since for every type T and U we also have the types $T \times U$, $T + U$, $T \to U$, and $\mathbb{P}\, T$, we must ensure that for all subsets A and B of D that interpret the types T and U, respectively, the sets $(A \times B)_\perp$, $(A + B)_\perp$, $(A \to \mathbb{U}B)_\perp$, and $(\mathbb{C}A)_\perp$ are also subsets of D. Naturally the interpretations $P^i_\perp$ of the primitive types must also be subsets of D $(1 \le i \le k)$.

One way to achieve this is to define D as the cpo that is the least fixed point of the recursive domain equation

$$D \;\cong\; P^1_\perp \oplus \ldots \oplus P^k_\perp \oplus (D \times D)_\perp \oplus (D + D)_\perp \oplus (D \to \mathbb{U}D)_\perp \oplus (\mathbb{C}D)_\perp .$$

We use the symbol $\cong$ rather than $=$ to indicate that D need not be *equal* to the right hand side. It need only be *isomorphic* to it, that means, there should be an order-preserving bijection between the two. Of course the functions $D \to \mathbb{U}D$ must be monotone.

Such recursive domain equations are studied for example in [Sto77, Plo83, Sch86, SHLG94], and there are various approaches for finding solutions to them, for instance the category-theoretic method described in [SP82]. It is beyond the scope of this thesis to describe the methods, but a solution is guaranteed to exist if the cpo-constructors used on the right hand side satisfy certain conditions.

In the above recursive domain equation, the only problematic constructors are monotone function space $\_ \to \_$ and lifted closed subsets $(\mathbb{C}\_)_\perp$. Of the other constructors, lifting $\_{}_\perp$, coalesced sum $\oplus$, pairing $\times$, and separated sum $(\_ + \_)_\perp$ are shown to be well-behaved in standard texts such as [Sch86, SHLG94], and the upward-closed set constructor $\mathbb{U}$ is dealt with in [Hec90].

The monotone function space constructor makes non-trivial solutions impossible, noted in [Sto77]. This is also true for arbitrary functions, and is caused by comparing cardinalities. The standard solution to this problem, presented first in [SS71, Sco75], is to restrict ourselves to *continuous* functions only. For the purpose of modelling a programming language, this is acceptable, since programs are computable, and computable functions are continuous. However, in a specification language we need to model unbounded nondeterminacy – which is not computable, as demonstrated in [Dij82] – and therefore we require non-continuous functions. Presently, we have no solution to this problem, and therefore restrict ourselves to continuous functions. That means we are guaranteed existence of a universe D, but our semantics does not cover expressions with unbounded nondeterminacy.

The second unusual constructor is the lifted closed-subset constructor $(\mathbb{C}\_)_\perp$. It is related to Plotkin's powerdomain constructor $\wp[\_]$ described in [Plo76], but slightly different. In particular, we do not exclude the empty set and sets that are not *finitely-generable*, that is non-computable infinite sets. Inclusion of the empty set should not be problematic, although it does force us to lift $\mathbb{C}\_$ in order to have a least element. However, it could be that inclusion of non-finitely generable sets causes problems similar to those caused by unbounded nondeterminacy. Recursive domain equations involving Plotkin's powerdomain constructor $\wp[\_]$ can indeed be solved, although it is necessary to forsake the category *CPO* in which the objects are cpos for the category *SFP* whose objects are certain limits of *CPO*. It is not known whether a similar construction can be used with $(\mathbb{C}\_)_\perp$. However, sets are much less central to our expression language than functions. It if turns out that $(\mathbb{C}\_)_\perp$ is not well-behaved in a suitable category, we have not given semantics to set expressions and "$\oplus(\mathbb{C}D)_\perp$" should be erased from the recursive domain equation specifying the universe D.

Environments map each variable of type T to an outcome, that is, an element of the type interpretation of T. We use ρ to stand for an environment. Environment ρ maps the variable x to the outcome $\rho\ x$. The environment $\rho[x \mapsto o]$ maps x to o and otherwise agrees with ρ. Environments are typed with contexts. We write $\rho : \Gamma$ to say that $dom\ \rho = dom\ \Gamma$, and for every variable $x \in dom\ \rho$, we have $\rho\ x \in \mathcal{T}[\![\Gamma\ x]\!]$.

It can be shown that for any expression E and environment ρ, the set $\mathcal{E}[\![E]\!]\rho$ is upward closed. This is easy to verify by examining the definitions of $\mathcal{E}[\![E]\!]\rho$ for all the shapes of E. The meaning of many shapes of expressions is explicitly upward closed. In other cases upward closedness follows from the facts that unions and intersections preserve upward closedness (lemma 39), and that each type interpretation is upward closed.

The meaning of expression E in the environment ρ is the set of outcomes $\mathcal{E}[\![E]\!]\rho$. It can be proven that the semantics respects the typing:

$$\Gamma \vdash E : T \quad \textit{implies} \quad \forall \rho : \Gamma . \mathcal{E}[\![E]\!]\rho \in \mathbb{U}\mathcal{T}[\![T]\!]\iota.$$

In this nondeterministic language $\mathcal{E}[\![E]\!]\rho$ is an element of $\mathbb{U}\mathcal{T}[\![T]\!]\iota$ rather than an element of $\mathcal{T}[\![T]\!]\iota$, as may be expected for a deterministic language. The proof is by induction on derivations in the type rules for $\Gamma \vdash E : T$, as in [Wad92a], using the typing rules. For each typing rule $\frac{\Gamma_i \vdash E_i : T_i}{\Gamma \vdash E : T}$ we need to show that $\forall \rho : \Gamma . \mathcal{E}[\![E]\!]\rho \in \mathbb{U}\mathcal{T}[\![T]\!]\iota$ follows from $\forall \rho_i : \Gamma_i . \mathcal{E}[\![E_i]\!] \in \mathbb{U}\mathcal{T}[\![T_i]\!]\iota$, for each of the i hypotheses.

In the following subsections we'll give the interpretations of the expressions, in separate subsections for **def**, **det**, **feas**, for refinement and equivalence, for application expressions, for constant expressions, for specificational expressions, for set expressions,

$$\begin{array}{rcll}
\mathcal{E}[\![\mathbf{def}E]\!]\rho & \hat{=} & \{\mathit{false}\}, & \text{if } \mathcal{E}[\![E]\!]\rho = \uparrow\{\bot\} \\
 & \hat{=} & \{\mathit{true}\}, & \text{otherwise} \\
\mathcal{E}[\![\mathbf{det}E]\!]\rho & \hat{=} & \{\mathit{true}\}, & \text{if } \exists\, v \in \mathcal{E}[\![E]\!]\rho.\mathcal{E}[\![R]\!]\rho = \uparrow\{v\} \\
 & \hat{=} & \{\mathit{false}\}, & \text{otherwise} \\
\mathcal{E}[\![\mathbf{feas}E]\!]\rho & \hat{=} & \{\mathit{false}\}, & \text{if } \mathcal{E}[\![E]\!]\rho = \{\} \\
 & \hat{=} & \{\mathit{true}\}, & \mathit{otherwise}
\end{array}$$

Figure 9.2: Meaning of definedness, determinacy, and feasibility

$$\begin{array}{rcll}
\mathcal{E}[\![E \sqsubseteq F]\!]\rho & \hat{=} & \{\mathit{true}\}, & \text{if } \mathcal{E}[\![E]\!]\rho \supseteq \mathcal{E}[\![F]\!]\rho \\
 & \hat{=} & \{\mathit{false}\}, & \text{otherwise} \\
\mathcal{E}[\![E \equiv F]\!]\rho & \hat{=} & \{\mathit{true}\}, & \text{if } \mathcal{E}[\![E]\!]\rho = \mathcal{E}[\![F]\!]\rho \\
 & \hat{=} & \{\mathit{false}\}, & \text{otherwise}
\end{array}$$

Figure 9.3: Meaning of refinement and equivalence

for quantifier expressions, and finally, for imperative expressions.

9.2.2 Definedness, Determinacy, and Feasibility

The meaning of the definedness-, determinacy-, and feasibility-testers is given in figure 9.2. The meaning of **def**E is the singleton set containing *false* if its meaning includes the undefined outcome (of the appropriate type). Otherwise it is the singleton set containing *true*. An expression is determined if its meaning contains a minimum outcome, or equally, if its meaning is equal to the upward-closure of the singleton set containing that minimum outcome. An expressions is feasible unless its meaning is the empty set.

9.2.3 Refinement and Equivalence

The meaning of refinement and equivalence is given in figure 9.3. The meaning of $E \sqsubseteq F$ is $\{\mathit{true}\}$, if the meaning of E is a superset of the meaning of F. Otherwise, it is $\{\mathit{false}\}$. Both sets $\{\mathit{true}\}$ and $\{\mathit{false}\}$ are indeed upward closed. The meaning of $E \equiv F$ is $\{\mathit{true}\}$ if the meanings of E and F are the same set. Otherwise, it is $\{\mathit{false}\}$. Trivially, mutual refinement is equivalence.

9.2.4 Applicative Expressions

Figure 9.4 gives meaning to the expressions from the λ calculus with constants, recursion, and **let** expressions. Semantics and types for some constants are given in the following subsection. The meaning of a constant expression is an upward closed set. For some constants, for example 7, this set will just be a singleton, for example $\{7\}$, which is indeed upward closed. But for other constants, in particular functional constants, the set will not be a singleton.

The meaning of a variable is simply the variable looked up in the environment, packed into an upward closed set. As for constant expressions, that set may or may

$\mathcal{E}[\![k]\!]\rho$	$\hat{=}$	$\uparrow\{\mathcal{C}[\![k]\!]\}$	constant
$\mathcal{E}[\![x]\!]\rho$	$\hat{=}$	$\uparrow\{\rho\, x\}$	variable
$\mathcal{E}[\![F\ X]\!]\rho$	$\hat{=}$	$\{y \mid f \leftarrow \mathcal{E}[\![F]\!]\rho, x \leftarrow \mathcal{E}[\![X]\!]\rho, y \leftarrow f\ x\}$	application
$\mathcal{E}[\![\lambda\, x.E]\!]\rho$	$\hat{=}$	$\uparrow\{\lambda\, v.\mathcal{E}[\![E]\!]\rho[x \mapsto v]\}$	abstraction
$\mathcal{E}[\![\mu\, x.E]\!]\rho$	$\hat{=}$	$\mu\, F$	recursion
where $F\ X$	$\hat{=}$	$\{y \mid v \leftarrow X, y \leftarrow \mathcal{E}[\![E]\!]\rho[x \mapsto v]\}$	
$\mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ F]\!]\rho$	$\hat{=}$	$\{w \mid v \leftarrow \mathcal{E}[\![E]\!]\rho, w \leftarrow \mathcal{E}[\![F]\!]\rho[x \mapsto v]\}$	local definition

Figure 9.4: Meaning of the applicative expressions

not be a singleton. The meaning of an application expression $F\ X$ is a set, constructed as follows. Apply each outcome f of F (these will be set-valued functions) to each outcome x of X, obtaining sets, and then union all those sets. The meaning of an abstraction expression $\lambda\, x.E$ in environment ρ is a set. It is the upward closure of the singleton set containing a function from a value v to the set obtained by interpreting E is the environment that maps x to v, and otherwise coincides with ρ. The functions in the set are monotone functions. Monotone functions are sufficient to interpret any specification language function, because these are restricted to being monotone (with respect to refinement) themselves. Here we are using λ in the target language purely as a convenient abbreviation. Equivalently, we could have written $\mathcal{E}[\![\lambda\, x.E]\!]\rho \mathrel{\hat{=}} \uparrow\{f \mid \forall\, v.f\ v = \mathcal{E}[\![E]\!]\rho[x \mapsto v]\}$, thereby eliminating references to λ from the target language.

The meaning of a recursive expression $\mu\, x : T.E$ is the least (with respect to $\mathcal{S}\sqsubseteq$) fixpoint of the set-to-set function F. F applied to a set X yields the union of E interpreted with x bound to each element of X. By construction, F is monotone with respect to $\mathcal{S}\sqsubseteq$ (which is $\supseteq$ by theorem 40), the order we use for sets representing nondeterminacy. This can be verified (without using any property of E) by noticing that $x \in \_$ is $\subseteq\!\to\!\Rightarrow$ monotone, conjunction and existential quantification are $\Rightarrow\!\to\!\Rightarrow$ monotone, $\{y \mid \_\}$ is $\Rightarrow\!\to\!\subseteq$ monotone, and $\uparrow$ is $\subseteq\!\to\!\subseteq$ monotone (lemma 36). By the Generalised Limit Theorem (from [HP72], repeated in [Nel89]), the least fixpoint of F exists, and is F^λ for some ordinal number λ. For any ordinal α, F^α is defined

$$F^\alpha \mathrel{\hat{=}} \bigsqcup\{F(F^\beta) \mid \beta < \alpha\}.$$

In our case, the order $\mathcal{S}\sqsubseteq$ is $\supseteq$, and therefore $\bigsqcup$ is (notationally confusing) $\bigcap$. For illustration, we have $F^0 = \bigcap\{\} = \uparrow\{\bot_T\}$, and that is the meaning of recursion if $F\ S = S$, that is, the recursion makes no progress. For natural n, we have $F^{n+1} = F(F^n)$.

As detailed in the previous section, if E, the body of the recursive expression $\mu\, x.E$, is determined, then we are guaranteed that F has a *non-empty* fixpoint, that is, $\mu\, x.E$ is as desired feasible. See theorem 43.

The meaning of the **let** expression $\mathbf{let}\ x = E\ \mathbf{in}\ F$ is constructed as the meaning of $(\lambda\, x.F)E$. However, this reshuffling of parts of a **let** expression need not have a type seen as a specification language expression in its own right, even though the **let** expression has. Therefore, **let** expressions are not merely syntactic sugar.

7	$:$	$\mathbb{N}$	
$\mathcal{C}[\![7]\!]$	$\hat{=}$	7	numbers
$\neg$	$:$	$\mathbb{B} \to \mathbb{B}$	
$\mathcal{C}[\![\neg]\!]$	$\hat{=}$	$\lambda_\perp\, v.\{\neg v\}$	negation
$+$	$:$	$\mathbb{R} \to \mathbb{R} \to \mathbb{R}$	
$\mathcal{C}[\![+]\!]$	$\hat{=}$	$\lambda_\perp\, a.\uparrow\{\lambda_\perp\, b.\{a+b\}\}$	addition
$(\_,\_)$	$:$	$x \to y \to x \times y$	
$\mathcal{C}[\![(\_,\_)]\!]$	$\hat{=}$	$\lambda x.\uparrow\{\lambda y.\uparrow\{(x,y)\}\}$	pairing
fst	$:$	$x \times y \to x$	
$\mathcal{C}[\![\mathit{fst}]\!]$	$\hat{=}$	$\lambda_\perp\, x.\uparrow\{\mathit{fst}\ x\}$	projections
$\mathbf{Inl}$	$:$	$x \to x + y$	
$\mathcal{C}[\![\mathbf{Inl}]\!]$	$\hat{=}$	$\lambda x.\uparrow\{\mathit{Inl}\ x\}$	constructors
$\_ \bigtriangledown \_$	$:$	$(x \to z) \to (y \to z) \to x + y \to z$	
$\mathcal{C}[\![\_ \bigtriangledown \_]\!]$	$\hat{=}$	$\lambda f.\uparrow\{\lambda g.\uparrow\{\lambda_\perp\, x.(f \bigtriangledown g)x\}\}$	dis
$\lambda_\perp\, x.E$	$\hat{=}$	$\lambda x.\mathit{if}\ x = \perp\ \mathit{then}\ \uparrow\{\perp\}\ \mathit{else}\ E$	notational convenience

Figure 9.5: Types and meaning of some constants

9.2.5 Constant Expressions

Each of the standard constants also has appropriate semantics, and a typescheme. Some examples are given in figure 9.5. The functions used on the right hand sides of the equations are well-known, except maybe $\bigtriangledown$, which can be pronounced 'dis', defined by: $(f \bigtriangledown g)(\mathit{Inl}\ a) \hat{=} f\ a$ and $(f \bigtriangledown g)(\mathit{Inr}\ b) \hat{=} g\ b$.

9.2.6 Example of Denotational Semantics of an Expression

Since there are a lot of sets within functions within sets in the denotational semantics we are proposing here, it may be useful to consider an example. We'll work out the semantics of $3 + 4$. We treat $+$ as a curried function applied to two arguments. The calculation will use some of the lemmas about upward closed sets presented earlier.

$$\begin{aligned} & \mathcal{E}[\![3+4]\!]\rho \\ = \quad & \text{app.} \\ & \{y \mid f \leftarrow \mathcal{E}[\![3+]\!]\rho, x \leftarrow \mathcal{E}[\![4]\!]\rho, y \leftarrow f\ x\} \end{aligned}$$

Now $\mathcal{E}[\![4]\!]\rho = \uparrow\{\mathcal{C}[\![4]\!]\} = \uparrow\{4\}$ is quite simple, and similarly, $\mathcal{E}[\![3]\!]\rho = \uparrow\{3\}$. The partial application $3+$ has semantics:

$\mathcal{E}[\![\sqcap x : T.E]\!]\rho$	$\hat{=}$	$\bigcup\{\mathcal{E}[\![E]\!]\rho[x \mapsto v] \mid v \in \mathcal{T}[\![T]\!]\}$	generalised choice
$\mathcal{E}[\![\bot_T]\!]\rho$	$=$	$\uparrow\{\bot\}$	undefined
$\mathcal{E}[\![\top_T]\!]\rho$	$=$	$\{\}$	miracle
$\mathcal{E}[\![E \sqcap F]\!]\rho$	$\hat{=}$	$\mathcal{E}[\![E]\!]\rho \cup \mathcal{E}[\![F]\!]\rho$	choice
$\mathcal{E}[\![\textbf{if } E \textbf{ fi}]\!]\rho$	$\hat{=}$	*if* $\mathcal{E}[\![E]\!]\rho = \{\}$ *then* $\uparrow\{\bot\}$ *else* $\mathcal{E}[\![E]\!]\rho$	miracle buster
$\mathcal{E}[\![G \rightarrow E]\!]\rho$	$\hat{=}$	*if* $\mathcal{E}[\![G]\!]\rho = \{true\}$ *then* $\mathcal{E}[\![E]\!]\rho$ *else* $\{\}$	guards
$\mathcal{E}[\![A \succ E]\!]\rho$	$\hat{=}$	*if* $\mathcal{E}[\![A]\!]\rho = \{true\}$ *then* $\mathcal{E}[\![E]\!]\rho$ *else* $\uparrow\{\bot\}$	assertions

Figure 9.6: Meaning of specificational expressions

$\mathcal{E}[\![3+]\!]\rho$

$=$ app.

$\{y \mid f \leftarrow \mathcal{E}[\![+]\!]\rho, x \leftarrow \mathcal{E}[\![3]\!]\rho, y \leftarrow f\ x\}$

$=$ constants

$\{y \mid f \leftarrow \uparrow\{\mathcal{C}[\![+]\!]\}, x \leftarrow \uparrow\{\mathcal{C}[\![3]\!]\}, y \leftarrow f\ x\}$

$=$ lemmas 44 and 45 remove both occurrences of $\uparrow$

$\{y \mid f \leftarrow \{\mathcal{C}[\![+]\!]\}, x \leftarrow \{\mathcal{C}[\![3]\!]\}, y \leftarrow f\ x\}$

$=$ def. of the constants

$\{y \mid f \leftarrow \{\lambda_\bot\, a.\uparrow\{\lambda_\bot\, b.\{a+b\}\}\}, x \leftarrow \{3\}, y \leftarrow f\ x\}$

$=$ drawing from singleton sets

$\{y \mid y \leftarrow \lambda_\bot\, a.\uparrow\{\lambda_\bot\, b.\{a+b\}\}\ 3\}$

$=$ $3 \neq \bot$, set comprehension

$\uparrow\{\lambda_\bot\, b.\{3+b\}\}$.

We return to the enclosing calculation:

$\{y \mid f \leftarrow \mathcal{E}[\![3+]\!]\rho, x \leftarrow \mathcal{E}[\![4]\!]\rho, y \leftarrow f\ x\}$

$=$ previous calculations

$\{y \mid f \leftarrow \uparrow\{\lambda_\bot\, b.\{3+b\}\}, x \leftarrow \uparrow\{4\}, y \leftarrow f\ x\}$

$=$ lemmas 44 and 45 remove the $\uparrow$

$\{y \mid f \leftarrow \{\lambda_\bot\, b.\{3+b\}\}, x \leftarrow \{4\}, y \leftarrow f\ x\}$

$=$ drawing from singleton sets

$\{y \mid y \leftarrow \lambda_\bot\, b.\{3+b\}\ 4\}$

$=$ $4 \neq \bot$, set comprehension

$\{3+4\}$

$=$ arithmetic

$\{7\}$.

We arrive at the expected singleton set!

9.2.7 Specificational Expressions

The meaning of specificational expressions is given in figure 9.6. The meaning of the generalised choice $\sqcap x : T.E$ in environment ρ is union of the meanings of E, in every environment $\rho[x \mapsto v]$, where $v \in \mathcal{T}[\![T]\!]\iota$.

The meaning of the undefined and miraculous expressions of type T are $\uparrow\{\bot\}$ (which is $\mathcal{T}[\![T]\!]\iota$) and $\{\}$ respectively.

$\mathcal{E}[\![T]\!]\rho$	$\hat{=}$	$\uparrow\{\mathcal{T}[\![T]\!]\iota\}$	type as set
$\mathcal{E}[\![\{E_1,...,E_n\}]\!]\rho$	$\hat{=}$	$\uparrow\{close\{v_1,...,v_n\} \mid v_i \leftarrow \mathcal{E}[\![E_i]\!]\rho\}$	enumerated sets
$*$	:	$(a \to b) \to \mathbb{P}\,a \to \mathbb{P}\,b$	mapping
$\mathcal{C}[\![*]\!]$	$\hat{=}$	$\lambda f.\uparrow\{\lambda s.\uparrow\{close(select^* s) \mid \forall v \in s.select\ v \in f\ v\}\}$	
$\bigcup$	:	$\mathbb{P}(\mathbb{P}\,a) \to \mathbb{P}\,a$	big union
$\mathcal{C}[\![\bigcup]\!]$	$\hat{=}$	$\lambda s.\uparrow\{close(\bigcup s)\}$	
$\cup$	:	$\mathbb{P}\,a \to \mathbb{P}\,a \to \mathbb{P}\,a$	binary union
$\mathcal{C}[\![\cup]\!]$	$\hat{=}$	$\lambda s.\uparrow\{\lambda t.\uparrow\{close(s \cup t)\}\}$	
$\{E \mid P \leftarrow F\}$	$\hat{=}$	$(\lambda P.E)^* F$	set comprehensions
$\{E \mid B\}$	$\hat{=}$	$\textbf{if}\ B\ \textbf{then}\ \{E\}\ \textbf{else}\ \{\}$	
$\{E \mid Q_1,...,Q_n\}$	$\hat{=}$	$\bigcup\{\{E \mid Q_2,...,Q_n\} \mid Q_1\}$	

Figure 9.7: Meaning of the set expressions

Choice between expressions is interpreted as set union, which preserves upward closedness. The meaning of the miracle buster is simply defined by cases. Guards and assertions are given meaning using conditional expressions testing their first argument for truth.

9.2.8 Set Expressions

The meaning of set expressions is given in figure 9.7. Any type T can be used as a set, in which case its meaning is simply the upward closure of the singleton set containing the interpretation of the type. The interpretation of any type is, as required here, a closed set, since any upward closed set is also closed. The meaning of an enumerated set expression is the upward closure of the set of all closed sets constructed by drawing one element from the interpretation of each enumerated expression.

The usual set-manipulating functions are available as constants. However, their definitions are not simply nondeterministic versions of the usual: they must also preserve closedness of the sets. We give mapping, big union, and binary union as examples. The meaning of big union and binary union is straightforward. The symbols $\bigcup$ and $\cup$ in the bodies of the definitions refer to normal mathematical big union and binary union.

The meaning of mapping a function expression F over a set expression S, that is F^*S is constructed as follows. For each outcome f of F, and for each outcome s of S, and for each element v of s, we *select* an element of $f\ v$, gather them into a set, and close that set. We do this for all possible selecting functions, and collect the resulting closed sets into a set, which is finally upward-closed. If for example s is empty, this set will contain only the empty set.

The meaning of set comprehensions is given in terms of mapping, conditional, and big union. The definitions are repeated in the figure 9.7.

9.2.9 Quantifier Expressions

Figure 9.8 gives the meaning of the universal and existential quantifiers. The meaning of $\forall x : T.E$ is constructed as follows. For each outcome of type T, interpret E with

$$\begin{aligned}
\mathcal{E}[\![\forall\, x : T.E]\!]\rho &\mathrel{\hat{=}} Min_{<_\forall}\{\mathcal{E}[\![E]\!]\rho[x \mapsto v] \mid v \in \mathcal{T}[\![T]\!]\iota\}\\
\mathcal{E}[\![\exists\, x : T.E]\!]\rho &\mathrel{\hat{=}} Min_{<_\exists}\{\mathcal{E}[\![E]\!]\rho[x \mapsto v] \mid v \in \mathcal{T}[\![T]\!]\iota\}
\end{aligned}$$

$$\{\} <_\forall \{\mathit{false}\} <_\forall \{\mathit{true}, \mathit{false}, \bot\} <_\forall \{\mathit{true}, \mathit{false}\} <_\forall \{\mathit{true}\}$$
$$\{\} <_\exists \{\mathit{true}\} <_\exists \{\mathit{true}, \mathit{false}, \bot\} <_\exists \{\mathit{true}, \mathit{false}\} <_\exists \{\mathit{false}\}$$

Figure 9.8: Meaning of the quantifications

x bound to that outcome, to obtain an upward closed set. Order the obtained upward closed sets by transitive $<_\forall$. The least set is the meaning of $\forall\, x : T.E$.

The meaning of $\exists\, x : T.E$ is constructed similarly, except that we take the least set under the transitive order $<_\exists$.

9.2.10 Semantics of State Transformers

A function type delivering a pair is given semantics like this:

$$\mathcal{T}[\![s \to a \times s]\!]\iota \mathrel{\hat{=}} (\mathcal{T}[\![s]\!]\iota \to \mathbb{U}(\mathcal{T}[\![a]\!]\iota \times \mathcal{T}[\![s]\!]\iota)_\bot)_\bot,$$

that is, such a function is either $\bot$ or a mapping from a value to a set of things that are undefined, or pairs. The interpretation of the state transformer type is analogous:

$$\mathcal{T}[\![ST\ s\ a]\!]\iota \mathrel{\hat{=}} (\mathit{State} \to \mathbb{U}(\mathcal{T}[\![a]\!]\iota \times \mathit{State})_\bot)_\bot.$$

A state transformer is undefined, or a mapping from a state to a set of things that are undefined, or pairs. Above *State* is an abbreviation:

$$\mathit{State} \mathrel{\hat{=}} (\mathbb{N} \to D_\bot)_\bot$$

A state is undefined, or a mapping from a natural to undefined or an outcome (all outcomes are represented in the universe D). The reference types are interpreted as naturals or an undefined error reference:

$$\mathcal{T}[\![\mathit{Ref}\ s\ a]\!]\iota \mathrel{\hat{=}} \mathbb{N}_\bot.$$

In order to accommodate representations for the types $ST\ s\ a$ and $\mathit{Ref}\ s\ a$ in the universe D, the recursive domain equation specifying D must be adjusted. We add "$\oplus(\mathbb{N} \to D_\bot)_\bot$" to the right hand side. Representations of state transformers and references are already available, in the form of representations of functions returning pairs, and the representation of the type $\mathbb{N}$. We add a representation of states. With the proviso about continuous versus monotone functions as discussed earlier, there is no problem about doing this.

In the definitions of $\mathcal{T}[\![ST\ s\ a]\!]\iota$ and $\mathcal{T}[\![\mathit{Ref}\ s\ a]\!]\iota$ the type s, respectively the types s and a, do not appear on the right hand sides. After they have played their role during type-checking, they have become meaningless and don't affect the semantics. Intuitively, for references this means: pointers are typed, but all pointers are represented in the same way.

With these models, states are slightly different from other functions: the body of the state-function is determined, whereas normal functions can have nondetermined bodies. That seems a reasonable choice since the only way to store a value in the state is by using the primitive state transformer *put* , and that involves binding, which extracts a single value from an expression.

However, like normal functions, the state-function is a partial function. The partial function is modelled by a total function that maps some arguments to $Lo\bot$. We refer to the set of those references for which this is not the case as the ***domain*** of the state. We define $\operatorname{dom}\sigma \mathrel{\hat{=}} \{n \in \mathbb{N} \mid \sigma\ n \neq Lo\bot\}$. We also assume that the domain of any state is finite, but this is not captured by the semantics. It is possible to distinguish a reference that is not allocated and one that is allocated, but stores the undefined value. The first will be mapped to $Lo\bot$ and the second to $Hi\bot$. The order on the interpretation of the state transformer type follows by construction from the rules for distributing order over a set.

Now let's examine the semantics of the state transformers. Operationally between the executions of two consecutive state transformers, the state can be changed nondeterministically by two occurrences. There could be an intervening garbage collection, examining which references are still accessible, and freeing the others. Or a concurrent state thread does a state-operation[2].

So far mathematically the states of two separate threads in a program are just that – separate. Yet operationally both state threads would be implemented using the same underlying state of the machine. So from the point of view of one thread we must model the changes that another thread may do by nondeterministic changes. Launchbury [LJ94] addresses these changes. Neither of these two intervening state changes is visible to any program. They leave invariant that part of the state that is accessible from the surrounding program.

Semantically the accessible references can be derived from the current environment ρ and the current state σ. Define $acc\ \rho\ \sigma$ to be the set of accessible (proper) references using the current environment ρ and current state σ. We have not defined $acc\ \rho\ \sigma$ formally, and suppose such a definition will be very long, but for illustration, here are some examples. If ρ maps the variable x to a reference r, then $r \in acc\ \rho\ \sigma$. Furthermore, if ρ maps f to a function that maps 3 to a set of references R, then $R \subseteq acc\ \rho\ \sigma$. As third example, if $r \in acc\ \rho\ \sigma$, and $\sigma\ r = \mathbf{Hi}\ r'$ where r' is another reference, then $r' \in acc\ \rho\ \sigma$. The formal definition of $acc\ \rho\ \sigma$ should list all the one-step accessible references, and define $acc\ \rho\ \sigma$ as the union of all references accessible in finitely many steps.

A state σ may be changed nondeterministically to state τ by a garbage collection or a foreign state operation as long as the states agree domain-restricted to $acc\ \rho\ \sigma$. Define $gc\ \rho\ \sigma$ to be the least such τ, that is $gc\ \rho\ \sigma \mathrel{\hat{=}} (acc\ \rho\ \sigma) \lhd \sigma$, where $\lhd$ denotes domain restriction. We get the lemma that $gc\ \rho\ ((acc\ \rho\ \sigma) \lhd \sigma) = gc\ \rho\ \sigma$, that is, garbage collection loses no accessible references, as expected.

[2]There is no theoretic need to accommodate this, but we'd like to keep some connection between the semantics and the implementation.

So at many points in the program the current state σ could be changed to any of $\{\tau \mid gc\ \rho\ \tau = gc\ \rho\ \sigma\}$. This set is a semantic way of expressing this nondeterministic change. The question is just at what points in the semantics this change should be inserted.

One thing to keep in mind is that this nondeterministic state transformation depends on the current environment ρ. However, the meaning of a constant k (such as the primitive state transformers) is $\mathcal{C}[\![k]\!]$ – it does not depend on the environment ρ. Building this nondeterministic state change into the semantics of the primitive state transformers or semicolon, they become 'environment-dependent constants'. In essence they remain constant, but the exact range of their nondeterminacy depends on the environment.

For example, the meaning of the trivial state combinator *return* cannot be given as

$$\mathcal{C}[\![\mathit{return}\,]\!] \ \hat{=}\ \lambda\, a.\!\uparrow\!\{\lambda\,\sigma.\!\uparrow\!\{(a,\sigma)\}\},$$

if we assume that the result state may be changed nondeterministically – with the range depending on the environment ρ. Rather, we'll use the meaning function $\mathcal{E}[\![\ ]\!]\rho$ for the primitive state combinators, and then we have

$$\mathcal{E}[\![\mathit{return}\,]\!]\rho \ \hat{=}\ \uparrow\!\{\lambda\, a.\!\uparrow\!\{\lambda\,\sigma.\!\uparrow\!\{(a,\tau) \mid gc\ \rho\ \sigma = gc\ \rho\ \tau\}\}\}.$$

Launchbury [LJ94] treats the problem of foreign state operations. He defines an 'equivalence modulo foreign state operations' on state transformers, and rewrites the semantics to apply nondeterministic foreign state changes in the middle of the semicolon operator. That is, $M;\ \lambda\, x.K$ is roughly: execute M on the initial state, bind the result to x, change the state nondeterministically, and then execute K.

For our purpose (dealing with foreign state operations *and* garbage collection) that is not sufficient. Just consider the equivalent pair

$$\mathit{new}\ 13;\ \lambda\, v.\mathit{new}\ 3 \ \equiv\ \mathit{new}\ 3.$$

The nondeterministic state change to make them equal cannot come at the semicolon: there's no garbage to collect. It cannot be before the execution of the second *new* on the RHS: v is in the environment, and so still considered accessible. Only after the end of the second argument of semicolon is it clear that the new reference v is garbage, since it goes out of scope.

Here are semantics that make the above examples of observably equivalent state transformers semantically equivalent: After each primitive state transformer, there's a nondeterministic state change preserving the accessible references. Further, the semantics of semicolon are modified: Executing $M;\ \lambda\, x.K$ is roughly: execute M on the initial state, binding the result to x. Then execute K, and finally change the state nondeterministically, preserving the accessible references. This makes all four primitive state transformers and semicolon environment-dependent constants. We'll just use the expression meaning function $\mathcal{E}[\![\ ]\!]\rho$.

Their semantics are given in figure 9.9.

The meaning of **run**E is the upward closed set obtained by applying the state-

$$\begin{array}{lcl}
\mathit{return} & : & a \rightarrow ST\ s\ a \\
\mathit{break} & : & a \rightarrow ST\ s\ a \\
\mathit{new} & : & a \rightarrow ST\ s\ (\mathit{Ref}\ s\ a) \\
\mathit{get} & : & \mathit{Ref}\ s\ a \rightarrow ST\ s\ a \\
\mathit{put} & : & \mathit{Ref}\ s\ a \rightarrow a\ ST\ s\ a \\
; & : & ST\ s\ a \rightarrow (a \rightarrow ST\ s\ b) \rightarrow ST\ s\ b
\end{array}$$

$$\begin{array}{lcl}
\mathcal{E}[\![\mathbf{run} E]\!]\rho & \hat{=} & \uparrow\{x \mid k \leftarrow \mathcal{E}[\![E]\!]\rho, \sigma \leftarrow \mathit{State} \setminus \{\bot\}, (x, \sigma') \leftarrow k\ \sigma\} \\
\mathcal{E}[\![\mathit{return}\ ]\!]\rho & \hat{=} & \uparrow\{\lambda\, a.\uparrow\{\mathit{return}_\rho\ a\}\} \\
\mathcal{E}[\![\mathit{break}]\!]\rho & \hat{=} & \uparrow\{\lambda\, a.\uparrow\{\lambda\, \sigma.\uparrow\{(a, \bot_{\mathit{State}})\}\}\} \\
\mathcal{E}[\![\mathit{new}\ ]\!]\rho & \hat{=} & \uparrow\{\lambda\, a.\uparrow\{\lambda_\bot\, \sigma.\mathit{new}_\rho\ a\ \sigma\}\} \\
\mathcal{E}[\![\mathit{get}\ ]\!]\rho & \hat{=} & \uparrow\{\lambda_\bot\, v.\uparrow\{\lambda_\bot\, \sigma.\mathit{get}_\rho\ a\ \sigma\}\} \\
\mathcal{E}[\![\mathit{put}\ ]\!]\rho & \hat{=} & \uparrow\{\lambda_\bot\, v.\uparrow\{\lambda\, a.\uparrow\{\lambda_\bot\, \sigma.\mathit{put}_\rho\ v\ a\ \sigma\}\}\} \\
\mathcal{E}[\![;\ ]\!]\rho & \hat{=} & \uparrow\{\lambda\, k.\uparrow\{\lambda\, m.\uparrow\{\mathit{semi}_\rho\ k\ m\}\}\}
\end{array}$$

$$\begin{array}{lcl}
\mathit{return}_\rho\ a\ \sigma & \hat{=} & \uparrow\{(a, \tau) \mid \mathit{gc}\ \rho\ \sigma = \mathit{gc}\ \rho\ \tau\} \\
\mathit{new}_\rho\ a\ \sigma & \hat{=} & \uparrow\{(v, \tau) \mid v \leftarrow \mathbb{N} \setminus \mathrm{dom}\, \sigma, \mathit{gc}\ \rho\ \tau = \mathit{gc}\ \rho\ (\sigma[v \mapsto x])\} \\
\mathit{get}_\rho\ v\ \sigma & \hat{=} & \mathit{if}\ v \not\in \mathrm{dom}\, \sigma\ \mathit{then}\ \uparrow\{\bot\}\ \mathit{else}\ \uparrow\{(\sigma\ v, \tau) \mid \mathit{gc}\ \rho\ \tau = \mathit{gc}\ \rho\ \sigma\} \\
\mathit{put}_\rho\ v\ a\ \sigma & \hat{=} & \mathit{if}\ v \not\in \mathrm{dom}\, \sigma\ \mathit{then}\ \uparrow\{\bot\}\ \mathit{else}\ \uparrow\{((), \tau) \mid \mathit{gc}\ \rho\ \tau = \mathit{gc}\ \rho\ (\sigma[v \mapsto x])\} \\
\mathit{semi}_\rho\ k\ m\ \sigma & \hat{=} & \uparrow\{(b, \tau) \mid (a, \sigma') \leftarrow k\ \sigma, l \leftarrow m\ a, (b, \sigma'') \leftarrow l\ \sigma', \mathit{gc}\ \rho\ \sigma'' = \mathit{gc}\ \rho\ \tau\}
\end{array}$$

$$\begin{array}{lcl}
\lambda_\bot\, x.E & \hat{=} & \lambda\, x.\mathit{if}\ x = \bot\ \mathit{then}\ \uparrow\{\bot\}\ \mathit{else}\ E
\end{array}$$

Figure 9.9: Types and meaning of the state combinators

transformer outcomes of E to every defined state, and gathering the first elements of the resulting pairs. If an undefined pair $\bot$ results, the whole expression is undefined. The trivial state transformer *return a* changes the inaccessible part of the state while returning a. The state transformer *break a* simply returns a while destroying the state. The state transformer *new a* fails if applied to an undefined state. Otherwise, it allocates a free reference, stores a in it, and changes the inaccessible part of the state nondeterministically. The state transformer *get v* fails if v is the undefined reference, or if the state is undefined, or if r is not in the domain of the state. Otherwise, it changes the inaccessible part of the state nondeterministically and returns the value stored at v. The state transformer *put v a* fails is the reference v is undefined, or if the state in undefined, or if the reference v is not in the domain of the state. Otherwise, it overwrites the state with $v \mapsto a$, and changes the inaccessible part of the state nondeterministically.

In summary, nondeterminacy is useful in giving semantics to state transformers in three ways:

1. The initial state of **run**E is arbitrary.
2. Allocation returns an arbitrary new reference.
3. The invisible state changes resulting from garbage collections and foreign state operations are captured by nondeterminacy.

9.3 Sketches of some Soundness Proofs

This section presents sketches of soundness proofs of some of the refinement laws with respect to the denotational semantics given in this chapter.

Refinement is a partial order with minimum $\bot$ and maximum $\top$. This is immediate since the interpretation of refinement, $\supseteq$, is a partial order with minimum $\uparrow\{\bot\}$ and maximum $\{\}$, which are the interpretations of $\bot$ and $\top$.

Equivalence is mutual refinement. Immediate, since set-equality, the interpretation of equivalence, is mutual set inclusion, which is the interpretation of refinement.

Equivalence is a congruence, that is, from $E \equiv F$ we conclude $G[E] \equiv G[F]$. Proof: The semantics of $G[E]$ depend on E only via $\mathcal{E}[\![E]\!]\rho$, which is equal to $\mathcal{E}[\![F]\!]\rho$ by assumption. Therefore $G[E] \equiv G[F]$.

Choice is greatest lower bound with respect to refinement. Immediate, since set union, the interpretation of choice, is the greatest lower bound with respect to set inclusion $\supseteq$, the interpretation of refinement.

Function application preserves miracles and distributes over choice in both arguments. That is, $E\ \top \equiv \top$ and $E\ (F \sqcap G) \equiv E\ F \sqcap E\ G$, and similarly for miracles and choices as functions. Furthermore, function application is monotone in both arguments. Proof. The semantics of an application $E\ X$ are $\{y \mid f \leftarrow \mathcal{E}[\![E]\!]\rho, x \leftarrow \mathcal{E}[\![X]\!]\rho, y \leftarrow f\ x\}$. It is easy to see that if either f or x are drawn from the empty set (the interpretation of $\top$), then the whole set will be empty. Furthermore, if $\mathcal{E}[\![E]\!]\rho$ or $\mathcal{E}[\![X]\!]\rho$ are unions, these unions can be distributed to the outside since unions preserve upward closed sets. Furthermore, $E\ X$ is refinement-monotone in E and X, since the set interpreting it is $\subseteq$-monotone in $\mathcal{E}[\![E]\!]\rho$ and $\mathcal{E}[\![X]\!]\rho$.

Lambda abstractions are monotone in their bodies, that is, from $E \sqsubseteq F$ we conclude $\lambda x.E \sqsubseteq \lambda x.F$. Proof. The semantics of $\lambda x.E$ is $\uparrow\{\lambda v.\mathcal{E}[\![E]\!]\rho[x \mapsto v]\}$. Upward closing is inclusion-monotone, and we have $E \sqsubseteq F$ by assumption in the environment ρ, therefore it is also true in environment $\rho[x \mapsto v]$.

The β transformation is $E[A/x] \equiv (\lambda x.E)A$ if A is determined. Proof. A is determined means semantically that there is a semantic value a such that $\mathcal{E}[\![A]\!]\rho = \uparrow\{a\}$. In the proof we'll use the function F that gives the outcomes of $E[A/x]$ in terms of the outcomes of A. It has the property that $F\ (\uparrow\{v\}) = \mathcal{E}[\![E]\!]\rho[x \mapsto v]$.

$$
\begin{array}{ll}
 & \mathcal{E}[\![E[A/x]]\!]\rho \\
= & \qquad \text{for some set-to-set function } F \\
 & F\ \mathcal{E}[\![A]\!]\rho \\
= & \qquad A \text{ det.} \\
 & F\ (\uparrow\{a\}) \\
= & \qquad \text{property of } F \\
 & \mathcal{E}[\![E]\!]\rho[x \mapsto a] \\
= & \qquad \text{drawing from singleton sets} \\
 & \{y \mid f \leftarrow \{\lambda v.\mathcal{E}[\![E]\!]\rho[x \mapsto v]\}, y \leftarrow \{a\}, y \leftarrow f\ v\} \\
= & \qquad \text{lemmas 44 and 45, using } E \text{ is monotone in } x \\
 & \{y \mid f \leftarrow \uparrow\{\lambda v.\mathcal{E}[\![E]\!]\rho[x \mapsto v]\}, y \leftarrow \uparrow\{a\}, y \leftarrow f\ v\} \\
= & \qquad \text{semantics of application and abstraction, and } \mathcal{E}[\![A]\!]\rho = \uparrow\{a\} \\
 & \mathcal{E}[\![(\lambda x.E)A]\!].
\end{array}
$$

Axiom proven sound.

For arbitrary expression A, the β transformation is a refinement: $E[A/x] \sqsubseteq (\lambda x.E)A$. Informal proof. For miraculous A, the law is trivially true. A feasible A can be expressed as a choice of determined expressions. For each of these determined expressions A_i, we apply the previous law to get $E[A_i/x] \equiv (\lambda x.E)A_i$. Since E is monotone is x, we have $E[A/x]$ is refined by a choice over the left hand sides, and, since application distributes over choice, a choice over the right hand sides is equivalent to $(\lambda x.E)A$.

9.3.1 Related Work on Powerdomains

Powerdomain constructors in denotational semantics correspond to the powerset constructor in set theory. There are (at least) three groups of powerdomain constructors. They are based on the Smyth order, the Hoare order, and the Egli-Milner order. Each group is useful for modelling different things. Powerdomains based on the Smyth order are useful for modelling demonic nondeterminacy, powerdomains based on the Egli-Milner order (e.g. the Plotkin powerdomain) are useful for modelling erratic nondeterminacy, or for modelling formal sets, whereas powerdomains based on the Hoare order are useful for modelling angelic nondeterminacy.

In this thesis, we use a Smyth-style powerdomain to model demonic nondeterminacy and a powerdomain based on the Egli-Milner order to model formal sets, that is, the sets of the specification language. In the following, we give a brief overview of the work about powerdomains by Plotkin and others, and then relate their work to ours.

Powerdomains were first proposed in [Plo76]. That paper develops the "Plotkin powerdomain", based on the Egli-Milner order. The aim is to model computational nondeterminacy. The simplest candidate construction $\mathbb{P} D$ for domain D is rejected, because it lets "nondeterminism mask nontermination". The kind of nondeterminacy Plotkin aims at is *computational*: Therefore he excludes the empty set and certain infinite sets. He excludes the empty set because a computation may have the outcome nontermination, but no computation has no outcome at all. He excludes infinite sets modelling unbounded nondeterminacy, since such sets are not computable, see for example [Dij82]. In Plotkin's terminology, the allowed sets are *finitely generable.*

Unfortunately, even on this restricted range, the Egli-Milner order has a number of defects. Firstly, if the base domain is not flat, then the Egli-Milner order is not antisymmetric. For example, take $(\mathbb{N}, \leq)$ as the base order. Then we have $\{0,2\}\mathcal{P}{\leq}\{0,1,2\}$ and $\{0,1,2\}\mathcal{P}{\leq}\{0,2\}$ and still the sets are distinct. Furthermore, we desire that singleton formation and the union operator are continuous. Continuity of these operations and antisymmetry are achieved by considering *closed* sets only, that is, sets that contain the limits of all chains they contain. The non-empty finitely-generable closed subsets of a cpo, ordered by the Egli-Milner order, are as desired a cpo with continuous singleton formation and union operation.

[Smy78] follows on from Plotkin's paper. Here too the aim is to find a powerdomain construction that can be used to model computational nondeterminacy, so Smyth also rules out the empty set and sets representing unbounded nondeterminacy. However, adopting the attitude that "a computation which *may* fail to yield any result is as good as worthless", that is, considering demonic nondeterminacy rather than erratic, he develops a simpler construction than Plotkin. It is now known as the "Smyth powerdomain".

The third of the classical powerdomains, the lower or Hoare powerdomain, is introduced in [Smy83]. It is useful for modelling angelic nondeterminacy. However, it is not relevant in the context of this thesis.

In more recent work, Heckmann [Hec90] examines the three classical powerdomain constructions to see whether they are are suitable for modelling formal sets — a motivation different from the original one, which was modelling nondeterminacy. He finds all three inappropriate. In particular, the Plotkin powerdomain does not include the empty set, but a representation of the formal empty set is definitely needed. If the empty set were added to the Plotkin powerdomain, there would be no least element anymore, because the empty set is incomparable to any other set, in particular $\{\bot\} \not\sqsubseteq \{\}$. Heckmann rejects putting the empty set artificially above $\{\bot\}$ or adding a new least element, "because the algebraic properties ... are messed up". Instead, he proposes the ***big set domain*** and the ***small set domain***, which are defined in terms of pairs containing an element of the Hoare and Smyth powerdomain each. The big set domain is isomorphic to the sandwich domain of [BDW88], and is bounded complete, whereas the small set domain is isomorphic to the mixed powerdomain with $\{\}$ of [Gun89, Gun90].

We use two powerdomains: one of the Smyth-kind to represent demonic nondeterminacy, and one based on the Egli-Milner order to model formal sets. However, since we are giving semantics to a specification language, our requirements are different.

To model demonic nondeterminacy, we use the powerdomain constructor $\mathbb{U}$. Given an expression E with outcomes in the partial order A, the meaning of E is an upward closed subset of A, that is, an element of $\mathbb{U}A = \{\uparrow S \mid S \subseteq A\}$. Upward closed sets are ordered by $\supseteq$, which, as detailed in theorem 40, is equivalent to ordering arbitrary subsets of A by the Smyth order. For any partial order A, $\mathbb{U}A$ is a complete lattice, and therefore also a cpo.

This approach differs from the one in [Smy78] because our motivation differs. We are not giving semantics to a nondeterministic programming language, but to a specification language. In particular, the empty set does make sense as the model for infeasible expressions, and it introduces no problems. Furthermore, we do require sets representing unbounded nondeterminacy. Since $\mathbb{U}A$ is a complete lattice, fixpoints of monotone functions can always be found as the least upper bound (here: intersection) of the approximations. However, to guarantee that the fixpoint is more than simply the empty set, some extra work is required, as detailed earlier.

To model outcomes of a set-type, we use the powerdomain construction $(\mathbb{C}A)_\perp$ for cpo A. This is a variation of Plotkin's powerdomain. $\mathbb{C}A$ are the closed subsets of cpo A, including those that are not finitely generable. They are not computable, but they do represent useful specifications with unbounded nondeterminacy, such as $\sqcap.n : \mathbb{N}.n$. Adding the non-finitely-generable sets causes no difficulties.

However, we have also not excluded the empty set which is needed to represent the empty set of the specification language, and that causes a problem. The empty set is incomparable to any other set, including $\{\perp_A\}$, and therefore $\mathbb{C}A$ does not have a least element. Despite Heckmann's objections [Hec90], we simply add an artificial least element. He also does so in [Hec92]. Therefore, the meaning of an expression of type $\mathbb{P}\ T$ is an upward closed set (like for any expression of any type), containing elements of the domain $(\mathbb{C}A)_\perp$, where domain A is the interpretation of type T.

Chapter 10

Conclusion

The first section of this chapter summarises the technical work of the thesis. The second draws conclusions from the experience gained in calculating with imperative expressions to compare them with the usual way of writing imperative programs, namely by using assignments. The third and fourth sections compare the work of this thesis to that of Ward [War94] and Flynn [Fly97]. The fifth section sets some areas in which the present work could be extended and improved.

10.1 Summary

The aim of this thesis is to present a formalism for calculating programs, including imperative programs. The two aspects, easy calculation and capturing imperative programming, are not easily combined, because the first is based on timeless unchanging mathematical expressions and the second is based on specified changes to a state. Rather than have imperative variables with assignments which destroy referential transparency and thereby make calculations clumsy, we choose to preserve referential transparency and bring state into the language in referentially transparent unchanging variables. To ensure that the program only ever has one state – the state of the machine memory – the state manipulating operations are parcelled up in the state monad. All the available operations ensure that at any time only one state value is used, and therefore the operations can be implemented as performing in-place updates of the real state, the machine's memory.

However, during the derivation of a program, we needn't limit ourselves to the operations of the state monad. We are free to bind the state to a variable, and manipulate or duplicate it in any way we like. It doesn't matter whether in these intermediate stages between specification and program we use the state single-threadedly[1]: these intermediate stages only exist on paper. The specification language they are written in must be as expressive as possible. In the program resulting from the derivation, such freedom is forbidden. We take the view that variables that are bound to states are part of the specification language, but not of the programming sublanguage.

[1]that is, there are never two states 'at the same time'

They share this property with other language constructs, in particular the generalised choice and guards. Using generalised choice and guards, it is possible to write expressions that describe their outcome by a predicate, but don't give an algorithm to find it. They are typically the starting points of transformational program derivations.

The specification language is a very rich expression language, which can roughly be described as a combination of the algorithmic language constructs of a modern functional programming language like Haskell, with firstly, language constructs capturing equivalence, refinement, and propositional and predicate logic, secondly, descriptive nondeterministic language constructs from the refinement calculus, and thirdly, the state monad to capture imperative programming.

The language is typed and nondeterministic. The expressions of the language are related by equivalence ($\equiv$) and by refinement ($\sqsubseteq$). The intuition behind refinement is: Expression E is refined by F, written $E \sqsubseteq F$, if a customer asking for an implementation of E would be satisfied if given an implementation of F. The technical meaning on the other hand is a precise formulation of: For any possible outcome of F there exists a possible outcome of E that approximates it. Refining is a combination of increasing termination and decreasing nondeterminacy.

The two extreme expressions $\bot$ and $\top$ are the expression that need never terminate but may deliver any outcome if it does, and the expression that definitely terminates delivering exactly the right outcome the user desires. Clearly $\top$ is not implementable – a miracle – and therefore excluded from the programming language.

Our notions of equivalence and refinement capture correctness, but there are many aspects they don't capture. For example, they say nothing about whether the expressions are descriptive specifications or algorithmic programs. So if the very first step in a derivation – usually a generalised choice – is already determined, and we derive a terminating algorithmic program from it, this means that all intermediate steps are equivalence rather than proper refinements. This occurs in our derivation of a combinator graph reducer.

Furthermore, refinement also doesn't capture any time or space complexity of the program, or anything about programming style. If the program has nondeterminacy, there are no requirements about fairness of the nondeterminacy. Fairness may be added during the derivation (a correctness preserving step), but technically that step is an equivalence. This occurs in our partial derivation of a printer control system.

One question arising here is whether the aim of combining calculational ease and imperative programs was achieved. We tested our specification language on some example derivations. There were four small examples: a printer control system, a musical synthesiser, an electric clock, and a pattern matching algorithm. It is appropriate to formulate them imperatively because they perform some IO operations. The first two also store information in the state that will influence the future behaviour of the program. Already in writing these small programs, the clumsiness of using imperative operations becomes obvious, which discourages the programmer from using imperative expressions. While an advocate of functional programming may consider that to be an advantage, ideally the calculus should allow easy use of state even for algorithms that can be expressed without state.

In another triplet of example derivations, the line, circle, and sphere drawing algorithms, state is not necessarily needed until the IO operations of displaying the figures are reached. The programs can be written as stateless expressions producing lists of integer pairs, which are then fed to a state transformer performing the IO operation of displaying them as pixels on the screen. The algorithms are however imperative in style, since each algorithm initialises a handful of variables and then repeatedly modifies them, producing a pixel of the figure each time. The algorithms use state statically, that is, the number of allocated variables is constant.

In the case of these graphics algorithms, imperative programs were derived indirectly, that is, the algorithms are first expressed in terms of a state-less higher order combinator. Then, an imperative implementation is provided for that combinator, immediately giving imperative programs for the algorithms. A host of state-less programs can thus be made imperative, possibly by the compiler. The calculations with state are then confined to the derivation of the imperative implementations of the combinators, however, there the clumsiness has to be faced.

This combinator approach can be applied to programs constructed from certain combinators. The programs could be left in the form with the combinator, and then several implementations given, targeted at different machine architectures.

The graphics algorithms used state in a static way. Some algorithms on the other hand need *dynamic* use of state, that is, the number of allocated variables is not fixed. As an example, a combinator graph reducer has been derived. The algorithm is imperative, with the state being a directed graph. The graph is locally modified by the program until it reaches a trivial form. The modifications are overwriting of existing vertices and creation of new vertices. Each local modification of the graph is potentially accessible from any other part of the graph and therefore not really local.

In deriving this program, we found it necessary to bind the state explicitly to variables, and to use it in many logical and algorithmic expressions. Moreover, to keep the expressions manageable we then introduced a generous set of auxiliary functions on state values. Of course, in the final program only the state monad operations manipulate state.

Our approach to designing a specification language with axioms is fairly practical. Nevertheless, we provide denotational semantics based on domain theory. Since the specification language contains not-executable constructs, operational semantics based on reduction axioms would be inappropriate. The main aim for the semantics is to make precise three language features: nondeterminacy, recursion, and state. We have not proven soundness of every refinement axiom with respect to the semantics, but believe this is possible. We don't claim that the set of refinement axioms given is minimal. It could be the case that a particular axiom can be derived from the others. We have given axioms covering logic, refinement and choice, and the basic language constructs. To give a complete set of axioms, one would have to axiomatise arithmetic, set theory, data types, and any other primitive constant that is used. Instead of doing this, we rely on intuition about these algebras from mathematics being carried over to expression refinement. Some of these algebras bring theoretical problems with them, but we assume that none of these are disastrous for program derivation in practice.

In the denotational semantics, nondeterminacy is treated by sets. Each expression denotes the set of its possible outcomes. Refinement ($\sqsubseteq$) is modelled by the superset relation ($\supseteq$). The sets of outcomes may contain the fictional undefined outcome to represent nontermination. Unbounded nondeterminacy is possible, for example the meaning of $\sqcap n : \mathbb{N}.(n \not\equiv \bot) \to n$ is the set of proper outcomes $\{0, 1, 2, ...\}$, a set that does not contain nontermination. Our calculus is concerned with total correctness, that is, an expression that need not terminate is as bad as an expression that will not terminate.

That relation 'as bad as' is captured by the Smyth order of sets [SS92], which gives rise to the weak powerdomain [Smy78], also known as the upper powerdomain. Rather than use the Smyth order as refinement and its antisymmetric closure as equivalence, we adopt the equivalent approach of letting each expression denote an upward closed set of outcomes. Then refinement is indeed simply supersetting, and equivalence is just set quality. The Smyth order is still used, namely in upward-closing the sets.

Recursion is given meaning as the least fixpoint of a monotone function. Upward closed sets form a lattice, so least fixpoints of monotone functions exist, and can be found by repeatedly applying the function to the least set.

State is modelled by a finite function from natural numbers to outcomes. Nondeterminacy is used three times in giving semantics to the state manipulating primitives: the initial state is arbitrary, allocation yields an arbitrary new reference, and between state transformers the inaccessible part of the state may change arbitrarily. This use of nondeterminacy leads to nondetermined expressions. However it does not lead to nondetermined whole programs, since the values of references and of states are not observable.

10.2 Comparison with the Imperative Refinement Calculus

We have presented a refinement calculus in which imperative programs can be specified and derived. That is also the purpose of [Bac80, Mor87, Mor88b]. They present 'the refinement calculus' which we will refer to here as the imperative refinement calculus (IRC). A comprehensive account is given in the textbook [Mor94]. In this section we will compare the two calculi, based on the experience of the derivations given earlier.

The crucial difference between the two is the way in which imperative programming is captured. In IRC assignments to program variables are used, whereas in our language the state monad is used.

The imperative refinement calculus as commonly presented has the following three weaknesses. The first is that as a consequence of using assignments there are two levels of reasoning: the level of expressions and the level of commands. One level of reasoning with one universally applicable set of laws would be simpler, and therefore preferable. In IRC at the level of expressions there is no nondeterminacy and no recursion. We reason about expressions by replacing subexpressions by equal subexpressions. Since the expression language in IRC is so simple there's usually not much reasoning to be done at expression level. At the level of commands 'specification statements' (nondeterministic assignments) give rise to nondeterminacy, and recursion gives rise to nontermination. Recursion is usually used in the form of tail-recursive loops.

We reason about commands by replacing one command in a program by an equivalent command or a command refining it. Here, equivalence of commands is equality as predicate transformers [Dij76], and refinement is lifted implication of predicate transformers. To derive a program (essentially a sequence of assignments with simple expressions) from a specification (usually one assignment or a nondeterministic assignment with relatively complicated expressions), we need to connect reasoning about expressions and reasoning about commands. This is done by translating assignments into substitutions: the semantics of the assignment $x := E$ is given in terms of a substitution, namely $wp\ (x := E)\ P \hat{=} P[E/x]$. Substitution also appears in the refinement laws. For example the law that introduces an assignment is: if $P \Rrightarrow Q[E/x]$ then $x : [P, Q] \sqsubseteq x := E$.

The second weakness of the imperative refinement calculus is that the poverty of the expression language forces one to express almost all calculations at the level of commands. Expressions are always determined and terminating. Therefore there can be no (possibly nondetermined) specification constructs in expressions. While that may be fine for programming, during program specification and derivation it is useful to have specification expressions. Recursion is also banned from expressions, since it introduces the possibility of undefined (nonterminating) expressions. Recursive functions in expressions are very convenient for programming, and exist in real imperative programming languages, but in IRC what could have been expressed by a recursively defined function must be put into a loop. One could add **if then else** expressions to IRC, since they preserve definedness and determinacy, but not the potentially undefined and nondetermined alternation expressions. One has to use an appropriate alternation command instead.

The third weakness is that there is no provision for dynamic use of state, that is, there are no pointers. There is no way to program with any linked data structure such as linked graphs. State comes only in the form of variables. They are declared by $[\![\mathbf{var}\ x \bullet C]\!]$ where C is a command that may use variable x. So the number of variables is fixed by the program text, that is, the use of state is static. Some papers [Bij89, But95] work around this weakness by simulating dynamic use of state. They add an explicit program variable, called a 'thought variable', that represents a mapping from references to values. Let's say the variable is σ of an appropriate array type. Then (in the notation of [Mor94]) allocation is the nondeterministic assignment $v, \sigma : [v \notin \mathrm{dom}\,\sigma, v \in \mathrm{dom}\,\sigma \wedge \mathrm{dom}\,\sigma_0 \lhd \sigma = \sigma_0]$. Dereferencing $(v\uparrow)$ is a shorthand for the application $\sigma\ v$, and updating $v\uparrow\ := E$ is the assignment $\sigma := \sigma[v \mapsto E]$. This only simulates dynamic use of state since (in a straightforward implementation) the amount of state is fixed: the whole of the array σ has to be allocated.

Apart from these weaknesses, the imperative refinement calculus has the advantages of being well understood and well known, and of being close to real imperative programming languages. It is quite reasonable to specify and derive a program in IRC, and then to transcribe it to Pascal or C.

A refinement calculus of expressions with the state monad addresses these three weaknesses of the imperative refinement calculus.

Firstly, there is only one level of reasoning. Everything is an expression. In particular, imperative programming is captured in the imperative expressions of the state monad. The main expression-forming constructs are function application and function abstraction. We reason by distributing function application over nondeterminacy, and then using the β rule $(\lambda x.E)\ F \equiv E[F/x]$ for determined F. Replacing subexpressions by equivalent subexpressions is equivalence, whether they are numbers, functions, or state transformers. Because everything is an expression we have higher-order functions, and in particular, higher-order imperative programming: state transformers can be bound to variables or encapsulated with **run**, and be subexpressions of otherwise not imperative expressions. That allows us to define higher-order functions corresponding to control structures like *for* loops in Pascal. Encapsulation allows an imperative implementation of a non-imperative specification.

Secondly, the expression language is rich: it includes specification constructs that introduce nondeterminacy as well as the algorithmic constructs of a higher-order functional programming language that bring recursion.

Thirdly, it is easy to make dynamic use of state. The references in the state monad are truly references, or pointers. The scope of the reference variable does not limit the 'lifetime' of the stored value. In $\mathbf{run}((\mathit{new}\ 3;\ \lambda v.\mathit{new}\ v);\ \lambda w.\mathit{get}\ w;\ \lambda u.\mathit{get}\ u)$ the scope of v is within the inner brackets, but the stored value 3 'survives' and is the value of the expression.

These features seem promising. However the price one has to pay for the state monad is that imperative programs become quite clumsy. The unchanging reference and the changing value stored at the reference are distinguished. Consequently every time we want to use the value stored at a reference, we have to dereference the reference, and bind the result to a new variable. This make programs longer and less clear, particularly if variable names are chosen badly. In some places in the examples we used the same variable name for a reference and the value stored there, prefixing the reference variable by 'v' to emphasise the connection. In addition this explicit dereferencing is clumsy in that the state has to be mentioned as an argument, even though it is not changed. For example, determining whether the value stored at a reference is even, is a function of *two* arguments, rather than just one, the reference. It can be expressed as a state-reader valued function $\lambda v.\lambda\sigma.\mathit{even}(\sigma\ v) : \mathit{Ref}\ s\ \mathbb{Z} \rightarrow \mathit{SR}\ s\ \mathbb{B}$, or even clumsier as the state-transformer valued function $\lambda v.\mathit{get}\ v;\ \lambda i.\mathit{return}\ (\mathit{even}\ i) : \mathit{Ref}\ s\ \mathbb{Z} \rightarrow \mathit{ST}\ s\ \mathbb{B}$.

This frequent explicit dereferencing means that state monad programs usually have lots of binding lambdas. Many calculation steps are applications of the monad laws, particularly the 'associative bind' law and the 'left return' law. These steps don't advance the derivation much conceptually; they just reshuffle variables and scopes. Although they can be made silently they are not completely trivial and it is not difficult to make errors in applying them.

On the whole, the clumsiness of the state monad discourages the use of state. That may be considered good in a masochistic way, if the derived programs are aimed at Haskell compilers that naturally don't compile a state transformer expression as directly as a Pascal compiler compiles a sequence of assignments. But a flexible language should allow easy imperative programming. Imperative programming is desirable because some algorithms rely on (dynamic) use of state for an acceptable time complexity. For others the most natural way to express them may be imperative. Admittedly what is natural depends on experience and opinion to some degree. However, most people would agree that Bresenham's line drawing algorithm for example is naturally expressed as repeated modification of updatable variables. But no one would prefer its state monad form to its assignment form! For algorithms with static use of state, assignments lead to clearer programs. In contrast, for algorithms with *dynamic* use of state a construction like the state monad seems unavoidable (witness the papers mentioned above treating dynamic state in IRC that make the state into a thought variable that is passed along, like the state in the state monad).

Another disadvantage of expression refinement with the state monad is that the language is not close to real imperative programming languages. It is close to Haskell with the state monad, but that's not the usual language or even the usual programming paradigm for most real programmers. One could specify and derive a program in expression refinement with the state monad and then transcribe it to Pascal, but that is not a good route. The transcription step is too big.

10.3 Comparison with Ward's Thesis

The work in this thesis in similar in aim to that of Nigel Ward [War94]. He presents a refinement calculus for a functional language, based on the ideas of the imperative refinement calculus. The language constructs he proposes are broadly similar to those presented here, except that he proposes language constructs with two kinds of nondeterminacy: demonic and angelic. For instance he proposes two infix binary choice operators, demonic choice $E[\!]F$, which is the same as our $E \sqcap F$, and angelic choice $E \diamond F$, which if required could be added to the language presented here as $E \sqcup F$. We prefer the notation $\sqcap, \sqcup$, together with $\sqsubseteq$ for refinement and $\bot, \top$ for what Ward calls *abort* and *magic*, to emphasise the lattice structure of the specification language. Accordingly, essential properties of demonic and angelic choice are

$$(X \sqsubseteq E) \wedge (X \sqsubseteq F) \equiv (X \sqsubseteq E \sqcap F)$$
$$(E \sqsubseteq X) \wedge (F \sqsubseteq X) \equiv (E \sqcup F \sqsubseteq X),$$

from which the demonic and angelic behaviour follows

$$E \sqcap \bot \equiv \bot$$
$$E \sqcup \bot \equiv E.$$

Both our languages are "truly nondeterministic" which means that λ abstractions with nondetermined bodies are themselves considered determined. Ward contrasts this with abstraction distributing over nondeterminacy as seemingly the only alternative, for example $\lambda x.1 \sqcap 2 \equiv \lambda x.1 \sqcap \lambda x.2$. He rightly identifies this proposed theorem as undesirable, for the left-hand-side can be refined to $\lambda x.\textbf{if } \textit{even } x \textbf{ then } 1 \textbf{ else } 2$, whereas the right-hand-side cannot.

However, there is another option. A λ abstraction with a nondetermined body could be seen as a nondetermined choice between all *functions*[2] that refine it. For example, the expression $\lambda x.1 \sqcap 2$ would be equivalent to $(\lambda x.1) \sqcap (\lambda x.2) \sqcap (\lambda x.\textbf{if } \textit{even } x \textbf{ then } 1 \textbf{ else } 2) \sqcap (\lambda x.(x \textit{ mod } 1) + 1) \sqcap \ldots$ A typical λ abstraction with nondetermined body would be equivalent to an unbounded choice between functions, and not itself determined. A

[2]in the usual mathematical sense of determined mappings

theorem along the lines of the following would relate the nondeterminacy inside the λ abstraction to nondeterminacy on the outside:

$$\lambda x.E \;\equiv\; \bigsqcap f.\forall x.(E \sqsubseteq f\ x) \rightarrow f,$$

where f ranges over (deterministic) functions of the appropriate type.

Like in Ward's and our systems

$$\lambda x.E \sqcap F \;\sqsubseteq\; \lambda x.E \sqcap \lambda x.F$$

would not in general be an equivalence. However, this third option is genuinely different. The difference becomes obvious when λ abstractions are themselves arguments to higher-order functions. Consider applying $f \hat{=} \lambda g.(g\ 1 = g\ 1)$ to $\lambda x.1 \sqcap 2$. The calculus of this thesis and Ward's yield

$$\begin{array}{ll} & f(\lambda x.1 \sqcap 2) \\ \equiv & \qquad \beta, \mathbf{det}(\lambda x.1 \sqcap 2) \\ & (\lambda x.1 \sqcap 2)1 = (\lambda x.1 \sqcap 2)1 \\ \equiv & \\ & (1 = 1) \sqcap (1 = 2) \sqcap (2 = 1) \sqcap (2 = 2) \\ \equiv & \\ & \mathit{True} \sqcap \mathit{False}, \end{array}$$

whereas with a suitable axiomatisation the third option would give

$$\begin{array}{ll} & f(\lambda x.1 \sqcap 2) \\ \equiv & \\ & (1 = 1) \sqcap (2 = 2) \\ \equiv & \\ & \mathit{True}. \end{array}$$

However, we have no axiomatisation of this understanding of λ abstractions with nondetermined bodies.

In contrast to our calculus, Ward's language is strict rather than lazy. This decision is somewhat a matter of taste. However, laziness allows more modularity in program construction, as has been often argued (for example in [Hug89, Hug90]) whereas strict semantics are more easily implementable. We feel that a specification language should aim for expressiveness rather than implementability. Of course the lazy/strict decision will affect the refinement laws. However, if required, a strict version of λ abstraction can be defined *within* our lazy language, e.g. as the abbreviation

$$\lambda_{\perp} x.E \;\hat{=}\; \lambda x.x \not\equiv \perp \succ E.$$

Ward briefly discusses a lazy version of his system.

Furthermore, Ward does not discuss imperative programming techniques in his system. Program development is aimed at implementations in functional languages. However, there seem to be no obstacles to adding the state monad to his language.

The main difference in approach between Ward's work and ours is that he presents the language as defined by the semantics rather than by axioms. He lists many "refinement laws" and advises the reader to generate new refinement laws by taking the dual of existing laws or proving new laws using the semantics. Therefore a programmer would be required to know the semantics Ward supplies — a substantial requirement, since the semantics he gives are *inspired* by the fairly straightforward predicate transformers, but a lot more complicated. Furthermore, Ward does not supply inference rules or other guidance about how to use the laws calculationally. In contrast we give a language defined by an inference system, that is, a list of axioms and a small number of inference rules. All laws are theorems — that is, provable from axioms or axioms themselves. We do provide semantics for our language to demonstrate the existence of non-trivial models. However, the user of the calculus is not required to know the semantics. In fact, one could give entirely different semantics to the language — as long as the same axioms are supported.

In particular, Ward does not treat improper boolean expressions in depth. Instead, he rather restricts boolean expressions occurring in laws to value expressions. However, not all non-value expressions are actually improper, and some value expressions are improper, namely non-terminating recursive expressions. In contrast, we axiomatise the behaviour of the logical connectives for all five possible truth values. In laws where restrictions have to be made, they are expressed in terms of **det**, **def**, **feas**, which are themselves fully axiomatised. We also characterise value expressions syntactically, but just as an easy guide to some of the determined (possibly undefined) expressions.

In giving semantics to his language, Ward has chosen a different route from the work presented here. He considers both angelic and demonic nondeterminacy, and therefore cannot map the specification language (which has a lattice structure) into sets. If he did so — like we do — demonic choice would be interpreted by set union, and angelic choice would be interpreted by intersection. In effect, the lattice $(\sqsubseteq, \sqcap, \sqcup, \bot, \top)$ would be interpreted by the lattice $(\supseteq, \cup, \cap, \mathit{Val}, \{\})$. However, this mapping is not appropriate since it does not reflect the intended meaning of angelic choice. We would have, for example, $\mathcal{E}[\![3 \sqcup 4]\!]\rho = \mathcal{E}[\![3]\!]\rho \cap \mathcal{E}[\![4]\!]\rho = \{3\} \cap \{4\} = \{\} = \mathcal{E}[\![\top]\!]\rho$.

It is useless to search for a different binary operation on sets to interpret $\sqcup$ since it would have to be a least upper bound with respect to $\supseteq$ to reflect that $\sqcup$ is the least upper bound with respect to $\sqsubseteq$, but $\cap$ is the unique operation that satisfies this. As [Hes90] says, angelic choice has nothing to do with intersection.

To avoid this shortcoming of set-based semantics, Ward maps the specification language into the lattice $\mathit{MBool} = \langle \mathbb{U}\mathit{Val} \rightarrow \mathbb{B} \rangle$, that is, the space of monotone functions from upward closed sets of values to $\{\mathit{tt}, \mathit{ff}\}$. This lattice does indeed model angelic and demonic nondeterminacy in the desired way. The semantics are informally: $\langle\langle E \rangle\rangle \rho S$ stands for "In environment ρ, expression E is guaranteed to yield an outcome in the upward closed set S."

Demonic and angelic choice are interpreted as follows.

$$\begin{aligned}
\langle\langle E \sqcap F \rangle\rangle \rho S &\mathrel{\hat{=}} \langle\langle E \rangle\rangle \rho S \wedge \langle\langle F \rangle\rangle \rho S \\
\langle\langle E \sqcup F \rangle\rangle \rho S &\mathrel{\hat{=}} \langle\langle E \rangle\rangle \rho S \vee \langle\langle F \rangle\rangle \rho S.
\end{aligned}$$

Both Ward's semantics and ours use upward closed sets for essentially the same reason, namely to model a nondeterministic total-correctness calculus with non-trivial semantic domains. That is, in some domains there are distinct semantic values u, v such that $u < v$. In fact, in our semantics such distinct semantic values exist even in simple domains such as that modelling integer expressions. However, undefined integer expressions could have been modelled without requiring an undefined semantic value. For functions, on the other hand, non-trivial semantic domains seem unavoidable.

Now refinement in a total-correctness calculus is a combination of

- narrowing demonic nondeterminacy
- widening angelic nondeterminacy
- improving termination.

Nondeterminacy suggests to use sets in the semantics, but if the expressions are actually determined, the sets can be singleton sets. Assume we have two determined expressions U and V such that $U \sqsubseteq V$ whose only possible outcomes are u and v such that $u < v$ respectively. Then in Ward's semantics

$$\begin{array}{ll} & U \sqsubseteq V \\ = & \quad \text{semantics of } \sqsubseteq \\ & \forall \rho, S. \langle\langle U \rangle\rangle \rho S \Rightarrow \langle\langle V \rangle\rangle \rho S, \end{array}$$

where S ranges over sets of semantic values. Now if S were not required to be upward-closed, we could choose $S = \{u\}$ to disprove the intended refinement. Therefore Ward requires the target sets of semantic values to be upward-closed. In our style of semantics, if $\mathcal{E}[\![E]\!]\rho$ were simply the set of possible outcomes of E in environment ρ, and as such, possibly not upward-closed, then we would have

$$\begin{array}{ll} & U \sqsubseteq V \\ = & \quad \text{semantics of } \sqsubseteq \\ & \forall \rho. \mathcal{E}[\![U]\!]\rho \supseteq \mathcal{E}[\![V]\!]\rho \\ = & \\ & \{u\} \supseteq \{v\} \\ = & \\ & \mathit{false}. \end{array}$$

which would not model the intended refinement. As mentioned in chapter 9, there are two equivalent solutions: Either we order the sets by the Smyth order instead of $\supseteq$, or we upward-close the sets $\mathcal{E}[\![E]\!]\rho$ and continue to use the simpler superset order.

10.4 Comparison with Flynn's Thesis

The recent thesis of Sharon Flynn [Fly97] broadly follows the same programme as this work. However, there are some differences in scope and emphasis, and in the design decisions taken. The main contribution of [Fly97] are given as, firstly, ways of structuring large specifications by combining abstractions with potentially infeasible bodies ("partial functions") and, secondly, a simple semantics. The denotational semantics she outlines is based on powerdomains and comparable to ours. This approach is indeed simpler than for instance Ward's semantics [War94] – but as discussed in section 10.3, it cannot model both demonic and angelic nondeterminacy at the same time.

In the following, we will outline and discuss the differences in scope and design decisions between [Fly97] and this thesis.

First of all, in Flynn's calculus function application and data type formation are strict. Our view on the advantages and disadvantages of laziness is discussed in subsection 10.5.1.

Secondly, the calculi differ in the types they provide. In Flynn's calculus, there are primitive types, functions and tuples, and *lists, bags,* and sets. The last three in particular facilitate model-oriented specification styles, as in Z or VDM. In contrast, in our calculus, there are primitive types, *sums,* tuples, functions, and sets, and *recursive and polymorphic* types. We don't provide lists and bags as type constructors, but they can be modelled in various ways using the given constructors, for instance as

$\mathbf{type}\ List\ a \mathrel{\hat{=}} \mu\, x.Empty\ \mid\ Cons\ a\ x$

$\mathbf{type}\ Bag\ a \mathrel{\hat{=}} a \rightarrow \mathbb{N}.$

We consider both recursive types and polymorphic types to be general useful tools in constructing various kinds of customised data structures and standard combinator functions.

Flynn has proposed and examined language constructs that are useful in making modular large specifications. They are biased choice $\overleftarrow{[]}$, function union $\dot{\cup}$, and biased function union $\overleftarrow{\cup}$. They can be expressed in terms of the core language[3] as follows:

$$\begin{aligned} E \overleftarrow{[]} F &\mathrel{\hat{=}} \mathbf{feas}E \rightarrow E \sqcap \neg\mathbf{feas}E \rightarrow F \\ \lambda x.E \,\dot{\cup}\, \lambda x.F &\mathrel{\hat{=}} \lambda x.E \sqcap F \\ \lambda x.E \,\overleftarrow{\cup}\, \lambda x.F &\mathrel{\hat{=}} \lambda x.E \overleftarrow{[]} F. \end{aligned}$$

The motivation for biased choice is to specify the "normal" case of a program execution first and guard it with a boolean expression capturing "normality". In case of an exception, this guarded expression is infeasible, and the right argument of biased choice performs the necessary error-handling. Here is an example:

$atm : (Request, DB) \rightarrow DB$

$atm\ (r, db) \mathrel{\hat{=}} (PINCorrect \rightarrow Process) \overleftarrow{[]}\ db.$

There are alternative ways of handling errors, for instance, the exception monad, see for example [Wad92c]. The motivation for function union and biased function union is similar, except that now there is an argument. We could build the function *atm* stepwise. First the normal case: $f \mathrel{\hat{=}} \lambda(r, db).PINCorrect \rightarrow Process$, and then the exceptional case: $g \mathrel{\hat{=}} \lambda(r, db).\neg PINCorrect \rightarrow db$, and then combine the two partial specifications to get $atm \mathrel{\hat{=}} f \,\dot{\cup}\, g$. Biased function union works similarly, but here would make the guard $\neg PINCorrect$ superfluous in g. Although the higher order functions $\lambda f.\lambda g.\lambda x.f\ x \sqcap g\ x$ and $\lambda f.\lambda g.\lambda x.f\ x \overleftarrow{[]}\ g\ x$ already exist in her language (and in ours), Flynn insists that $\dot{\cup}$ and $\overleftarrow{\cup}$ are "syntactic" rather than language constructs. The reason is that she has imposed syntactic restrictions in order to ban potentially infeasible expressions. Therefore, according to these restrictions, the above functions f and g are not well-formed, since their bodies are potentially infeasible. However, they may still appear as arguments of

[3]Here we use our syntax. Similar expressions can be given in Flynn's syntax.

$\dot{\cup}$ and $\overleftarrow{\cup}$. But now still a proof must be given that the combined function has in fact got a feasible body, since that is not ensured by use of $\dot{\cup}$ or $\overleftarrow{\cup}$ alone, for example consider $\lambda x.x = 1 \to 1 \overleftarrow{\cup} \lambda x.x = 2 \to 2$. A feasibility proof is still required. In this thesis, we have also imposed syntactic restrictions to outlaw potentially infeasible expressions, but since as [Nel89] says "syntax is bad", maybe it would be better to remove these syntactic restrictions, particularly in the light of the usefulness of $\dot{\cup}$ and $\overleftarrow{\cup}$. Naturally, the programmer would then have to ensure feasibility is not lost during the refinement by other means.

A further difference between the calculi concerns the interpretation of nondeterminacy. Flynn interprets nondeterminacy as *erratic*, that is, $E \,[\!]\, F$ could evaluate to either E or F without any preference for defined or undefined outcomes. Therefore $\bot$ and $\bot \,[\!]\, 3$ are not equivalent. We share this operational intuition. However, in a total correctness calculus one is always interested in the worst possible outcome and therefore a program that could fail is as useless as one that definitely fails. That means $\bot \sqcap 3 \equiv \bot$, and we have *demonic* nondeterminacy. In fact, the total correctness view is present in Flynn's work, in form of the axiom for definedness of a binary choice: $\delta(E \,[\!]\, F) \equiv \delta E \wedge \delta F$, where δE stands for "E is defined". There are a number of algebraic advantages to regarding $\bot \sqcap 3$ and $\bot$ as equivalent ($\equiv$) rather than just "refinement equivalent" ($\sqsubseteq\!\!\sqsupseteq$) as Flynn does. The first is that refinement is antisymmetric (with respect to equivalence) and therefore a partial order rather than just a preorder. Secondly, the properties of demonic binary choice are not only associativity, commutativity, idempotency, unit $\top$, but in addition it has the zero $\bot$. Thirdly, the connection between refinement and choice is expressed more clearly by

$$E \sqsubseteq F \;\equiv\; E \sqcap F \equiv E$$

whereas in Flynn's calculus we only have

$$E \sqsubseteq F \;\equiv\; \neg\delta E \vee (E \,[\!]\, F \equiv E).$$

The first formula of the two captures the connection between refinement as a partial order with a minimal and maximal element, and demonic choice as the greatest lower bound operation. If desired, angelic choice can be added as the least upper bound operation. In our view the mathematical elegance of refinement as a complete lattice is compelling. However, with the demonic view of nondeterminacy, the axiom for properness/determinacy of a choice becomes more complex. Instead of Flynn's

$$\Delta(E \,[\!]\, F) \equiv \Delta E \wedge \Delta F \wedge (E \equiv F)$$

we have

$$\mathbf{det}(E \sqcap F) \equiv (\mathbf{det}E \wedge (E \sqsubseteq F)) \vee (\mathbf{det}F \wedge (F \sqsubseteq E)).$$

Continuing with our viewpoint of the refinement lattice, we prefer our generalized choice of the form $\sqcap x : T.E$ (E with x bound to an arbitrary outcome of type T) to Flynn's "specification expressions" $[\!]/S$ (an arbitrary element of set S). The advantages are that $\bigsqcap x.E$ is axiomatised concisely as the greatest lower bound and that it does not require any theory of sets. Furthermore, we have only one language construct that introduces infeasibility: the guard. Generalized choice does at worst ***inherits*** potential infeasibility from its body (typically of the shape $F \rightarrow G$) but does not ***introduce*** infeasibility. This design choice makes guarding play a singular crucial role in our calculus: it is the connection between the refinement lattice of expressions and the implication lattice of propositions. Many other language constructs (assertions, **if** ... **fi**, biased choice, ...) can be expressed, using guards, as alternations. A generalized angelic choice can easily be added as the general least upper bound $\bigsqcup x : T.E$.

Naturally, Flynn's design choice of interpreting nondeterminacy as erratic also influences her denotational semantics. Broadly, they are similar to ours (and share the weakness of not providing a model for angelic nondeterminacy). Both our semantics employ the Smyth order to model the *refinement* order, but unlike this thesis her semantics uses a variant of the Plotkin order for the *definedness* order. The definedness order is used to find least fixpoints as models for recursive definitions. Incidentally, none of her axioms forces the use of *least* fixpoints; any fixpoint would satisfy the relevant axioms. In contrast, our view of nondeterminacy as demonic blurs the distinction between the refinement order and the definedness order; we use the Smyth order for both. We do use a variant of the Plotkin order to model the order of set-values, but that is a completely different purpose, and not central to the semantics.

10.5 Further Work

This section describes possible extensions to our work. Firstly, with laziness and recursion, the language has sensible expressions that cannot be introduced into a program refinement using the present set of axioms. A general theorem introducing recursion is missing. Secondly, we ponder an alternative to adding imperative features to a rich expression language, namely adding a rich expression language to the imperative refinement calculus.

10.5.1 Laziness, Recursion, and Infinite Values

Our specification language is 'lazy'. Semantically, that means that a variable can be bound to undefinedness. Intuitively, it means that to evaluate a function application, instead of evaluating the argument, and then the body of the function, one rather evaluates the body of the function straight away, and only evaluates as much of the argument as is needed to evaluate the body. In terms of the theorems of the calculus, laziness means that β reduction is an equivalence for arbitrary determined, but possibly undefined, arguments.

We chose laziness because we consider it to be an important tool in specification. Laziness separates *what* is being calculated from *how much* is calculated. This point is frequently argued by lazy functional programmers, for instance in [Hug89, Hug90]. Unfortunately, implementations of lazy programming languages are often not as efficient as this naive way of thinking seems to promise. The problem is that in an average program, many, if not most function arguments will be evaluated sooner or later, and delaying their evaluation only adds a large book-keeping overhead. Trying to determine which arguments will be used is known as strictness analysis.

However, in our specification language we have already admitted features (foremost generalised choice) that provide greater expressiveness at the expense of executability. We view laziness in the same way: it is a language feature that adds expressiveness, possibly at the expense of efficient executability. Laziness definitely does not increase the set of functions that can be implemented: There are algorithms that can be elegantly and modularly expressed if the language is lazy, but all computable functions can also be written in a strict language.

By the way, making function application lazy can be seen as making the semantics more homogeneous. For if we deal with recursive expressions by unfolding, that is, $\mu f.F[f] \equiv F[\mu f.F[f]]$, then even in a language with strict application, the functional F that produces the recursively defined function is lazy. For us, there is no distinction between such a functional and a normal higher-order function $\lambda f.F[f]$.

To make lazy function application useful in more than the most pathological examples, we require lazy data structures, that is, the tuple and sum constructors are lazy. A consequence is that we can write defined recursive expressions of these types too, rather than only of function types. The values of these expressions may be finite or infinite.

Yet, our theorems for introducing such expressions into a program derivation are limited. For instance, the theorem that we use to introduce recursively defined functions is repeated here. For $<$ a well founded order on T, and $F[f]$ monotone in f,

$$(\forall x.(E \sqsubseteq F[\lambda y.y < z \succ\!\!- E[y/x]])) \Rightarrow (\lambda x.E \sqsubseteq \mu f.\lambda x.F[f]).$$

The resulting recursively defined function takes arguments of type T. It need not be strict, and need not be primitive recursive (for instance the Ackermann function), but still there are many functions that cannot be generated by this theorem. These are functions that produce potentially infinite results, like $\mathit{iterate}\ f\ x \mathrel{\hat{=}} x : \mathit{iterate}\ f\ (f\ x)$, or even the familiar $\mathit{map} : (a \to b) \to [a] \to [b]$. The arguments of their recursive calls may be exactly the same as the original arguments, and there is no well founded order such that $X < X$.

As discussed in the section about refinement in chapter 3, the desired general theorem that introduces recursive expressions (of any type) would be of the shape

$$(E \sqsubseteq F[E]) \Rightarrow (E \sqsubseteq \mu x.F[x]), \text{ where } x \text{ is fresh},$$

with some condition that ensures the unfolding-function F does make *progress*, that it is *productive*. But what is progress ? It is too much to require that F be not strict, that is, $F[\perp] \not\equiv \perp$, since if $E \equiv \perp$, the strict $F[x] \mathrel{\hat{=}} x$ should not be forbidden. It is clearly too much to require that $X \sqsubset F[X]$ for all X, since it would rule out F having any fixpoint at all! It would also rule out any candidate F, since if $X \equiv \top$, no inequivalent, more refined $F[\top]$ exists. But the weakened requirement $X \sqsubseteq F[X]$ for all X is too little: it would not rule out $F[x] \equiv x$, for arbitrary E. The question remains open: What does it mean for F to 'make progress' ?

A possible line of attack is to say that if we can observe a certain amount of information from X, we must be able to observe 'one step' more from $F[X]$. For example, for lists, we would require something like

$$(take\ n\ xs) +\!\!+ \perp \ \sqsubseteq\ take\ (n+1)\ F[xs],$$

for every natural n.

It seems that we need comparisons of infinite values, that is, theorems like for instance the take-lemma of [BW88]: If $take\ n\ xs \equiv take\ n\ ys$ for each natural n, then $xs \equiv ys$, whether they are finite or infinite. Such laws can be proven sound with respect to a denotational semantics (informally done in [BW88]), or an operational semantics using co-induction (see [Gor94b, Gor94a]). Whatever style of semantics is chosen, such theorems are only true in programming languages, not for specification languages, since the observers are programs. If we restrict ourselves to programs, a progress-characterisation along the lines of our tentative axiom 'operational recursion' seems reasonable.

In summary, our specification language offers more expressiveness than our present theorems can introduce into a program derivation. We desire a theorem that introduces recursive expressions of any type, into arbitrary specification contexts.

10.5.2 The Imperative Refinement Calculus Extended

The advantage of imperative expressions using the state monad is their referential transparency. Their disadvantage is that they are quite clumsy. A Pascal programmer will be quite unwilling to accept this clumsiness, especially since for programs with only static use of state, the Pascal versions are very clear, even if not referentially transparent.

Weakest preconditions provide a way of reasoning about imperative programs with static use of state. The well-known imperative refinement calculus is based on weakest precondition semantics. It deals with undefinedness, nondeterminacy, and miracles, but only at the level of commands. The expressions are always assumed to be feasible, defined, and determined.

One way to combine expression refinement and imperative programming would be to extend the expression language of the imperative refinement calculus. One could use our specification language, without the state monad operations. The weakest precondition semantics would have to be extended to deal with miracles, undefinedness, and nondeterminacy at expression level.

Then 'functional programs' (expressions) could be embedded in imperative programs. It would also be useful to embed imperative programs in expressions in the way that **run**E encapsulates the state transformer E in a state-less expression. One could extend the expression language by adding expressions of the form **var** $x : T \bullet C \bullet E$, where $x : T$ is a list of typed variables whose scope is the command C and the expression E. The intended meaning is that of expression E with x replaced by the final value of x after command C. For example **var** $x : \mathbb{N} \bullet x := 17;\ x := x - 1 \bullet x + 4$ would be equivalent to 20. Referential transparency for expressions is preserved by demanding that C have no free variables except x. So the above expression could be substituted for 20 anywhere. Exploring such a language would have to show whether it is at all useful to distinguish referentially transparent variables and assignable variables. It would be reasonable to allow C in **var** $x : T \bullet C \bullet E$ to refer to referentially transparent variables, but not to assignable variables. On the other hand, having two different kinds of variables is not desirable for simplicity.

This approach is appealing in that it combines well-known syntax from both programming paradigms, imperative and functional. It addresses the weakness of IRC of having a poor expression language. However, it doesn't remove the two levels from the language, and it doesn't add dynamic use of state.

Bibliography

[ABH$^+$92] C. J. Aarts, R. C. Backhouse, P. Hoogendijk, T. S. Voermans, and J. van der Woude. A relational theory of datatypes. Available via anonymous ftp from `ftp.win.tue.nl` in directory `pub/math.prog.construction`, September 1992.

[Bac80] R.-J. R. Back. Correctness preserving program refinements: Proof theory and Applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[Bac90] R. Backhouse. On a Relation of Functions. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business: A Birthday Salute to E. W. Dijkstra.* Springer Verlag, 1990.

[Bar72] J. M. Barbour. *Tuning and Temperament.* Da Capo, 1972. Reprint of the first ed., East Lansing, 1951.

[BCJ84] H. Barringer, J. H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[BdBM$^+$91] R. C. Backhouse, P. de Bruin, G. Malcolm, T. S. Voermans, and J. van der Woude. Relational catamorphisms. In Möller B., editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers B.V., 1991.

[BdM93a] R. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Moeller, H. Partsch, and S. Schuman, editors, *Formal Program Development IFIP TC2/WG 2.1 State-of-the-Art Seminar, Rio de Janeiro, January 1992*, pages 43–46, 1993.

[BdM93b] R. Bird and O. de Moor. Solving optimization problems with catamorphisms. In *Proceedings of the Second International Conference on Mathematics of Program Construction, Oxford, 29 June - 3 July 1992*, volume 669 of *Lecture Notes in Computer Science.* Springer Verlag, 1993.

[BDW88] P. Bunemann, S. B. Davidson, and A. Watters. A semantics for Complex Objects and Approximate Queries: Extended Abstract. Internal report ms-cis-87-99, University of Pennsylvania, October 1988. Also in 7th ACM Principles of Database Systems.

[BF94] A. Bunkenburg and S. Flynn. Expression Refinement: Deriving Bresenham's algorithm. In *Functional Programming, Glasgow 1994*, Workshops in Computing. Springer Verlag, 1994. Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland.

[BFL$^+$94] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A practioner's guide.* FACIT. Springer Verlag, 1994.

[BGJ89] R. Bird, J. Gibbons, and G. Jones. Formal Derivation of a Pattern Matching Algorithm. *Science of Computer Programming*, 12:93–104, 1989.

[Bij89] A. Bijlsma. Calculating with Pointers. *Science of Computer Programming*, 12:191–205, 1989.

[Bir80] R. S. Bird. Tabulation Techniques for Recursive Programs. *ACM Computing Surveys*, 12:403–417, 1980.

[Bir84] R. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4), October 1984.

[Bir87] R. Bird. An Introduction to the Theory of Lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 3–42. Springer Verlag, 1987.

[Bir88] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computer Science*, pages 151–218. Springer-Verlag, 1988.

[Bir90] R. Bird. A Calculus of Functions for Program Derivation. In D. A. Turner, editor, *Research Topics in Functional Programming*, 1987 University of Texas Year of Programming Series, pages 287–308. Addison Wesley, 1990.

[Bre65] J. E. Bresenham. An Algorithm for Computer Control of a Digital Plotter. *IBM Syst. J.*, 4(1):25 - 30, 1965.

[Bre77] J. Bresenham. A Linear Algorithm for Incremental Digital Display of Circular Arcs. *Communications of the Association for computing machinery*, 20(2):100 - 106, February 1977.

[But95] M. Butler. Calculational Derivation of Algorithms on Tree-based Pointer Structures. Technical Report DSSE-TR-95-2, University of Southampton, October 1995.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice Hall, 1988.

[Che86] J. H. Cheng. *A Logic for Partial Functions.* PhD thesis, University of Manchester, 1986.

[CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *Proceedings of the 3rd Refinement Workshop*, pages 51–69. Springer Verlag, 1991. Also available as technical report UMCS-90-3-1 from University of Manchester.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Dij82] E. W. Dijkstra. The equivalence of bounded nondeterminacy and continuity. In *Selected Writings on Computing: a personal approach.* Springer Verlag, 1982.

[dM92] O. de Moor. *Categories, Relations and Dynamic Programming.* Technical Monograph, PRG-98, Oxford University Laboratory, Programming Research Group, April 1992.

[dP92] U. de'Liguoro and A. Piperno. Must preorder in non-deterministic untyped λ-calculus. In J.-C. Raoult, editor, *CAAP 92, 17th Colloquium on Trees in Algebra and Programming*, volume 581 of *Lecture Notes in Computer Science*, pages 203–220. Springer Verlag, 1992.

[Dro82] R. G. Dromey. *How to Solve it by Computer.* Prentice Hall International, 1982.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Springer Verlag, 1990.

[FGS$^+$85] H. Fuchs, J. Goldfeather, S. Spach, J. Austin, F. Brooks, J. Eyles, and J. Poulton. Fast Spheres, Shadows, Textures, Transparencies and Image Enhancements in Pixel-Planes. In *SIGGRAPH 85*, 1985.

[Fly97] S. Flynn. *A Refinement Calculus for Functional Programs.* PhD thesis, Glasgow University, 1997.

[Gor94a] A. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Gor94b] A. Gordon. A tutorial on co-induction and functional programming. In K. Hammond, D. N. Turner, and P. S. Sansom, editors, *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 78–95. Springer Verlag, 1994.

[GS93] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Springer Verlag, 1993.

[Gun89] C. A. Gunter. The mixed powerdomain. Internal report MS-CIS-89-77, Logic & Computation 18, University of Pennsylvania, December 1989.

[Gun90] C. A. Gunter. Relating Total and Partial Correctness Interpretations of Non-Deterministics Programs. In P. Hudak, editor, *Principles of Programming Languages (POPL 90)*, pages 306–319. ACM, 1990.

[Hec90] R. Heckmann. Set domains. In Jones N, editor, *ESOP 90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 177–196. Springer Verlag, 1990.

[Hec92] R. Heckmann. Power Domains Supporting Recursion and Failure. In J.-C. Raoult, editor, *CAAP 92, 17th Colloqium on Trees in Algebra and Programming*, volume 581 of *Lecture Notes in Computer Science*, pages 165–181. Springer-Verlag, 1992.

[Heh94] C. R. Hehner. Abstractions of Time. In A. W. Roscoe, editor, *A Classical Mind*. Prentice Hall, 1994.

[Hes90] W. H. Hesselink. Modalities of nondeterminacy. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J Misra, editors, *Beauty is our Business: A Birthday Salute to E.W. Dijkstra*, pages 182–192. Springer-Verlag, 1990.

[Hin69] R. Hindley. The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[HMT88] R. Harper, R. Milner, and M. Tofte. The definition of Standard ML: Version 2.0. Technical report, Laboratory for Foundations of Computer Science, Edinburgh University, 1988.

[Hoa89] C.A.R Hoare. Lectures on Category Theory. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 5 of *NATO ASI Series F: Computer and System Sciences*, 1989.

[Hoo93] P. F. Hoogendijk. (Relational) programming laws in Boom hierarchy of types. In *Proceedings of the Second International Conference on Mathematics of Program Construction, Oxford, 29 June - 3 July 1992*, volume 669 of *Lecture Notes in Computing Science*. Springer Verlag, June/July 1993.

[HP72] P. Hitchcock and D. Park. Induction Rules and Termination Proofs. In *IRIA Conference on Automata, Languages, and Programming Theory*, 1972.

[HPW91] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.1. Technical report, Yale University, August 1991.

[HPW92] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[Hug89] J. Hughes. Why Functional Programmming matters. *Computer Journal*, 32(2):98–107, April 1989.

[Hug90] J. Hughes. Why Functional Programmming matters. In D. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[HW89] P. Hudak and P. Wadler. Report on the programming language Haskell, a non-strict purely-functional programming language. Technical report, Glasgow University and Yale University, January 1989.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation.* PhD thesis, University of Utrecht, Utrecht, The Netherlands, 1993.

[JM94] C. B. Jones and C. A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[Joh95] T. Johnsson. Fold-unfold transformations on state monadic interpreters. In *Functional Programming, Glasgow 1994.* Springer Verlag, 1995. Proceedings of the 1994 Glasgow Workshop on Functional Programming, Ayr, Scotland.

[Jon72] C. B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. *ACM SIGPLAN Notices*, 7(1):150–169, January 1972.

[Jon86] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, 1986.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 2nd edition, 1990.

[Jon93] M. P. Jones. Release Notes for Gofer 2.28. Included as part of the standard Gofer distribution, February 1993.

[JS90a] C. B. Jones and R. C. F. Shaw, editors. *Case Studies in Systematic Software Development.* Prentice Hall Internationa;, 1990.

[JS90b] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI design*, pages 13–70. North Holland, 1990.

[JS93] G. Jones and M. Sheeran. Designing Arithmetic Circuits by Refinement in Ruby. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science.* Springer Verlag, 1993.

[KL93] D. J. King and J. Launchbury. Functional graph algorithms with depth-first search. In J. T. O'Donnell and K. Hammond, editors, *Functional Programming, Glasgow 1993.* Springer Verlag, July 1993. Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 5 - 7 July 1993.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* North Holland, 1952.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6:323–350, 1977.

[Kol76] G. Koletsos. Sequent calculus and partial logic. Master's thesis, Manchester University, 1976.

[Kol81] G. Koletsos. Notational and logical completeness in three-valued logic. *Bull. of the Greek Mathematical Society*, 22:121–141, 1981.

[Lau93] J. Launchbury. Lazy imperative programming. In *Proc. ACM Sigplan Workshop on State in Programming Languages*, pages 46–56, June 1993. (available as YALEU/DCS/RR-968, Yale University).

[LJ94] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.

[LP95] J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[Luk20] J. Łukasiewicz. O logice trójwartościowej. *Ruch Filozoficzny*, pages 169–171, 1920. Translated as "On three-valued logic" in Polish Logic 1920-39, S. McCall (ed.), Oxford U.P., 1967.

[Mal89] G. Malcolm. Homomorphisms and Promotability. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer Verlag, 1989.

[McC67] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North Holland Publishing Company, 1967.

[Mee86] L. Meertens. Algorithmics - Towards Programming as a Mathematical Activity. *Mathematics and Computer Science*, 1, 1986. CWI Monographs (J. W. de Bakker, M. Hazewinkel, J. K. Lenstra, eds.) North Holland, Puhl. Co.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[MJ95] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer Verlag, 1995.

[Mog89] E. Moggi. Computational lambda calculus and monads. *Logic in Computer Science*, June 1989. California, IEEE.

[Mor87] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287 – 306, 1987.

[Mor88a] C. Morgan. Auxiliary variables in data refinement. *Information Processing Letters*, 29(6):293–296, December 1988.

[Mor88b] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10:403 – 419, 1988.

[Mor89] J. M. Morris. Laws of Data Refinement. *Acta Informatica*, 26:287 – 308, 1989.

[Mor90a] J. M. Morris. Programming by Expression Refinement: the KMP algorithm. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business: A Birthday Salute to E. W. Dijkstra*, pages 327–338. Springer Verlag, 1990.

[Mor90b] J.M. Morris. Programming by Expression Refinement: a Sequence of Examples. *Structured Programming*, 11:189–197, 1990.

[Mor94] C. Morgan. *Programming from Specifications*. Prentice Hall, 2nd edition, 1994.

[Mor96a] J. M. Morris. Reasoning equationally in the presence of undefinedness. *Science of Computer programming*, 1996. Submitted for publication.

[Mor96b] J. M. Morris. Undefinedness and nondeterminacy in program proofs. Submitted, 1996.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nel89] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[NH93] T. S. Norvell and E. C. R. Hehner. Logical specifications for functional programs. In *Proceedings of the Second International Conference on Mathematics of Program Construction, Oxford, 29 June - 3 July 1992*, volume 669 of *Lecture Notes in Computer Science*, pages 269–290. Springer Verlag, 1993.

[oPCG95] Mathematics of Program Constructions Group. Fixed-point calculus. *Information Processing Letters*, 53:132–136, 1995. (C. Aarts, R. Backhouse, E. Boiten, H. Doornbos, N. v. Gasteren, R. v. Geldrop, P. Hoogendijk, E. Voermans, J. v. d. Woude).

[Pat93] J. W. Patterson. Fast spheres. In R. J. Hubbold and R. Juan, editors, *Eurographics '93*. Eurographics Association, Blackwell Publishers, 1993.

[PH$^+$96] J. Peterson, K. Hammond, et al. Report on the Programming Language Haskell. Technical Report YALEU/DCS/RR-1106, Yale University, May 1996. http://haskell.cs.yale.edu/haskell-report/haskell-report.html.

[PJW93] S. Peyton Jones and P. Wadler. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages*, 1993.

[Plo76] G. Plotkin. A Powerdomain Construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

[Plo83] G. Plotkin. Domains. Lecture Notes, Department of Computer Science, University of Edinburgh, 1983.

[Sch86] D. A. Schmidt. *Denotational Semantics — A Methodology for Language Development.* Allyn and Bacon, 1986.

[Sco75] D. Scott. Data types as lattices. In A. Dold and B. Eckmann, editors, *Logic Conference, Kiel 1974*, volume 499 of *Lecture Notes*, Berlin, 1975. Springer Verlag.

[SHLG94] V. Stoltenberg-Hansen, I. Lindström, and E. R. Griffor. *Mathematical Theory of Domains.* Cambridge Tracts in Theoretical Computer Science 22. Cambridge University Press, 1994.

[Smy78] M. B Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–26, 1978.

[Smy83] M. B. Smyth. Powerdomains and Predicate Transformers: A Topological View. In J. Diaz, editor, *ICALP 83*, volume 154 of *Lecture Notes in Computer Science*, pages 662–676. Springer Verlag, 1983.

[Sne93] J. L. A. Snepscheut. *What computing is all about.* Texts and Monographs in Computer Science. Springer Verlag, 1993.

[SP82] M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comp.*, 11:761–783, 1982.

[Spi89] M. Spivey. A Categorical Approach to the Theory of Lists. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science.* Springer Verlag, 1989.

[Spr82] R. F. Sproull. Using program transformations to derive line-drawing algorithms. *ACM Transactions on Graphics*, 1(4):259 – 273, October 1982.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics of computer languages. In *Proceedings of the Symposium in Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*, 1971.

[SS92] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514 – 523, 1992.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[vG88] A. J. M. van Gasteren. *On the shape of mathematical arguments.* PhD thesis, Technische Universiteit Eindhoven, 1988.

[Wad92a] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. (Special issue of selected papers from 6'th Conference on Lisp and Functional Programming.).

[Wad92b] P. Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[Wad92c] P. Wadler. Monads for Functional Programming. In *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and Systems Sciences.* Springer Verlag, August 1992.

[War94] N. Ward. *A Refinement Calculus for Nondeterministic Expressions.* PhD thesis, University of Queensland, 1994.

[Wir90] N. Wirth. Drawing lines, circles, and ellipses in a raster. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business: A Birthday Salute to E. W. Dijkstra.* Springer Verlag, 1990.