Projections as Subtypes in Lazy Functional Languages

Mark Kambites

Third year project report submitted March 11, 1999 toward the degree of
M.Math. Mathematics and Computer Science from the Departments of
Mathematics and Computer Science, at the University of York.

Abstract: The concept of *subtypes* as sets of values of a given type has been used as a mathematical tool for reasoning about, and hence proving correctness of, functional programs. Subtypes in functional programming are equivalent to the pre- and post-conditions used for formal specification in an imperative context.

We explore the possibility of representing such subtypes as functions expressable within a lazy functional language. These can then be incorporated into actual programs, providing extra verification of program correctness at run-time.

Contents

List of Figures ix

Chapter 1. Introduction 1

Chapter 2. Mathematical Background 3
- 2.1. Sets with Structure 3
- 2.2. The Trouble with Types as Sets 5
- 2.3. Atomic Types 6
- 2.4. Product Types 6
- 2.5. Disjoint Union Types 7
- 2.6. Function Types 8
- 2.7. More Complex Types 9
- 2.8. Some Special Functions 10
- 2.9. Composition Algebras 11
- 2.10. Conclusion 12

Chapter 3. Overview and Review 13
- 3.1. The Need for Constraints 13
- 3.2. Static Methods of Constraining Values 14
- 3.3. Dynamic Constraint Checking 15
- 3.4. Formal Specification Techniques 16
- 3.5. Constraints as Subtypes 17
- 3.6. Subtypes and Functional Languages 17
- 3.7. Subtypes for Reasoning and Proof — Reade's Tree Balancing 17
- 3.8. Subtypes in Practice — Runciman's Projections 18
- 3.9. A Theory of Subtypes — Möller's Composition Algebra 20
- 3.10. Conclusion 21

Chapter 4. Specification and Analysis of Requirements 23
- 4.1. Practicability 23
- 4.2. Soundness 23
- 4.3. Ease of Use 24
- 4.4. Ease of Reasoning 25
- 4.5. Sufficient Power 26

4.6. Efficiency 27
4.7. Conclusion 27

Chapter 5. The Algebra of Functional Subtypes 29
5.1. Initial Intuitions 29
5.2. Downward Closure 31
5.3. Downward Closure and Laziness 33
5.4. Downward Closure on Flat Domains 33
5.5. Downward Closure on Complex Domains 34
5.6. Downward Closure on Function Domains 37
5.7. Generalising Downward Closure 39
5.8. Intradependent Subtypes 40
5.9. Conclusion 42

Chapter 6. Subtype Combinators 43
6.1. Preliminaries 43
6.2. Flat Types and Complex Types 43
6.3. Subtypes of Atomic Types 44
6.4. Subtypes of Tuple Types 45
6.5. Subtypes of Lists 46
6.6. Subtypes of Algebraic Types 47
6.7. Deriving Combinators for Algebraic Types 47
6.8. Subtypes of Functions 48
6.9. Algebraic Types Revisited 49
6.10. Conclusion 50

Chapter 7. Subtypes in Action 53
7.1. The Problem 53
7.2. Preliminaries 53
7.3. The Subtype of Fully Defined Integers 54
7.4. Normalised Trees 54
7.5. Put Trees and Taken Trees 56
7.6. Ordered Trees 57
7.7. Trees Containing or Not Containing a Given Value 58
7.8. Subtypes of Functions 59
7.9. Testing and Integrating the Subtypes 60
7.10. Conclusion 61

Chapter 8. Conclusion 63
8.1. Evaluation 63
8.2. Evaluation of the Downward Closed Subtyping System 63

8.3. Evaluation of the Intradependent Subtyping System 64
8.4. Conclusion 64
8.5. Suggestions for Future Work 65
8.6. Acknowledgements 65

Appendix A. Source Code for the Subtype Combinator Library 67
A.1. Downward Closed Subtype Combinators 67
A.2. DrIFT Rule for Downward Closed Subtype Combinators 69
A.3. Intradependent Subtype Combinators 70
A.4. DrIFT Rule for Intradependent Subtype Combinators 70

Appendix B. Miscellaneous Subtype Definitions 73
B.1. Integer Subtypes 73
B.2. List Subtypes 74
B.3. Subtypes of Ordered Types 75

Appendix C. Source for the Balanced Trees Example 77
C.1. 2-3 Tree Header (Data Type and Derived Combinators) 77
C.2. 2-3 Tree Body (Subtypes and Algorithms) 78

Bibliography 85

List of Figures

1.1 Chapter dependency of this report. 2

2.1 A partial order without a supremum. 4

2.2 The domain of Booleans. 6

2.3 The domain of Boolean pairs, in an eager language. 6

2.4 The domain of Boolean pairs, in a lazy language. 7

2.5 The disjoint union of the type of Booleans with itself, in an eager language. 7

2.6 The disjoint union of the type of Booleans with itself, in a lazy language. 8

2.7 The domain of functions from the Booleans to the Booleans. 9

2.8 The domain of the type MyBoolPair. 9

CHAPTER 1

Introduction

In most computer programs, data flows between different sections of code, often written at different times by different people. To prevent misunderstandings, to allow the early detection of errors and to aid reasoning about programs, it is helpful to have some means of specifying *constraints* upon particular values. Such constraints can be modelled as *subtypes* — subsets of the values of a particular type.

In the past, subtypes have generally been used for *reasoning* about programs. Our intention is to *implement* subtypes as functions in a lazy functional language, providing an extra level of program verification at run-time.

Much of the material in this report should be equally applicable to any lazy functional programming language. Where a particular language is required, for example purposes or for a concrete implemention, that chosen is Haskell 1.3.

It is not the intention of this report to provide an introduction to Haskell, or to functional programming in general. The reader without a grounding in functional programming is advised to consult the excellent textbook [**BW88**]. The language Haskell is officially documented in [$\mathbf{P^+96}$], and a good introduction is provided by [**Tho96**].

Chapter 2 of this report provides a basic grounding in the mathematics of domain theory and its application to datatypes in functional programming. Some knowledge of this material is necessary to understand the rest of this report, but the reader with prior knowledge of the theories underlying functional programming may prefer to omit this section, or to use it for reference only.

In Chapter 3, we examine in depth the motivation for imposing constraints upon values, and look at various alternative methods employed for the purpose. We then introduce the concept of a subtype, comparing it with the other methods described and reviewing previous work in the area.

Chapter 4 explains in more detail our intention to implement subtypes in practice, and conducts an analysis of our requirements.

Chapter 5 explores the problem of finding a suitable mathematical model for subtypes which can be implemented in a lazy functional context.

Chapter 6 sees the development of a set of *combinators* — constructs which can be used to define safe, robust subtypes for practical use in the lazy functional language Haskell. In Chapter 7 we put our combinators to the test, by applying them to a practical problem.

Chapter 8 concludes, evaluating work done and suggesting some directions which further research might profitably take.

Appendices contain complete Haskell/DrIFT source code for the combinator library developed in Chapter 6 and the example from Chapter 7, as well as some miscellaneous examples of subtype definitions.

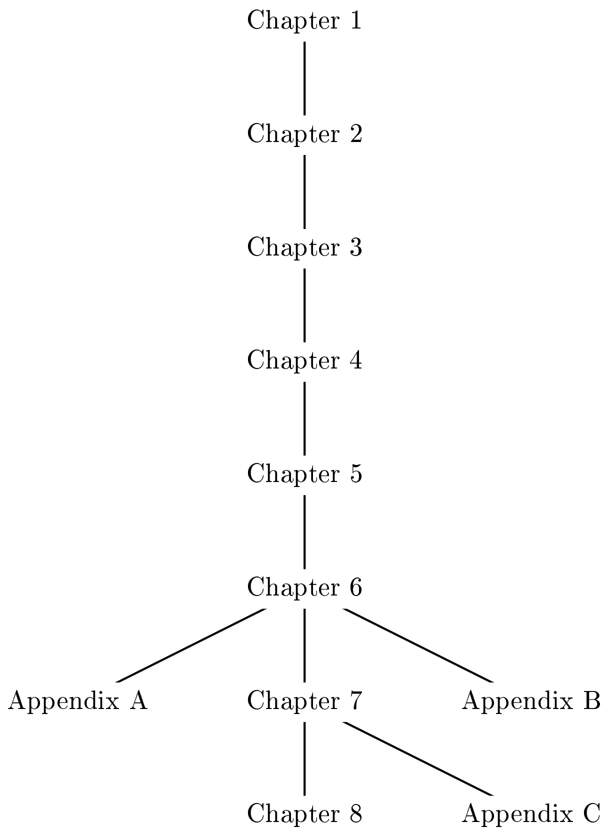

FIGURE 1.1. Chapter dependency of this report.

CHAPTER 2

Mathematical Background

Much of the work in this report is founded in the mathematics of *domain theory*, of which the reader will require some understanding. This chapter provides an outline of the required material.

For the reader who would like more depth, the definitive works in this area are those by Scott [**Sco76, Sco82**]. Gentler introductions are to be found in many books on programming language semantics — [**NN92, Sch86, Sto77, Ten91**] all devote chapters to domain theory. Be warned, though, that introductory texts tend to present simplified versions of Scott's theories, and different authors simplify in different ways. In particular, the definition of a "domain" itself differs significantly from text to text.

2.1. Sets with Structure

DEFINITION 2.1. A *partial order* or *poset* is a pair $(S, \sqsubseteq)$, where S is a set, and $\sqsubseteq$ is a binary relation on S which satisfies:

1. $\forall x \in S, x \sqsubseteq x$ [reflexivity]
2. $\forall x, y, z \in S, x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$ [transitivity]
3. $\forall x, y \in S, x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$ [anti-symmetry]

□

Technically speaking, then, a partial order is a *pair*, and is distinct from each of the set and the operation which together comprise it. In practice, however, it is common to speak of the pair and the set interchangeably. Often the name of the latter is used for both. We shall follow this convention. In particular, we may talk of a *subset* of a partial order $(S, \sqsubseteq)$, by which we mean variously a subset of S, or a partial order $(T, \sqsubseteq)$ where $T \subseteq S$. The exact meaning intended should be clear from the context.

EXAMPLE 2.1. Consider the set $\mathbb{C}$ of complex numbers with the relation $\sqsubseteq$ defined by $x \sqsubseteq y$ if and only if $arg(x) = arg(y)$ and $|x| \leq |y|$. We can prove that $(\mathbb{C}, \sqsubseteq)$ is a partial order by showing that it satisfies the three required axioms:

1. For any $x \in \mathbb{C}$, we have $arg(x) = arg(x)$ and $|x| \leq |x|$, so $x \sqsubseteq x$.
2. For any $x, y, z \in \mathbb{C}$, if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $arg(x) = arg(y) = arg(z)$ and $|x| \leq |y| \leq |z|$ so $x \sqsubseteq z$
3. For any $x, y \in \mathbb{C}$, if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $arg(x) = arg(y)$ and $|x| \leq |y|$ and $|y| \leq |x|$, so $x = y$.

□

We can represent partial orders graphically as *Hasse diagrams* . The reader has already seen a simple example — Figure 1.1 represents the partial order of *dependency* on the chapters of this report. Each vertex represents a member of our set — in this case a chapter. Each edge represents the fact that, in order to read the chapter at the lower end, one requires a knowledge of material from the chapter at the upper end. Where a dependency can be inferred from other dependencies using the transitive property of our relation, we do not draw a corresponding edge — if we did, our diagram would quickly get too confusing. In our example, it is implicit that Chapter 4 depends upon Chapter 1. Similarly, we do not explicitly represent the fact that each element is related to itself — this can safely be assumed.

DEFINITION 2.2. If $(S, \sqsubseteq)$ is a partial order and $T \subseteq S$, then an *upper bound* of T is an element $s \in S$ such that $\forall t \in T, t \sqsubseteq s$.

An upper bound s of T is the *least upper bound* or *supremum* of T, if for any upper bound u of T we have $s \sqsubseteq u$. In this case we write $s = \sqcup T$, or if T is a doubleton set $\{x, y\}$ we sometimes write $s = x \sqcup y$. □

DEFINITION 2.3. If $(S, \sqsubseteq)$ is a partial order and $T \subseteq S$, then a *lower bound* of T is an element $s \in S$ such that $\forall t \in T, s \sqsubseteq t$.

An lower bound s of T is the *greatest lower bound* or *infimum* of T, if for any lower bound l of T we have $l \sqsubseteq s$. In this case we write $s = \sqcap T$, or if T is a doubleton set $\{x, y\}$ we sometimes write $s = x \sqcap y$. □

An upper bound of T — even the supremum — may or may not be contained in T. It is evident, though, that any upper bound which *is* contained in T must be the supremum. A subset need not have any upper bounds — there are two ways in which it can fail to do so:

1. It can contain two elements which simply do not have a common upper bound. In our example of the complex numbers, any set containing both 1 and i will have no upper bound, since such an element would have the same argument as each, which is plainly impossible.
2. Even if every pair of elements in the subset has an upper bound, it does not follow that the entire subset has an upper bound. Consider the usual ordering $\leq$ defined upon the positive integers. Any set of two positive integers has an upper bound — their sum, for example. However, the entire set of positive integers has no upper bound, since there is no (finite) number larger than all the positive integers.

Also, the existence of an upper bound does not guarantee a *least* upper bound, since upper bounds need not be related to one another. In the partial order shown in Figure 2.1, for example, x and y are both upper bounds of $\{a, b\}$, but neither is the supremum.

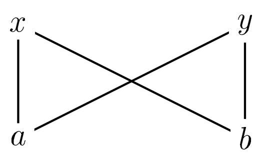

FIGURE 2.1. A partial order without a supremum.

It should be apparent from the symmetry of definition that all of the above observations on upper bounds and suprema apply equally to lower bounds and infima.

DEFINITION 2.4. A non-empty subset T of a partial order is called *directed* if every doubleton subset of T has a supremum in T. □

EXAMPLE 2.2. In Figure 2.1 the entire set $\{a, b, x, y\}$ is *not* directed, since the doubleton subset $\{x, y\}$ has no supremum. The subset $\{a, b, x\}$ *is* directed, since each pair of its elements has a supremum. □

DEFINITION 2.5. A partial order $(S, \sqsubseteq)$ is called *complete* if S has an infimum in S, and every directed subset of S has a supremum in S. □

DEFINITION 2.6. A *domain* is a complete partial order. □

As mentioned above, some authors use different definitions of a domain. In particular, it is common to demand that a domain be a *complete lattice*, in which *every* subset, directed or otherwise, has a supremum.

For our purposes it makes no real difference which definition we use. Any complete partial order can be extended to a complete lattice, by the addition of extra elements to act as suprema. We choose not to use such extensions because, in our application, it is difficult to assign a concrete meaning to the extra elements. However, so long as we take care never to make direct use of the *non-existence* of these extra elements, we can reason about a complete partial order, and be assured that our results apply equally to the corresponding complete lattice. An example of care being taken in this respect will be seen in Section 5.5, in the proof of Proposition 5.6.

2.2. The Trouble with Types as Sets

One of the appealing features of a functional programming style is the ease with which we can *reason* about functional programs. Of particularly significance in this respect is the absence of a global "state" on which "side-effects" can be exercised. Our reasoning processes are simplified if we can assume that the net effect of an expression evaluation is characterised entirely by the resulting value.

Unfortunately, Turing showed that, in any programming language with more than a certain degree of expressiveness, it is possible to write a program which never halts [**Coh97**]. In functional terminology, it is possible to write an expression, evaluation of which never terminates.

In a sense, we can think of the *evaluator* of the expression as a global resource over which the expression evaluated can exercise a degree of control. In particular, it can have the "side-effect" of causing the evaluator to *fail completely to return a value*. In most languages, this failure can take the form either of indefinite looping, or of an error condition which terminates the evaluation process completely.

With a lazy evaluation strategy, things are yet more complicated. An expression of a compound type can be evaluated "by need", one component at a time. Any or all of these component evaluations may fail to terminate, so whether a particular value at a particular point will cause a failure depends upon whether those components which cannot be evaluated are "needed".

It would be very tiresome if we had to consider each of these possibilities as a special case, whenever we reasoned about programs. We can address this problem by introducing *extra values of the expression type* to model various possible combinations of definedness and undefinedness. We shall see how this works firstly with *atomic types* and then with a selection of *compound types*.

2.3. Atomic Types

Atomic types contain fixed values which are either built into the programming language, or enumerated by the programmer. Examples in many languages include the integer and Boolean types, as well as enumeration types. If an expression of an atomic type is evaluated, the result will be either a fully-defined member of the type, or non-termination. To model the latter possibility, we introduce one extra "undefined" value. It is convenient also to introduce a partial ordering relation, to encapsulate the qualitative distinctness of our new element from the other, fully-defined values.

DEFINITION 2.7. If T is a set, then the *lifting* of T is the complete partial order $(T \cup \{\perp\}, \sqsubseteq)$ where $x \sqsubseteq y \iff x = y \vee x = \perp$. □

The new element $\perp$ (pronounced "bottom" or "undefined") is used to model non-termination, while the relation $\sqsubseteq$ can be read as "is less defined than". Informally, "$x \sqsubseteq y$" means "if evaluation of an expression involving y does not terminate normally, then evaluation of the same expression with x substituted for y will also fail to terminate". The type of booleans, for example, can be modelled by the domain which is represented by the Hasse diagram shown in Figure 2.2.

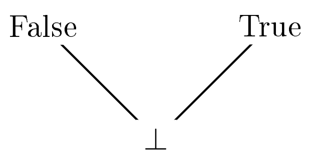

FIGURE 2.2. The domain of Booleans.

2.4. Product Types

A *product type* contains tuples of values of given types. Examples include the type of Boolean pairs, or the type of triples consisting of two integers and a string.

In a language with an *eager* evaluation strategy, a value of a product type is evaluated either wholly or not at all, just like one of an atomic type. If it *is* evaluated, either an error occurs or a fully-defined value is returned. In these languages, then, we can apply the same solution as for atomic types, modelling the type with the complete partial order which is the *lifting of the product type.* Figure 2.3 shows the domain of Boolean pairs in an eager language.

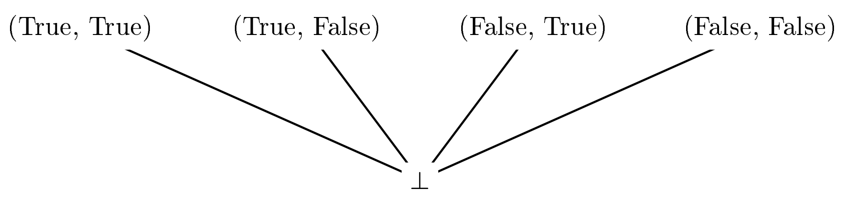

FIGURE 2.3. The domain of Boolean pairs, in an eager language.

With a *lazy* evaluation strategy, however, things are slightly more complicated. It is possible to evaluate any component of a tuple independently, and any or all such evaluations may result in errors. We need a model which allows us to express undefinedness of each component. We can construct such a structure from the complete partial order models of the component types:

DEFINITION 2.8. If $(S_1, \sqsubseteq_1), (S_2, \sqsubseteq_2), \ldots, (S_n, \sqsubseteq_n)$ are complete partial orders then the *product domain* of $S_1 \ldots S_n$ is the complete partial order $(S_1 \times S_2 \times \cdots \times S_n, \sqsubseteq)$ where $(s_1, s_2, \ldots, s_n) \sqsubseteq (t_1, t_2, \ldots, t_n)$ if and only if $s_1 \sqsubseteq_1 t_1, s_2 \sqsubseteq_2 t_2, \ldots, s_n \sqsubseteq_n t_n$. □

The domain of Boolean pairs in a lazy language is pictured in Figure 2.4.

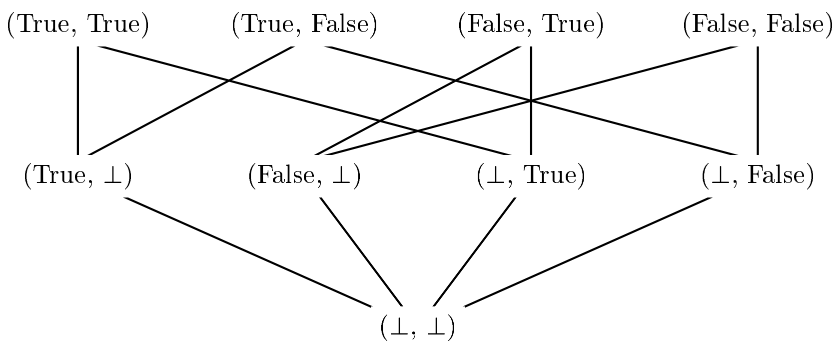

FIGURE 2.4. The domain of Boolean pairs, in a lazy language.

2.5. Disjoint Union Types

A *disjoint union type* or *sum type* contains members taken from the union of a number of types. To insist that types were disjoint before forming their union would raise difficult questions about the definition of equality between members of different types. Rather, we choose to guarantee disjointness by *tagging* each value with an indicator of its origin.

As with product types and atomic types, evaluation of an expression of a disjoint union type in an *eager* language either terminates, or it does not. So we can model a union type with the complete partial order which is the *lifting of the union type*. Figure 2.5 shows the disjoint union of the type of Booleans with itself. Subscript numbers are used as tags.

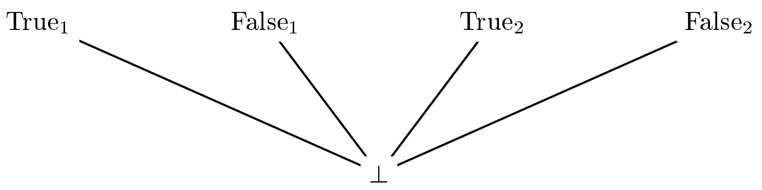

FIGURE 2.5. The disjoint union of the type of Booleans with itself, in an eager language.

In a *lazy* language, it is common to allow the use of the *tag* of an element of a disjoint union type for pattern matching. An attempt to evaluate an expression of such a type can fail either when attempting to evaluate the *tag*, or at any point during the evaluation of the actual element. Once again, we can model this situation with a structure formed from the complete partial order models of the underlying types, with a new "bottom" element:

DEFINITION 2.9. Suppose $(S_1, \sqsubseteq_1), (S_2, \sqsubseteq_2), \ldots, (S_n, \sqsubseteq_n)$ are complete partial orders with $\forall i, j \in \{1, \ldots, n\}, S_i \cap S_j = \emptyset \wedge \perp_S \notin S_i$. Then the *disjoint union domain* of $S_1 \ldots S_n$ is the complete partial order $(T_1 \cup T_2 \cup \cdots \cup T_n \cup \{\perp_S\}, \sqsubseteq)$, where $x \sqsubseteq y$ if and only if $x = \perp_S$ or $\exists i | x \in S_i \wedge y \in S_i \wedge x \sqsubseteq_i y$. □

Figure 2.6 shows the Hasse diagram of the disjoint union of the type of Booleans with itself, in a lazy language.

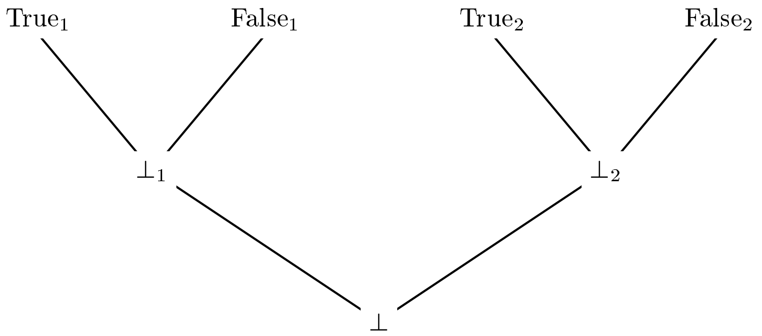

FIGURE 2.6. The disjoint union of the type of Booleans with itself, in a lazy language.

Notice that it makes sense to define the disjoint union of a *single* type. The result is effectively a *lifting* of the type, with a new $\perp$ element less defined than all the other elements.

2.6. Function Types

After our treatment of product and tagged union types, we might be forgiven for assuming that functions, too, need to be modelled differently for eager and lazy languages. In fact, virtually every language is lazy with regard to *function* evaluation. This is because the operation of a function is considered only with regard to those parameter values with which it is actually called. A truly eager approach to function evaluation would involve the complete tabulation of every function the first time it was called, with a single non-termination result generating an immediate error!

DEFINITION 2.10. The *function domain* of a set D and a complete partial order $(E, \sqsubseteq_E)$, written $D \to E$, is the complete partial order $(S, \sqsubseteq)$ where S is the set of all functions from D to E, and $f \sqsubseteq g$ if and only if $\forall x \in D, f(x) \sqsubseteq_E g(x)$. □

In Figure 2.7 we show the domain of Boolean functions. Functions are shown as sets of mappings.

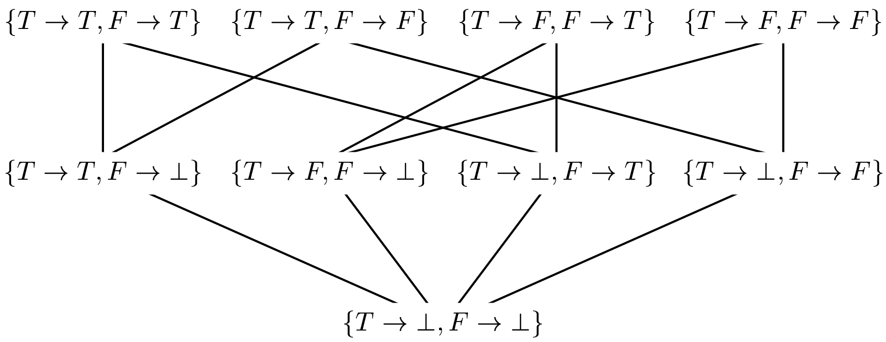

FIGURE 2.7. The domain of functions from the Booleans to the Booleans.

2.7. More Complex Types

Other compound types provided in programming languages can usually be represented in terms of those described above. Haskell, for instance, provides *algebraic types* such as:

```
data Contact = Telephone Int
             | Address String String String
```

In general, algebraic types can be viewed as *disjoint unions of product types*. The above example, for example, is equivalent to the disjoint union of `Int` and `String` × `String` × `String`. `Telephone` and `Address` act as distinguishing tags.

Notice that we could define a tuple type as an algebraic type. For example:

```
data MyBoolPair = MyTag Bool Bool
```

However, this new type does not have quite the structure we might expect of a pair. It is actually the disjoint union of a single product type, which is a *lifting* of that type. The structure of this domain is shown in Figure 2.8.

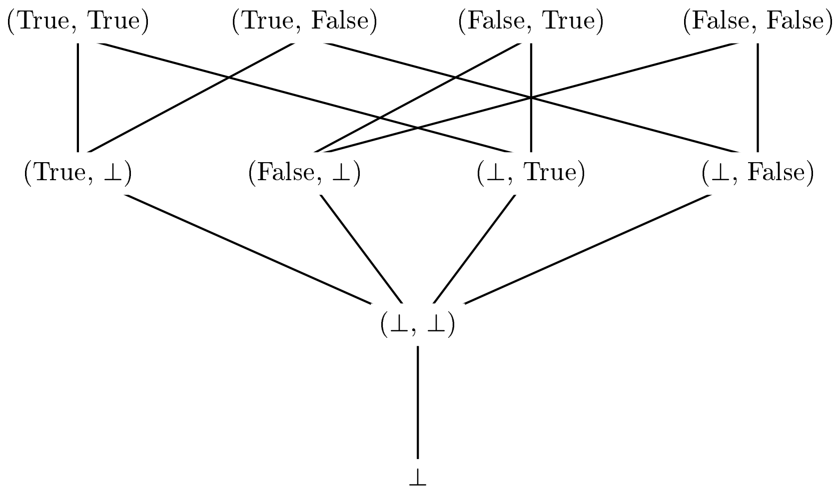

FIGURE 2.8. The domain of the type MyBoolPair.

In fact, the tuple types provided in the Haskell language are of this new, lifted kind. Haskell tuples, then, are modelled as having an extra $\perp$ element, which is less defined even than the tuple of undefined elements. Wadler [**Wad85**] provides a discussion of the motivation for and consequences of this approach.

Lists, also, can be thought of as algebraic types. In a Haskell-like language without a list structure, we might define a list of Booleans with:

```
data [Bool] = Bool : [Bool]
            | []
```

2.8. Some Special Functions

DEFINITION 2.11. A function $f : D \to E$ is *monotonic*[1] with respect to partial orders $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ if and only if $\forall x, y \in D, x \sqsubseteq_D y \implies f(x) \sqsubseteq_E f(y)$. □

EXAMPLE 2.3. The function $f : \mathit{Bool} \to \mathit{Bool}$ defined by $f(\perp) = True$ and $f(True) = f(False) = False$ is *not* monotonic. In the domain of Booleans $\perp \sqsubseteq True$, but it is not true that $f(True) = True \sqsubseteq False = f(\perp)$. □

The use of the $\perp$ element to model non-termination immediately imposes the restriction that only monotonic functions can be computed. A formal proof of this is beyond the scope of this dissertation. As an informal explanation, suppose $f : D \to E$ is not monotonic but can be computed. Then there exist $x, y \in D$ with $x \sqsubseteq_D y$ but *not* $f(x) \sqsubseteq_E f(y)$. So there is an expression E involving an occurrence $f(y)$ evaluation of which fails to terminate, but which does terminate with $f(x)$ substituted for $f(y)$. But substituting $f(x)$ for $f(y)$ is simply substituting x for an occurrence y. So E is an expression involving an occurrence of y, evaluation of which fails to terminate, but which does terminate when x is substituted for y. But this means that we cannot have $x \sqsubseteq y$, which is a contradiction.

We saw in Definition 2.8 that the set of all functions between two domains forms a complete partial order. The subset of such functions which are *monotonic* also forms a complete partial order, with the ordering relation defined in the same way.

DEFINITION 2.12. A function $f : D \to E$ is *strict* if and only if $f \perp_D = \perp_E$.

A curried function $f : X_1 \to X_2 \to \cdots \to X_N \to Y$ is *strict in its mth argument* if $f x_1 x_2 \ldots x_{m-1}$ is strict for all $x_1 \in X_1$, $x_2 \in X_2$, ... , $x_{m-1} \in X_{m-1}$. □

EXAMPLE 2.4. Consider the Haskell functions defined as follows:

```
firsttwo1, firsttwo2 :: [a] -> (a,a)
firsttwo1 (x:y:_) = (x, y)
firsttwo2 xs = (head xs, head (tail xs))
```

`firsttwo1` is strict, since $\perp$ will not match the pattern in the definition, and the result will be $\perp$. `firsttwo2` is not strict, since $\perp$ will match the pattern in the definition, and the result will be $(\perp, \perp)$. □

[1]This definition is confusingly similar to one familiar from elementary analysis. In real analysis, a function is described as monotonic if it is either non-decreasing or non-increasing on its entire domain. By the definition above, though, it is monotonic with respect to $(R, \leq)$ only if it is non-decreasing, and with respect to $(R, \geq)$ only if it is non-increasing. A more appropriate description of this new property might be "monotonically *increasing*", but the unqualified use of "monotonic" in this context is standard in computer science.

DEFINITION 2.13. The *identity function on D* is $id_D : D \to D$ defined by $\forall x \in D, id_D(x) = x$. $\square$

DEFINITION 2.14. A function $f : D \to D$ is a *partial identity* if and only if $f \sqsubseteq id_D$. $\square$

What does this definition mean? Recalling the definition of the function domain, we see that $f \sqsubseteq id \iff \forall x \in D, f(x) \sqsubseteq id_D(x) \iff \forall x \in D, f(x) \sqsubseteq x$. So a partial identity is a function which maps every element of a domain to itself, or to some element less defined than itself.

COROLLARY 2.1. If f is a partial identity, then $f\bot \sqsubseteq \bot \implies f\bot = \bot$, so f is necessarily strict. $\square$

PROOF. The result is immediate from Definitions 2.12 and 2.14. $\square$

DEFINITION 2.15. The *fixed point set* of a function $f : D \to D$ is $F(f) = \{x \in D | f(x) = x\}$. If x is a member of the fixed point set of f, we say x is a *fixed point* of f. $\square$

EXAMPLE 2.5. The identity function id_D on D has a fixed point set consisting of the entire of D. The function defined on the real numbers by $f(x) = x^2$ has fixed points 0 and 1. $\square$

DEFINITION 2.16. A function $f : D \to D$ is *idempotent* if and only if $f \circ f = f$. $\square$

Equivalently, we can say that a function is idempotent if its image set is a subset of its fixed points set, or if its restriction to its image set is an identity function.

EXAMPLE 2.6. Consider the Haskell function which calculates the absolute value of an integer, defined by:

```
abs :: Int -> Int
abs x | x >= 0  = x
abs x           = -x
```

`abs` is plainly idempotent, since its image set is the set of non-negative integers, and it maps each such integer to itself. $\square$

DEFINITION 2.17. A *projection* is an idempotent partial identity function. $\square$

EXAMPLE 2.7. The identity function id_D on any domain D is a projection. It is certainly a partial identity, and it is also idempotent, since $id_D(id_D(x)) = x = id_D(x)$.

The idempotent function `abs` from Example is *not* a projection, since it not a partial identity. $\square$

2.9. Composition Algebras

We have already observed that the set of monotonic functions between two domains forms a complete partial order. In fact, we can give it even more structure.

DEFINITION 2.18. A *monoid* is a triple $(S, \circ, 1)$ where S is a set, $1 \in S$ and $\circ$ is a binary operation satisfying:

1. $\forall f, g \in S, f \circ g \in S$ [closure]
2. $\forall f, g, h \in S, (f \circ g) \circ h = f \circ (g \circ h)$ [associativity]
3. $\forall f \in S, (f \circ 1) = (1 \circ f) = f$ [identity]

□

EXAMPLE 2.8. The set of integers with the operation of multiplication and identity 1 form a monoid. □

Möller [**Möl96**] defines the following useful structure, combining the properties of a monoid and a complete partial order:

DEFINITION 2.19. A composition algebra is a quintuple $(S, \circ, 1, \sqsubseteq, 0)$ where S is a set, $0 \in S$, $1 \in S$, $\circ$ is a binary operation on S and $\sqsubseteq$ is a binary relation on S, satisfying:

1. $(S, \circ, 1)$ is a monoid.
2. $(S, \sqsubseteq)$ is a complete partial order with infimum 0.
3. The operation $\circ$ is monotonic in both arguments.

□

A prime example of a composition algebra, and the one with which we shall be most concerned, is the domain of monotonic functions between two complete partial orders, with the operation of function composition.

DEFINITION 2.20. A *subalgebra* of a composition algebra $(S, \circ, 1, \sqsubseteq, 0)$ is a composition algebra $(T, \circ, 1, \sqsubseteq, 0)$ such that $T \subseteq S$. □

PROPOSITION 2.2. Suppose $(S, \circ, 1, \sqsubseteq, 0)$ is a composition algebra, and $T \subseteq S$. Then $(T, \circ, 1, \sqsubseteq, 0)$ is a subalgebra of $(S, \circ, 1, \sqsubseteq, 0$ if and only if the following conditions are satisfied:

1. $0, 1 \in T$ [identity and infimum]
2. $\forall f, g \in T, f \circ g \in T$ [closure]
3. Every directed subset of T has a supremum in T.

□

2.10. Conclusion

To avoid reasoning explicitly about errors and infinite loops in programs, we introduce extra undefined and partially defined elements into our models of types. These can be used to model various combinations of termination and non-termination. We also introduce an ordering relation $\sqsubseteq$ to represent the concept of one element being "less defined than" another. Together, the type and the relation and the set form a structure known as a *domain*, or *complete partial order*.

We have also seen that the set of monotonic functions between two domains forms a special structure called a *composition algebra*. We shall make extensive use of all of these concepts in the chapters that follow.

CHAPTER 3

Overview and Review

In this chapter, we examine in depth the motivation for imposing constraints upon values in programs. We look at various alternative methods employed for this purpose, including static, dynamic, and formal specification approaches. We then proceed to introduce the concept of a *subtype*, comparing it with the other methods described. We examine the special significance subtypes hold in functional languages, and review previous work in the area.

3.1. The Need for Constraints

Many occasions arise in computer programming when it is useful to specify a set of constraints to which a particular value should conform.

For example, suppose that a programmer intends to implement a function to search for a particular item in an ordered list, and return a Boolean value to indicate its presence or absence. If she is not able to specify that the list must be ordered, another programmer may erroneously call the function with an unordered list. So what happens when a function is called with a value which the implementor did not intend it to handle? There are four distinct possibilities:

1. The equations (in a declarative language) or algorithm (in a procedural language) supplied by the implementor do not mathematically well-define the function on the given parameter values. This will cause an error at run-time, resulting from an invalid operation, such as division by zero, or from a failure of pattern matching.

2. The function is mathematically well-defined on the given parameters, but in such a way that the language is unable to evaluate or execute it to give an result. This again will cause an error, although this time it may take the form of the evaluation or execution looping indefinitely. This may be more difficult to trace.

3. The function is well-defined on the supplied parameters, both mathematically and in the context of the language evaluation strategy. The algorithm is quite applicable, but its result is not what the caller expects. This will result in a legal, but incorrect, return value. Perhaps some inconsistency of this value, either internal or with other data, will result in the detection of the error at a later stage. At this point, it should prove possible to trace the origin of the failure, although the process may be greatly complicated by the lateness of its discovery. But perhaps it will never be detected, either because the return value is quite plausible, or because subsequent stages themselves fail to detect errors, and gradually iron out any evidence of inconsistency. In this case, the entire program may return a plausible but erroneous value, with potentially disastrous consequences in the real world.

4. By chance, the function has been written in such a way that it works with the unexpected value, producing the result expected by the caller. This poses no *immediate* problem, but subsequent *changes* to the implementation of the function may have unexpected effects on the program as a whole.

Similarly, the programmer implementing a *call* to a function is inclined — maybe even entitled — to assume that, if he provides correct parameter values, the return value will be appropriate. An error in the implementation of the function may result in an erroneous result, but even if the value is plainly ridiculous, it is largely a matter of chance whether and when the problem is detected.

The explicit placing of constraints upon values can help to minimise these problems in either or both of two different ways. Static and formal reasoning methods aim to ensure that a value can never be in violation of its constraint. Dynamic methods, on the other hand, aim to ensure that any constraint violations which *do* arise during program execution are caught as quickly as possible. We shall look at each of these approaches in turn.

3.2. Static Methods of Constraining Values

The static type systems of most programming languages are of little help in this regard. Often, the programmer is allowed to specify only such type information as the compiler actually needs to facilitate its own internal table management. Worse, in some weakly typed languages the programmer may be allowed to specify more type information even than the compiler actually checks. This may engender a false belief that the compiler will prevent the possibility of invalid parameter values, preventing the programmer from giving any further consideration to the prospect. If we implement our ordered-list-search function in C [**KR88**], for example, we might write:

```
typedef list orderedlist;
bool search(orderedlist l, item x);
```

We then declare variables of type `orderedlist` only when we know that they will be ordered. However, the compiler regards `list` and `orderedlist` as synonymous, and will make no attempt to stop us calling `search` with a basic, unordered `list`.

A more powerful type system is provided in the functional language Cayenne [**Aug99**]. Cayenne has *dependent types* — types which are first-class objects, and can be stored in records and passed as parameters just like any other value. Result types of functions can depend upon the function arguments, and types of record components can depend upon other components. This system is sufficiently powerful to allow the encoding of predicate logic in terms of types. Effectively, programs can be specified as types, and the compiler can be asked to *prove* properties about functions.

For example, it is possible to specify that, if an equality test function `(==)` is defined upon a type, then it must actually represent an equivalence relation. The type checker will then seek to prove that any `(==)` function declared does indeed have the properties of reflexivity, symmetry and transitivity. A consequence of this flexibility is that it is not, in general, possible to decide to if two Cayenne type expressions are equivalent in finite time.

Research [**Bet98**] has recently been conducted into the possibility of subtyping in the presence of dependent types, but it was not possible to obtain this in time for analysis in this report.

Kieburtz [**Kie83**][1] suggests that a language could be constructed in which the programmer can specify precisely a domain for each function. The compiler would then conduct a detailed static analysis of the program, to ensure that each function was called only with parameters within its domain. For example, suppose that we define functions `push` and `pop`, with domains consisting of the set of all stacks, and the set of non-empty stacks respectively. The compiler would analyse the structure of each expression in our program sufficiently deeply to discover that `pop(pop(push(push(EmptyStack))))` is a legitimate expression, while `pop(pop(push(EmptyStack)))` is not.

The problem with this approach, as with any purely static method of constraint, is that it can take no account of data which is not available until run-time. In practice, most programs do not have all their data available for static analysis at the time of compilation.

In response to this problem, Martin [**Mar83**] proposes an even more radical approach. He suggests that the programmer specify a codomain as well as a domain for each function. The static type checker then prevents the composition of two functions, unless the codomain of the inner function is a subset of the domain of the outer function. In our example of the stack, the functions `push` and `pop` would have codomains consisting of the sets of non-empty stacks and of all stacks, respectively.

Under Martin's system, the expression `pop(pop(push(push(EmptyStack)` is illegal, since the domain of the outer `pop` does not entirely contain the entire codomain of the inner `pop`. To circumvent this problem, Martin introduces extra primitive functions, termed *filters*, which apply different functions to a value, depending upon the value itself. In our example, we would use a filter `ifempty`, which takes a stack and two functions, applying one or other function to the stack depending upon whether the stack is empty. Our illegal expression would then be rewritten as `ifempty(pop(push(push(EmptyStack))), f, pop)` where `f` is any function with the empty stack in its domain.

The most obvious thing for the function `f` to do is to raise an exception. In this case, Martin's approach can be viewed as using the static type checker to force the programmer to implement appropriate dynamic checks.

3.3. Dynamic Constraint Checking

Some strongly typed languages explicitly provide for a few particularly common and useful constraints to be specified statically, but checked dynamically. Ada [**Bar96**], for example, allows numeric function parameters and variable values to be restricted to subranges. These constraints are checked at run-time, with violations raising an exception. However, even these systems are seldom powerful enough to express exactly the constraints upon the values that can reasonably be passed to a particular function, or which a particular function might reasonably be expected to return.

Another approach to this problem, popular in object-orientated languages, is to create a special "abstract data type" to hold values satisfying a particular constraint. The data in an object of such a type can be modified only via certain "methods" — operations which are designed to ensure that the constraint, termed a "data invariant", cannot be violated. An "ordered list" abstract data type, for example, might have methods to perform insertion, removal and searching, all of which are implemented to ensure the preservation of the ordering property. [**Som89**]

[1] The author was unable to obtain this paper. The summary of Kieburtz's suggestion provided here is based on a description provided by Martin [**Mar83**].

This approach can work quite well in some circumstances, but problems arise when we come to implement functions which *preserve* constraints but do not *require* them. For instance, suppose we have a function to remove all even numbers from a list of integers. If implemented in the obvious way, it will work equally well on unordered and ordered lists, and will preserve the ordering property, if present.

However, if our function is written to work directly upon the basic list type, then it may use methods of access not supported by our ordered list type. In this case, we will not be able to use it with ordered lists. There are several ways to circumvent this problem. We could implement our function twice — for ordered and for unordered lists. We could have a method in our ordered list type to apply an arbitrary function directly to the underlying basic list, and then *verify* that the ordering constraint is still satisfied. We could ensure that the function only uses methods common to both types. None of these solutions seem wholly satisfactory.

Another technique frequently employed in imperative programming, is that of introducing *assertions* into programs. An assertion is a statement that the programmer *expects* a particular predicate to be true at a particular point in the program. It is evaluated at run-time, and if it is not true an error is assumed to have occurred, and an exception is raised.

For example, our function to search an ordered list could begin with the assertion that its list parameter is ordered, and end with the assertion that its result value is still ordered.

Assertions are less frequently supported in lazy functional languages. Haskell, for instance, currently provides no explicit mechanism for a programmer to make an assertion. In such languages, there arises issues of of whether and when an assertion can be tested without forcing evaluations and interfering with lazy operation. Members of the Standard Haskell Committee [**Has**] have recently discussed this issue, and seem to have concluded that the risks of introducing an explicit assertion mechanism outweigh the benefits.

3.4. Formal Specification Techniques

Another approach to the problem of parameter constraints is to ensure that somewhere there is written a definitive statement of what each function in a program is intended to do. This information can then be made available to all programmers, perhaps in the form of comments in the source code, or maybe in a specially written document.

One common system in the world of imperative programming involves the specification for each function of two conditions, termed the *precondition* and the *postcondition* [**Pre97**]. The precondition is predicated on the function's arguments and on any relevant global variables, while the postcondition is predicated on the arguments, the function's return value and relevant globals. The function implementor's task is to guarantee that, if the precondition is satisfied on calling the function, then the postcondition will be satisfied on exit.

This method allows the programmer to know exactly where he stands. Not only is he aware of what he may or may not assume about the arguments to his own function and what it is expected to return. He also knows exactly how he can safely employ other functions to achieve his ends.

Further, sufficiently precise specification of functions means that formal proof techniques can be applied to programs. These can be used to show that entire programs, do (or just as frequently, do not) satisfy their specifications.

3.5. Constraints as Subtypes

The concept of *subtypes* as subsets of the values of a given static type has been well-known for some time. Subtypes can be used as mathematical tools for reasoning about computer programs, and hence proving their correctness. A constraint upon the values which can legitimately be passed to a function statically declared to have domain T can be expressed as the subtype containing exactly those members of the static type T which satisfy the constraint.

Similarly, the set of plausible return values from a particular function can be treated as a subtype, with a membership test providing a sanity check on the operation of the function.

Further, since functions have type, a specific property of a function can be expressed as the subtype containing exactly those functions (with the same domain and codomain) which have that property.

3.6. Subtypes and Functional Languages

A fundamental tenet in the theory of functional programming is that the effect of a call to a function is characterised entirely by the single value returned — there can be no "side effects"[2]. Similarly, if we assume that a function has exactly one parameter, the eventual result value of a call to a function is dictated exactly by the parameter passed — there can be no dependence upon global variables or resources.

A useful consequence of this is that we can represent any precondition it is possible to define upon a unary function as the subtype containing exactly those members of the domain type which satisfy the precondition. Since we can use currying to express any multi-parameter function as a higher-order unary one, it seems reasonable to expect that the same benefit will apply to functions of more than one variable, so long as we provide appropriately for higher-order subtypes.

For post-conditions the situation is slightly more involved. A postcondition on a unary function may be predicated on both the argument and the result. However, *for a given value of the argument*, the postcondition can be represented as the subtype containing exactly those members of the codomain type which will satisfy the post-condition. In other words, the post-condition can be represented as a subtype which is *a function of the argument value.*

Furthermore, it is common practice to implement a complex function, or even an entire program, as a *pipeline* of composed functions. The insertion of a subtype into such a pipeline is the closest equivalent in functional programming to a general assertion about state, in an imperative language.

3.7. Subtypes for Reasoning and Proof — Reade's Tree Balancing

A good example of the use of subtypes as reasoning tools is provided by Reade [**Rea92**]. He introduces a new equational algorithm for the removal of values from 2-3 trees, and makes extensive use of subtypes to prove formally its correctness. We shall not be concerned directly with his equations, or with the intimate details of his proof. However, he raises some pertinent points, and defines some interesting subtypes. In Chapter 7 we shall use the latter as a benchmark when evaluating our system of subtype combinators.

[2]Provided always that we model non-termination as as if it were a value (namely $\perp$) in the function's codomain. See section 2.8 for a discussion of the consequences of this assumption.

Reade takes a *subtype set*, which we shall term just a subtype, to be an arbitary set of values of the same type. He notes that this definition makes a subtype equivalent to a predicate on a type — namely its characteristic function. He introduces two combinators, allowing the natural construction of subtypes of product and function types:

DEFINITION 3.1. If X_i is a subtype of type T_i for all $i \in \{1..n\}$ then $X_1 \times X_2 \times \cdots \times X_n$ is the subtype of $T_1 \times T_2 \times \cdots \times T_n$ containing values of the form $(x_1, x_2, \ldots, x_n)$ such that $x_1 \in X_1$, $x_2 \in X_2$, $\ldots$, $x_n \in X_n$.

DEFINITION 3.2. If X and Y are subtypes of static types S and T respectively, then $X \to Y$ is the subtype of $S \to T$ defined by $\{f \in S \to T | \forall x \in X, f(x) \in Y\}$.

If we take X to be the subtype of values in the domain satisfying a particular precondition, and Y to be the subtype of values in the codomain satisfying a particular *argument-independent* postcondition, then to say that $f \in X \to Y$ is exactly equivalent to the statement that f satisfies the specification consisting of the given precondition and postcondition.

Also, if the $\to$ operator is taken to be right associative, it can be seen that it extends in an intuitive way to higher-order curried functions. For example, if $f \in (X_1 \to X_2) \to X_3$ and $x_1 \in X_1$ and $x_2 \in X_2$ then we will have $f x_1 x_2 \in X_3$.

Reade is primarily concerned with subtypes of 2-3 trees, which he defines as follows:

DEFINITION 3.3. A 2-3 tree is either empty (E), or a 2-node of the form $Tr2(t_1, a, t_2)$ or a 3-node of the form $Tr3(t_1, a, t_2, b, t_3)$.

He defines various different subtypes, such as the subtypes of *balanced* trees of a particular depth and in general:

DEFINITION 3.4. A 2-3 tree T is *balanced with depth k* **iff** $T = E$ and $k = 0$ **or** $T = Tr2(t_1, a, t_2)$ where t_1, t_2 are balanced with depth $k - 1$ **or** $T = Tr3(t_1, a, t_2, b, t_3)$ where t_1, t_2, t_3 are balanced with depth $k - 1$.

DEFINITION 3.5. A 2-3 tree T is *balanced* **iff** there exists a positive or zero integer k such that T is balanced with depth k.

These particularly interesting subtype definitions are discussed in some depth in the following chapters, and will serve as a useful benchmark for testing the expressive power of a system of subtypes.

While Reade uses only subtypes of fully-defined values in his proof, he notes that there could also be applications for subtypes containing partially-defined values. For example, $\{\perp\} \to \{\perp\}$ is none other than the subtype of *strict* functions of a given type.

3.8. Subtypes in Practice — Runciman's Projections

A different approach to subtypes is taken by Runciman [**Run95**]. His intention is to represent a subtype S of a type T as a *projection* $s : T \to T$, such that the members of S are exactly the fixed points of s. Such projections can be implemented in a functional programming language such as Haskell, making subtypes first-class objects of the language in their own right. Subtypes can then be integrated into programs, and used to provide a degree of verification at run-time.

For example, the set of natural[3] numbers considered as a subtype of the integers, can be represented by the Haskell function:

```
nat :: Int -> Int
nat n | n >= 0  =  n
```

We can now express the restriction that a function is intended only to deal with natural number arguments by composing the subtype onto the right:

```
g = f . nat
```

For natural number arguments, g will behave identically with f, but for negative arguments we have `g x = f` $\perp$ which, if f is strict, will be $\perp$.

Of course, we must be aware that a domain subtype violation will not *always* raise an error. In our example, if f is not strict, lazy evaluation may result in a violation not being discovered. If the argument is not evaluated then we can say that the incorrect value did not affect program execution *at this particular point*, but we may still have missed an opportunity to detect an error which may cause problems elsewhere or at a later date.

Similarly, codomain restriction can be represented by composing subtypes on the left, and general assertions by the insertion of subtypes into a composed pipeline.

Runciman notes that the operation of composition on the domain and codomain sides with given subtypes is, in itself, a projection on the set of functions of the given type, and hence can be thought of as specifying a subtype of such functions. He defines these new subtypes with an infix operator:

```
(->>) :: (a->a) -> (r->r) -> ((a->r) -> (a->r))
sa ->> sr = (.sa) . (sr.)
```

This definition is broadly comparable to Reade's arrow combinator, but it is not quite equivalent. Reade insists only that if the argument is a member of the domain subtype, then the result will be a member of the codomain subtype. By contrast, Runciman demands that a non-member of the domain subtype cannot be allowed to pass unchanged into the function. Rather, some member must be substituted for it as the function parameter.

For example, consider the function $f : Int \to Int$ defined by $f(x) = 3x$. By Reade's definition, this is a member of the subtype $Nat \to Nat$, since we have:

$$x \in Nat \implies x \geq 0 \implies 3x \geq 0 \implies f(x) \geq 0 \implies f(x) \in Nat.$$

However, f is not a fixed point of Runciman's `Nat ->> Nat` since, for example,

```
f -1 = 3 * -1 = -3
```

but

```
((Nat ->> Nat) f) f -1 = (Nat . f . Nat) -1
                       = (Nat . f) (Nat -1)
                       = (Nat . f) <undefined>
                       = <undefined>
```

Like Reade's combinator, the `->>` operator also extends naturally to curried functions of multiple arguments, with the expected consequences.

Runciman also hoped that the intersection of two subtype sets — the imposition of **two** restrictions upon a value — could be represented by the composition of

[3] In line with common practice in computer science, we take the natural numbers to be the **non-negative** integers.

their respective functions. Indeed this seems very natural, but it transpires that the set of projections from a type T to itself is not, in general, closed under the operation of function composition. Some further restriction will be needed upon the projections admissible as subtypes, in order to guarantee that they can be composed without undesirable consequences. It is the resolution of this issue, and the general development of Runciman's idea, with which much of this report is concerned.

3.9. A Theory of Subtypes — Möller's Composition Algebra

Möller [**Möl89**] presents a way of introducing assertions into the applicative language CIP-L [**B$^+$85**]. An expression E whose value must satisfy an assertion A, is replaced by the conditional expression `if A then E else error`, which is abbreviated to $A \triangleright E$.

Evidently, this idea has much in common with Runciman's conception of subtype projections in functional languages, but there are two important differences.

1. CIP-L has strict call-time semantics — it is not "lazy". It will be seen later that the lack of lazy evaluation dramatically simplifies the requirements upon an implementation of subtypes, by avoiding worries about partially defined values.

2. The sublanguage of Boolean expressions in CIP-L is far richer than in most functional languages. Most significantly, CIP-L permits the use of *quantifiers.*

In [**Möl89**], Möller extends his system in the light of Runciman's idea. He abstracts away from the particular linguistic framework of CIP-L, towards a more general, algebraic representation of subtypes as idempotent partial identities on complete partial orders, which he terms *pretypes.*

He defines the *partial identity belonging to an assertion* C to be the function $id_C = \lambda x.C \triangleright x$. Again, this definition is broadly comparable with Runciman's subtypes, without being quite the same:

1. The fixed points of id_C are not, as we might expect, exactly the values for which the assertion C holds. Rather, they include the value $\perp$ regardless of whether it satisfies C. For Runciman, then, $\perp$ is in the subtype as it must be in all subtypes, while for Möller, considering only strict call-time semantics, this is not an issue.

2. Möller requires his partial identities to be fully strict — a non-member must be mapped to $\perp$. In contrast, Runciman demands only that non-members are mapped to some member less defined than themselves. The former approach is difficult to reconcile with a lazy evaluation strategy, since in the worst case it may involve evaluating an infinite structure to see if it satisfies an assertion. In the latter case, though, the structure can be checked for subtype membership only as and when necessary. An undefined element is introduced where sufficient evidence becomes available for the subtype projection to infer non-membership.

These differences reflect in obvious ways the differing requirements of subtyping in a strict and in a lazy language. For reasons such as these, Möller's work is not *directly* applicable to the problem of implementing *lazy* subtypes. However, it will prove useful to draw upon some of his ideas.

3.10. Conclusion

We have seen that it is useful in programming to be able to specify *constraints* to which particular values must conform. We use the concept of *subtypes*, which are subsets of the values of a particular type, for reasoning about programs, where more precise constraints are required than are supported by the static type system of the language under consideration. Subtypes are especially useful in functional languages, because the lack of "side-effects" makes them equivalent to pre-conditions, post-conditions and general assertions.

We have seen that *partial identities* on a type T can be used to represent subtype sets. We have also introduced the idea that such functions could be implemented in a lazy functional language and incorporated into programs, to provide a degree of dynamic constraint checking. It is with the development of this idea that much of the rest of this report is concerned.

CHAPTER 4

Specification and Analysis of Requirements

Following on from the work of Runciman, [**Run95**] our intention is to develop a systematic way of implementing subtypes into a lazy functional programming language, to provide a degree of runtime error checking. For example purposes, we shall use the language Haskell 1.3. The principles, though, should be transferable to other, similarly lazy languages.

We start with the intuition that a subtype is some subset of the values of a given type. Our idea, following [**Run95**], is to represent subtypes of a type T as restricting functions from T to T. Such a function representing a subtype S will leave members of S unchanged, but cause an *error* if it detects non-membership. Semantically, it will map each value either to itself, or to some value *less defined* than itself. We recall from Chapter 2 that such a function is called a *partial identity.*

4.1. Practicability

Our intention is to implement subtypes in practice, so whatever system we choose, it has to be *capable of computation.* We recall from Chapter 2 that only *monotonic* functions are capable of computation. So our subtype functions will need to be *monotonic partial identities.*

4.2. Soundness

Our objective in implementing subtypes in practice is to facilitate run-time *checking* of value constraints in a functional program. So long as the program is functioning correctly, the subtyping system should act merely as a passive observer. It is of the utmost importance that our system of subtyping be *sound*. By this we mean that *incorporation of appropriate and correctly defined subtypes into a correctly implemented program will not change the result of the program.*

Suppose a subtype function s is intended to represent a set S. If the program in which it is used is operating correctly, then all values to which the subtype is applied will be in S. In this situation, we want the subtype to change nothing. Semantically, we want every member of S to be a fixed point of s.

In a lazy language, we treat both infinite values and partially defined values as "first-class citizens". We might expect that they can be members or non-members of subtypes, just like fully-defined values. If this is to be the case, it will have to be possible for them to be fixed points of our subtypes. This will entail a degree of *laziness* in our subtypes, in two different ways:

Firstly, *it is essential that a subtype does not attempt an eager evaluation of an infinite member.* Even if such a structure is fully-defined, a subtype attempting complete evaluation will loop indefinitely. We recall from Chapter 2 that a function which loops indefinitely is semantically equivalent to $\perp$, so such a subtype effectively treats as an infinite item as a non-member.

EXAMPLE 4.1. It might seem plausible to define a subtype of *ordered lists of integers* in Haskell as follows:

```
ordered :: Int -> Int
ordered xs | isordered xs = xs
    where isordered (x:y:zs) = (x <= y) && (isordered (y:zs))
```

Suppose we now want a function which takes the least n elements of an ordered list of integers. Of course, this simply means taking the first n elements. Haskell provides a standard function, called `take`, to do just this. However, if we use `take`, the correct operation of the program will be dependent upon the list being ordered. We might indicate this by implementing our `least` function as follows:

```
least :: Int -> [Int] -> [Int]
least n = (take n) . ordered
```

So how does the function work in practice? With simple, finite lists, it behaves exactly as we would expect. `least 3 [2,3,5,7,11,13]` is now `[2,3,5]`. `least 3 [1,4,6,4,1]` is $\perp$. However, Haskell supports infinite lists, so we can define the entire list of natural numbers:

```
nat :: [Int]
nat = 0 : map (+1) nat
```

The list `nat` is certainly ordered, so we might expect to be able to take the `least 3` elements of it. However, `least 3 nat` will loop forever! Instead, we could implement `ordered` as follows:

```
ordered :: Int -> Int
ordered (x:y:zs) = x : if (x <= y) then (ordered (y:zs))
                                   else (error "Unordered list")
ordered xs       = xs
```

Now the evaluation of `ordered` is lazier. `least 3 [2,3,5,7,11,13]` is still `[2,3,5]`, and now `least 3 nat` is `[1,2,3]`. However, the price to pay is that `least 3 [1,4,6,4,1]` is now `[1,4,6]`. In being lazy, our subtype function has become unaware that the list ceases to be ordered in the unevaluated portion. Of course, `least 4 [1,4,6,4]` will have an undefined element introduced.

Runciman [**Run95**] suggests that this behaviour is actually more intuitive than mapping the entire list to $\perp$. The author finds it difficult to agree with this assessment but, intuitive or not, such behaviour is inevitable if we want our `ordered` subtype to work with infinite lists. □

We said that a subtype must be lazy in *two ways*. In a lazy language, it is quite permissible to manipulate partially defined data values, provided that evaluation of the undefined components is never attempted. it is necessary that *a subtype does not attempt an eager evaluation of an undefined component of a member*.

Best of all would be if subtypes were completely lazy, attempting no extra evaluation at all. Failing that, though, a subtype could be allowed to eagerly evaluate a component, so long as undefinedness of that component would indicate non-membership of the subtype.

4.3. Ease of Use

Another important feature of a subtyping system is that it should be *easy to use*. If integrating subtypes into a program is difficult, not only will programmers have to spend more time doing so, but they will make mistakes. Erroneously implemented subtypes can manifest themselves in a number of ways:

1. They may mistakenly detect "errors" which do not exist. Much time could be wasted searching for such errors.

2. They may fail to detect errors which they would detect if correctly implemented. Even in this case, we are worse off than without subtypes at all. A programmer mistakenly believing that a subtype will detect a particular suspected source of an error, may discount it. He will then waste his time checking other sources. Worse, a program may find its way into the real world, complete with an error which should have been detected.

3. They may violate the soundness principle, actively interfering with the correct operation of the program, without causing a detectable error. The consequences of such interference are potentially disastrous.

In order to ensure that subtypes are easy to use, it will be helpful to provide a clear, simple framework of *combinators* — methods which the programmer can use to create safe, robust subtypes with known properties. This issue will be discussed in depth in Chapter 6.

4.4. Ease of Reasoning

As we have frequently had cause to note, one of the primary advantages of a functional programming style is the ease with which we can reason about functional programs. We saw in Section 3.7 that subtypes can be very useful in the reasoning process. A good, practical subtyping system will enable us to incorporate subtypes which have been used for reasoning about a program directly into an implementation of the program.

Further, even where value constraints have not already been calculated during the derivation of a program it may be possible to derive them later by a reasoning process. Finally, we may want to reason about programs *after* subtypes have been inserted. For example, optimising transformations could be applied after the insertion of subtypes. For all these reasons, *we will need to be able to reason about subtypes.* We would like subtypes to have some nice mathematical properties.

Firstly, once a subtype has been applied to a value, further applications of the same subtype should be superfluous. It should not need to be applied twice, or three times, or twenty-seven times to realise its full error detection capability. If the number of applications required to find an error is bounded, then the required recursion should be built into the subtype. If an unbounded number of applications are required, how will the programmer *using* the subtype know how many times to apply the subtype?

With a functional representation of subtypes, what we are seeking is the mathematical property of *idempotence.* If we know that our subtypes are idempotent, then we need not worry about how many times a subtype should be applied to a value. Furthermore, where the same subtype is applied to a value twice, it may be possible to eliminate the redundant check by a process of cancellation.

EXAMPLE 4.2. An integer is either positive or it is not. There is no additional category of "positive positive integers". It is intuitively obvious that any reasonable definition of a "positive" subtype function will have the property of idempotence. □

EXAMPLE 4.3. It is less clear that a "long long list" the same as a "long list". Indeed Runciman [**Run95**] provides (as a counterexample) a definition which ascribes different meanings to the two:

```
long :: [a] -> [a]
long (x:y:zs) = x : long (y:zs)
```

By this definition `[1,2]` is a `long` list, but not a `long long` list! This behaviour makes `long` very difficult to reason about, so we do not want to admit it as a subtype. □

When we apply two subtype constraints to a value, we are specifying a single, more strict constraint — namely the *conjunction* of two constraints. Similarly, when we apply two functions to a value, we are applying a single function — the *composite* of the two functions. If subtypes are to be represented by functions, it would seem natural that conjunction be represented by composition. For this, we will need to know that the composition of two subtype functions is a subtype function.

EXAMPLE 4.4. Suppose we have defined subtype functions `evenint` and `primeint` over the integers. Then we would like the composite `even . prime` to be a subtype function with the fixed point 2. □

Further, the conjunction of two constraints is the same regardless of the order in which they are applied. It would be nice if the composite of two subtype functions was the same, regardless of the order of composition. We would like subtype compositon to be *commutative.* This property, also, would help to facilitate cancellation of redundant subtypes. If we can commute subtypes to make identical ones adjacent, then we will be able to cancel them.

4.5. Sufficient Power

Finally, if our system of subtypes is to be of any use *it must be sufficiently expressive to provide representations for the constraints programmers actually want to apply to values.*

So what constraints *do* programmers actually require? To answer this question with any degree of authority would probably require a large survey of programs and programmers. The time constraints upon this project did not permit a comprehensive investigation of this nature. The author's own experience of programming suggests that most value constraints in real programs are of a simple nature.

Common examples include integer values which must be positive, lists which must be ordered, strings which must be shorter than a certain length, and stacks which must be non-empty. Such simple subtypes are often readily expressed in functional languages as *guards* — predicates which must be satisfied during pattern matching. Guards, though, tend to restrict laziness, by forcing evaluation of the values on which they are predicated. If our system of subtypes is to be of real use, it should support a good range of such simple constraints, and it should check them lazily.

More complex types tend to bring with them more complex subtypes. Reade's balanced 2-3 trees [**Rea92**], which we saw in Chapter 3, provide a good example of the kind of subtype that can arise. If we can create a subtype system sufficiently powerful to represent these, then we shall have done well.

In many functional languages, functions are treated as first-class citizens. It would seem natural, then, if our subtype system extended smoothly to allow definition of *subtypes of function types* with particular characteristics.

4.6. Efficiency

It is obviously desirable that *the incorporation of subtypes into a program should have a minimal impact on the time and space efficiency of the program.*. Several of the requirements upon which we have already settled will be of assistance in this respect. We have described how mathematical properties such as idempotence and commutativity might allow the application of optimising transformations after the introduction of subtypes. Also, the quality of laziness will obviously help to reduce the performance impact of subtypes.

4.7. Conclusion

In this chapter, we have explained our intention to implement subtypes in practice, and identified our main objectives. Our system of subtypes should have the following characteristics:

1. *Practicability*: Our system must be capable of actual implementation.
2. *Soundness*: Appropriate and correctly implemented subtypes should not interfere with the operation of a correctly implemented program. *Laziness* should be preserved wherever possible.
3. *Ease of use*: A system of *combinators* should be available to make incorporation of subtypes into a program as easy and foolproof as possible.
4. *Ease of reasoning*: Subtypes should have as many nice mathematical characteristics as possible. In particular, the properties of *idempotence*, *commutativity* and *composition closure* would be desirable.
5. *Sufficient power*: Our system must be sufficiently expressive to provide representations of the constraints we actually want to impose. Also, in a language where functions are first-class citizens, we would like be able to define subtypes of functions.
6. *Efficiency*: The incorporation of subtypes into programs should have a minimal impact on time and space efficiency.

In practice, it may not be possible to achieve all of these objectives. If this is the case, we will need to balance conflicting requirements against one another.

CHAPTER 5

The Algebra of Functional Subtypes

In this chapter, we explore the mathematical problem of choosing an appropriate functional representation for a subtype. Following on from the work of Runciman [**Run95**] and of Möller [**Möl96**], we attempt to construct a sound mathematical basis for the incorporation of subtypes into lazy functional programs as a run-time constraint checking system.

In our exploration of this area, we state and prove a number of pure mathematical propositions. Where no source is acknowledged for such a result or proof, the author has derived it independently. However, given the volume of pure mathematical literature and the limited time available for this project, it may be that some such results are already recorded and proved elsewhere.

5.1. Initial Intuitions

Following on from [**Run95**] and from the previous chapter, we start with the intuition that subtypes of a type T will be represented by partial identity functions from T to T. Our intention is to implement our subtype functions on real computers, so these functions will need to be *monotonic*. Thus, we need concern ourselves only with *monotonic partial identity functions*.

We noted in Chapter 2 that all partial identities are strict — they have $\perp$ as a fixed point. If we use such functions to model subtypes, we can represent only those subtypes which contain the undefined value $\perp$. This is no particular problem. Our subtypes are to be *lazy*, so constraint violations can be detected only when the values concerned are actually evaluated. Evaluation of a $\perp$ value always loops indefinitely or causes an error, so the program would fail anyway.

We saw in Chapter 2 that the set of *all* monotonic functions from a type T to itself forms a *composition algebra*. The following result tells us that the same kind of structure can be used to model the *subset* of such functions which are *partial identities*.

THEOREM 5.1. Suppose $(T, \sqsubseteq_T)$ is a complete partial order. Then $(P, \circ, 1, \sqsubseteq, 0)$ is a composition algebra, where P is the set of monotonic partial identity functions from T to T, 0 is the everywhere-$\perp$ function on T, $\sqsubseteq$ is the usual pointwise extension to functions of $\sqsubseteq_T$, $\circ$ is the operation of function composition, and 1 is the identity function on T. □

PROOF. We know from Chapter 2 that $(F, \circ, 1, \sqsubseteq, 0)$ forms a composition algebra, where F is the set of all monotonic functions from T to T. We shall show that $(P, \circ, 1, \sqsubseteq, 0)$ is a subalgebra of $(F, \circ, 1, \sqsubseteq, 0)$:

1. By the reflexivity axiom of a complete partial order, $1 \sqsubseteq 1$, so 1 is a partial identity, and hence $1 \in P$.
2. By definition of the infimum we have $\forall x \in T, 0 \sqsubseteq x$, so certainly $0 \sqsubseteq 1$, and hence $0 \in P$.

3. Suppose $s, t \in P$. Then by the definition of a partial identity, $s \sqsubseteq 1$ and $t \sqsubseteq 1$. Now by the definition of the $\sqsubseteq$ ordering on functions, we have $\forall x \in T, (s(x) \sqsubseteq 1(x) = x) \wedge (t(x) \sqsubseteq 1(x) = x)$. So $(s \circ t)(x) = s(t(x)) \sqsubseteq t(x) \sqsubseteq x$. So $(s \circ t) \sqsubseteq 1$. Hence, $(s \circ t) \in P$.
4. Suppose S is a directed subset of P. Then by transitivity of set inclusion, it is also a directed subset of F. But F is a complete partial order, so there exists a supremum $\sqcup S$ of S in F. Now $1 \in F$, and by the definition of a partial identity, 1 is an upper bound of S. So by the definition of the supremum, $\sqcup S \sqsubseteq 1$. Hence $\sqcup S \in P$.

Thus, $(S, \circ, 1, \sqsubseteq, 0)$ is a subalgebra of $(F, \circ, 1, \sqsubseteq, 0)$, and hence a composition algebra. □

This result tells us that the set of monotonic partial identity functions has the useful mathematical property of *composition closure*. Another feature which we identified as desirable is the property of *idempotence* — the application of a subtype *twice* should not detect violations which a single application does not. Also, we would like to know that subtypes *commute*. Unfortunately, simple partial identities do not, in general, fulfil either of these requirements:

EXAMPLE 5.1. On the domain of Boolean pairs, define functions f and g by:

```
f,g :: Bool -> Bool

f (True, True) = (True, undefined)

g (True, True) = (True, True)
```

Now f and g do not commute, and `(f . g)` is not idempotent, since:

```
(f . g) (True,True)   = f (g (True, True))
                      = f (True, True)
                      = (True, undefined)

(g . f) (True,True)   = g (f (True, True))
                      = g (True, undefined)
                      = undefined

((f.g) . (f.g)) (True, True)  = f (g ((f.g) (True,True)))
                              = f (g (True, undefined))
                              = f undefined
                              = undefined
```

□

It is perhaps some consolation that the requirements of composition closure and idempotence on monotonic partial identities automatically imply the property of commutativity.

THEOREM 5.2. Suppose S is a set of monotonic idempotent partial identity functions, which is closed under the operation of function composition. Then the members of S commute. □

Möller [**Möl96**] provides a proof of this result using some preliminaries. We use a similar technique to prove it directly.

PROOF. Suppose $f, g \in S$. Then by monotonicity of composition, we have $f \circ g \sqsubseteq f \circ 1 = f$ and $f \circ g \sqsubseteq 1 \circ g = g$. Now S is closed under composition, so $f \circ g \in S$. Members of S are idempotent, so $f \circ g = (f \circ g) \circ (f \circ g)$.

Now by monotonocity of composition, we have $f \circ g = (f \circ g) \circ (f \circ g) \sqsubseteq g \circ f$. By the same argument with f and g reversed, we can also obtain $g \circ f \sqsubseteq f \circ g$, so by antisymmetry of the relation $\sqsubseteq$ we must have $f \circ g = g \circ f$. □

Following Möller [**Möl96**], we shall use the term *pretypes* for monotonic projections. It would be convenient if the set of all pretypes over a complete partial order formed a subalgebra. Unfortunately, as we observed in Chapter 3, the set of all projections is *not* closed under the operation of composition, and nor is the set of pretypes:

EXAMPLE 5.2. Consider the functions f and g defined by the Haskell equation:

```
f, g :: Bool -> Bool
f (True, _)    = (True, undefined)
g (True, True) = (True, True)
```

It is easy to see that `f` and `g` are both pretypes, but the composite function `f . g` is not a pretype, since it is not idempotent:

```
(f.g) (True, True) = f (g (True, True))
                   = f (True, True)
                   = (True, undefined)

((f.g) . (f.g)) (True, True) = f (g ((f.g) (True, True)))
                             = f (g (True, undefined)
                             = f undefined
                             = undefined
```

□

This means that if we admit all monotonic projections as subtypes, then we cannot assume that the composite of two subtypes will be another subtype. Instead, we must seek some *subset* of the set of pretypes which is closed under composition or, better still, forms a subalgebra. What kind of subset of the set of projections might satisfy these criteria? There certainly are such sets — the trivial composition algebra containing only 0 and 1 is an example. But are any of them sufficiently expressive for our purposes?

5.2. Downward Closure

Consider again Example 5.2. We can see that it relies upon the fact that an element is mapped first *into* the fixed point set of f, and then out of it again. For applications of projections to do this, it is necessary that there be a "hole" in the fixed point set of f — an element which is not a fixed point f, but which is less defined than some fixed point of f. What happens if we insist that there be no such "holes"?

DEFINITION 5.1. Suppose $(S, \sqsubseteq)$ is a complete partial order. A subset T of S is called *downward closed* if and only if $\forall x \in S, y \in T, x \sqsubseteq y \implies x \in T$.

A partial identity function $s : S \to S$ is called *downward closed* if and only if its fixed point set is downward closed. □

Runciman [**Run95**] identifies the set of monotonic downward closed projections as a possible set of subtypes Möller [**Möl96**] proves that it is closed under function composition. In fact, we can go further:

THEOREM 5.3. The set of downward closed pretypes over a complete partial order forms a composition algebra. □

PROOF. We know that the set of all monotonic

Suppose $(T, \sqsubseteq)$ is a complete partial order. Let D be the set of downward closed pretypes over T, and P be the set of monotonic partial identities over T. By theorem 5.1, $(P, \circ, 1, \sqsubseteq, 0)$ is a composition algebra. From the definition of a pretype, D is a subset of P. We shall show that $(D, \circ, 1, \sqsubseteq, 0)$ is a *subalgebra* of $(P, \circ, 1, \sqsubseteq, 0)$.

1. $y \sqsubseteq x \wedge 1(x) = x \implies 1(y) = y$ is evidently true, so 1 is downward closed. Also, $1(1(\mathrm{x})) = 1(\mathrm{x})$ is true by definition, so 1 is idempotent. Thus, $1 \in D$.

2. Suppose $y \sqsubseteq x \wedge 0(x) = x$. Then $0(x) = x = \perp$, and $y \sqsubseteq x = \perp$, so $0(y) = \perp = y$. So 0 is downward closed. Also, $0(0(x)) = \perp = 0(x)$ so 0 is idempotent. Thus, $0 \in D$.

3. Suppose $f, g \in D$. Then by definition, f and g are idempotent partial identities with the property of downward closure. Suppose further that $(f \circ g)(y) = y$ and $x \sqsubseteq y$. Then
$$\begin{aligned}
& y = (f \circ g)(y) && \\
& \quad = f(g(y)) && \\
& \quad \sqsubseteq g(y) && \text{(since } f \text{ is a partial identity)}\\
& \quad \sqsubseteq y && \text{(since } g \text{ is a partial identity)}\\
\implies\ & f(g(y)) = g(y) = y && \text{(by antisymmtery of } \sqsubseteq)\\
\implies\ & f(x) = x \text{ and } g(x) = x && \text{(since } f \text{ and } g \text{ are downwardly closed)}\\
\implies\ & (f \circ g)(x) = f(g(x)) = f(x) = x &&
\end{aligned}$$
So $(f \circ g)$ is downward closed. Also, for any $x \in T$ we have:
$$\begin{aligned}
g(g(x)) &= g(x) && \text{(by idempotence of } g)\\
f(g(x)) &\sqsubseteq g(x) && \text{(since } f \text{ is a partial identity)}
\end{aligned}$$
Now, since g is downward closed this means that:
$$\begin{aligned}
g(f(g(x))) &= f(g(x)) &&\\
((f \circ g) \circ (f \circ g))(x) &= f(g(f(g(x)))) &&\\
&= f(f(g(x)) &&\\
&= f(g(x) && \text{(by idempotence of f)}
\end{aligned}$$
So $(f \circ g)$ is idempotent. Hence, $(f \circ g) \in D$.

4. Suppose S is a directed subset of D. For each $x \in T$, put $E(x) = \{f(x) | f \in S\}$. Now for each x, if $f(x), g(x) \in E$ then $f(x) \sqcup g(x) = (f \sqcup g)(x) \in E$ so each $E(x)$ is directed. Also, each $E(x)$ is a subset of the complete partial order T, so $E(x)$ has a supremum $\sqcup E(x) \in E(x)$. Define a function $s : T \to T$ by $s(x) = \sqcup E(x)$.
   Now for each x, $s(x) \in E(x)$, so there exists $f \in S | s(x) = f(x)$. But f is a partial identity, so $s(x) = f(x) \sqsubseteq 1(x)$. Thus s is a partial identity.
   Also, suppose $s(y) = y$ and $x \sqsubseteq y$. Then $y = s(y) \in E(y)$, so there exists $f \in S$ with $f(y) = y$. But f must be downward closed, so we have

$x = f(x) \sqsubseteq s(x)$. Also, s is a partial identity, so $s(x) \sqsubseteq x$. Hence, s(x) = x. So s is downward closed.

Now suppose $f \in S$. For each $x \in T$ we have $f(x) \in E(x)$, so $f(x) \sqsubseteq \sqcup E(x) = s(x)$. So $f \sqsubseteq s$. Thus, s is an upper bound of S. Suppose $u \in D$ an upper bound of S. Then for each $x \in T$ we have $\forall f \in S, f \sqsubseteq u$, so $\forall f(x) \in E(x), f(x) \sqsubseteq u(x)$, so $u(x)$ is an upper bound of $E(x)$. But this means that $s(x) \sqsubseteq u(x)$. So $s \sqsubseteq u$. Hence, $s \in D$ is the supremum of S.

Thus, D contains suprema for all directed subsets.

□

So the set of downward closed pretypes has all the nice algebraic properties that we seek — composition closure, commutativity and idempotence.

5.3. Downward Closure and Laziness

Another compelling argument for downward closed subtypes comes from the requirement for *laziness*.

PROPOSITION 5.4. All completely lazy partial identities of product types are downward closed. □

This statement does not lend itself well to formal proof, at least with the particular mathematical framework which we are using. We can, however, present an intuitive argument.

Suppose s is a completely lazy partial identity on S. Suppose further that $x, y \in S$ with $s(y) = y \wedge x \sqsubseteq y$. Now there must be some component of x which is defined, but which is undefined in $s(x)$. This component is fully defined in x, and hence takes the same value y. Now suppose we try to evaluate *only* this component.

x and y are defined on this component, so there is no way to distinguish between them without further evaluation. But s is completely lazy, and so conducts no such further evaluation, and so it cannot tell x from y. Thus, we must have $s(x) = s(y) = y$, so $x \sqsubseteq s(x)$. But since s is a partial identity, we must also have $s(x) = x$. Thus, s is downward closed.

A similar arguments applies to algebraic types.

5.4. Downward Closure on Flat Domains

We have seen that monotonic downward closed projections have nice mathematical and lazy properties. Before choosing them as our model for subtypes, we must also assess whether the set of them is sufficiently comprehensive for our purposes. Is there downward closed pretype corresponding to each subtype we might reasonably want to represent?

For types with flat domains, there is no problem in this respect:

THEOREM 5.5. All partial identities on a flat domain are downward closed and idempotent. □

PROOF. Suppose $(S, \sqsubseteq)$ is a flat domain and let $f : S \to S$ be a partial identity. Now suppose that $x, y \in S$ with $x \sqsubseteq y$ and $f(y) = y$. Then by the definition of a flat domain, either $x = y$ or $x = \perp$. If $x = y$ then $f(x) = f(y) = y = x$. If $x = \perp$ then $f(x) \sqsubseteq x = \perp$, so $f(x) = \perp = x$. Thus, f is downward closed.

Now suppose $z \in S$. Then $f(z) \sqsubseteq z$. So by the definition of a flat domain, either $f(z) = z$ or $f(z) = \bot$. If $f(z) = z$ then $f(f(z)) = f(z)$ is immediate. If $f(z) = \bot$ then we must have $f(f(z)) = f(\bot) = \bot = f(z)$. So f is idempotent. ☐

We have already observed that partial identities are necessarily strict, so $\bot$ must be a member of all representable subtypes. For atomic types, then, the requirement of downward closure places no further restrictions on the subtypes that can be represented.

If, like Möller [**Möl96**], we consider only the case of an eager language, then the domainsof product and sum types are also "flat", and enjoy the same benefit. For the purposes of subtyping in such a language, then, downward closed subtypes provide a neat solution, with mathematical neatness and a comprehensive set of representable subtypes.

Our intention, though, is to implement subtypes in a *lazy* language. We must consider the effect of the restriction of downward closure on more complex domains.

5.5. Downward Closure on Complex Domains

Our idea is that a subtype function models the subtype set consisting of its fixed points. Hence, if subtype functions are downward closed, so are their subtype sets they represent. This somewhat curtails our freedom in deciding which partially defined values of a type are considered members of a subtype.

EXAMPLE 5.3. The subtype of Boolean pairs containing only the fully-defined pairs and $\bot$ has no representation as a downward closed projection. ☐

Even more seriously, it transpires that we have lost the freedom to choose even which fully-defined values are to be in our subtypes. In Section 3.7 we identified as one example of a subtype likely to arise in practice, the set of *balanced 2-3 trees.*

PROPOSITION 5.6. There is no monotonic downward closed projection on the domain of partially- and fully-defined 2-3 trees whose set of fully-defined fixed points is exactly the set of balanced fully-defined 2-3 trees. ☐

PROOF. Suppose for a contradiction that f is such a function. Let $X = Tr2(E,E)$, $Y = Tr2(E,Tr2(E,E))$ and $Z = Tr2(Tr2(E,E),Tr2(E,E))$. From Definition 3.5 it is plain that X and Z are balanced, but Y is not. Now X, Y and Z are all fully-defined, so from the definition of f, we know that $f(X) = X$ and $f(Z) = Z$, but $f(Y) \neq Y$.

f is downward closed, and $P \sqsubseteq X$ and $Q \sqsubseteq Z$, so we must have $f(P) = P$ and $f(Q) = Q$. Now $P \sqsubseteq Y$ and $Q \sqsubseteq Y$, and f is monotonic, so it follows that $P = f(P) \sqsubseteq f(Y)$ and $Q = f(Q) \sqsubseteq f(Y)$ — ie. $f(Y)$ is an upper bound of $\{P,Q\}$.

But Y is the least[1] upper bound of $\{P,Q\}$, so this means that $Y \sqsubseteq f(Y)$. Also, f is a partial identity, so $f(Y) \sqsubseteq Y$. So by antisymmetry of the relation $\sqsubseteq$ we must have $f(Y) = Y$. But we know that $f(Y) \neq Y$, so this is a contradiction. ☐

COROLLARY 5.7. There is no completely lazy implementation of the set of balanced 2-3 trees.

PROOF. Immediate from Propositions 5.4 and 5.6. ☐

[1] In fact in our complete *partial order* representation, Y is the *only* upper bound, but were we reasoning with the corresponding complete *lattice* this would not be the case — cf. Section 2.1.

Intuitively, this corollary is perhaps to be expected. We cannot represent the set of balanced binary trees in a lazy manner, because there are two components of our type, capable of independent evaluation, such that the values of one which are legitimate in our subtype are dependent upon the value taken by another. We shall call such subtypes *intradependent.*

We cannot check such a subtype without looking at two components, at least. But what if only one component is evaluated? If we are to be lazy, we must treat the list as ordered and allow that component through unchanged. But the value of an expression is not, in general, permitted to depend upon the evaluation order. It follows that, if each component is allowed through the subtype check unchanged when evaluated first, then it must always be allowed though unchanged whenever it is evaluated. So all components must be allowed through unchanged, and hence the entire value is unchanged. So our checking system, in being lazy, has not checked the subtype at all!

If a subtyping system were built into a language interpreter or compiler, then it might be possible to circumvent this problem. A subtype function needing further evaluation of a value to complete a check could be *suspended* in the background. The program would continue provisionally, as though the subtype function had verified the value. As and when further component values became available, the suspended function could be "woken up" to resume its check. Semantically, the subtype check is implemented so as to return different values dependent upon the evaluation order! Semantically, the subtype check it is a *function of the evaluation order* as well as of its parameter.

Another solution to this problem will be seen in Section 5.8.

We can generalise this this result mathematically, as well as intuitively. In order to do so, we will need a

DEFINITION 5.2. Suppose $(S, \sqsubseteq)$ is a complete partial order. Now for any $x \in S$, the *downward closure set of* x is $\{y \in S \mid y \sqsubseteq x\}$. For any $T \subseteq S$, the *downward closure set of* S is $\{y \in S \mid \exists x \in S | y \in D(x)\}$.

If $f : S \to S$ is a monotonic downward closed projection then the *fixed point descendent set of* x *with respect to* f is $FD_f(x) = D(x) \cap F(f)$. □

THEOREM 5.8. Suppose f is a monotonic downward closed projection on a complete partial order $(S, \sqsubseteq)$. Then for all $x \in S$, we have $f(x) = \sqcup FD_f(x)$.

□

PROOF. Suppose f is as stated, and $x \in S$, Now for any $y \in FD_f(x)$, we know by definition that $y \sqsubseteq x$ and $f(y) = y$. Now by monotonicity of f, we have $f(y) \sqsubseteq f(x)$. So $f(x)$ is an upper bound of $FD_f(x)$.

By idempotence of f, we have $f(f(x)) = f(x)$, and since f is a partial identity we have $f(x) \sqsubset x$. So $f(x) \in FD_f(x)$. So if u is an upper bound of $FD_f(x)$ we must have $f(x) \sqsubseteq u$. Thus, $f(x)$ is the supremum of $FD_f(x)$. □

THEOREM 5.9. Suppose S is a subtype set of a type T. Then S has a representation as a monotonic downward closed projection if and only if $\forall x \in T, \exists y \in S | y = \sqcap(D(S) \cap D(x))$. □

PROOF. Suppose f is such a representation. Then by definition $F(f) = S$, and since f is downward closed, $F(f) = D(F(f)) = D(S)$. Now for any $x \in T$, we have $D(S) \cap D(x) = F(f) \cap D(x) = FD_f(x)$. By Theorem 5.8, $f(x) = \sqcup(FD_f(x)) =$

$\sqcup(D(S) \cap D(x))$, and since f is idempotent, this means $\sqcup(D(S) \cap D(x)) \in F(f) = S$ as required.

Conversely, assume that $\forall x \in T, \exists y \in S | y = \sqcup(D(S) \cap D(x))$. Now define $f : T \to T$ by $f(x) = \sqcup(D(x) \cap D(S))$. We must show that f is a monotonic, downward closed, idempotent, partial identity with fixed points which are exactly the members of S.

Monotonicity: If $x \sqsubseteq y$ then plainly $D(x) \subseteq D(y)$, so $(D(S) \cap D(x)) \subseteq (D(S) \cap D(y))$, so $f(x) = \sqcup(D(S) \cap D(x)) \sqsubseteq \sqcup(D(S) \cap D(y)) = f(y)$.

Downward closure: Suppose $f(y) = y$ and $x \sqsubseteq y$. Then by our assumption, $y = f(y) = \sqcup(D(S) \cap D(x)) \in S$. Now by the definition of the downward closure set, this means that $x \in D(S)$. Also, $x \in D(x)$, so $x \in (D(S) \cap D(x))$. But x is the supremum of D(x), so it must be the supremum of $(D(S) \cap D(S))$. So $x = \sqcup(D(S) \cap D(x)) = f(x)$.

Partial identity: x is certainly an upper bound of $D(x)$ and hence of $D(x) \cap D(S)$, so by definition of the supremum, $f(x) = \sqcup(D(x) \cap D(S)) \sqsubseteq x$.

Fixed points: Suppose $x \in S$. Then $x \in D(S)$. Also, $x \in D(x)$, so certainly $x \in D(S) \cap D(x)$. But x is the supremum of $D(x)$, so it must be the supremum of $D(S) \cap D(x)$. So $x = \sqcup(D(S) \cap D(x)) = f(x)$. Conversely, suppose $f(x) = x$. Then by our assumption, $x = f(x) = \sqcup(D(S) \cap D(x)) \in S$

Idempotence: For any $x \in T$, by our assumption we have $f(x) = \sqcup(D(S) \cap D(x)) \in S$. But we know that members of S are fixed points of f, so we must have $f(f(x)) = f(x)$.

So S has a representation as a monotonic downward closed projection. □

THEOREM 5.10. Suppose S is a subtype set of a type T. Then S has a representation as a monotonic downward closed projection if and only if every subset of $D(S)$ with a supremum in T has a supremum in S, and for every $x \in T$, $D(x) \cap D(S)$ has a supremum in T. □

PROOF. Suppose f is such a representation, and suppose $V \subseteq D(S)$ with $x = \sqcup V \in T$. Then certainly $V \subseteq D(x)$, so $V \subseteq (D(S) \cup D(x))$. Now x is an upper bound of $D(x)$, and hence of $D(S) \cup D(x)$. Suppose u is an upper bound of $D(S) \cup D(x)$. Then u is an upper bound of V, so we must have $x \sqsubseteq u$. Thus, x is the supremum of $(D(S) \cup D(x))$. Now by Theorem 5.9, $x \in S$.

Conversely, suppose every subset of $D(S)$ with a supremum in T has a supremum in S, and every $D(x) \cap D(S)$ has a supremum in T. Then for any x, $(D(S) \cap D(x))$ has a supremum in T and hence in S, so $D(S) \sqsubseteq (D(S) \cap D(x))$

□

Theorem 5.10 gives us a relatively easy way to distinguish subtypes which cannot be represented by downward closed projections by algebraic means.

EXAMPLE 5.4. Consider the subtype S of ordered lists of integers. Consider the lists `2 : ⊥ : []` and `⊥ : 1 : []`. Both also have more defined instances which are ordered, so they are members of $D(S)$. But their supremum is `2 : 1 : []`, which is not in S. So by Theorem 5.10, S has no representation as a monotonic downward closed partial identity. Worse, by Proposition 5.4, there is no lazy partial identity representation of S. This is to be expected, because the subtype of ordered lists is *intradependent* — the validity of the tail is dependent upon the value of the head. □

There is another, more positive, corollary of Theorem 5.8. The following result, which was stated and proved from first principles in [**Run95**], tells us that when there *is* a monotonic downward closed function representation of a given subtype set, it is unique.

COROLLARY 5.11. A monotonic downward closed projection is characterised entirely by its fixed points. □

PROOF. Immediate from Theorem 5.8 and uniqueness of the supremum. □

5.6. Downward Closure on Function Domains

In Chapter 4 we specified that our system of subtypes should extend smoothly to allow definition of subtypes of functions with particular characteristics. In Chapter 3 we identified one way in which we may wish to constrain a function — by constraining either or both of the argument and the result. We introduced Runciman's `->>` combinator, which constrains both.

For our purposes here, it is more convenient to treat domain and codomain restriction separately. We could facilitate such restriction with the Haskell functions defined by:

```
precond :: (a->a) -> ((a->r) -> (a->r))
precond s = (.s)

postcond :: (r->r) -> ((a->r) -> (a->r))
postcond s = (s.)
```

The function `postcond` takes a subtype projection s over a type r and returns a (polymorphic) subtype projection p over a type $a \to r$. For a given function type $a \to r$, the fixed points of p are exactly those functions which always return fixed points of s — those functions whose *image sets* are subsets of the subtype set represented by s. The relationship between the structures of p and s is so close that almost any domain theoretic property of s will transfer to p. Certainly, if s is a monotonic, downward closed projection then p will be likewise:

PROPOSITION 5.12. If s is a monotonic downward closed projection over a type R, then $p =$ `postcond` s is a monotonic downward closed projection over the type $A \to R$, for any A. □

PROOF. **Monotonicity:** That p is monotonic follows directly from monotonicity of s and of function composition.

Partial identity: Let $f \in T$. We know that s is a partial identity, and for any $x \in A$, we have $f(x) \in R$, so $p(f)(x) = s(f(x)) \sqsubseteq f(x)$. Thus, $p(f) \sqsubseteq f$.

Idempotence: We know that s is idempotent, so $p(p(f)) = p(s \circ f) = s \circ s \circ f = s \circ f = p(f)$ so p is idempotent.

Downward closure: Suppose $p(g) = g$ and $f \sqsubseteq g$. Then for any $x \in T$, we have $s(g(x) = p(g)(x) = g(x)$, and $f(x) \sqsubseteq g(x)$. But s is downward closed, so we can derive $s(f(x)) = f(x)$. So $p(f)(x) = s(f(x)) = f(x)$, so $p(f) = f$.

□

At first sight, the `precond` and `postcond` functions seem very similar in nature. The `precond` takes a subtype projection s over a type a and returns a (polymorphic) function q over a type $a \to r$. However, things get more complex when we attempt to identify the fixed points of q.

The fixed points of `postcond s` were those members of $a \to r$ whose *image sets* were contained within the subtype represented by s. The image set of a function on a given domain is a concrete property of the function itself. Unfortunately, there is no concrete property of a function which is to the domain as the image set is to the codomain.

One of the main purposes of a domain theoretic model of type is to avoid partial functions. All functions are total on their respective types, but they may return the value $\perp$ to signify undefinedness. With this in mind, it does not actually make any sense to "restrict the domain" of a function to a set S. Rather, the equivalent is to demand that a function map all non-members of S be mapped to $\perp$. Any functions which do map all non-fixed-points of s to $\perp$ are certainly fixed points of q. Further, it is obvious that the set of such functions is downward closed, so we might hope that q will have all the nice properties we expect of a subtype.

EXAMPLE 5.5. Consider the Haskell functions defined by:

```
recip :: Real -> Real
recip x | x != 0 = 1 / x

nonzero :: Real -> Real
nonzero x = if (x!=0) then x else (error "Subtype violation")
```

`Nonzero` is a monotonic downward closed projection over the real numbers, and so we can think of it as representing a subtype — the subtype of non-zero real numbers. The function `recip` is a fixed point of `precond nonzero`, since it maps every number not in the subtype `nonzero` to $\perp$. □

These are not the only fixed points of q. If f is not strict in the components which q undefines, then we will have $q(f) = f$, even if f does not map the elements in question to $\perp$.

EXAMPLE 5.6. Consider the Haskell function defined by:

```
f :: Real -> Real
f x  = 2
```

f is a fixed point of `precond nonzero`, and indeed of any precondition subtype! □

Unfortunately, this latter category of fixed points of q is *not* downward closed.

EXAMPLE 5.7. Consider the function f defined in Example 5.6 and the functions g and s defined by:

```
g :: Real -> Real
g x | x==x = 2

s x = undefined
```

Now s is a downward closed projection, and $g \sqsubseteq f$, but:

```
           f 2 = 2

postcond s f 2 = (f.s) 2
               = f (s 2)
               = 2
```

```
                g 2 = 2

     postcond s g 2 = (g.s) 2
                    = g (s 2)
                    = g undefined
                    = undefined
```

So `postcond s f` $\neq$ `postcond s g`, so `postcond s` is not downward closed. $\square$

Nonetheless, projections of the form `precond s` and `postcond t` compose and commute idempotently provided that `s` and `t` have the same properties. These properties were proven for the `->>` combinator in [**Run95**]. We can show that they hold for our `precond` and `postcond` functions by observing that they can be expressed in terms of `->>`:

```
precond s  = (s ->> ok)
postcond t = (ok ->> t)
```

There is scope for much more investigation in this area.

5.7. Generalising Downward Closure

We have seen that downward closed subtypes give us all the mathematical properties we desire, but that they are somewhat limited in their expressive power. Might there be some way in which we can *generalise* the requirement of downward closure — preserving the properties of composition closure, idempotence and commutativity, while increasing the number of subtypes which can be represented? The following proposition gives us a necessary condition for such a set:

PROPOSITION 5.13. The members of a set S of projections over a type T compose idempotently if and only if $\forall f, g \in S, \forall x \in T, f(x) = x \implies f(g(x)) = g(x)$. $\square$

PROOF. Suppose $\forall f, g \in S, \forall x \in T, f(x) = x \implies f(g(x)) = g(x)$. Then for any $y \in T, f(g(f(x))) = g(f(x))$. So $((g \circ f) \circ (g \circ f))(x) = g(f(g(f(x)))) = g(g(f(x))$. But g is idempotent, so $g(g(f(x)) = g(f(x))$. Hence, $g \circ f$ is idempotent.

Conversely, suppose $\exists f, g \in S, x \in T | f(x) = x \wedge f(g(x)) \neq g(x)$. Then $(g \circ f)(x) = g(f(x)) = g(x)$. But $((g \circ f) \circ (g \circ f))(x) = \mathrm{g(f(g(f(x))} = \mathrm{g(f(g(x))}$. Now f and g are partial identities, so $((g \circ f) \circ (g \circ f))(x) = g(f(g(x))) \sqsubseteq f(g(x)) \sqsubseteq g(x) = (g \circ f)(x)$. But we know $f(g(x)) \neq g(x)$, so we must have $((g \circ f) \circ (g \circ f))(x) \neq (g \circ f)(x)$. So $(g \circ f)$ is not idempotent. $\square$

Of course, this is not a *sufficient* condition, since we must also be sure that our set is closed under composition. It is easy to see how this condition is satisfied by the set of monotonic *downward closed* projections:

> If g is a partial identity then $g(x) \sqsubseteq x$, so if f is downward closed and $f(x) = x$ then $f(g(x)) = g(x)$.

This argument is not particularly dependent upon the properties of the usual domain ordering relation $\sqsubseteq$. We can use choose any ordering with certain characteristics and define properties analogous to partial identity and downward closure.

THEOREM 5.14. Suppose $\leq$ is a transitive, antisymmetric relation over T satisfying $\forall x, y \in T, x \leq y \implies x \sqsubseteq y$. Then let S be the set of monotonic idempotent functions satisfying:

1. $\forall f \in S, \forall x \in T, f(x) \leq x$ [pseudo-partial-identity]
2. $\forall f \in S, \forall x, y \in T, f(y) = y, x \leq y \implies f(x) = x$ [pseudo-downward-closure]

Then S is closed under composition and members of S commute. □

PROOF. Suppose $\leq$ is such a relation, and let S be as stated. We must show that the composition of any two members of S is in S. Let $f, g \in S$. Then:

Monotonicity: Monotonicity of $f \circ g$ is guaranteed by monotonicity of f and g, and of function composition.

Pseudo-partial-identity: For any $x \in T$, $(f \circ g)(x) = f(g(x)) \leq g(x) \leq x$, so $(f \circ g)$ is a pseudo-partial identity.

Pseudo-downward-closure: Suppose $x, y \in T$ with $(f \circ g)(y) = y$ and $x \leq y$. Then:

$$
\begin{aligned}
& y = (f \circ g)(y) \\
& \quad = f(g(y)) \\
& \quad \leq g(y) && \text{(since } f \text{ is a pseudo-partial-identity)} \\
& \quad \leq y && \text{(since } g \text{ is a pseudo-partial-identity)} \\
\implies & f(g(y)) = g(y) = y && \text{(by antisymmtery of } \leq\text{)} \\
\implies & f(x) = x \text{ and } g(x) = x && \text{(since } f \text{ and } g \text{ are pseudo-downwardly-closed)} \\
\implies & (f \circ g)(x) = f(g(x)) = f(x) = x
\end{aligned}
$$

So $(f \circ g)$ is pseudo-downward-closed.

Partial identity: Suppose $x \in T$. Then we know $((f \circ g)(x) \leq x$, which by the condition on our relation means that $((f \circ g)(x) \sqsubseteq x$.

Idempotence: For any $x \in T$, suppose $f(x) = x$. We know that g is a pseudo-partial-identity, so $g(x) \leq x$. Now f is pseudo-downward-closed, so this means that $f(g(x)) = g(x)$. So by Proposition 5.13, members of S compose idempotently,

So $f \circ g \in x$. Thus, S is closed under composition. Commutativity follows from Theorem 5.2. □

Note that there is no requirement that the relation $\leq$ be reflexive. If S includes the identity function then reflexivity becomes implicit, but it is not necessary to do so.

5.8. Intradependent Subtypes

Due to the time available, the content of this section is of a more intuitive, and less formal nature. All the material below would probably admit of further formalisation.

So we can choose an arbitrary relation $\leq$ over a type T and the composition closure of an arbitrary set S of idempotent functions over T, so long as S and $\leq$ satisfies the three conditions from Theorem 5.14. What should we choose? One of the major problems with downward closure is that it does not allow us to represent some apparently simple subtypes. The example we used previously was the subtype of *ordered lists*. Can we choose $\leq$ and S such that S contains a representative of this subtype?

Suppose we define $\leq$ on the type of lists integers by $x \leq y$ if and only if $x \sqsubseteq y$ and x contains no undefined *elements* (although it may have an undefined tail).

This relation certainly satisfies the conditions for applying Theorem 5.14, so there is a largest subset of our type of lists which satisfies the conditions of pseudo-partial-identity and pseudo-downward-closure with respect to $\leq$. Let S that set. Now ordered lists can be represented by the function defined in Haskell as follows:

```
ordered :: [Int] -> [Int]
ordered []     = []
ordered (x:xs) | seq x True = x : orderedabove x xs
       where orderedabove _ []              = []
             orderedabove x (y:ys) | x <= y  = y : orderedabove y ys
```

The `seq` expression is a standard Haskell function used to *force evaluation* of its first parameter, before returning its second.

It is intuitively easy to see that `ordered` satisfies the conditions of Theorem 5.14 with $\leq$, so it is in S. Indeed, so are most other useful subtypes of lists. Further, we know that members of S has the mathematical properties of composition closure, idempotence and commutativity.

But have we forgotten about the need for laziness? We concluded earlier that we could not implement `ordered` in a maximally lazy manner, because of the way in which it is *intradependent*. As we would expect, the `ordered` subtype we have just defined is *not* maximally lazy. The nature of the definition means that the subtype eagerly evaluates the `head` of a list, even if only asked for the `tail`. Semantically, the fixed points of the `ordered` function are all fully-defined ordered lists (finite or infinite), as well as ordered lists undefined only in the tail. Lists with undefined *elements* are not in the subtype. For example:

```
tail [undefined, 2]              = [2]
(tail . ordered) [undefined, 2]  = undefined
```

This property of "head-strictness" characterises those projections over lists which satisfy this requirements of Theorem 5.14 for this particular definition of the relation $\leq$. Hence, projections with the same characteristics will commute and compose idempotently.

The idea generalises to other algebraic and product types, with varying degrees of success. In general, the author believes that sets of projections over a compound type which are eager and strict with respect to particular components are characterised by particular relations, with which they satisfy the requirements of Theorem 5.14. Unfortunately, the time available did not permit the formalisation of this proposition, and its conclusive proof or disproof.

With more complex types, it transpires that there is often no single ordering relation which allows the expression of every intradependent subtype set we require. Of course, there is no guarantee of idempotent composition of members of *different* sets of subtypes which satisfy Theorem 5.14 with *different* relations.

Notice that we can define $\leq$ by $x \leq y$ if and only if x or fully defined or $x = \perp$. Now the conditions of Theorem 5.14 are satisfied if S is the set of all *totally eager* monotonic projections. In effect, we have reduced our system of subtypes to that which we would use in a language with an eager evaluation strategy.

Since deriving the above material, the author has become aware that theories exist which involve the modelling of types as relations, with membership corresponding to reflexivity. It sounds as though there may have some bearing on matters, but there was not sufficient time to investigate.

5.9. Conclusion

In this chapter, we have explored various possible functional representations of subtypes, seeking one that would best fit the requirements identified in Chapter 4. We started with the intuition that subtypes should be *partial identities*, and that in order to be computable they must be *monotonic*.

Of the mathematical properties we sought, the set of monotonic partial identities is closed under composition, but lacks idempotence and commutativity. The set of monotonic idempotent partial identities, or pretypes, is not closed under composition. However, we showed that any set of pretypes which *is* closed under composition also has the property of commutativity.

We can guarantee idempotent composition closure and commutativity by taking the set of *downward closed* monotonic projections over a type. Indeed, if subtypes are to be maximally lazy then they must be downward closed. Unfortunately, insisting upon downward closed severely limits the expressive power of our subtypes. Even apparently simple constraints, such as that a list must be ordered, cannot be represented. Also, while subtypes which apply postconditions to functions are downward closed, those that apply preconditions are not.

To provide more expressive power, we abandoned our requirement for maximal laziness. We looked for a generalisation of downward closure which would allow the implementation of intradependent subtypes. We found that we could take an arbitrary set of projections over a type T as subtypes, and expect them to have desirable mathematical properties, so long as there exists a *relation* over T with certain properties. This allowed us to express intradependent subtypes, such as that of *ordered lists* of a given type.

In widening our set of subtype functions to include some that are not downward closed, we lose the property of complete laziness. However, depending upon the relation chosen, we can guarantee that only certain components of a subtype are eagerly evaluated. So long as we are happy to consider elements with these component undefined as "non-members", our subtypes remain sound.

CHAPTER 6

Subtype Combinators

In this chapter, we put our ideas into practice by developing a library of *subtype combinators*. These will allow us easily to construct subtypes for safe, practical use in the lazy functional language Haskell 1.3.

We showed in Chapter 3 that only subtypes with the property of *downward closure* can be fully lazy. The range of subtypes which can be implemented in such a way is somewhat limited. We begin by creating a system which will allow us to construct downward closed subtypes only. To give ourselves more scope, we then proceed to relax the demand for complete laziness and implement some constructors for *intradependent subtypes*.

The full code of the combinator library we create is to be found in Appendix A. For the reader hungry for more examples of our combinators in use, Appendix B provides a small collection of miscellaneous subtypes.

6.1. Preliminaries

For readability of subsequent definitions, we follow [**Run95**] and make a static type declaration:

```
type SubtypeOf a = a -> a
```

Declaring a function to be of this type does *not*, of course, guarantee that it is a partial identity, or that it has any of the other nice properties we would like to associate with subtypes. The static type system of Haskell is not sufficiently powerful for us to express such concepts. It seems likely that a language with more a flexible static type system, such as Cayenne [**Aug99**], might allow us to express more precisely the restrictions we wish to place upon subtypes.

6.2. Flat Types and Complex Types

For the purposes of subtype construction, we divide types into two categories – *flat* types and *complex* types. Theorem 5.5 tells us that on flat types, all monotonic partial identities are downward closed and idempotent. On these types, we shall allow the user to specify an arbitrary subtype, in the form of a predicate.

On complex types, however, the requirements for the preservation of laziness and of other nice properties need to be much much more stringent. Subtypes of these types can be constructed only in limited ways, from known subtypes. To enforce this distinction, we define two Haskell *classes*:

```
class Subtypical t

class (Subtypical t) => FreeSubtypical t
```

The class `Subtypical` will contain *all* types of which subtypes are required. The class `FreeSubtypical` contains only the flat types. So which types should be in each class? Broadly speaking, the members of the Haskell `Enum` class are flat types [**Run95**]. We would like to declare:

```
instance Enum t => Subtypical t
instance Enum t => FreeSubtypical t
```

Unfortunately, for reasons of implementation, Haskell does not permit these declarations. Instead, we shall content ourselves with defining individual types to be members of the classes as and when we need to. For example:

```
instance Subtypical Int
instance FreeSubtypical Int
```

Following Runciman [**Run95**], we shall also declare some subtypes which are applicable to any type:

```
ok, ko :: (Subtypical t) => SubtypeOf t
ok = id
ko = const
       (error "Subtype violation: expression should not be evaluated")
```

The `ok` function over any given type represents the universal subtype — the subtype containing the entire type. It will be useful when a combinator requires us to specify a subtype for a component, the value of which we are not interested in. The `ko` function represents the subtype containing nothing but $\perp$. It can be used as a subtype for a "placeholder" component, which should never be evaluated.

6.3. Subtypes of Atomic Types

We have already observed that on types with flat domains we can safely allow the user to specify an arbitrary subtype with a *predicate*. We can facilitate this with a combinator defined as follows:

```
only :: (FreeSubtypical a) => (a -> Bool) -> SubtypeOf a
only p n | p n = n
```

EXAMPLE 6.1. The subtype of *even integers* can be defined with:

```
evenint :: SubtypeOf Int
evenint = only ((==0) . (‘mod‘ 2))
```

We can use our subtype to define a function which divides an even number by two, causing an error if it is presented with an odd number:

```
halve :: Int -> Int
halve = (‘div‘ 2) . evenint
```

Now `halve 4` and `halve -68` are `2` and `-34` respectively, but `halve 5` is undefined. □

We can also use `only` to define *dependent subtypes* — functions which return different subtypes, depending on their arguments. In doing so, we are effectively defining an entire set of subtypes at a stroke. Note that *dependent* subtypes are entirely different from the întradependent subtypes introduced in Chapter 5.

EXAMPLE 6.2. We can generalise `evenint` to create a function which takes an integer parameter n, and returns the subtype function representing the set of integers divisible by n:

```
divisibleby :: Int -> SubtypeOf Int
divisibleby n = only ((==0) . (`mod` n))
```

Now we can define a function to perform only exact integer division:

```
divide :: Int -> Int -> Int
divide m d = div (divisibleby d m) d
```

□

These subtypes certainly halt program execution upon detecting an error, but the error messages produced are not always as helpful as they could be. In HUGS 1.3, for example:

```
? divide 5 2

Program error: {only (flip v90 0 . flip v117 2) 5}
```

We can improve this behaviour by defining:

```
only_msg :: FlatSubtypical a
                        => String -> (a -> Bool) -> SubtypeOf a
only_msg _ p x | p x = x
only_msg s _ _       = error ("Subtype violation: " ++ s)
```

Now `divisibleby` can be re-expressed:

```
divisibleby n = only_msg ("value not divisible by " ++ show n)
                         ((==0) . (`mod` n))
```

The same problem in HUGS 1.3 now produces:

```
Program error: Subtype violation: value not divisible by 2
```

6.4. Subtypes of Tuple Types

An obvious way in which to specify a constraint upon a tuple is to specify constraints upon each of its components. For pairs, for example, we can define a combinator:

```
subPair :: (Subtypical a, Subtypical b) =>
                 SubtypeOf a -> SubtypeOf b -> SubtypeOf (a,b)

subPair s t (x,y) = (s x, t y)
```

Example 6.3. We can define the subtype of pairs of integers, the first of which is even and second divisible by three, as follows:

```
s :: SubtypeOf (Int,Int)
s  = subPair evenint (divisibleby 3)
```

Now `s (2,3)` is `(2,3)`, but `s (4,5)` is `(4,`$\perp$`)`. An attempt to evaluate the second component of the latter will cause an error. □

Example 6.4. We can define a subtype which expresses the idea that the second component of a pair should never be evaluated:

```
t :: (Subtypical a) => SubtypeOf (a, a)
t  = subPair ko ok
```

This constraint could be applied to the parameter of a function which is intended to be lazy in the second component of the pair, to provide a check that it really *is* lazy. □

Now we have a general way of constructing subtypes of pairs, provided that we have subtypes of the components. We can express this as follows:

```
instance (Subtypical a, Subtypical b) => Subtypical (a,b)
```

So how powerful is `subPair`? Of course, the answer depends upon the range of available component subtypes. Suppose that we have representations for every conceivable subtype of each component type. Then it is plain that `subPair` allows us to represent any subtype of pairs, provided that there is no intradependency between the components.

6.5. Subtypes of Lists

How can we specify subtype constraints upon a list? A list falls into one of three categories:

1. *Undefined*: Since our subtypes are to be represented by partial identities, the undefined list is a member of every representable subtype.
2. *Empty*: An empty list is either a member of a given subtype, or it is not. We can specify membership or non-membership with a Boolean flag.
3. *Non-empty*: Such a list is effectively a pair, consisting of an element and a list, so we can specify separate constraints upon each. Additionally, there is the possibility that no non-empty lists are permitted in our subtype at all. This we can represent with a Boolean flag.

With this in mind, a suitable combinator for lists would be as follows:

```
subList     :: (Subtypical a)
            => (Bool)                                     -- Empty
            -> (Bool, SubtypeOf a, SubtypeOf [a])         -- Cons
            -> SubtypeOf [a]

subList     (True)   _            []        = []
subList     _        (True, s, t) (x:xs)    = (s x) : (t xs)
```

The first argument relates to the empty list — it is simply a Boolean value indicating whether the empty list is a member of our subtype. The second argument is a tuple, explaining how the subtype should handle non-empty lists. It consists of a Boolean flag, indicating whether such lists are acceptable at all, and two subtype constraints, to be applied to the element and the list respectively.

EXAMPLE 6.5. We can use our new combinator to define the set of lists, all of whose members satisfy a particular constraint.

```
listOf :: (Subtypical a) => SubtypeOf a -> SubtypeOf [a]
listOf s = subList (True) (True, s, listOf s)
```

□

EXAMPLE 6.6. We can define a dependent subtype of lists *no longer than* a given length:

```
asshortas :: (Subtypical a) => Int -> SubtypeOf [a]
asshortas n = subList (True) (n!=0, ok, (asshortas (n-1)))
```

□

6.6. Subtypes of Algebraic Types

We can easily generalise the above treatment of lists, to arbitrary algebraic types. For each constructor of such a type, we want to specify whether the constructor is admissible in the subtype at all, and if so, what constraints must be applied to its parameters.

EXAMPLE 6.7. Consider the algebraic type:

```
data Animal = Cow Char
            | Iguana
            | Sloth String (Int, Int)
```

A suitable subtype constructor for this type would be:

```
subAnimal :: (Bool, SubtypeOf Char)                          -- Cow
          -> (Bool)                                          -- Iguana
          -> (Bool, SubtypeOf String, SubtypeOf (Int, Int)) -- Sloth
subAnimal (True,sa) _      _            (Cow a)     = Cow (sa a)
subAnimal _         (True) _            (Iguana)    = Iguana
subAnimal _         _      (True,sa,sb) (Sloth a b) = Sloth (sa a) (sb b)
```

We can now declare the subtype of cows, or that of Sloths with short strings:

```
cow        = subAnimal (False, ko) (True) (False, ko, ko)
shortSloth = subAnimal (False, ko) (False) (True, asshortas 3, ok)
```

Notice the use of `ko` to pad out tuples which begin with `False`. In theory it makes no difference what value we use, but by using `ko` we can perform a subtype check on the correctness of our subtype combinator! If the `ko` terms are evaluated, then an error will be raised to indicate that the combinator is not working correctly. □

Subtypes generated only with combinators of this kind are completely lazy. In a subtype definition, the value of each component in the result is seperately defined in terms only of that same component in the argument. Hence, it will never be necessary for the subtype to evaluate more components than are actually required "downstream".

They are also partial identities — each component is mapped with a subtype, which must map it to a less defined value. Similarly, they are idempotent, so long as all the functions supplied are idempotent. It follows from various results in Chapter 5 that these subtypes are downward closed, and hence they have the desired mathematical properties of idempotent composition closure and commutativity.

6.7. Deriving Combinators for Algebraic Types

It would be less than ideal if the user of our library had to write his own combinators by hand for each algebraic type. Mistakes might be made in doing so, and invalid subtypes introduced. These might then be combined using valid combinators to produce more invalid subtypes, compounding the problem.

Unfortunately, Haskell's type system is not sufficiently powerful to support a single definition polymorphic across different algebraic types. To circumvent this problem, Winstanley [**Win97**] has developed a Haskell preprocessor, called

DrIFT[1]. DrIFT scans a Haskell program looking for type declarations, and for special directives. The latter tell DrIFT to execute *rules* — Haskell functions within DrIFT which generate new Haskell code, appending it to the bottom of the input file. DrIFT is described in detail in [**Win97**]. The program itself, and associated documentation are available for download from the World Wide Web [**Dri**].

A DRiFT *rule* is a function which take as a parameter a structure containing information about a Haskell data type. It returns a data structure representing some Haskell source code, which is added to the program. DrIFT comes supplied with a selection of rules for common tasks, but we shall need to write our own. With an appropriate DrIFT rule, we would be able to write the following:

```
{-! for Colour derive : subtypes !-}

instance Subtypical Colour

data Colour = Transparent
            | RGB Int Int Int
```

DrIFT would then produce a definition equivalent to:

```
subColour :: (Bool)
          -> (Bool, SubtypeOf Int, SubtypeOf Int, SubtypeOf Int)
          -> SubtypeOf Colour

subColour (True) _       Transparent         = Transparent
subColour _ (True,a,b,c) (RGB Int Int Int)   = (a Int, b Int, c Int)
```

We have implemented an appropriate DRiFT rule, the code for which can be found in Appendix A.

6.8. Subtypes of Functions

In Chapter 3, we noted that one of the most useful features of subtypes in functional languages was their equivalence to preconditions and postconditions. We have not conducted extensive investigations into subtypes of functions, but we do know that we can define a downward closed argument-independent postcondition combinator:

```
postcond :: (Subtypical b) => SubtypeOf b -> SubtypeOf (a -> b)
postcond s = (s.)
```

The corresponding precondition combinator is not downward closed, but we know that the set of precondition and postcondition combinators commute and compose idempotently, so it should be safe to define them provided we do not use any other subtypes of functions.

```
precond :: (Subtypical a) => SubtypeOf a -> SubtypeOf (a -> b)
precond s = (.s)
```

We can now define Runciman's `->>` operator [**Run95**] in terms of `precond` and `postcond`:

```
(->>) (Subtypical a, Subtypical b) =>
           SubtypeOf a -> SubtypeOf b -> SubtypeOf (a->b)
```

[1] DeRiving Instances From Types. DrIFT was formerly known as Derive, but has been renamed for legal reasons.

```
a ->> b = (precond a) . (postcond b)
```

A further interesting idea is to define an argument-dependent postcondition combinator, as follows.

```
deppostcond :: (a -> SubtypeOf b) -> (SubtypeOf (a -> b))
deppostcond s f x = s x (f x)
```

This could be combined with `precond` to give a single "specifying operator" for a function:

```
(+>>) :: (SubtypeOf a) -> (a -> SubtypeOf b) -> (SubtypeOf (a -> b))
s +>> t = (precond s) . (deppostcond t)
```

In principle, a subtype constructed using this operator would be able to specify the correct operation of any function *exactly*. However, we have not had time to investigate thoroughly the mathematical properties of `deppostcond` and `+>>`, so we shall omit them from our library for now.

The whole area of subtypes of functions requires investigation in much more detail, so the functional subtypes we provide in our combinator library should be regarded as provisional.

6.9. Algebraic Types Revisited

We noted in Chapter 5 that downward closed projections can only represent a certain, very limited range of subtypes. In Section 5.8, we explored the possibility of using an analogous quality to that of downward closure, but with a relation other than the usual partial ordering $\sqsubseteq$. We can make use of this idea to develop more powerful combinators. For lists, for example, we could define:

```
depsubList :: (Subtypical a)
           => ([a] -> (Bool))
           -> ([a] -> (Bool, SubtypeOf a, SubtypeOf [a]))
           -> SubtypeOf [a]

depsubList f   _   []     | p     = []
    where (p) = f []

depsubList _   f   (x:xs) | p     = (s x) : (t xs)
    where (p,s,t) = f (x:xs)
```

`depsubList` is similar to `sublist`, except that the parameter relating to each constructor is not a tuple, but a function. Each function takes as a parameter a tree, and returns a tuple of the same nature as those passed to `subList`.

In theory, this function is powerful enough to represent arbitrary subtypes of lists. Each function passed to `depSublist` could be defined so as to consider exactly whether it wanted its parameter to be in the subtype, and return either `(False, ko, ...., ko)` or `(True, ok, ..., ok)` as appropriate. Even if we use `depsubList` in this rather heavy-handed way, we can still be sure that it will return a partial identity function. However, we may well lose the nice properties of commutativity and idempotence.

Our intention, though, is to treat `depsubList` rather more delicately. The easiest way to explain this is with an example. We could implement the subtype of *ordered lists* with `depsubList`, as follows:

```
ordered :: (Ord a, Subtypical a) => SubtypeOf [a]
ordered = depsubList (const True)
                     (\(x:_) -> (seq x True, ok, orderedabove x))

orderedabove :: (Eval a, Ord a, Subtypical a) => a -> SubtypeOf [a]
orderedabove x = depsubList (const True)
                    (\(y:_) -> (seq y (x <= y), ok, orderedabove y))
```

The definition of `ordered` says that the empty list is always ordered, and that the list `(x:xs)` is ordered if `xs` is ordered above `x`. The definition of `orderedabove` says that the empty list is always ordered above everything, and that the list `(y:ys)` is ordered above `x` if `x` is less than or equal to `y`, and `ys` is ordered above `y`.

It is reasonably straightforward to deduce that this definition simplifies to that of the `ordered` function given in Section 5.8. We recall that the latter evaluated the head eagerly but the tail lazily, and in fact there is a simple way to deduce these facts directly from the definition above.

Consider the lambda expression definining the second parameter to `depsubList` in the definition of `ordered`. We can see that the pattern has an underscore in the position for the tail of the list – the tail is discarded. From this, we can immediately infer that the subtype does *not* eagerly evaluate the tail of the list. We have used a `seq` expression to force evaluation of the head of the list in the returned tuple. The first component is evaluated as a *guard* by our subtype, so this guarantees that the subtype will be strict in the head of the list.

This, we hope, makes the subtype slightly easier to reason about. Notice that we use a `seq` expression in the definition of `orderedabove`, even though y is explicitly used in the same expression. This is because our ordered function is polymorphic, and we cannot guarantee that the ordering relation supported by every type will be strict in y!

Thus, our subtype is *sound* so long as we consider lists with undefined elements to be non-members. This seems quite reasonable.

The idea extends to more general algebraic types. Again, we have written a DrIFT rule to derive an appropriate intradependent subtype combinator. The code for this is to be found in Appendix B, and we shall see it in action in Chapter 7.

As discussed in Section 5.8, our belief is that subtypes with the same strictness properties will commute and compose idempotently. However, this needs more formalisation before we can depend upon it.

Given the uncertainty surrounding the mathematical properties of these combinators, and the fact that they are somewhat more complicated to use, we shall provide them in a separate Haskell module. This allows the user to choose whether or not to include them in his programs.

6.10. Conclusion

In this chapter, we have put our ideas for a subtyping system into practice, by constructing a subtype combinator library. We began by implementing combinators which produce downward closed subtypes. These are *completely lazy* and hence *sound*. They have nice mathematical properties, such as *commutativity* and *idempotent compositon*. However, they allow the expression only of a very limited range of subtypes.

We attempted to solve the latter problem by implementing a collection of what we termed *intradependent combinators*. These give us a great deal of expressive power, but at a price. Subtypes are no longer completely lazy, and the user must define and use them carefully to ensure soundness. The mathematical properties of such subtypes are not entirely clear, and still require further investigation.

CHAPTER 7

Subtypes in Action

In this chapter, we put our subtyping system to the test by applying it to a practical problem taken from the literature.

7.1. The Problem

In Section 3.7 we mentioned that we would use Reade's 2-3 tree subtypes as a benchmark for evaluating our system of subtype combinators. To recap, Reade [**Rea92**] introduces a new equational algorithm for the removal of values from 2-3 trees. His algorithm works in tandem with an insertion algorithm by Hoffman and O'Donnell [**HO82**] Reade makes extensive use of subtypes to prove the correctness of both the removal and the insertion algorithms.

Our intention is to attempt an implementation of the subtypes used in Reade's proof, using our subtype combinator library. It will then be possible to incorporate constraint checks into a program containing a Haskell implementation of the algorithm itself. This will provide further verification of correctness of both the algorithm and the implementation.

7.2. Preliminaries

Reade's algorithm is expressed as a set of equations, which translate very easily into a functional programming language. In Appendix C we provide a Haskell implementation. The type of 2-3 trees itself is implemented as a Haskell algebraic type. In addition to the standard `Empty`, `Tr2` and `Tr3` nodes, Reade's algorithm uses extra `Put` and `Taken` nodes for intermediate values.

```
data Tree23 = Empty
            | Tr2 Tree23 Int Tree23
            | Tr3 Tree23 Int Tree23 Int Tree23
            | Put Tree23 Int Tree23
            | Taken Tree23
```

To implement Reade's subtypes, we shall need combinators for both simple downward closed tree subtypes, and intradependent subtypes. We can use our DrIFT rules to derive them:

```
{-! for Tree23 derive : downsubtypes, depsubtypes !-}
```

The complete combinator definitions created by DrIFT can be found in Appendix C. Due to the rather complex nature of the type and the simple formatting technique employed by our DrIFT rule, they are rather difficult to read! We provide here a brief explanation of the action of the combinators.

```
subTree23 :: (Bool)
          -> (Bool, SubtypeOf Tree23, SubtypeOf Int,
                                          SubtypeOf Tree23)
```

```
              -> (Bool, SubtypeOf Tree23, SubtypeOf Int,
                    SubtypeOf Tree23, SubtypeOf Int, SubtypeOf Tree23)
              -> (Bool, SubtypeOf Tree23, SubtypeOf Int, SubtypeOf Tree23)
              -> (Bool, SubtypeOf Tree23)
              ->  SubtypeOf Tree23
```

`subTree23` takes five parameters, each of which is a tuple relating one of the constructors of `Tree23`. For each constructor, we can specify a Boolean value indicating whether that constructor is acceptable at all, followed by a collection of subtypes to apply to the components associated with the constructor. The returned value is a subtype of `Tree23`.

Provided the constraints passed to this function are idempotent, downward closed and completely lazy, then the subtypes generated will be likewise. It follows that they commute, and they compose to make subtypes with similar properties. Wherever possible, we would like to define tree subtypes with this first combinator.

```
depsubTree23 :: (Tree23 -> (Bool))
             -> (Tree23 -> (Bool, SubtypeOf Tree23,
                                  SubtypeOf Int, SubtypeOf Tree23))
             -> (Tree23 -> (Bool, SubtypeOf Tree23, SubtypeOf Int,
                            SubtypeOf Tree23, SubtypeOf Int,
                            SubtypeOf Tree23))
             -> (Tree23 -> (Bool, SubtypeOf Tree23, SubtypeOf Int,
                            SubtypeOf Tree23))
             -> (Tree23 -> (Bool, SubtypeOf Tree23))
             -> SubtypeOf Tree23
```

`depsubTree23` also takes five parameters, each a function relating to one of the constructors of `Tree23`. Each function takes as a parameter a tree, and returns a tuple of the same nature as those passed to `subTree23`. Again, the returned value is a subtype of `Tree23`. Subtypes generated with this technique are in general neither downward closed nor completely lazy. They are as eager as the functions passed to `depsubTree23`. Nor can we assume, for now, that they have nice mathematical properties[1].

7.3. The Subtype of Fully Defined Integers

Reade's first subtype definition is the following[2]:

DEFINITION 7.1. Let $\mathcal{L}$ be the set of fully-defined integers. □

As we observed in Chapter 5, $\perp$ is a fixed point of all partial identities. Hence, there is no way to represent a set of fully defined integers. Indeed with lazy subtyping, there would be no point in doing so, since we would never detect the problem until it caused a failure.

7.4. Normalised Trees

Reade defines the sets of *normalised trees*, of given depth and in general, as follows:

[1]Although the question of when they *do* is deserving of more study — cf. Sections 5.8 and 6.9.

[2]Throughout this chapter we use definitions equivalent to those used by Reade. However, for reasons of context, the definitions here are not necessarily verbatim quotations.

DEFINITION 7.2. Let $\mathcal{N}_k$, for $k \geq 0$, be the sets defined inductively by

$$\begin{aligned}\mathcal{N}_0 &= \{Empty\} \\ \mathcal{N}_{k+1} &= \{Tr2(t_1, a, t_2) | a \in \mathcal{L}, t_1, t_2 \in \mathcal{N}_k\} \\ &\quad \cup \{Tr3(t_1, a, t_2, b, t_3) | a, b \in \mathcal{L}, t_1, t_2, t_3 \in \mathcal{N}_k\},\end{aligned}$$

For convenience, we also define $\mathcal{N}_{-1} = \emptyset$. The set of all normalised trees is denoted

$$\mathcal{N} = \bigcup \{\mathcal{N}_i | i \geq 0\}$$

□

A *normalised tree* is essentially the same as the balanced tree we saw defined in Chapter 3. The definition is reformulated slightly to exclude the new `Put` and `Taken` nodes.

The recursive definition lends itself well to implementation with the combinator library. A normalised tree *of depth k* is easily implemented as a completely lazy, downward closed, recursive subtype, the structure of which almost exactly mirrors Reade's definition:

```
normdepth :: Int -> SubtypeOf Tree23
normdepth (-1) = ko

normdepth 0 = subTree23 (True)
                        (False, ko, ko, ko)
                        (False, ko, ko, ko, ko, ko)
                        (False, ko, ko, ko)
                        (False, ko)

normdepth k | k > 0 = subTree23 (False)
                    (True, normdepth (k-1), ok, normdepth (k-1))
  (True, normdepth (k-1), ok, normdepth (k-1), ok, normdepth (k-1))
                    (False, ko, ko, ko)
                    (False, ko)
```

More concisely, we can incorporate all three cases into a single definition:

```
normdepth k = | k >=(-1) = subTree23 (k==0)
                        (k>0, normdepth (k-1), ok, normdepth (k-1))
     (k>0, normdepth (k-1), ok, normdepth (k-1), ok, normdepth (k-1))
                        (False, ko, ko, ko)
                        (False, ko)
```

The parameters to `subTree23` represent the following properties of normalised trees:

1. The `Empty` tree is a normalised tree of depth k if and only if $k = 0$.
2. A tree with a `Tr2` node as its root is normalised with depth k if and only if $k > 0$ and each of its two child trees are normalised with depth $k - 1$. The contents of the integer data field is irrelevant.
3. A tree with a `Tr3` node as its root is normalised with depth k if and only if $k > 0$ and each of its three child trees are normalised with depth $k - 1$. The contents of the integer data fields are irrelevant.
4. A tree with a `Put` node at its root is not normalised.

5. A tree with a `Taken` node at its root is not normalised.

Implementing the type of *normalised trees* in general, with unspecified depth, proves less straightforward. We noted in Chapter 3 that subtyping in the language CIP-L was made easier by the availability of *quantifiers*. If we had an existential quantifier in Haskell, it would be easy to move from the specific cases of trees balanced with particular depths, to a more general case of trees balanced with any depth. Unfortunately, no such construct is available.

The problem for us is that a subtype of balanced trees has an essential intradependency between components. Previously, we have always been able to resolve such problems at minimal cost. One of the dependent components was always of a relatively simple nature, and eager evaluation of it did not present a problem.

In this case, it seems that the only solution is to eagerly calculate the depth of an arbitrary branch of the tree, so that we can used our normdepth function. For this, we need a small auxilliary:

```
depth :: Tree23 -> Int
depth (Empty)         = 0
depth (Tr2 t _ _)     = (depth t) + 1
depth (Tr3 t _ _ _ _) = (depth t) + 1
depth (Put t _ _)     = (depth t) + 1
depth (Taken t)       = (depth t) + 1
```

Now we can implement the general type of normalised lists as follows:

```
normalised :: SubtypeOf Tree23
normalised = depsubTree23
   (const True)
   (\(Tr2 t _ _) -> (True, normdepth (depth t), ok,
                     normdepth (depth t))
   (\(Tr3 t _ _ _ _) -> (True, normdepth (depth t), ok,
                         normdepth (depth t), ok,
                         normdepth (depth t))
   (const (False, ko, ko, ko))
   (const (False, ko))
```

This definition does what is required, although it eagerly evaluates the leftmost branch of the tree. Of course, if this branch of the tree were infinite, this would cause an infinite loop — semantically, the subtype would return $\perp$. This should not pose a problem, since we consider that an infinite tree cannot be balanced, and returning $\perp$ simply indicates non-membership of the subtype!

7.5. Put Trees and Taken Trees

DEFINITION 7.3. Let $\mathcal{T}$, for $k \geq 0$, be the set of normalised trees of depth k extended to allow for a possible `Put` node at the root where the subtrees are normal trees of depth k:

$$\mathcal{P}_k = \mathcal{N}_k \cup \{Put(t_1, a, t_2) | a \in \mathcal{L}, t_1, t_2 \in \mathcal{N}_k\}$$

□

DEFINITION 7.4. Let $\mathcal{T}$, for $k \geq 0$, be the set of normalised trees of depth k extended to allow for a possible `Taken` node at the root where the subtree is normalised with depth $k - 1$:

$$\mathcal{T}_k = \mathcal{N}_k \cup \{Taken(t) | t \in \mathcal{N}_{k-1}\} \qquad \text{(for } k \geq 0\text{)}$$

. □

`Put` trees and `Taken` trees are slight variations on normalised trees, where the root node can be a `Put` or `Taken` node respectively. They are easily implemented, in a way similar to that used for normalised trees:

```
putdepth :: Int -> SubtypeOf Tree23
putdepth k = subTree23 (k==0)
                       (k>0, normdepth (k-1), ok, normdepth (k-1))
        (k>0, normdepth (k-1), ok, normdepth (k-1), ok, normdepth (k-1))
                       (k>0, normdepth k, ko, normdepth k)
                       (False, ko)

takedepth :: Int -> SubtypeOf Tree23
takendepth k = subTree23 (k==0)
                         (k>0, normdepth (k-1), ok, normdepth (k-1))
        (k>0, normdepth (k-1), ok, normdepth (k-1), ok, normdepth (k-1))
                         (False, ko, ko, ko)
                         (k>0, normdepth (k-1))
```

7.6. Ordered Trees

Definition 7.5. The set $\mathcal{O}$ of all *ordered* trees is given by:

$$\mathcal{O} = \{\mathcal{B}_{i,j} | i, j \in \mathcal{L}, i < j\}$$

where, for all $i, j \in \mathcal{L}, i < j$

$$\begin{aligned} \mathcal{B}_{i,j} = &\{Tr2(t_1, a, t_2) | a \in \mathcal{L}, i < a < j, t_1 \in \mathcal{B}_{i,a}, t_2 \in \mathcal{B}_{a,j}\} \\ &\cup \{Tr3(t_1, a, t_2, b, t_3) | a, b \in \mathcal{L}, i < a < b < j, t_1 \in \mathcal{B}_{i,a}, t_2 \in \mathcal{B}_{a,b}, t_3 \in \mathcal{B}_{b,j}\} \\ &\cup \{Empty\} \end{aligned}$$

□

The subtype of *ordered trees* is intradependent, but this time the dependencies involve the integer data members of the tree. Reade's trees are not designed to hold undefined values, so eager evaluation of these should not pose a problem.

If we implemented the subtype using Haskell's integer type, we would find that we need four similar but distinct subtypes. These would represent trees *ordered in general*, *ordered between two given numbers*, *ordered below a given number* and *ordered above a given number*. We can eliminate the need for four distinct subtypes with a simple extension of the type of integers:

```
data InfInt = Finite Int
            | PlusInfinity
            | MinusInfinity
            deriving Eq
```

The presence of `deriving Eq` causes the Haskell interpreter to derive an equality[3] test operator (`==`) for our type. We shall also need ordering relations. Haskell's

[3] Mathematically this definition is a little dubious, since it does not really make much sense to say that $\infty = \infty$. This need not concern us here, though, since we shall never actually used the equality test between infinities.

`Ord` class is such that we need specify only one ordering relation. The others will be automatically deduced.

```
    instance Ord InfInt where
        MinusInfinity <= _        = True
        _ <= PlusInfinity         = True
        _ <= MinusInfinity        = False
        PlusInfinity <= _         = False
        (Finite x) <= (Finite y) = x <= y
```

Now we can define the subtypes of ordered trees in general and between two values, as follows:

```
ordered :: SubtypeOf Tree23
ordered = ordbet MinusInfinity PlusInfinity

orderedbetween :: Int -> Int -> SubtypeOf Tree23
orderedbetween x y :: ordbet (Finite x) (Finite y)

ordbet :: InfInt -> InfInt -> SubtypeOf Tree23
ordbet x y = depsubTree23
        (const True)
        (\(Tr2 _ n _) -> (x <= Finite n && Finite n <= y,
                          ordbet x (Finite n), ok, ordbet (Finite n) y))
        (\(Tr3 _ m _ n _) -> (x <= Finite m && Finite m <= Finite n
                                                           && Finite n <= y,
                          ordbet x (Finite m), ok,
                          ordbet (Finite m) (Finite n),
                          ok, ordbet (Finite n) y))
        (const (False, ko, ko, ko))
        (const (False, ko))
```

Reade also defines a larger subtype of ordered trees which allow `Put` nodes and `Taken` nodes at the root. A definition for this type would be related to the definition of `ordered` in much the same way that the definitions of `putdepth` and `takendepth` are related to the definition of `normdepth`.

7.7. Trees Containing or Not Containing a Given Value

Reade formally defines his subtypes $\mathcal{C}_n$ and $\mathcal{D}_n$ in terms of his `member23` function, which he does not provide. For our purposes an informal definition will suffice. Broadly speaking, $\mathcal{C}_n$ is the subtypes of trees (without `Put` and `Taken` nodes, but not necessarily balanced or ordered) which contain the value n. $\mathcal{D}_n$ is the subtype of trees, (with the same restrictions) which do *not* contain the value n.

The subtype of trees which do *not* contain a particular value is easy to implement:

```
notcontains :: Int -> SubtypeOf Tree23
notcontains x = depsubTree23
     (const False)
     (\(Tr2 _ n _) -> (n /= x,
                       notcontains x,
                       ok,
                       notcontains x)
```

```
    (\(Tr3 _ m _ n _) -> (m /= x && n /= x,
                          notcontains x,
                          ok,
                          notcontains x,
                          ok,
                          notcontains x)
    (const (False, ko, ko, ko))
    (const (False, ko))
```

Unfortunately, the subtype of trees which do *contain* a particular value proves impossible to implement with any degree of laziness. Without an ordering property on our tree, we have *total intradependence*. Whether each node is permitted *not* to contain the value depends entirely on whether any other node in the tree contains it! The only way to implement this subtype would be with an exhaustive, eager search of the tree:

```
eagercontains :: Int -> SubtypeOf Tree23
eagercontains x = depsubTree23
    (const False)
    (\t -> (member23 t x, ok, ok, ok))
    (\t -> (member23 t x, ok, ok, ok, ok, ok))
    (const (False, ko, ko, ko))
    (const (False, ko))
```

What we *can* easily implement is a "contains" check for trees known (or assumed) to be ordered. In many circumstances, this will suffice.

```
contains :: Int -> SubtypeOf Tree23
contains x = depsubTree23
    (const False)
    (\(Tr2 _ n _) -> (True,
                      if x < n then (contains x) else ok,
                      ok,
                      if x > n then (contains x) else ok))
    (\(Tr3 _ m _ n _) -> (True,
                          if x < m then (contains x) else ok,
                          ok,
                          if m < x && x < n then (contains x) else ok,
                          ok,
                          if n < x then (contains x) else ok))
    (const (False, ko, ko, ko))
    (const (False, ko))
```

7.8. Subtypes of Functions

In Definition 3.1 we introduced Reade's $\rightarrow$ subtype combinator. To recap, if X and Y are subtypes of static types S and T respectively, then $X \rightarrow Y$ is the subtype of functions $f : S \rightarrow T$ satisfying the condition that if $x \in X$ then $f(y) \in Y$. We also saw, in Chapter 3, that this operator is not the same as Runciman's `->>`, which we have available in our combinator library.

For example, Reade uses his operator to express the fact that his function `remove`, when applied to an empty tree, returned an empty tree:

$$remove \in \mathcal{L} \rightarrow \mathcal{N}_0 \rightarrow \mathcal{N}_0.$$

However, `remove` is certainly not a fixed point of our subtype:

```
ok -> (normdepth 0) -> (normdepth 0)
```

That would imply that it was an error to call `remove` with any value other than an empty tree!

It might be possible to implement subtypes closer to those defined with Reade's arrow operator, using the `+>>` combinator suggested in Section 6.8. This is deserving of more investigation. However, in the absence of such a combinator, we shall refrain from attempting to implement subtypes of functions as such. Instead, we shall use the `->>` operator simply as a useful tool for applying subtypes to function arguments and results.

7.9. Testing and Integrating the Subtypes

There has not been time to test the subtypes very exhaustively. However, brief tests have suggested that they work as intended. Trials with various small trees have suggested that all of our subtypes recognise the values we expect, and preserve laziness to the extent we expect. For example, we can check that the `ordered` subtype is strict only in the integer components of a tree by evaluating expressions such as:

```
leftrootelement (ordered (Tr3 undefined 1 undefined 3 undefined))
```

There are many ways in which the subtypes we have defined could be employed to provide run-time validation of programs using Reade's algorithm. For example, consider the following expression:

```
foldr remove23 (foldr insert23 Empty (take 100 primes))
               (take 100 primes)
```

This, of course, should have the value `Empty`, and indeed with our implementation of Reade's subtypes, it does. Assuming that the list `primes` has already been evaluated past the 100th element, evaluation of this expression takes 9,120 reductions and 19,303 cells. With a check added for normalisation after every call to `insert23` and `remove23`, this becomes takes 298,876 reductions and 1,205,627 cells! Adding an check for ordering instead of normalisation has a similar effect (234,877 reductions and 836,695 cells).

This apparently large increase is perhaps not surprising, since `insert23` and `remove23` are very simple functions, and these are relatively complex subtype constraints. Checking that each call to `insert23` has actually inserted the appropriate value, using our `contains` subtype that assumes an ordered tree, only doubles the number of reductions and cells required for evaluation of the above expression.

In practice, we would probably only want to test such a complex constraint this frequently during an initial testing phase of a simple function. For these purposes, the performance impact is perhaps bearable. But it is probably too great for the subtypes of this kind to stay in the program permanently.

A single test of ordering, between building and deconstructing the tree takes only 11,429 reductions and 27,621 cells. Adding an normalisation constraint as well increases the total to 14,339 reductions and 39,627 cells.

Our subtypes also seemed capable of detecting errors. For example, we introduced an elementary "error" into our implementation of Reade's algorithm, by transposing two variable names in an auxilliary function definition, used by `remove23`. This caused the above expression, with subtype checking, to evaluate to `Tr3 Empty 229 Empty 157 Empty`. The incorporation of an `ordered` constraint check after each call to `remove23` detected the error, as expected.

Other tests produce similar results. In general, the subtypes seem to work as expected. They have a large impact on performance when applied as postconditions to small, frequently used functions. Of course, applying the checks less frequently reduces the impact on performance.

7.10. Conclusion

We set out to implement Reade's subtypes in practice, using our subtype combinator library. With regard to his subtypes of *trees*, we were quite successful. Most subtypes were implemented easily, in a manner that closely mirrored the form of the original definition. Only the subtype of *all trees containing a given value* caused insurmountable problems, forcing us to completely abandon laziness and eagerly search the entire tree. In fact, it is plain that there can be no lazy way to carry out such a check. Even if our subtype could be "suspended", using a method such as that suggested in Section 5.5, a lazy subtype would only be able to detect the absence of a given element when the entire tree had been evaluated.

We had less success with other subtypes. Reade's subtypes of functions were defined with a combinator of an entirely different nature to the `->>` combinator implemented in our subtype library. More research needs to be done into whether it will be possible to implement subtypes of this nature by some other means. Also, we were unable to implement Reade's subtype of fully-defined integers, although this is of no consequence in practice.

When we tried using our subtypes, the impact upon program performance was, as we might expect, highly dependent on the frequency with which we check constraints. By applying constraints as postconditions to simple, frequently-used functions, we increased the number of reductions and cells required many times over. Restricting constraint checks to only after larger operations predictably produced better performance.

CHAPTER 8

Conclusion

8.1. Evaluation

Our intention in this project was to implement subtypes as functions in a lazy functional language, to provide an extra level of program verification at run-time. In Chapter 4, we identified six requirements which we hoped our subtyping system would satisfy. We shall evaluate our subtyping system with respect to each of these criteria.

We demonstrated beyond doubt the *practicability* of our system of subtypes in Chapter 6, with a successful implementation in the lazy functional language Haskell 1.3.

The *efficiency* of the system is difficult to assess without more extensive testing than we have been able to conduct. In the example in Chapter 7, we had a wide degree of choice as to how frequently we checked constraints. Addition of subtype testing increased reductions and cells by anything from 5% to 2000% depending upon where constraint checks were placed.

These figures could certainly be improved. Optimisations could be made to the implementation of the combinators and the subtypes. Also, optimising transformations could be applied after the incorporation of subtypes. It is worth noting, though, that the more often we apply transformations to our subtypes, the less sure we can be of their correctness. Ultimately, if the program is correct, it may be possible by process of reasoning to transform constraint checks out of existence!

Whether the performance penalty imposed by subtype checking is acceptable for any given application depends, of course, on the nature of the application. However, it seems probable that most programs will resemble our example in having a wide choice of positions in which constraint checks could be implemented. This might allow an appropriate compromise to be found between efficiency and thorough checking.

Downward closed subtypes and intradependent subtypes strike different balances between the remaining four criteria of *ease of use*, *ease of reasoning*, *sufficient power* and *soundness*. We shall assess separately with respect to these requirements our systems for subtyping with *downward closed combinators* alone, and with *intradependent combinators* as well.

8.2. Evaluation of the Downward Closed Subtyping System

If we restrict our attention to downward closed combinators, then subtypes are *completely lazy*. If correctly implemented and used, then they will do nothing to impede the operation of a correct program. Subtypes generated with these combinators, then, are completely *sound*.

Such subtypes also have all the nice mathematical properties we hoped for — they are *idempotent*, *commutative* and *closed under composition*. Their *ease of reasoning*, then, should be good.

It is always difficult for the designer of a system to assess the *ease of use* of his own product. It is difficult to see the product from the perspective of one who is not intimately familiar with it! However, in so far as the author *can* assess, the downward closed subtyping system is quite simple to use. The user can employ library functions to create and combine subtypes arbitrarily, without worrying about the soundness of the resulting types. Using a DrIFT rule, he can derive subtype combinators for his own algebraic data types, with the same properties as those supplied for standard types.

The author found the process of defining subtypes with the combinators extremely intuitive. The structure such definitions often mirrors closely the way in which subtype sets are defined abstractly, so implementation becomes an extremely straightforward process.

Unfortunately, the downward closed subtyping system does not perform well at providing *sufficient power*. Of compound types, only relatively simple subtypes with no dependence between components, can be implemented. Even such an apparently uncomplicated constraint as that a list should be ordered is not expressible within the system.

We can define a limited range of subtypes of functions. We can define the subtype of functions satisfying a given argument-independent postcondition. However, if we want to retain downward closure we cannot define the subtype of functions satisfying a given precondition. Further research is needed in this area.

8.3. Evaluation of the Intradependent Subtyping System

Once we bring in the *intradependent combinators*, laziness of subtypes is no longer complete. It is still possible to ensure that subtypes are *sound*, but the onus is now on the user to ensure that eagerness in subtypes is limited to components which can safely be evaluated.

Analogous remarks apply to *ease of reasoning*. Intradependent subtypes do not, in general commute or compose idempotently, although they may do so in certain, special circumstances. This issue is in need of further investigation.

With regard to *ease of use*, the programmer still has standard combinators, and can still derive custom ones with DrIFT. Now, though, he must think carefully about how he employs them, if the subtypes created are not to be overly eager. However, we showed in Section 6.9 how he can ensure laziness in particular components, by discarding their values in lamba expressions.

More positively, the intuitive nature of subtype definition, at least as perceived by the author, still seems to remain when using the intradependent combinators.

The advantage of the intradependent subtyping system is that it provides the maximum possible degree of expressiveness. In theory, we can use the intradependent combinators to define an arbitrary projection over a given compound type. We have made no more progress, though, on the issue of subtypes of functions.

8.4. Conclusion

We have discovered that our original objectives, as identified in Chapter 4 were not entirely compatible. We can have complete laziness and nice mathematical properties, or we can have a great degree of expressiveness, but we cannot have both.

In an attempt to resolve this problem, we have implemented a system whereby subtypes are generated by two sets of combinators. If we restrict ourselves to the first set, we get a subtyping system which is completely lazy, easy to use and easy to reason with, but we cannot express even some apparently simple subtypes. If we use both sets, we can express a much larger range of subtypes, but the mathematical properties of our subtypes are less clear, and the user must take steps to ensure sufficient laziness.

In practice, the system seems to work quite well. in Chapter 7 we applied it to a practical problem, and found it easy to implement most of the subtypes we wanted in a reasonably lazy manner. Of those we could *not* implement lazily, one was by nature impossible to check without eager evaluation. One would have been rendered useless by the lazy evaluation strategy. The others were subtypes of functions, where more research is needed.

8.5. Suggestions for Future Work

There are many theoretical issues deserving of further investigation. We have explained only intuitively the exact circumstances under which subtypes generated using the intradependent combinators can be expected to commute, and to compose idempotently. This needs to be mathematically formalised, probably using Theorem 5.14. Also, we have barely scratched the surface where subtypes of functions are concerned. Much more work could be carried out in this area.

On a practical level, there is also much work that could be carried out. The efficiency of the subtype combinators — both the standard ones and those generated with DrIFT — could doubtless be improved. The DrIFT rules could be extended to support Haskell's named record types.

An alternative approach would be implement a similar subtyping system in a language with a more powerful static type system, such as Cayenne [**Aug99**]. Could we allow the programmer to define arbitrary subtype functions, and then let the static type checker *prove* that they had the required properties? As mentioned in Chapter 3, research [**Bet98**] has been carried out into subtyping in Cayenne, but it was impossible to obtain this in time for consideration in this report.

Yet another possibility is that of providing support for subtype projections in compilers and interpreters. We briefly touched upon the extra options this might make available in Section 5.5.

8.6. Acknowledgements

I'm particularly indebted to Malcolm Wallace, who converted DrIFT to work with HUGS 1.3 for me. Thanks are also due to Kirsty, for putting up with me when I babbled about domain theory. For his explanations of the techniques of lazy evaluation, I would like to thank $\perp$. Last, but not least, I must thank my supervisor Colin Runciman, for his many helpful comments and suggestions. Responsibility for all errors, of course, remains my own.

APPENDIX A

Source Code for the Subtype Combinator Library

A.1. Downward Closed Subtype Combinators

```
File: DownSubtypes.lhs

This (literate) Haskell script contains definitions for downward
closed subtype combinators.

> module DownSubtypes where

The ->> function subtype combinator is right-associative. I don't entirely
know why the declaration of this fact needs to be up here at the top of
the file, but it doesn't work otherwise.

> infixr 5 ->>

First, a declaration of the type of subtypes. Remember that declaraing
a function to be of this type does not guarantee that it has the
properties required of a subtype!

> type SubtypeOf t = t -> t

"Subtypical" is the class of types for which subtypes can be defined.
Standard members are the basic subtypes ok and ko, which contain
everything and nothing respectively.

> class Subtypical t where
>    listof :: (SubtypeOf t) -> (SubtypeOf [t])
>    ok, ko :: SubtypeOf t

>    listof = map
>    ok = id
>    ko _ = error "ko subtype: possible placeholder argument evaluation"


If we have some types with subtypes, then we can deduce subtypes for
pairs, lists and functions over those types.

> instance (Subtypical s, Subtypical t) => Subtypical (s -> t)
> instance (Subtypical s) => Subtypical [s]
> instance (Subtypical a, Subtypical b) => Subtypical (a,b)


"FreeSubtypical" is the class of types for which we can use "only" to
construct subtypes - basically the flat types. "only" and "only_msg"
are declared in the class, simply so that we can overload the latter
to provide a more helpful error message. No other overloading should
be used.
```

```
> class (Subtypical t) => FreeSubtypical t where
>    only :: (t->Bool) -> SubtypeOf t
>    only = only_msg "only: Subtype violation"
>
>    only_msg :: String -> (t->Bool) -> SubtypeOf t
>    only_msg _ p x | p x = x
>    only_msg s _ _ = error ("Subtype violation: " ++ s)

-------------------------------------------------------------------
-------------------------------------------------------------------
Next comes the downward closed list combinator.

> subList     :: (Subtypical a) => (Bool)
>             -> (Bool, SubtypeOf a, SubtypeOf [a])
>             -> SubtypeOf [a]

> subList     (True)   _             []        = []
> subList     _       (True, s, t)    (x:xs)     = (s x) : (t xs)

-------------------------------------------------------------------
-------------------------------------------------------------------
Now the combinators for pairs and triples. If you want bigger
tuples, it shouldn't be difficult to add them.

> subPair   :: (Subtypical a, Subtypical b)
>           => SubtypeOf a -> SubtypeOf b -> SubtypeOf (a,b)
>
> subPair s t (x, y) = (s x, t y)

> subTriple   :: (Subtypical a, Subtypical b, Subtypical c)
>             => SubtypeOf a -> SubtypeOf b -> SubtypeOf c
>             -> SubtypeOf (a,b,c)
>
> subTriple s t u (x,y,z) = (s x, t y, u z)


-------------------------------------------------------------------
-------------------------------------------------------------------
Finally, the combinators for subtypes of functions. Of these, only
postcond-generated subtypes are downward closed, but the others are
commutative, idempotent and closed under composition. The issue of
subtypes of function needs a lot of thought, so these (except perhaps
postcond) should perhaps be regarded as a provisional suggestion.

> postcond :: (Subtypical a) => SubtypeOf a -> SubtypeOf (b -> a)
> postcond s = (s.)

> precond :: (Subtypical a) => SubtypeOf a -> SubtypeOf (a -> b)
> precond s = (.s)

> (->>)  :: (Subtypical a, Subtypical b) =>
                 SubtypeOf a -> SubtypeOf b -> SubtypeOf (a->b)
> a ->> b = (precond a) . (postcond b)
```

A.2. DrIFT Rule for Downward Closed Subtype Combinators

```
-- DrIFT rule to generate downward closed subtype combinators for
-- algebraic types. This could be made a lot more efficient and
-- elegant, but it does the job! Hopefully it's self-explanatory.

dst :: Data -> Doc
dst d = typeDec $$ empty $$ eqnDecs
    where outName :: Doc
          outName = text ("sub" ++ (name d))

          typeDec :: Doc
          typeDec =     outName
                     <> text " :: "
                     <> nest 1 (
                         (vcat . map typeTuple . body) d
                         $$ text ("SubtypeOf " ++ name d)
                     )
          typeTuple :: Body -> Doc
          typeTuple b =    text "(Bool"
                        <> hcat (map typeTupleElem (types b))
                        <> text ") -> "

          typeTupleElem :: Type -> Doc
          typeTupleElem t = text ", SubtypeOf"
                            <#> typename t

          eqnDecs :: Doc
          eqnDecs = vcat (map (eqnDec (body d)) [1..(length (body d))])

          eqnDec :: [Body] -> Int -> Doc
          eqnDec bs i =      eqnLeft bs i
                         <#> eqnMiddle (bs!!(i-1))
                         <#> equals
                         <#> eqnRight (bs!!(i-1))

          eqnLeft bs i =     outName
                         <#> (hsep (zipWith (blankAllButPos i) [1..] bs))

          eqnMiddle b =     text ('(' : constructor b)
                        <#> hsep (varNames' (types b))
                        <#> char ')'

          eqnRight b =      text (constructor b)
                        <#> hsep (zipWith applyList (varNames (types b)) (varNa$

          applyList x y = (text "(") <> x <#> y <> (text ")")

          blankAllButPos :: Int -> Int -> Body -> Doc
          blankAllButPos i j b | i == j = eqnTerm (types b)
          blankAllButPos _ _ _          = text " _"

          eqnTerm :: [Type] -> Doc
          eqnTerm ts = parenList ((text "True") : varNames ts)


-- I'm sure DrIFT must come with a function to do this, but if so I
```

```
-- can't find it!

typename :: Type -> Doc
typename (Arrow a b) = typename a <#> text "->" <#> typename b
typename (Apply a b) = typename a <#> char ' ' <#> typename b
typename (Con a) = text a
typename (Var a) = text a
typename (List a) = char '[' <> typename a <> char ']'
typename (Tuple as) = (parenList . map typename) as
```

A.3. Intradependent Subtype Combinators

```
File: DepSubtypes.lhs

This file contains combinators for intradependent subtypes. Currently
only lists are included.

> module DepSubtypes where

> import DownSubtypes

--------------------------------------------------------------------
Here are the nondownward closed list combinators. For usage
instructions, see the guide to usage.

> depsubList :: (Subtypical a) => ([a] -> (Bool))
>                              -> ([a] -> (Bool, SubtypeOf a, SubtypeOf [a]))
>                              -> SubtypeOf [a]
>
> depsubList f    _    []     | p     = []
>     where (p) = f []
>
> depsubList _    f   (x:xs) | p     = (s x) : (t xs)
>     where (p,s,t) = f (x:xs)
```

A.4. DrIFT Rule for Intradependent Subtype Combinators

```
-- This DrIFT rule generates a combinator for intradependent subtypes of
-- an algebraic type. It uses the "typenames" functions, declared with
-- dst, above.

idst :: Data -> Doc
idst d = typeDec $$ eqnDecs
    where outName :: Doc
          outName = text ("depsub" ++ (name d))

          mainTypename :: Doc
          mainTypename = text (name d)

          typeDec :: Doc
          typeDec =      outName
                     <> (text " :: ")
                     <> nest 1 (
                          (vcat . map itypeTuple . body) d
```

```
                 $$ text ("SubtypeOf " ++ name d)
           )

itypeTuple :: Body -> Doc
itypeTuple b =   text "("
              <> mainTypename
              <> text " -> (Bool"
              <> (hcat . map typeTupleElem . types) b
              <> text ")) -> "

typeTupleElem :: Type -> Doc
typeTupleElem t = text ", SubtypeOf"
              <#> typename t

eqnDecs :: Doc
eqnDecs = vcat (map (eqnDec (body d)) [1..(length (body d))])

eqnDec :: [Body] -> Int -> Doc
eqnDec bs i =      eqnLeft bs i
               <#> eqnConsPattern (bs!!(i-1))
               <#> text "| p"
               <#> equals
               <#> eqnRight (bs!!(i-1))
               $$  eqnWhere (bs!!(i-1))

eqnLeft bs i = outName
           <#> hsep (zipWith (blankAllButPos i) [1..] bs)

eqnConsPattern b = text ('(' : constructor b)
               <#> hsep (varNames' (types b))
               <#> char ')'

eqnRight b = text (constructor b)
         <#> hsep (zipWith applyList (varNames (types b)) (varNames' $

eqnWhere b   = text "    where"
           <#> parenList ((text "p") : (varNames (types b)))
           <#> equals
           <#> text " f "
           <#> eqnConsPattern b


applyList x y = (text "(") <> x <#> y <> (text ")")

blankAllButPos :: Int -> Int -> Body -> Doc
blankAllButPos i j b | i == j = text " f"
blankAllButPos _ _ _          = text " _"
```

APPENDIX B

Miscellaneous Subtype Definitions

B.1. Integer Subtypes

```
-- Some examples of integer subtypes

> module IntSubtypes where

First we must import modules containing the combinators, and some
polymorphic subtype definitions.

> import OrdSubtypes
> import DownSubtypes


Now we declare that we want subtypes of type "Int", and we want to
be able to define them with "only". This is safe, since "Int" is a
flat type - there are no partially defined values.

> instance Subtypical Int
> instance FreeSubtypical Int


We can use the standard "ordered" subtypes to declare some common
types:

> positive, negative, nonnegative, nonpositive, digits, natural :: SubtypeOf Int
> positive    = greaterthan 0
> negative    = lessthan 0
> nonnegative = nolessthan 0
> nonpositive = nogreaterthan 0

> digits      = inclosedinterval 0 9

> natural = nonnegative


Next we have a dependent subtype of numbers with a given remainder o
division by something else.

> remainder :: Int -> Int -> SubtypeOf Int
> remainder r b = only (\x -> (mod x b == r))

Numbers divisible by a particular number:

> divisibleby :: Int -> SubtypeOf Int
> divisibleby = remainder 0

Even and odd numbers. "even" and "odd" are predicates in the Haskell
```

prelude, so we choose different names.

```
> evenint = divisibleby 2
> oddint = remainder 1 2
```

Here are subtypes of (positive) units, primes and composites. These are provided for illustration only - for real use one would want a rather more efficient primality test!

```
> unit, prime, composite :: SubtypeOf Int
> unit        = only (==1)
> prime       = inordlist primes . positive
> composite   = notinordlist primes . greaterthan 1
```

The definition for the list of primes comes from Bird and Wadler (see bibliography).

```
> primes :: [Int]
> primes = map head (iterate sieve [2..])
> sieve (p : xs) = [ x | x <- xs, x ‘mod‘ p /= 0]
```

B.2. List Subtypes

```
> module ListSubtypes where

> import DownSubtypes
> import DepSubtypes
> import IntSubtypes
> import OrdSubtypes

> aslongas, asshortas :: (Subtypical a) => Int -> SubtypeOf [a]
> aslongas 0 = ok
> aslongas n = subList (False) (True, ok, (aslongas (n-1)))

> asshortas 0 = subList (True) (False, ko, ko)
> asshortas n = subList (True) (True, ok, (asshortas (n-1)))

> infinite :: (Subtypical a) => SubtypeOf [a]
> infinite = subList (False) (True, ok, infinite)

> listOf :: (Subtypical a) => SubtypeOf a -> SubtypeOf [a]
> listOf s = subList (True) (True, s, listOf s)
```

Below this point are intradependent subtypes. Use with care!

This is the dependent subtype of lists ordered according to any relation.

```
> totalordered :: (Eval a, Subtypical a) => (a -> a -> Bool) -> SubtypeOf [a]
> totalordered r = depsubList (const True)
>                       (\(x:_) -> (seq x True, ok, totalorderedabove r x))
>
> totalorderedabove :: (Eval a, Subtypical a) => (a -> a -> Bool)
>                                          -> a -> SubtypeOf [a]
> totalorderedabove r x = depsubList (const True)
>                  (\(y:_) -> (seq y (r x y), ok, totalorderedabove r y))
```

Now we can easily define the set of conventionally ordered lists....

```
> ordered :: (Eval a, Ord a, Subtypical a) => SubtypeOf [a]
> ordered = totalordered (<=)

....or of lists ordered the other way....

> decreasing :: (Eval a, Ord a, Subtypical a) => SubtypeOf [a]
> decreasing = totalordered (>=)

....or of lists of integers, where each integer is a factor of the
previous one....

> factorList :: SubtypeOf [Int]
> factorList = totalordered (\r x -> mod x r == 0)
```

B.3. Subtypes of Ordered Types

```
-- Some examples of subtypes of ordered types

> module OrdSubtypes where

First we must import the file containing the subtype definitions.

> import DownSubtypes

We declare some dependent subtypes which might be useful for any
ordered type. We start with unbounded ranges:

> nolessthan       :: (Ord a, FreeSubtypical a) => a -> SubtypeOf a
> nolessthan        = only . (<=)

> nogreaterthan    :: (Ord a, FreeSubtypical a) => a -> SubtypeOf a
> nogreaterthan     = only . (>=)

> greaterthan      :: (Ord a, FreeSubtypical a) => a -> SubtypeOf a
> greaterthan       = only . (<)

> lessthan         :: (Ord a, FreeSubtypical a) => a -> SubtypeOf a
> lessthan          = only . (>)


And now some ranges:

> inopeninterval        :: (Ord a, FreeSubtypical a) => a -> a -> SubtypeOf a
> inopeninterval a b     = (greaterthan a) . (lessthan b)

> inclosedinterval      :: (Ord a, FreeSubtypical a) => a -> a -> SubtypeOf a
> inclosedinterval a b   = (nolessthan a) . (nogreaterthan b)

> notinopeninterval        :: (Ord a, FreeSubtypical a) => a -> a -> SubtypeOf a
> notinopeninterval a b  = only (\x -> x <= a || x >= b)

> notinclosedinterval        :: (Ord a, FreeSubtypical a) => a -> a -> SubtypeOf a
> notinclosedinterval a b    = only (\x -> x < a || x > b)


The "between" subtype is provided for readability and for compatibility
```

with some earlier definitions I wrote. With hindsight, its use is probably best avoided because of the ambiguity between the open and closed intervals!

```
> between :: (Ord a, FreeSubtypical a) => a -> a -> SubtypeOf a
> between = inopeninterval
```

The following subtypes allow the definition of things that are, or are not, in a given ordered list:

```
> inordlist :: (Ord a, FreeSubtypical a) => [a] -> SubtypeOf a
> inordlist = only . ordcontains

> notinordlist :: (Ord a, FreeSubtypical a) => [a] -> SubtypeOf a
> notinordlist xs = only (not . ordcontains xs)
```

A little auxilliary search routine....

```
> ordcontains (x:xs) y | x == y    = True
>                      | x < y     = ordcontains xs y
> ordcontains _ _                  = False
```

APPENDIX C

Source for the Balanced Trees Example

C.1. 2-3 Tree Header (Data Type and Derived Combinators)

```
-- Import relevant subtype combinator library files

import IntSubtypes
import DownSubtypes

-- The Tree datatype itself -------------------------------------------

data Tree23 = Empty
            | Tr2 Tree23 Int Tree23
            | Tr3 Tree23 Int Tree23 Int Tree23
            | Put Tree23 Int Tree23
            | Taken Tree23
            deriving Show

instance Subtypical Tree23

-- Instruct DRiFT to derive subTree23 and depsubTree23 functions:

{-! for Tree23 derive : downsubtypes, depsubtypes !-}
{-* Generated by Derive-v1.0 : Look, but Don't Touch. *-}

-- The functions below have been reformatted very slightly to make them
-- fit on the page. Otherwise, they are exactly as our DrIFT rule
-- generated them.

subTree23 :: (Bool) ->
     (Bool, SubtypeOf Tree23, SubtypeOf Int, SubtypeOf Tree23) ->
     (Bool, SubtypeOf Tree23, SubtypeOf Int, SubtypeOf Tree23,
                                      SubtypeOf Int, SubtypeOf Tree23) ->
     (Bool, SubtypeOf Tree23, SubtypeOf Int, SubtypeOf Tree23) ->
     (Bool, SubtypeOf Tree23) ->
     SubtypeOf Tree23
subTree23 (True)  _  _  _  _ (Empty ) = Empty
subTree23  _ (True,aa,ab,
      ac)  _  _  _ (Tr2 aa' ab' ac' ) = Tr2 (aa aa') (ab ab') (ac ac')
subTree23  _  _ (True,aa,ab,ac,ad,
 ae)  _  _ (Tr3 aa' ab' ac' ad' ae' ) = Tr3 (aa aa') (ab ab')
                                                  (ac ac') (ad ad') (ae ae')
subTree23  _  _  _ (True,aa,ab,
    ac)  _ (Put aa' ab' ac' ) = Put (aa aa') (ab ab') (ac ac')
subTree23  _  _  _  _ (True,aa) (Taken aa' ) = Taken (aa aa')

depsubTree23 :: (Tree23 -> (Bool)) ->
(Tree23 -> (Bool, SubtypeOf Tree23, SubtypeOf Int,
                                                     SubtypeOf Tree23)) ->
```

```
(Tree23 -> (Bool, SubtypeOf Tree23, SubtypeOf Int,
                       SubtypeOf Tree23, SubtypeOf Int, SubtypeOf Tree23)) ->
(Tree23 -> (Bool, SubtypeOf Tree23, SubtypeOf Int,
                                                    SubtypeOf Tree23)) ->
(Tree23 -> (Bool, SubtypeOf Tree23)) ->
SubtypeOf Tree23
depsubTree23  f  _  _  _  _ (Empty ) | p = Empty
    where (p) =  f  (Empty )
depsubTree23  _  f  _  _  _ (Tr2 aa' ab' ac' ) | p = Tr2 (aa aa') (ab ab')
                                                                  (ac ac')
    where (p,aa,ab,ac) =  f  (Tr2 aa' ab' ac' )
depsubTree23  _  _  f  _  _ (Tr3 aa' ab' ac' ad' ae' ) | p = Tr3 (aa aa')
                                         (ab ab') (ac ac') (ad ad') (ae ae')
    where (p,aa,ab,ac,ad,ae) =  f  (Tr3 aa' ab' ac' ad' ae' )
depsubTree23  _  _  _  f  _ (Put aa' ab' ac' ) | p = Put (aa aa') (ab ab')
                                                                   (ac ac')
    where (p,aa,ab,ac) =  f  (Put aa' ab' ac' )
depsubTree23  _  _  _  _  f (Taken aa' ) | p = Taken (aa aa')
    where (p,aa) =  f  (Taken aa' )

--  Imported from other files :-
```

C.2. 2-3 Tree Body (Subtypes and Algorithms)

```
-- Include the necessary files from the subtype combinator library,
-- and the headers

import DownSubtypes
import DepSubtypes
import TreeHeader

------------------------------------------------------------------------
-- Subtypes of normalised trees, with a given depth and in general.

depth Empty           = 0
depth (Tr2 t _ _)     = (depth t) + 1
depth (Tr3 t _ _ _ _) = (depth t + 1)

normdepth :: Int -> SubtypeOf Tree23
normdepth k | k>=(-1)  = subTree23 (k==0)
                        (k>0, normdepth (k-1), ok, normdepth (k-1))
          (k>0, normdepth (k-1), ok, normdepth (k-1), ok, normdepth (k-1))
                        (False, ko, ko, ko)
                        (False, ko)

normalised :: SubtypeOf Tree23
normalised = depsubTree23
              (const True)
              (\(Tr2 t _ _) -> (True, normdepth (depth t),
                                              ok, normdepth (depth t)))
              (\(Tr3 t _ _ _ _) -> (True, normdepth (depth t),
                         ok, normdepth (depth t), ok, normdepth (depth t)))
              (const (False, ko, ko, ko))
              (const (False, ko))
```

```
-------------------------------------------------------------------------
-- Put trees, of given depth and in general.

putdepth :: Int -> SubtypeOf Tree23
putdepth k | k>=(-1) = subTree23 (k==0)
                       (k>0, normdepth (k-1), ok, normdepth (k-1))
          (k>0, normdepth (k-1), ok, normdepth (k-1), ok, normdepth (k-1))
                       (k>0, normdepth k, ko, normdepth k)
                       (False, ko)

puttree :: SubtypeOf Tree23
puttree = depsubTree23
            (const True)
            (\(Tr2 t _ _) -> (True, normdepth (depth t), ok,
                                                normdepth (depth t)))
            (\(Tr3 t _ _ _ _) -> (True, normdepth (depth t), ok,
                                  normdepth (depth t), ok, normdepth (depth t)))
            (\(Put t _ _) -> (True, normdepth (depth t), ok,
                                                normdepth (depth t)))
            (const (False, ko))


-------------------------------------------------------------------------
-- Taken trees, of given depth and in general.

takendepth k | k>=(-10 = subTree23 (k==0)
                         (k>0, normdepth (k-1), ok, normdepth (k-1))
         (k>0, normdepth (k-1), ok, normdepth (k-1), ok, normdepth (k-1))
                         (False, ko, ko, ko)
                         (k>0, normdepth (k-1))

takentree :: SubtypeOf Tree23
takentree = depsubTree23 (const True)
                         (\(Tr2 t _ _) -> (True, normdepth (depth t),
                                                ok, normdepth (depth t)))
                         (\(Tr3 t _ _ _ _) -> (True, normdepth (depth t),
                           ok, normdepth (depth t), ok, normdepth (depth t)))
                         (const (False, ko, ko, ko))
                         (const (True, normalised))


-------------------------------------------------------------------------
-- Ordered trees containing a particular element, and all trees NOT
-- containing the given element.

contains :: Int -> SubtypeOf Tree23
contains x = depsubTree23
        (const False)
        (\(Tr2 _ n _) -> (True,
                          if x < n then (contains x) else ok,
                          ok,
                          if x > n then (contains x) else ok))
        (\(Tr3 _ m _ n _) -> (True,
                              if x < m then (contains x) else ok,
                              ok,
                              if m < x && x < n then (contains x) else ok,
                              ok,
```

```
                            if n < x then (contains x) else ok))
        (const (False, ko, ko, ko))
        (const (False, ko))


notcontains :: Int -> SubtypeOf Tree23
notcontains x = depsubTree23
        (const True)
        (\(Tr2 _ n _) -> (n /= x,
                         if x < n then (notcontains x) else ok,
                         ok,
                         if x > n then (notcontains x) else ok))
        (\(Tr3 _ m _ n _) -> (m /= x && n /= x,
                             if x < m then (notcontains x) else ok,
                             ok,
                             if m < x && x < n then (notcontains x) else ok,
                             ok,
                             if n < x then (notcontains x) else ok))
        (const (False, ko, ko, ko))
        (const (False, ko))


--------------------------------------------------------------------------
-- Ordered trees, between particular values and in general.

-- To avoid writing functions separate but similar functions "ordered",
-- "orderedbelow", "orderedabove" and "orderedbetween", we define a type
-- with representations for unbounded values.

data InfInt = Finite Int
            | PlusInfinity
            | MinusInfinity
            deriving Eq

instance Ord InfInt where
    MinusInfinity <= _       = True
    _ <= PlusInfinity        = True
    _ <= MinusInfinity       = False
    PlusInfinity <= _        = False
    (Finite x) <= (Finite y) = x <= y


ordered :: SubtypeOf Tree23
ordered = ordbet MinusInfinity PlusInfinity

orderedbetween :: Int -> Int -> SubtypeOf Tree23
orderedbetween x y = ordbet (Finite x) (Finite y)

ordbet :: InfInt -> InfInt -> SubtypeOf Tree23
ordbet x y = depsubTree23
        (const True)
        (\(Tr2 _ n _) -> (x <= Finite n && Finite n <= y,
                         ordbet x (Finite n), ok, ordbet (Finite n) y))
        (\(Tr3 _ m _ n _) -> (x <= Finite m && Finite m <= Finite n
                                                  && Finite n <= y,
                         ordbet x (Finite m), ok, ordbet (Finite m)
                         (Finite n), ok, ordbet (Finite n) y))
```

```
        (const (False, ko, ko, ko))
        (const (False, ko))



-----------------------------------------------------------------------------
-- Trees of integers satisfying a particular constraint

-- This subtype is not used by Reade (except in so far as he requires
-- all values in trees to be defined), but is quite useful anyway.

treeof :: SubtypeOf Int -> SubtypeOf Tree23
treeof s = subTree23 (True)
                     (True, treeof s, s, treeof s)
                     (True, treeof s, s, treeof s, s, treeof s)
                     (False, ko, ko, ko)
                     (False, ko)

------------------------------------------------------------------------
-- Reade's (and Hoffman and O'Donnell's) Algorithms in Haskell
--
-- These implementations seem to work correctly, but I don't claim
-- to have checked them rigorously. Any errors are far more likely
-- to be my own than Reade's (MEK)

insert23 :: Int -> Tree23 -> Tree23
insert23 n t = checktop (put n t)

put :: Int -> Tree23 -> Tree23
put n Empty = Put Empty n Empty

put n (Tr2 t1 a t2) | n==a      = Tr2 t1 a t2
                    | n<a       = tr2 (put n t1) a t2
                    | otherwise = tr2 t1 a (put n t2)

put n (Tr3 t1 a t2 b t3) | n==a      = Tr3 t1 a t2 b t3
                         | n==b      = Tr3 t1 a t2 b t3
                         | n<a       = tr3 (put n t1) a t2 b t3
                         | n<b       = tr3 t1 a (put n t2) b t3
                         | otherwise = tr3 t1 a t2 b (put n t3)

put n _ = error "put of un-normalised tree"

tr2 :: Tree23 -> Int -> Tree23 -> Tree23
tr2 (Put t1 a t2) b t3 = Tr3 t1 a t2 b t3
tr2 t1 a (Put t2 b t3) = Tr3 t1 a t2 b t3
tr2 (Taken t1) a Empty = error "tr2(Taken_,_,Empty)"
tr2 (Taken t1) a (Tr2 t2 b t3) = Taken (Tr3 t1 a t2 b t3)
tr2 (Taken t1) a (Tr3 t2 b t3 c t4) = Tr2 (Tr2 t1 a t2) b (Tr2 t3 c t4)
tr2 Empty a (Taken t1) = error "tr2(Empty,_,Taken,_)"
tr2 (Tr2 t1 a t2) b (Taken t3) = Taken (Tr3 t1 a t2 b t3)
tr2 (Tr3 t1 a t2 b t3) c (Taken t4) = Tr2 (Tr2 t1 a t2) b (Tr2 t3 c t4)
tr2 o1 o2 o3 = Tr2 o1 o2 o3

tr3 :: Tree23 -> Int -> Tree23 -> Int -> Tree23 -> Tree23
tr3 (Put t1 a t2) b t3 c t4 = Put (Tr2 t1 a t2) b (Tr2 t3 c t4)
tr3 t1 a (Put t2 b t3) c t4 = Put (Tr2 t1 a t2) b (Tr2 t3 c t4)
```

```
tr3 t1 a t2 b (Put t3 c t4) = Put (Tr2 t1 a t2) b (Tr2 t3 c t4)
tr3 (Taken t1) a (Taken t2) b (Taken t3) = Taken (Tr3 t1 a t2 b t3)
tr3 (Taken t1) a (Taken t2) b t3 = Tr2 (Tr2 t1 a t2) b t3
tr3 t1 a (Taken t2) b (Taken t3) = Tr2 t1 a (Tr2 t2 b t3)
tr3 (Taken t1) a (Tr2 t2 b t3) c (Taken t4) = Tr2 (Tr2 t1 a t2) b (Tr2 t3 c t4)
tr3 (Taken t1) a (Tr3 t2 b t3 c t4) d (Taken t5) = Tr2 (Tr3 t1 a t2 b t3) c
                                                              (Tr2 t4 d t5)
tr3 (Taken t1) a Empty b t2 = error "tr3(Taken_,_,E,_,_)"
tr3 (Taken t1) a (Tr2 t2 b t3) c t4 = Tr2 (Tr3 t1 a t2 b t3) c t4
tr3 (Taken t1) a (Tr3 t2 b t3 c t4) d t5 = Tr3 (Tr2 t1 a t2) b (Tr2 t3 c t4)
                                                                       d t5
tr3 t1 a Empty b (Taken t2) = error "tr3(_,_,E,_,Taken)"
tr3 t1 a (Tr2 t2 b t3) c (Taken t4) = Tr2 t1 a (Tr3 t2 b t3 c t4)
tr3 t1 a (Tr3 t2 b t3 c t4) d (Taken t5) = Tr3 t1 a (Tr2 t2 b t3) c
                                                             (Tr2 t4 d t5)
tr3 Empty a (Taken t1) b t2 = error "tr3(E,_,Taken,_,_,_)"
tr3 (Tr2 t1 a t2) b (Taken t3) c t4 = Tr2 (Tr3 t1 a t2 b t3) c t4
tr3 (Tr3 t1 a t2 b t3) c (Taken t4) d t5 = Tr3 (Tr2 t1 a t2) b
                                                        (Tr2 t3 c t4) d t5
tr3 o1 o2 o3 o4 o5 = Tr3 o1 o2 o3 o4 o5

checktop :: Tree23 -> Tree23
checktop (Put t1 a t2) = Tr2 t1 a t2
checktop (Taken t) = t
checktop other = other

remove23 :: Int -> Tree23 -> Tree23
remove23 n t = checktop (remove n t)

----------------------------------------------------------------------------
remove :: Int -> Tree23 -> Tree23
remove n Empty = Empty

remove n (Tr2 t1 a t2) | n == a    = (merge t1 t2)
                       | n < a     = (tr2 (remove n t1) a t2)
                       | otherwise = (tr2 t1 a (remove n t2))

remove n (Tr3 t1 a t2 b t3) | n == a    = (leftPut (merge t1 t2) b t3)
                            | n == b    = (rightPut t1 a (merge t2 t3))
                            | n < a     = (tr3 (remove n t1) a t2 b t3)
                            | n < b     = (tr3 t1 a (remove n t2) b t3)
                            | otherwise = (tr3 t1 a t2 b (remove n t3))

remove n _ = error "remove of un-normalised tree"

-------------------------------------------------------------------------
merge :: Tree23 -> Tree23 -> Tree23
merge Empty Empty = Taken Empty
merge (Tr2 t1 a t2) (Tr2 t3 b t4) = tr3 (Taken t1) a (merge t2 t3) b (Taken t4)
merge (Tr2 t1 a t2) (Tr3 t3 b t4 c t5) = tr3 (Taken t1) a (merge t2 t3)
                                                            b (Tr2 t4 c t5)
merge (Tr3 t1 a t2 b t3) (Tr2 t4 c t5) = tr3 (Tr2 t1 a t2) b (merge t3 t4)
                                                            c (Taken t5)
merge (Tr3 t1 a t2 b t3) (Tr3 t4 c t5 d t6) = tr3 (Tr2 t1 a t2)
                                               b (merge t3 t4) c (Tr2 t5 d t6)
merge _ _ = error "merge of inappropriate trees"
```

```
leftPut:: Tree23 -> Int -> Tree23 -> Tree23
leftPut (Tr2 t1 a t2) b t3 = Tr3 t1 a t2 b t3
leftPut (Tr3 t1 a t2 b t3) c t4 = Put (Tr2 t1 a t2) b (Tr2 t3 c t4)
leftPut (Taken t1) a t2 = Tr2 t1 a t2
leftPut Empty _ _ = error "leftPut(E,_,_)"
leftPut (Put _ _ _) _ _ = error "leftPut(_,_,_)"

rightPut:: Tree23 -> Int -> Tree23 -> Tree23
rightPut t1 a (Tr2 t2 b t3) = Tr3 t1 a t2 b t3
rightPut t1 a (Tr3 t2 b t3 c t4) = Put (Tr2 t1 a t2) b (Tr2 t3 c t4)
rightPut t1 a (Taken t2) = Tr2 t1 a t2
rightPut _ _ Empty = error "rightPut(_,_,E)"
rightPut _ _ (Put _ _ _) = error "rightPut(_,_,_)"


---------------------------------------------------------------------
-- Reade did not provide his member23 function, so I have
-- implemented it myself in the obvious way

member23 :: Tree23 -> Int -> Bool

member23 (Tr2 x p y) m | m < p     = member23 x m
                       | p < m     = member23 y m
                       | otherwise = True

member23 (Tr3 x p y q z) m | m < p          = member23 x m
                           | p < m && m < q = member23 y m
                           | q < m          = member23 z m
                           | otherwise      = True

member23 Empty _ = False

-----------------------------------------------------------------
-- A few simple test functions

nat :: [Int]
nat = 1 : map (+1) nat

listtotree :: [Int] -> Tree23
listtotree = foldr ((ok ->> ok ->> normalised) insert23) Empty

treeminuslist :: Tree23 -> [Int] -> Tree23
treeminuslist t = foldr remove23 t

leftrootelement :: Tree23 -> Int
leftrootelement (Tr2 _ n _) = n
leftrootelement (Tr3 _ n _ _ _ ) = n

--------------------------------------------------------------------
-- Maps elements of the second list to true or false, depending on
-- whether they are in the first list.
searchfornums :: [Int] -> [Int] -> [Bool]
searchfornums xs = map (member23 t)
    where t = foldr insert23 Empty xs
```

Bibliography

[Aug99] L. Augustsson. Cayenne — a language with dependent types. *ACM SIGPLAN Notices*, 34:239–250, 1999.

[B+85] F. L. Bauer et al. The wide spectrum language CIP-L., 1985.

[Bar96] J. G. P. Barnes. *Programming in Ada 95*. Addison-Wesley, 1996.

[Bet98] G. Betarte. *Dependent Record Types and Algebraic Structures in Type Theory*. PhD thesis, Dept. of Computing Science, University of Göteborg., 1998.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[Coh97] D. I. A Cohen. *Introduction to Computer Theory (2nd edition)*. Wiley, 1997.

[Con71] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[Cur86] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science. Pitman, 1986. Excellent introduction to FP maths (partial orders, category theory) in appendix, also much on sequential algorithms.

[Dri] The DrIFT Homepage. http://www.dcs.gla.ac.uk/~nww/Derive/derivehome.html.

[Has] The Standard Haskell Homepage. http://www.cs.chalmers.se/~rjmh/Haskell/.

[HO82] C. M. Hoffman and M. J. O'Donnell. Programming with equations. *ACM Trans. Programming Language Syst.*, 4:83–112, 1982.

[Kie83] R. Kieburtz. Precise typing of abstract data type specifications. In *Proc. 10th Symp. on Principles of Programming Languages*, pages 109–116, 1983.

[KR88] B. W. Kernighan and D. M. Richie. *The C Programming Language, 2nd edition*. Prentice Hall, 1988.

[Mar83] J. J. Martin. Precise typing and filters. *Information Processing Letters*, 17:109–112, 1983.

[MFF87] R. N. McKenzie, McNulty G. F., and Taylor W. F. *Algebras, Lattices, Varieties*. Brooks/Cole, 1987.

[Möl89] B. Möller. Applicative assertions. *Lecture Notes in Computer Science*, 375:348–362, 1989.

[Möl96] B. Möller. Assertions and recursions. *Lecture Notes in Computer Science*, 1074:163–184, 1996.

[NN92] H. R. Nielson and F. Neilson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[P+96] Peterson et al. Report on the programming language Haskell (version 1.3). http://www.haskell.org/definition/, 1996.

[Pre97] R.S. Pressman. *Software Engineering: A Practitioner's Approach (4th edition)*. McGraw Hill, 1997.

[Rea89] C. Reade. *Elements of Functional Programming*. International Computer Science. Addison-Wesley, 1989.

[Rea92] C. M. P. Reade. Balanced trees with removals: an exercise in rewriting and proof. *Science of Computer Programming*, 18:181–204, 1992.

[Run95] C. Runciman. Subtype constraints as first-class values. Talk given at 48th meeting IFIP WG2.1, Gunzberg, Germany, and associated notes, 1995.

[Sch86] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[Sco76] D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976.

[Sco82] D. Scott. Domains for denotational semantics. *Automata, Languages, and Programming*, IX:577–613, 1982.

[Som89] I. Sommerville. *Software Engineering (3rd edition)*. Addison-Wesley, 1989.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Ten91] R. D. Tennent. *Semantics of Programming Languages*. Prentice-Hall, 1991.

[Tho96] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[Wad85] P. Wadler. A splitting headache - and its cure! strict vs lazy semantics for pattern matching in lazy languages. Unpublished, 1985.

[Win97] N. Winstanley. Reflections on instance derivation. In *Proc. 1997 Glasgow Functional Programming Workshop*, September 1997.