Making Uniqueness Typing Less Unique

Thesis submitted for the degree of Doctor in Philosophy

December 14, 2008

Edsko Jacob Jelle de Vries

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidate's own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Voor mijn ouders

Voor het helpen met de GW-Basic programma's uit de Weet-Ik!,
en het herstellen van `autoexec.bat`.

Summary

Chapter 1, p. 11

Computer science distinguishes between two major programming paradigms: imperative and functional programming. Central to **imperative programming** is the notion of some form of state (or memory) together with a list of instructions that inspect and modify that state. The canonical example of this paradigm is the Turing machine. **Functional programming** on the other hand is centred around a mathematical language with a notion of evaluation of expressions in this language. The notion of state—the core concept in imperative programming—is completely absent. The canonical example of the functional paradigm is the lambda calculus.

Although neither Turing machines nor the lambda calculus support a notion of **interaction**, it is not difficult to see how interaction can be added to a Turing machine: it suffices to allow external entities to modify the machine state. Adding interaction to the lambda calculus is a much more difficult problem, and there are many solutions proposed in the literature.

One option is to regard an interactive program as a function from the **world state** to a new world state. For example, we might have a function `getChar` that, given a `world` object, reads a character from the console and produces a new `world` object; and a function `putChar` that, given a character and a `world` object, echos the character to the console and produces a new `world` object. We can then define a program that reads a character from the console and echoes it back to the user as follows:

```
λworld. putChar (getChar world)
```

The main problem with this approach is the potential **duplication** of the world state. For example, we can easily define a function that returns a pair of world states, one in which the character `'a'` has been printed and one in which the character `'b'` has been printed:

```
λworld. (putChar ('a', world), putChar ('b', world))
```

Since such a program has no semantic interpretation, we have two options: we can dismiss this approach, or we can impose a **type system** which analyses the programs written by the user and rejects those without a semantic interpretation. **Uniqueness typing** is such a type system: it can be used to ensure that objects such as the world state are used in a **single-threaded** manner.

Uniqueness typing was developed for a language based on graph rewriting rather than the lambda calculus. This is unfortunate as it means that recent advances in type systems in the wider functional programming community cannot readily be incorporated in uniqueness typing. The main thesis we will defend in this dissertation is that the graph rewriting background is inessential to uniqueness typing. We claim that it is possible to define a uniqueness type system that is sufficiently similar to conventional type systems for lambda calculus based functional programming languages that techniques developed for such languages can be applied without major difficulties. In short, we propose to **make uniqueness typing less unique**.

Two main themes have been interwoven in this thesis. In the first, we show how to define a uniqueness type system for the lambda calculus. This system is initially strongly based on the original uniqueness type system, but through a series of simplifications we show how to make it simpler and more like a conventional lambda calculus type system. In the second theme we offer evidence of the effectiveness of these simplifications by taking advantage of them to incorporate type system extensions originally developed for languages without uniqueness typing.

Chapter 4, p. 115 The original uniqueness type system distinguishes between objects that are **non-unique** (may freely be duplicated), **unique** now (have not been duplicated) but that may become non-unique later, and objects that are **necessarily unique** (unique now and must forever remain unique). The first simplification we propose is that it is sufficient to **distinguish only between non-unique and unique objects** (without a notion of "necessarily unique"), and we develop a lambda calculus type system based on this idea. We then show how to extend this type system with **arbitrary rank types**, based on work by Peyton Jones, Vytiniotis, Weirich, and Shields (2007).

Chapter 5, p. 129 Uniqueness types often involve inequality constraints between uniqueness attributes ("this object must be unique if that other object is"). The second simplification we propose is to **remove these constraints by supporting arbitrary boolean expressions as uniqueness attributes**, simplifying both the type system and its implementation. We show again how to incorporate higher rank types, and briefly discuss how to support **generalized algebraic data types** based on work by Peyton Jones, Vytiniotis, Weirich, and Washburn (2006).

Chapter 6, p. 143 As mentioned, the original uniqueness type system distinguishes between non-unique, unique and necessarily unique objects. We suggested that this may be simplified by distinguishing only between non-unique and unique objects, but a disadvantage of this approach is that it relies on a form of closure typing. This makes types difficult to read and understand, and so in the final system we suggest that it may be better to **distinguish only between non-unique and necessarily unique** objects (and no longer support a notion of "unique now, but may become non-unique later). This removes the need for closure typing, without sacrificing expressiveness *provided* that we are careful in the types we assign to library functions and **take advantage of polymorphism**. As a further simplification, we propose to **regard uniqueness attributes and types as one syntactic category** (and distinguish using a kind system). This makes the presentation and formalization of the type system more uniform, and gives us additional expressive power for free. We show how to support higher rank types, **impredicativity** and **records and variants** based on work by Leijen (2008a) and Gaster and Jones (1996).

Chapter 7, p. 159 We then give a formalization of this (core) type system using the proof assistant Coq. After proving numerous technical lemmas, we provide a number of lemmas about the type system such as well-kindedness, weakening and exchange, followed by the main **soundness** theorem (*progress* and *type preservation* or *subject reduction*) based on the call-by-need lambda calculus.

Chapter 8, p. 201 Finally, we discuss some of the design choices we have made, potential alternatives, and the relation between our work and related work, and identity future work. We conclude that indeed we have made uniqueness typing less unique: the system we have developed is simple, easy to understand and recent type system extensions are readily incorporated. The only exception appears to be that our system allows less impredicative instantiation than the type system on which it is based. We explain why and discuss some ways in which we might solve this problem.

Acknowledgements

Choosing a PhD thesis topic is a difficult and time consuming task (Cham, 1998), and when I eventually settled on one I wanted to make sure that an expert in the field thought the project worthwhile. At the time, our university was organizing IFL05 and one of the participants was Prof. Rinus Plasmeijer, head of the Software Technology Research Group in Nijmegen—the team that develops the language *Clean*; I took advantage and asked him for his opinion. He thought that it was a good proposal, but told me that another PhD student in Hungary was working on the exact same problem. This was a bit of a set-back, but Rinus and I got talking—first during a lovely walk in Glendalough, Wicklow and later in a small Italian restaurant just off O'Connell Street in the heart of Dublin—and it turned out that Rinus had a huge list of open problems and would only be too happy if I started working on one of them. A few days later this was followed up with an email listing some of these projects. Many of them were detailed funding proposals for PhD positions, but somewhere in the middle was a throw-away remark that the relation between uniqueness typing and polytypic programming needed clarification. This sounded like an interesting problem and it was not too far off from my original thesis topic, so I suggested that I begin work on this. It soon emerged that to be able to fully answer the question, we needed a uniqueness type system with support for higher rank types and the result is the thesis that you are currently reading. As well as suggesting the original research problem, Rinus has been very helpful throughout my PhD: he gave feedback on my papers, was enthusiastic about my ideas and provided encouragement when needed. Rinus, your involvement was greatly appreciated!

Further motivation and feedback was given by my colleagues in the Foundations and Methods Group, especially Arthur Hughes, Andrew Butterfield, Hugh Gibbons, Glenn Strong and Shane Ó Conchúir. I am indebted to David Gregg, who got me an invitation for the Verifying Optimizing Compilers seminar in Dagstuhl. A huge thank-you also to my office mate of four years and fellow student of eight, John Gilbert. Although our thesis topics are not at all related, I enjoyed the many discussions and our mutual interest in exotic teas; moreover, John proofread all my papers and most of this thesis.

The larger functional programming community has generally been very supportive, and this thesis has benefited from many discussions with people in conferences, summer schools, and in emails. There are far too many names to list here, but I would like so single out a few people. Dimitri Vytiniotis, Bastiaan Heeren, Stefan Holdermans, Fenrong Liu and Amr Sabry have been very helpful discussing various aspects of their papers. Additionally, Bastiaan provided feedback on my first paper and invited me to present my work in Utrecht, which was appreciated, and Stefan has provided very valuable feedback on sections of this thesis. My understanding of the single-threaded polymorphic lambda calculus was much improved by discussions with Adam Megacz, who also assisted with some aspects of my formal soundness proof. Thanks!

I do not know whether I could have completed that soundness proof without the guidance of Arthur Charguéraud, one of the authors of the *Engineering Formal Metatheory* paper. His generous help in getting the foundations of the proof right was invaluable. Likewise, I could not have done the proof without the aid of the people on the Coq-club mailing list, whose kind and helpful replies to many questions were greatly appreciated.

A thank-you also to Daan Leijen, who invited me to do an internship with him in Microsoft Research, Redmond. Although my work in Microsoft was not immediately related to my thesis, I learned a great deal and it is likely I would not have had the ideas which became the final chapter if not for my work with Daan. Moreover, I remember my time there very fondly; it was a great summer!

I thank my supervisor, Dr David M Abrahamson, and my funding agency, the Irish Research Council for Science, Engineering and Technology (IRCSET) for supporting my research, and the two examiners of the thesis, Prof. Matthew Hennessy and Dr Sven-Bodo Scholz.

Finally, where would I be without my better half, Wendy Verbruggen? As well as proof-reading everything I have written (and few errors escape her careful eyes!), sharing my life with her makes it infinitely more enjoyable.

Edsko de Vries, Dublin, December 14, 2008

Contents

Chapters that have been published or presented prior to this thesis are marked with a star (∗). Details can be found on the first page of each chapter.

List of Figures

Notational Conventions

For consistency, we make the (arbitrary) decision to use "expression" to refer to programs and "term" to refer to either expressions or types. In a language where no distinction is made between programs and types, "expression" and "term" are synonymous.

We will use a `monospaced` font for code listings (concrete syntax) and an *italic* font for mathematical formulae. Operator names appearing in mathematical formulae, such as fv(e), or abstract syntax, such as $(\mathrm{let}\ x = e\ \mathrm{in}\ e')$, will be typeset in a roman font.

We will use the following notational conventions throughout this dissertation.

Shorthand

$\overline{a}$	$a_0, a_1, \ldots, a_n$ (where it is assumed that n is clear from the context)
$[\overline{a := \tau}]$	$[a_1 := \tau_1, a_2 := \tau_2, \ldots, a_n := \tau_n]$
$\forall a_1 a_2 \ldots a_n \cdot \tau$	$\forall a_1 \cdot \forall a_2 \cdot \ldots \cdot \forall a_n \cdot \tau$

Meta-level variables

Meta-level variables can be superscripted (T') or subscripted (T_1) to distinguish different instances of the variable. We will generally use superscripts if the number of occurrences is fixed, and subscripts if the number is variable (as in $T_1, \ldots, T_n$).

T	term
Γ	typing environment
σ	type (possibly involving universal quantifiers)
ρ	type without top-level quantifiers
τ	monotype (type not involving universal quantifiers)
ν	uniqueness attribute (type of kind $\mathcal{U}$)
κ	kind
e	expression
$\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{u}, \mathfrak{v}$	skolem constants
S	substitution
C	context
E	evaluation context

Object-level variables

x, y, z	expression or term variable (we will not use x for a type variable in a context where there is a distinction between types and expressions)
a, b, c	type variable (of arbitrary kind, though generally not of kind $\mathcal{T}$ or $\mathcal{U}$)
t, s, r	type variable (of kind $\mathcal{T}$, i.e., requiring an attribute)
u, v, w	uniqueness variable (type variable of kind $\mathcal{U}$)

Types and uniqueness attributes

$\mathbb{N}$	natural numbers
$\mathbb{Z}$	integers
$\mathbb{B}$	booleans
$\bullet$	unique
$\times$	non-unique
$\forall(a : \kappa) \cdot \sigma$	Universal quantification (in an implicitly typed language)
$\Pi(a : \kappa) \cdot \sigma$	Universal quantification (in an explicitly typed language)
τ^{ν}	Type τ with attribute ν
$\sigma \xrightarrow{\nu} \sigma'$	Shorthand for $(\sigma \to \sigma')^{\nu}$
$\sigma \xrightarrow[\nu_c]{\nu_f} \sigma'$	Shorthand for $(\sigma \underset{\nu_c}{\longrightarrow} \sigma')^{\nu_f}$; ν_c is the closure attribute of the function

Kinds

$\star$	Kind of value-types
$\mathcal{T}$	Kind of "base" types (types without a uniqueness attributes)
$\mathcal{U}$	Kind of uniqueness attributes (when considered as types)

Universal algebra

$\mathrm{fv}(t)$	the free variables in t; if it is not clear from the context, we may use fev for the free expression variables, ftv for the free type variables and fuv for the free uniqueness variables
$t[x := t']$	substitution: replace x by t' in t
$S_2 \circ S_1$	composition
$t_1 \approx t_2$	identity (in the sense of equational logic)
$t_1 \overset{?}{\approx} t_2$	a (unification) equation (which may or may not have a solution)
$C[e]$	Substitution of e for the "hole" in context C

Terms and their types

$\Gamma \vdash e : \sigma$	e has type σ in environment Γ
$e :: \sigma$	user-supplied type annotation on e

Relations between types

$u \leq v$	Implication between uniqueness attributes (if v is unique, u must be unique)
$\sigma \preceq \sigma'$	Subsumption (σ is more general than σ')

1 Introduction

1.1 Computability

At the heart of computer science is the notion of *computability*. Although there is no universally agreed definition of computability, there is a well-known and widely accepted hypothesis, known as the Church-Turing thesis, which states that a function is computable if and only if it can be expressed as a Turing machine or (equivalently) as an expression in the lambda calculus.

These two models of computation, Turing machines and the lambda calculus, are quite different and they form the basis of two fundamentally different programming paradigms: *imperative* programming and *functional* programming. The fact that both models are nevertheless equivalent is quite astounding, and one of the most fundamental results in computer science.

A Turing machine (Turing, 1937) can read from and write to an infinite tape containing symbols chosen from some finite alphabet. At any point in time, the machine is in a particular state (chosen from a finite set of states) and can read exactly one symbol from the tape. The machine contains a *finite state machine* (known as its transition function) which tells it, given the current state of the machine and the current symbol on the tape, which symbol to write to the tape, in which direction to move the tape—left or right—and what next state to go to.

The behaviour of the Turing machine is completely determined by the symbol alphabet, the set of states, and the transition function. An important class of such configurations are known as *Universal Turing Machines*. A universal Turing machine is one which can simulate all others. Effectively, it runs an interpreter or virtual machine which can read the definition of another Turing machine from the tape and then behave like that other machine.

Turing machines are a model for imperative programming languages, which include virtually all mainstream languages from Visual Basic to C++. Programs in these languages are essentially lists of instructions which are executed one by one and have access to some form of memory (the "tape") usually modelled by *variables* which can be read from or written to.

In the lambda calculus (Church, 1932) on the other hand this notion of "state" is completely absent. A program in the lambda calculus is an expression, which is built up as follows:

1. A *variable* (x) is an expression

2. If e is an expression, then the *abstraction* ($\lambda x \cdot e$) is an expression

3. If e and e' are expressions, then the *application* ($e\ e'$) is an expression

An abstraction should be thought of as an (anonymous) function definition. For example, $\lambda x \cdot x$ is the identity function (the function that returns its argument unchanged). Mathematicians might write that function as $x \mapsto x$. It is also related to the notation $f(x) = x$, but note that in the lambda calculus function definitions do not get a name (f).

"Running" a lambda expression means simplifying (evaluating) it. For example, the application of the identity function $(\lambda x \cdot x)$ to an argument y, $(\lambda x \cdot x)\ y$, can be simplified to just y. Common programming constructs such as numbers, booleans, conditionals ("`if`-statements") or recursion (looping constructs) can be elegantly encoded in the pure lambda calculus. Some of these encodings are rather ingenious, and the interested reader is urged to consult a text book on this topic such as (Hankin, 1994, especially Chapter 6, *Computability*). Pragmatically inclined readers will be happy to learn that these encodings have practical applications too (Jansen et al., 2007).

The lambda calculus is a model for pure functional programming languages such as *Clean* or *Haskell* which do not have a notion of global state. Although the concept of a *variable* arises in both programming paradigms, they are quite different: like variables in mathematics, variables in pure functional languages do not actually vary. There is no assignment statement!

1.2 Type systems

Both the Turing machine and the lambda calculus model have the property that they cannot get "stuck". It is impossible to write a program (a transition function or a lambda expression) which cannot be executed in the model. This property however does not scale to programming languages derived from these basic computational models.

For example, suppose we extend the core lambda calculus with the constants `tt`, `ff`, `if`, `rec`, $\texttt{k}_n$ (for all natural numbers n), `inc`, `dec` and `zero`, denoting the booleans true and false, the conditional, recursion, constants for each natural number, functions to increment and decrement a natural number, and a check for zero, respectively. This language is another classic called PCF (Plotkin, 1977).

As for the core lambda calculus, executing a program in PCF means evaluating or simplifying it, but now there are ill-formed terms such as `inc tt`; it does not make sense to increment a boolean constant, and evaluation will get stuck when presented with such a program.[1]

To solve this problem, we introduce a *type system* which weeds out the invalid from the valid programs. Unfortunately, it is impossible to give a (decidable) algorithm which accepts only the valid programs and rejects only the invalid programs, and so type systems must be approximations: some programs will be rejected even when they would not get stuck when evaluated.

Naturally, we would like to minimize the number of valid programs that get rejected. However, if a valid program does get rejected, it is equally important that a programmer understands why. There is therefore a trade-off between accuracy and clarity of type systems, and a large part of computer science including this dissertation is dedicated to the search for good locations in this design space.

We started with program evaluation and described a type system as a device to separate the good from the bad so that evaluation does not get stuck. Alternatively, we can approach it from the other side and start with the definition of a type system and treat evaluation as secondary. This leads to a second fundamental result in computer science, known as the *Curry-Howard* isomorphism[2]: the typed lambda calculus (that is, the lambda calculus together with a type system) corresponds directly to a *logic*.

[1] We consider reduction to be defined for closed terms only, and consider constants such as `inc` to be bound.

[2] It is hard to pinpoint the exact origin of the isomorphism. As Sørensen and Urzyczyn put it, "Surprisingly, the ancient Greeks probably did not know about the Curry-Howard isomorphism. But some authors trace the roots of the idea back to Schönfinkel, Brouwer, Heyting, or even Frege and Peano". See (Sørensen and Urzyczyn, 2006, Section 4.8) for a historical account.

From this perspective, the type of a term is a logical proposition (such as "for all natural numbers n, $n + 0 = n$") and the term itself is a *proof* of the proposition. The particular type system determines the logic (or, equivalently, the logic determines the type system). The Curry-Howard isomorphism is an amazing result which brings logic firmly within the realm of computer science, and the proofs in this dissertation (Chapter 7) are in fact programs in a lambda calculus p. 159 known as the *Calculus of Inductive Constructions*.

1.3 Interaction

We mentioned before that the behaviour of a Turing machine is completely specified by its transition function. In his original paper, Turing distinguishes between two kinds of machines:

> "If at each stage the motion of a machine is *completely* determined by the configuration, we shall call the machine an "automatic machine" (or *a*–machine). For some purposes we might use machines (choice machines or *c*–machines) whose motion is only partially determined by the configuration [...] In this paper I deal only with automatic machines.
>
> (Turing, 1937, emphasis in original)

"Turing machines" are what Turing himself would have called *a*–machines. Some people have cited this as evidence that the Church-Turing thesis is wrong (Wegner, 1997; Goldin and Wegner, 2005) because Turing machines cannot be interactive: the input to the machine must be fully determined when the machine is started, and the machine cannot communicate with the rest of the world during the computation.

An "automatic" machine becomes a "choice" machine as soon as we allow the machine's tape to be modified by external entities: the tape itself becomes a means of communication. This is essentially what happens in "real" computers (memory-mapped I/O); for example, we can write to the computer's screen by modifying one particular area of memory[1], or find out which key was pressed on the computer's keyboard by reading another.

It is less obvious how to add interaction to the lambda calculus. Indeed, for a long time this was seen as one of the major obstacles for functional programming. The issues are now better understood, and two workable solutions have emerged (uniqueness typing and monadic I/O). Nevertheless, the last word on this issue has not yet been said.

There are two main approaches to adding interaction to functional programming languages, known as *stream based* I/O and *environment based* I/O (Achten and Plasmeijer, 1995). In the stream based approach, a program converts a stream of input data into a stream of output data. The input stream is explicitly allowed to depend on the output stream; that is, the input stream does not need to be fully constructed before the program is run (it is created lazily). Stream based I/O can also be defined in terms of continuations, but the two approaches turn out to be equivalent (Hudak and Sundaresh, 1988) so we will not discuss the details here.

Although stream based I/O has fallen out of favour with functional programmers (Hudak et al., 2007; Peyton Jones, 2001), interestingly it has recently seen a come back in imperative models of computation: *Persistent Turing Machines* are an extension of Turing machines which have two additional tapes, one of which is used for input and one of which is used for output (Goldin, 2000). It is claimed that PTMs are a better model for interactive computation than plain Turing machines.

[1] Segment `0xB800` for the old assembly hackers amongst us!

In environment based I/O, the state of the world is represented by some token object `world`. A program becomes a function that given a `world` produces a new `world`. For example, we might have a function `getChar` that, given a `world` object, reads a character from the console and produces a new `world` object, and a function `putChar` that, given a character and a `world` object, echos the character to the console and produces a new `world` object. We can then define a program that reads a character from the console and echoes it back to the user as follows:

```
λworld. putChar (getChar world)
```

I/O models that rely on uniqueness typing, such as the one used in *Clean*, and I/O models that rely on monads, such as the one used in *Haskell*, both fall into the class of environment based I/O. The difference between the two approaches will become clear in the next section.

1.4 Purity

The absence of state in pure functional programming languages has various consequences, one of which is *definiteness*: the same expression used twice must evaluate to the same result.[1] For example, the value returned by a function must only depend on the arguments provided. In imperative languages this is not true. For example, consider the following C fragment:

```
x = getc(stdin) + getc(stdin);
```

This program reads two characters from the keyboard and adds them together. It seems obvious that the calls to `getc` will (potentially) return two different values; after all, the user can enter two different characters. Nevertheless, this is a violation of definiteness: the value returned by `getc` does not just depend on its argument (`stdin`).

One of the most important advantages of the absence of state is that reasoning about and proving properties of programs becomes easier. For example, it justifies making use of basic laws from arithmetic such as "for all natural numbers n, $n + n = 2 \times n$". In imperative languages, these laws cannot be applied; the C program above is certainly not equivalent to the following program:

```
x = 2 * getc(stdin);
```

Some imperative languages such as Eiffel advocate the *Command-Query Principle* (Meyer, 1997, Chapter 23), which splits the language into procedures (commands) which can have side effects but cannot return a value, and functions (queries) which return a value but cannot have a side effect. For example, the Eiffel equivalent to the second C program is

```
io.read_integer
x := 2 * io.last_integer
```

The Command-Query principle improves the situation for reasoning, because this program is in fact equivalent to

```
io.read_integer
x := io.last_integer + io.last_integer
```

However, reasoning is still quite limited because such identities only hold in between commands. For example, the Eiffel program that corresponds to the *first* C program is

[1] Section 2.8 contains a more detailed discussion of these issues.

```
io.read_integer ; a := io.last_integer
io.read_integer ; b := io.last_integer
x := a + b
```

Although *a* and *b* are both the result from a call to `io.last_integer`, they (potentially) have different values. Hence, the value of a function call is not solely determined by its arguments.

Side-effecting functions in a functional programming language manipulate a token `world` object. For example, we can define a function to read two integers and add them as

```
λworld. let (a, world')   = freadi world
            (b, world'')  = freadi world'
        in (a + b, world'')
```

This program is similar to the C program, except that `freadi` returns a new `world` object (as well as an integer). This new `world'` is then used in the second call to `freadi`. Of course, the `world` does not *actually* represent the entire state of the world; nevertheless, it makes clear that the second call to `freadi` happens in a "different world" and thus may return a different result.

Since the `world` object does not actually represent the world but is a token object only, one may wonder what would happen if we define

```
λworld. let (a, world')   = freadi world
            (b, world'')  = freadi world
        in (a + b, world'')
```

Now both calls to `freadi` get passed the *same* world state, so by definiteness they should return the same result. Since we cannot guarantee this, we must outlaw this definition. The difference between the monadic approach and the approach based on uniqueness typing is the mechanism used to forbid definitions such as the one above.

In the monadic approach, the programmer never directly manipulates `world` objects. Instead, a number of primitive functions from `world` to `world` are defined, and the programmer is provided a way to string these together (the monadic interface). Thus it becomes impossible to even *define* the function above, and programs are "correct by construction".

In uniqueness typing, the programmer does directly manipulate `world` objects, but a specialized type system guarantees that every `world` object is used at most once (that is, the object must be *unique*). Since the same `world` object is used twice in the example above, the type checker will reject the program. A more detailed comparison can be found in Section 2.8.7. p. 66

1.5 Overview of this dissertation

The pure functional programming language *Clean* (Plasmeijer and van Eekelen, 2002) is based on uniqueness typing. Unfortunately, like the type system of Haskell 98 (Peyton Jones, 2002), Clean's uniqueness type system only supports rank-1 types (the rank of a type will be defined in the next chapter). While this suffices for a large class of programs, it does not suffice for all and the use of higher rank types is becoming more widespread. For Clean, the need for higher rank types has become particularly acute due to the extensive use of data-type generic (or polytypic) programming (Alimarine, 2005; Schreur and Plasmeijer, 2004; Plasmeijer and Achten, 2005; van Weelden et al., 2005; Koopman and Plasmeijer, 2007; Smetsers et al., 2008), which relies essentially on support for higher rank types (Hinze, 2000, especially Section 3.1.3, *Specializing generic values*).

There are various proposals for extending Haskell's type system to support higher rank types (Vytiniotis et al., 2006; Peyton Jones et al., 2007; Leijen, 2008a). Unfortunately, uniqueness typing as used in Clean is too far removed from Haskell's type system for these proposals to p. 78 carry over easily. This is caused in part by Clean's background of graph rewriting (Section 3.2) rather than the lambda calculus. We will argue that the difference is inessential and propose a new uniqueness type system which is similar enough to the standard type systems for functional languages that techniques for Haskell or ML can be applied without major modifications in the context of Clean: we propose to make uniqueness typing less unique.

The dissertation is structured as follows. The technical background is given in Chapter 2. This chapter is long but important: as well as providing context and listing the theory that we will assume known in the remainder, this chapter supports our thesis that uniqueness typing can be firmly grounded on the theory for traditional functional type systems.

This is followed by an overview of related work in Chapter 3. We interpret "related work" as alternative proposals for substructural type systems with the purpose of maintaining referential transparency when adding side effects to a purely functional language. Of all systems considered, however, only the type system by Hage, Holdermans, and Middelkoop (2007)—which cites our p. 86 first paper on the topic (de Vries et al., 2007b)—supports higher rank types (Section 3.2.8), and even they present an inference algorithm only for the rank-1 fragment.

Chapter 4, based on (de Vries et al., 2007b), redefines uniqueness typing for the lambda calculus instead of for graph rewriting and presents our first attempt at scaling Clean's uniqueness type system to higher rank types. While this chapter achieves what we set out to do—define a uniqueness type system that supports higher rank types—the resulting type system is rather complex. In particular, the types (as used by and presented to the user) can be daunting.

Chapters 5 and 6, based on (de Vries et al., 2008), propose various simplifications to the *core* type system (supporting only rank-0 types). The simplified core system is an improvement over Clean's uniqueness type system even when not using higher rank types, as the system is simpler and yet more expressive. Moreover, the system becomes so similar to Haskell's type system that integrating proposals for Haskell, such as (Peyton Jones et al., 2007) or (Leijen, 2008a), becomes entirely straightforward.

We formalize the core type system in Chapter 7 and prove that the type system is sound (progress and type preservation). That is, we prove that if the type system accepts a program, then that program will not get stuck when evaluated.

Finally, we present our conclusions and identity future work in Chapter 8.

Background

This chapter provides the technical background that we will need in the remainder and establishes terminology. Most sections in this chapter constitute large topics in their own right, and we can give but a brief overview here. An accessible introduction to type systems is given in (Pierce, 2002); a more theoretical but highly recommended textbook is (Sørensen and Urzyczyn, 2006). For substructural type systems we refer to (Walker, 2005) or (Wadler, 1993) (the latter deals with linear logic only). Finally, an excellent tutorial on type inference is (Peyton Jones et al., 2007). We delay an explanation of uniqueness typing itself until Chapter 3 on related work. p. 77

Although this chapter is long, it will set the context for the following chapters and it is an important part of our thesis that uniqueness typing can be firmly based on the standard functional type theory. Moreover, although every section is tailored to the specific requirements of our work, we have tried to include references to related work wherever possible to provide the reader with follow-up material. Finally, various sections contain discussions of a less technical nature in which we discuss some design choices in the foundations of our work.

2.1 Logic

As mentioned in the introduction, type systems can be viewed as logics. For the type systems we consider in this dissertation, however, the corresponding logics are all *constructive*: the law of the excluded middle does not hold. Since this is a contentious issue, Section 2.1.1 presents a slightly tongue-in-cheek introduction to and justification for constructive logic. The remaining two subsections introduce two different styles of presenting *proofs* in logic: natural deduction style and sequent style.

2.1.1 On constructive logic

In the Western world, logic is generally traced back to Aristotle. In the fourth book of the *Metaphysica*, Aristotle gives what he considers two fundamental laws:

> “ Τὸ γὰρ αὐτὸ ἅμα ὑπάρχειν τε καὶ μὴ ὑπάρχειν ἀδύνατον τῷ αὐτῷ καί κατά τὸ αὐτό (...) Ἀλλὰ μὴν οὐδὲ μεταξὺ ἀντιφάσεως ἐνδέχεται εἶναι οὐθέν, ἀλλ' ἀνάγκη ἢ φάναι ἢ ἀποφάναι ἓν καθ' ἑνὸσ ὁτιοῦν.[1]
>
> Aristotle, *Metaphysica*, Γ.3 and Γ.7 (*ca.* 330 BC)

[1] It is impossible for the same attribute at once to belong and not to belong to the same thing and in the same relation (...) Nor indeed can there be any intermediate between contrary statements, but of one thing we must either assert or deny one thing, whatever it may be. (Tredennick, 1933)

Less well-known are the texts written by members of the Mohist school founded by a teacher called Mozi (墨子) in ancient China (these texts are themselves also collectively known as the 墨子). There are many parallels between these texts and the ancient Western texts. In particular, the same two laws are stated:

> Furthermore, the Mohist Canons propose basic principles regulating disputations. The first says that of two contradictory propositions, one must be false: they cannot be true at the same time: "是不俱当，不俱当必或不当。" This is exactly the Law of Non-Contradiction! Secondly, the texts say that two contradictory propositions cannot be both false, one of them must be true: "谓辩无胜，必不当，说在辩。" This, of course, is the Law of Excluded Middle.
>
> (Zhang and Liu, 2007, *Some Thoughts on Mohist Logic*)

Today, few people object to the law of non-contradiction, but the law of the excluded middle is not (quite) so widely accepted. One reason for rejecting it is the existence of logical paradoxes. Interestingly, both the ancient Greek and the Chinese were aware of these paradoxes. A famous paradox is attributed to the Cretan philosopher who claimed that Κρῆτες ἀεί ψεύσται, "Cretans are always liars"[1]. In the 墨子 we find 以言为尽悖，悖, "To claim that all saying contradicts itself is self-contradictory". Consider the following paradox:

This statement is false.

(This statement asserts something about itself.) By the law of the excluded middle, that statement must either be true or false. If it is true, then it must be false (since that is what it asserts): contradiction. But if it is false, then it must be false that it is false (again, since that is what it asserts), and thus (by another application of the same law) it must be true: contradiction again. Hence, this can be considered as a counter-example to the law of the excluded middle.

While this may seem like linguistic curiosity, this kind of self-reference forms the basis for many of the most fundamental results in mathematics in the twentieth century (see Hofstadter, 1999, for an engaging exposition). The most famous of these is probably Russell's paradox (van Heijenoort, 1967), which upset the foundations of set theory. Consider the set R of all sets that are not members of themselves; is R a member of R? Again, by the law of the excluded middle, the answer must be yes or no; but both answers lead to contradiction.

Russell's paradox led him to invent type theory (see Kamareddine et al., 2004, for a historical overview) which ensured that sets such as R cannot be defined. This disqualifies our counter-example, and the law of the excluded middle is still accepted.

But there are other, perhaps more serious, objections to the law of the excluded middle. The most important is that logic or mathematics should not attempt to define "truth" (a metaphysical notion), but define provability instead:

> If "to exist" does not mean "to be constructed", it must have some metaphysical meaning. It cannot be the task of mathematics to investigate this meaning or to decide whether it is tenable or not. We have no objection against a mathematician privately admitting any metaphysical theory he likes, but [...] we study mathematics as something simpler, more immediate than metaphysics.
>
> (Heyting, 1966, *Intuitionism: An Introduction*)

[1] And mean and fat, too: "Even one of their own men, a prophet from Crete, has said about them, 'The people of Crete are all liars, cruel animals, and lazy gluttons.' " (Titus 1:12, *New Living Translation*).

This philosophical stance is usually traced back to the Dutch mathematician Brouwer, and was developed into a formal logic by another Dutch mathematician, and student of Brouwer's, Arend Heyting ("Hey" is pronounced /ɦɛi/, not like the English "hey", /ɦeɪ/) and the Russian Andrey Kolmogorov (Kuiper, 2004, Chapter 7). Incidentally, Brouwer himself did not agree with this development:

> We willen tonen, dat de wiskunde onafhankelijk is van de zogenaamde logische wetten, (wetten van redenering of van menselijk denken). Dit schijnt paradox, want wiskunde wordt gewoonlijk gesproken en geschreven als bewijsvoering, afleiding van eigenschappen, en in de vorm van een aaneenschakeling van syllogismen.[1]
>
> (Brouwer, 1907, *Over the Grondslagen der Wiskunde*)

A proof of a property σ in constructive logic can be viewed as an algorithm (a program) that *constructs* a proof of σ. This makes it extremely suited as the "logic of computer science", and we will see various other aspects of this view later in this chapter. A common interpretation of constructive logic defines very concretely what is accepted as a proof of a compound statement; this is known as the Brouwer-Heyting-Kolmogorov or BHK interpretation (Troelstra and Schwichtenberg, 1996, Section 2.5.1):

- A proof of $\sigma \wedge \sigma'$ is a pair of proofs (e, e') such that e proves σ and e' proves σ'
- A proof of $\sigma \vee \sigma'$ is either a proof of σ or a proof of σ'
- A proof of $\sigma \longrightarrow \sigma'$ is an function transforming a proof of σ into a proof of σ'
- $\perp$ (false) does not have any proofs
- A proof of $\forall x \in \sigma . \sigma'$ is a function that returns a proof of $\sigma'[x := e]$ for any e in σ (this is a generalization of proofs of $\sigma \longrightarrow \sigma'$)
- A proof of $\exists x \in \sigma . \sigma'$ is a pair $(e,\ p)$ of an element e in σ together with a proof p of $\sigma'[x := e]$ (this is a generalization of proofs of $\sigma \wedge \sigma'$)

Lest the reader got the impression that the case of the law of the excluded middle has been settled, let me assure you that this is not the case. Consider the following puzzle.

> Three gods A, B, and C are called, in some order, "True", "False", and "Random". True always speaks truly, False always speaks falsely, but whether Random speaks truly or falsely is a completely random matter. Your task is to determine the identities of A, B, and C by asking three yes-no questions; each question must be put to exactly one god. The gods understand English, but will answer all questions in their own language, in which the words for "yes" and "no" are "da" and "ja", in some order. You do not know which word means which.
>
> (Boolos, 1996, *The Hardest Logic Puzzle Ever*)

The exact solution is not important (it is given in Boolos's paper) but Boolos makes the following observation:

[1] We want to show, that mathematics is independent of the so called logical laws, (laws of reasoning or of human thought). This seems paradoxical, for usually mathematics is expressed, orally or in writing, in the form of argumentation, deduction of properties, by means of a chain of syllogisms. (Kuiper, 2004)

> Mathematicians and philosophers have occasionally attacked the idea that excluded middle is a logically valid law. We can't hope to settle the debate here, but [...] it is clear from The Hardest Logic Puzzle Ever [...] that our ability to reason about alternative possibilities, even in everyday life, would be almost completely paralysed were we to be denied the use of the law of excluded middle.

Needless to say, I cannot conclude this section on that quote. Instead, I will leave the reader with the following observation from Rabern and Rabern (2008): what happens when we ask each god:

Are you going to answer "ja" to this question?

2.1.2 Natural deduction

The idea of a mathematical proof is as old as mathematics, but the study of the structure of proofs (structural proof theory) as a separate subject is a relatively recent development. It is generally traced back to Gentzen (Negri and Plato, 2001), who developed two different styles of presenting proofs: natural deduction style and sequent style. We will almost exclusively use natural deduction style in this dissertation; however, we will briefly present sequent style in the next section because we will need it in the discussion of related work.

A *judgement* in natural deduction takes the form $\Gamma \vdash \sigma$ (read as "Γ proves σ"), where Γ is a list[1] of assumptions and σ is a proposition that is supposed to follow from those assumptions. The
p. 21 specific logic dictates what is considered to be a valid judgement. Figure 2.1 shows propositional logic in natural deduction style.

For every logical connective (implication $\rightarrow$, conjunction $\wedge$ and disjunction $\vee$) there are *introduction* rules and *elimination* rules. The rules all have the shape

$$\frac{\text{premises}}{\text{conclusion}}\ \text{Rule name}$$

Such a rule states that the *conclusion* can be derived using rule *name* if you can find a proof for the *premises* of the rule. If a rule has no premises, the conclusion can trivially be derived and the rule is known as an axiom. An introduction rule for a connective $\diamond$ tells us how to *prove* a judgement $\sigma \diamond \sigma'$; an elimination rule tells us what *follows* from an assumption $\sigma \diamond \sigma'$. For example, to prove $\sigma \wedge \sigma'$ (σ and σ'), we need to prove both σ and σ' (rule $\wedge$I), but from an assumption $\sigma \wedge \sigma'$ we can conclude both σ and σ' (rules $\wedge\text{E}_1$ and $\wedge\text{E}_2$).

A *typing judgement* for a program e takes the form $\Gamma \vdash e : \sigma$, and a *typing derivation* is a proof that e has type σ. In this context, Γ is a list of assumptions about the types of the free variables in e; throughout this dissertation we will implicitly assume an invariant that every variable occurs at most once inside an environment.[2] Of course, this is a change in syntax only; a logic *is* a type system, a typing judgement *is* a logical judgement and a typing derivation *is* a logical derivation. To be completely explicit about it, Figure 2.1 also shows the simply typed lambda calculus, which is the Curry-Howard mirror image of propositional logic. The two systems are near identical; the simply typed lambda calculus can be seen as a syntax for encoding proofs in propositional logic (as well as a typed programming language).

[1] Although we will not get back to this point until Section 2.6, it is important to note that we do not use a *set* of assumptions but a *list* of assumptions.

[2] Strictly speaking, this will make some programs (such as $\lambda x \cdot \lambda x \cdot x$) untypeable. This can be resolved informally by referring to the "Barendregt convention", but we will not worry about such technicalities until our formal proofs in Chapter 7, where we will discuss this issue at some length in Section 7.2.

Propositional logic

$$\frac{\sigma \in \Gamma}{\Gamma \vdash \sigma}\ \textsc{Ax}$$

$$\frac{\Gamma, \sigma \vdash \sigma'}{\Gamma \vdash \sigma \rightarrow \sigma'}\ {\rightarrow}\mathrm{I} \qquad \frac{\Gamma \vdash \sigma \rightarrow \sigma' \qquad \Gamma \vdash \sigma}{\Gamma \vdash \sigma'}\ {\rightarrow}\mathrm{E}$$

$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \sigma'}{\Gamma \vdash \sigma \wedge \sigma'}\ \wedge\mathrm{I} \qquad \frac{\Gamma \vdash \sigma \wedge \sigma'}{\Gamma \vdash \sigma}\ \wedge\mathrm{E}_1 \quad \frac{\Gamma \vdash \sigma \wedge \sigma'}{\Gamma \vdash \sigma'}\ \wedge\mathrm{E}_2$$

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma \vee \sigma'}\ \vee\mathrm{I}_1 \quad \frac{\Gamma \vdash \sigma'}{\Gamma \vdash \sigma \vee \sigma'}\ \vee\mathrm{I}_2 \qquad \frac{\Gamma \vdash \sigma \vee \sigma' \qquad \Gamma, \sigma \vdash C \qquad \Gamma, \sigma' \vdash C}{\Gamma \vdash C}\ \vee\mathrm{E}$$

Simply-typed lambda calculus

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{Var}$$

$$\frac{\Gamma, x : \sigma \vdash e : \sigma'}{\Gamma \vdash \lambda x \cdot e : \sigma \rightarrow \sigma'}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e : \sigma \rightarrow \sigma' \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash e\, e' : \sigma'}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash e' : \sigma'}{\Gamma \vdash (e, e') : \sigma \wedge \sigma'}\ \textsc{Pair} \qquad \frac{\Gamma \vdash e : \sigma \wedge \sigma'}{\Gamma \vdash \mathrm{fst}\, e : \sigma}\ \textsc{Fst} \quad \frac{\Gamma \vdash e : \sigma \wedge \sigma'}{\Gamma \vdash \mathrm{snd}\, e : \sigma'}\ \textsc{Snd}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathrm{Left}\, e : \sigma \vee \sigma'}\ \textsc{Left} \quad \frac{\Gamma \vdash e : \sigma'}{\Gamma \vdash \mathrm{Right}\, e : \sigma \vee \sigma'}\ \textsc{Right}$$

$$\frac{\Gamma \vdash e : \sigma \vee \sigma' \qquad \Gamma, x : \sigma \vdash e : C \qquad \Gamma, x : \sigma' \vdash e' : C}{\Gamma \vdash \mathrm{case}\, e\, \mathrm{of}\, \{\mathrm{Left}\, x \rightarrow e; \mathrm{Right}\, x \rightarrow e'\} : C}\ \textsc{Case}$$

Figure 2.1: Natural deduction

$$\dfrac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\,\text{Ax}$$

$$\dfrac{\Gamma \vdash e' : \sigma \qquad \Gamma, x : \sigma' \vdash e : \sigma''}{\Gamma, y : \sigma \to \sigma' \vdash e[x := y\, e'] : \sigma''}\,{\to}\text{Left} \qquad \dfrac{\Gamma, x : \sigma \vdash e : \sigma'}{\Gamma \vdash \lambda x \cdot e : \sigma \to \sigma'}\,{\to}\text{Right}$$

$$\dfrac{\Gamma \vdash e' : \sigma \qquad \Gamma, x : \sigma \vdash e : \sigma'}{\Gamma \vdash e[x := e'] : \sigma'}\,\text{Cut}$$

Figure 2.2: Sequent Calculus

2.1.3 Sequent calculus

Judgements in the sequent calculus take the same form as judgements in natural deduction[1], but the logic is presented differently. Instead of an introduction rule and elimination rule for each connective, there is a "left" introduction rule which introduces the connective as an assumption
p. 22 and a "right" introduction rule which introduces the connective as a conclusion. Figure 2.2 shows the sequent calculus for the implicational fragment of propositional logic (that is, considering implication as the only connective).

It is not immediately obvious how function application can be typed in the sequent calculus, since there is no explicit application rule. However, we can derive a rule for application as follows (adopted from Barendregt and Ghilezan, 2000, Section 4).

$$\dfrac{\dfrac{\Gamma \vdash e : \sigma \to \sigma' \qquad \dfrac{\dfrac{\Gamma \vdash e' : \sigma \qquad \dfrac{\overline{x : \sigma' \in \Gamma, x : \sigma'}}{\Gamma, x : \sigma' \vdash x : \sigma'}\,\text{Ax}}{\Gamma, y : \sigma \to \sigma' \vdash x[x := y\, e'] : \sigma'}\,{\to}\text{Left}}{\Gamma, y : \sigma \to \sigma' \vdash y\, e' : \sigma'}\,=}{\Gamma \vdash (y\, e')[y := e] : \sigma'}\,\text{Cut}}{\Gamma \vdash e\, e' : \sigma'}\,= \tag{2.1}$$

Sequent calculi are important for proving proof-theoretic properties, but are less intuitive to work with and we will not make use of them except in the discussion of the work of Dana Harrington in
p. 91 Section 3.4.

2.2 Universal Algebra

p. 29 Before we will start introducing type systems in Section 2.3, we will first introduce a number of results from universal algebra. Universal algebra is the study of the syntax and semantics of terms and related operations; we will only need a few results from universal algebra, and we will not
p. 39, p. 59 need them until sections 2.5 and 2.8, but introducing them here avoids interrupting the flow later in this chapter.

Most of the definitions in this section are taken from (Baader and Nipkow, 1998).

[1] For classical logic, the judgements take a slightly different form: there can be an arbitrary number of conclusions after the turnstile. However, for constructive logic the number of conclusions is limited to one, as in natural deduction.

2.2.1 Basic Definitions

Terms are built up from (uninterpreted) function symbols and variables. For example, if f is a binary function and x and y are variables, then $f(x, y)$ is a term. A *signature* Σ defines the arity of each function. For example, the signature of the types in the simply typed lambda calculus with natural numbers is $\{(\rightarrow) \mapsto 2, \mathbb{N} \mapsto 0\}$ with a binary operator $(\rightarrow)$ (the function space constructor) and a constant (nullary) operator for the type of natural numbers.

We can be explicit about which variables we want to allow by considering the set of terms $T(\Sigma, X)$ over a signature Σ and set of variables X. We always assume that $\text{dom}(\Sigma) \cap X = \emptyset$, that is, no symbol can be both a variable and a function symbol. A *ground term* is a term which is built-up from function symbols only (contains no variables).

A *substitution* is a function $S : X \longrightarrow T(\Sigma, X)$. The set of variables for which S is not the identity is called the *domain* of the substitution (note that $X \subset T(\Sigma, X)$). A substitution can be extended to a function $\hat{S}$ over terms in the obvious way. The composition $S_2 \circ S_1$ of two substitutions S_2 and S_1 is defined as $(S_2 \circ S_1)(x) = \hat{S}_2(S_1(x))$.

A substitution S is *more general* than a substitution S' if there is a substitution S'' such that $S' = S'' \circ S$; S' is said to be an *instance* of S.

An identity $T \approx T'$ is a pair of terms (possibly containing variables). For example, we can assert that a binary function f is associative by giving the identity $f(x, f(y, z)) \approx f(f(x, y), z)$. A set of identities E leads to *equational logic*, shown in Figure 2.3. p. 24

2.2.2 Unification

Given a set of identities E and two terms T and T', the problem of finding a substitution or *unifier* S such that

$$E \vdash ST \approx ST'$$

known as *equational unification*. We will often use the notation

$$E \vdash T \overset{?}{\approx} T'$$

for denoting a unification problem. We will sometimes omit the set of identities (E) if this is clear from the context, simply writing $T \overset{?}{\approx} T'$.

A unifier S is *most general* if is more general than any other unifier: that is, if for all other unifiers S' there exists a unifier S'' such that $S' = S'' \circ S$.[1] Most general unifiers are useful in the context of type inference, where they allow us to infer most general types (Section 2.5.1). p. 41

A unifier S is *reproductive* if for all other unifiers S', $S' \circ S = S'$. It is not hard to see that a reproductive unifier must be most general (take $S'' = S'$). Although for our purposes a reproductive unifier is not more useful than a most general unifier, it is sometimes easier to prove that a unifier is reproductive than to prove that it is most general.

General equational unification is undecidable, but we will consider two special cases in the remainder of this section, both of which are decidable: *syntactic unification* or simply *unification* (unification under the empty set of identities) and *boolean unification* (unification under Huntington's set of postulates).

[1] We identify substitutions which map equivalent terms to equivalent terms.

$$\frac{(T \approx T') \in E}{E \vdash T \approx T'}\ \textsc{Assump}$$

$$\frac{}{E \vdash T \approx T}\ \textsc{Refl} \qquad \frac{E \vdash T' \approx T}{E \vdash T \approx T'}\ \textsc{Sym} \qquad \frac{E \vdash T \approx T' \qquad E \vdash T' \approx T''}{E \vdash T \approx T''}\ \textsc{Trans}$$

$$\frac{E \vdash T \approx T'}{E \vdash ST \approx ST'}\ \textsc{Subst} \qquad \frac{E \vdash T_1 \approx T_1' \qquad \ldots \qquad E \vdash T_n \approx T_n'}{E \vdash f(T_1, \ldots, T_n) \approx f(T_1', \ldots, T_n')}\ \textsc{Closure}$$

Figure 2.3: Equational logic

2.2.3 Syntactic unification

Syntactic unification (often simply "unification") is the problem of finding a substitution S such that $\emptyset \vdash ST \approx ST'$ or, equivalently, such that $ST = ST'$. Syntactic unification was invented by Robinson (1965), but the (slightly more general) description we give here is based on a well-known paper by Martelli and Montanari (1982). To unify two terms T and T', start with the singleton set of equations $\{T \approx T'\}$ and repeatedly apply any of the following transformations.

1. Select any equation of the form
$$T \approx x$$
where T is not a variable and x is a variable, and rewrite it as $x \approx T$.

2. Select any equation of the form
$$x \approx x$$
where x is a variable and erase it.

3. Select any equation of the form
$$f(T_1, T_2, \ldots, T_n) \approx g(T_1', T_2', \ldots, T_m')$$
If $f \neq g$ stop with failure. Otherwise we must have $n = m$; replace the equation by
$$\begin{gathered} T_1 \approx T_1' \\ T_2 \approx T_2' \\ \vdots \\ T_n \approx T_n' \end{gathered}$$

4. Select any equation of the form
$$x \approx T$$
where x is a variable which occurs somewhere else in the set of equations and where $T \neq x$. If x occurs in T, stop with failure (this is known as the "occurs check"). Otherwise, apply the substitution $[x := T]$ to all other equations (without erasing the equation $x \approx T$).

If no transformation applies, the result will be of the form $\{x_1 = T_1, x_2 = T_2, \ldots, x_n = T_n\}$, which is the substitution (or unifier) S that we are looking for. Moreover, this unifier will be a *most general unifier* (*mgu*): it will be more general than any other unifier (Martelli and Montanari, 1982, Section 2).

Commutativity	Unit elements
$x \vee y \approx y \vee x$	$x \vee 0 \approx x$
$x \wedge y \approx y \wedge x$	$x \wedge 1 \approx x$
Distributivity	Complements
$x \vee (y \wedge z) \approx (x \vee y) \wedge (x \vee z)$	$x \vee \neg x \approx 1$
$x \wedge (y \vee z) \approx (x \wedge y) \vee (x \wedge z)$	$x \wedge \neg x \approx 0$

Figure 2.4: Huntington's postulates

The algorithm as described here is non-deterministic and inefficient; we refer the reader to the Martelli and Montanari paper for suggestions for improvement. For efficiency reasons, the occurs check is often omitted in the implementation of logic programming languages (Overton, 2003, p. 31); however, an implementation of unification for the purposes of type inference (Section 2.5.1) *must* include the occurs check since it is easily violated; for example, the type error in

```
self_apply f = f f
```

is detected by the occurs check.

2.2.4 Boolean Unification

Although Boolean Algebra is often attributed to Boole, this is not entirely accurate. The algebra proposed by Boole (1854) is not the same algebra as Boolean Algebra (Hailperin, 1981; Burris, 2001): the operations in Boole's algebra do not line up with the operations in Boolean algebra[1], and as Hailperin (1981) writes:

> If we look carefully [..], we find [Boole] carrying out operations, procedures, and processes of an algebraic character, often inadequately described by present-day standards and, at times, making no sense to a modern mathematician.

Boolean Algebra in its current form was developed by a contemporary of Boole, the logician William Jevons (see Styazhkin, 1969, for a historical perspective). Its most common presentation as the series of independent axioms shown in Figure 2.4 is due to Huntington (1904).

Boolean unification is the problem of finding a substitution S such that $B \vdash ST \approx ST'$, where B is the set of Huntington's postulates. There are two algorithms for boolean unification known as *Löwenheim's formula* and *successive variable elimination*. However, before we can describe these methods, we will need some preliminary results first.

We define the following convenient derived operator, which acts as a conditional; read as "if b then P else Q":

$$P \lhd b \rhd Q \qquad \equiv \qquad (b \wedge P) \vee (\neg b \wedge Q)$$

One reason that this operator is a convenient abstraction is that the other boolean operators distribute nicely over the conditional (we will prove these results in Section A.1.6): p. 228

$$B \vdash (P \lhd b \rhd Q) \vee R \approx P \vee R \lhd b \rhd Q \vee R \tag{2.2}$$

$$B \vdash (P \lhd b \rhd Q) \wedge R \approx P \wedge R \lhd b \rhd Q \wedge R \tag{2.3}$$

$$B \vdash \neg(P \lhd b \rhd Q) \approx \neg P \lhd b \rhd \neg Q \tag{2.4}$$

[1] Nor with the operations in a Boolean ring.

We will also need the following special case of (2.2):

$$B \vdash P \lhd b \rhd Q \approx (P \lhd b \rhd Q) \vee (P \wedge Q) \tag{2.5}$$

Since $B \vdash T \approx T'$ if and only if $B \vdash \neg T' \lhd T \rhd T' \approx 0$, it suffices to find a method to find a substitution S such that $B \vdash ST'' \approx 0$ for some term T''. Like syntactic unification, boolean unification has the property that if there exists a unifier S such that $B \vdash ST \approx ST'$, then there will exist a most general unifier; indeed, there will be a reproductive unifier. We will prove this shortly by giving an algorithm to produce a reproductive unifier for a unification equation $B \vdash T \overset{?}{\approx} T'$.

As long as we are doing unification over the algebra $(\Sigma = \{\wedge, \vee, \neg, 0, 1\}, X)$, finding *some* unifier such that $B \vdash ST \approx ST'$ is straight-forward, since we can reduce any ground expression in this algebra (that is, any term without variables) to either 0 or 1. Hence, we can simply try all substitutions $[x := 0]$ or $[x := 1]$ for all variables $x \in \mathrm{fv}(T) \cup \mathrm{fv}(T')$, evaluate both sides, and compare for (syntactic) equality. For example, to unify $B \vdash x \overset{?}{\approx} y$, we can try the substitutions

$$\begin{bmatrix} x := 0 \\ y := 0 \end{bmatrix} \qquad \begin{bmatrix} x := 0 \\ y := 1 \end{bmatrix} \qquad \begin{bmatrix} x := 1 \\ y := 0 \end{bmatrix} \qquad \begin{bmatrix} x := 1 \\ y := 1 \end{bmatrix}$$

to find two unifiers, $[x := 0, y := 0]$ and $[x := 1, y := 1]$. Similarly, to unify $B \vdash x \wedge y \overset{?}{\approx} x$ we try the same set of unifiers to find $[x := 0, y := 0]$, $[x := 0, y := 1]$ and $[x := 1, y := 1]$.

Although none of the unifiers we found are most general, we can translate any unifier into a most general unifier using a method known as *Löwenheim's formula*. If S is a unifier such that $B \vdash ST \approx 0$, then we can calculate a most general unifier S' using[1]

$$S'(x) := Sx \lhd T \rhd x$$

(see Baader and Nipkow, 1998, Section 10.4.3.) For example, consider trying to unify $B \vdash x \overset{?}{\approx} y$. I.e., we are trying to find a unifier S such that $B \vdash S(\neg y \lhd x \rhd y) \approx 0$. Based on the unifier $[x := 0, y := 0]$ we would find the following most general unifier:

$$\begin{aligned} \begin{bmatrix} x := 0 \lhd (\neg y \lhd x \rhd y) \rhd x \\ y := 0 \lhd (\neg y \lhd x \rhd y) \rhd y \end{bmatrix} &\approx \begin{bmatrix} x := \neg(\neg y \lhd x \rhd y) \wedge x \\ y := \neg(\neg y \lhd x \rhd y) \wedge y \end{bmatrix} \\ &\approx \begin{bmatrix} x := (y \lhd x \rhd \neg y) \wedge x \\ y := (y \lhd x \rhd \neg y) \wedge y \end{bmatrix} \approx \begin{bmatrix} x := x \wedge y \\ y := x \wedge y \end{bmatrix} \end{aligned}$$

It is easy to verify that this unifier is indeed more general than all other unifiers we have seen so far; it is also more general than the most obvious unifier $[x := y]$ (with "more general" strictly in p. 23 the sense of Section 2.2.1).

When we introduce uninterpreted constants into the algebra, however, we can no longer guess a unifier by instantiating all variables to either 0 or 1. For reasons that will become clear in p. 46 Section 2.5.3, we will refer to such constants as *skolem constants*. Suppose x is a variable and $\mathfrak{u}$ and $\mathfrak{v}$ are skolem constants (uninterpreted function symbol of arity 0); then $B \vdash x \overset{?}{\approx} \mathfrak{u} \vee \mathfrak{v}$ has a trivial unifier $[x := \mathfrak{u} \vee \mathfrak{v}]$, but we can no longer guess this unifier in the manner just outlined.

[1] The formula given by Baader and Nipkow (1998) uses an exclusive disjunction $\ovee$ instead of an inclusive disjunction, but $B \vdash (\neg x \wedge y) \ovee (x \wedge z) \approx (\neg x \wedge y) \vee (x \wedge z)$.

Fortunately, there is another method for boolean unification. It is known as *successive variable elimination* and was already described by Boole (1854, Chapter VII, *On Elimination*). Our explanation here is based on a combination of the method described in Brown (2003, Chapter 7, *Solutions of Boolean Equations*) and the method described in Baader and Nipkow (1998, Section 10.4.5, *Successive variable elimination*). It is however slightly different from both; the version we give transfers easily to an implementation in a functional language (the algorithms given in the books are a specification more than an implementation), and uses the operations in a boolean algebra rather than in a boolean ring (that is, we use inclusive rather than exclusive disjunction). Since it is therefore slightly non-standard, we provide a proof that the method is correct. The structure of this proof closely follows the proof given by Baader and Nipkow.

We are looking for a unifier S such that $B \vdash ST \approx 0$. If T is a ground term (contains no variables), then $ST = T$ for any substitution S. Hence, if T is ground, then either $B \vdash T \approx 0$ and the empty substitution will be a suitable "unifier" (which is trivially reproductive), or $B \nvdash t \approx 0$ and the unification equation has no solutions. The output of the successive variable elimination therefore is a substitution S and a *consistency condition*. The consistency condition cc is a ground term and we must have that $B \vdash cc \approx 0$ (this can easily be verified because any ground term can be reduced to 0 or 1). If $B \nvdash cc \approx 0$, the unification equation has no solutions.

If T is not ground, we choose a variable $x \in \mathrm{fv}(T)$. We eliminate x from T (in a manner to be described shortly) to obtain a term T'. We then recursively try to find a unifier S' such that $B \vdash S'T' \approx 0$, and then use S' to construct S. Since the number of free variables in t is finite and decreases on every recursion, this process must terminate.

Theorem 1 (Successive variable elimination) *Let T be a term with at least one free variable x, and let T_s be $T[x := s]$. Observe that $B \vdash T_x \approx T_1 \lhd x \rhd T_0$. We eliminate x from T by defining $e = T_1 \wedge T_0$. Then,*

1. *Every unifier of $T_x \overset{?}{\approx} 0$ is a unifier of $e \overset{?}{\approx} 0$.*
2. *If S_e is a reproductive unifier of $e \overset{?}{\approx} 0$ and $x \notin dom(S_e)$, then*

$$S_t := S_e \circ [x := T_0 \vee (x \wedge \neg T_1)]$$

is a reproductive unifier of $T_x \overset{?}{\approx} 0$.

Proof (1). Let S be a unifier of $T_x \overset{?}{\approx} 0$, i.e. $B \vdash ST_x \approx S(T_1 \lhd x \rhd T_0) \approx 0$. We need to show that $B \vdash Se = S(T_1 \wedge T_0) \approx 0$.

$$\begin{array}{ll} & S(T_1 \lhd x \rhd T_0) \approx 0 \\ & \quad \{ \text{ by (2.5) } \} \\ \Rightarrow & S\Big((T_1 \lhd x \rhd T_0) \vee (T_1 \wedge T_0)\Big) \approx 0 \\ & \quad \{ \text{ distribute } S \ \} \\ \Rightarrow & S(T_1 \lhd x \rhd T_0) \vee S(T_1 \wedge T_0) \approx 0 \\ & \quad \{ \ a \vee b = 0 \Rightarrow a = 0 \text{ and } b = 0 \ \} \\ \Rightarrow & S(T_1 \wedge T_0) \approx 0 \qquad \square \end{array}$$

Proof (2). Let S_e be a reproductive unifier of $B \vdash e = T_1 \wedge T_0 \overset{?}{\approx} 0$. First we show that S_t is a unifier of $B \vdash T_x \approx 0$:

$$\begin{array}{ll}
 & S_t T_x \\
 & \quad \{ \text{ expand } T_x \text{ and distribute } S \ \} \\
= & S_t T_1 \lhd S_t x \rhd S_t T_0 \\
 & \quad \{ \text{ expand } S_t x \ \} \\
= & S_t T_1 \lhd S_e\Big(T_0 \vee (x \wedge \neg T_1)\Big) \rhd S_t T_0 \\
 & \quad \{ \ S_t T_1 = S_e T_1 \text{ since } x \notin \mathrm{fv}(T_1)\text{; similarly for } T_0 \ \} \\
= & S_e T_1 \lhd S_e\Big(T_0 \vee (x \wedge \neg T_1)\Big) \rhd S_e T_0 \\
 & \quad \{ \text{ extract } S_e \ \} \\
= & S_e\Big(T_1 \lhd T_0 \vee (x \wedge \neg T_1) \rhd T_0\Big) \\
 & \quad \{ \text{ Definition of } P \lhd b \rhd Q \ \} \\
= & S_e\bigg(\Big((T_0 \vee (x \wedge \neg T_1)) \wedge T_1\Big) \vee \Big(\neg(T_0 \vee (x \wedge \neg T_1)) \wedge T_0\Big)\bigg) \\
 & \quad \{ \text{ Distributivity } \} \\
\approx & S_e\bigg(\Big((T_0 \wedge T_1) \vee ((x \wedge \neg T_1) \wedge T_1)\Big) \vee \Big(\neg(T_0 \vee (x \wedge \neg T_1)) \wedge T_0\Big)\bigg) \\
 & \quad \{ \text{ Complements; zero element for conjunction } \} \\
\approx & S_e\bigg((T_0 \wedge T_1) \vee \Big(\neg(T_0 \vee (x \wedge \neg T_1)) \wedge T_0\Big)\bigg) \\
 & \quad \{ \text{ De Morgan } \} \\
\approx & S_e\bigg((T_0 \wedge T_1) \vee \Big((\neg T_0 \wedge \neg(x \wedge \neg T_1)) \wedge T_0\Big)\bigg) \\
 & \quad \{ \text{ Complements; zero element for conjunction } \} \\
\approx & S_e(T_0 \wedge T_1) \\
 & \quad \{ \text{ By assumption } \} \\
\approx & 0 \qquad \square
\end{array}$$

Finally, we need to prove that S_t is reproductive, i.e., for all unifiers S', $S' \circ S_t = S'$. For $y \neq x$ we have

$$\begin{array}{ll}
 & S'(S_t y) \\
 & \quad \{y \neq x\} \\
= & S'(S_e y) \\
 & \quad \{ \text{ by assumption, } S_e \text{ is reproductive} \} \\
\approx & S'y \qquad \square
\end{array}$$

The tricky part of the proof is the case for x. By assumption, S' is a unifier of $B \vdash T_x \approx T_1 \lhd x \rhd T_0 = (x \wedge T_1) \vee (\neg x \wedge T_0) \approx 0$. But that means that $B \vdash S'(x \wedge T_1) \approx 0$ and $B \vdash S'(\neg x \wedge T_0) \approx 0$. We will need these two results in the final part of the proof:

$$\begin{array}{ll}
 & S'(S_t x) \\
 & \quad \{ \text{ expand } S_t x \ \} \\
= & S'(S_e(T_0 \vee (x \wedge \neg T_1))) \\
 & \quad \{ \ S_e \text{ is reproductive } \} \\
\approx & S'(T_0 \vee (x \wedge \neg T_1)) \\
 & \quad \{ \text{ distribute } S' \ \} \\
= & S'T_0 \vee (S'x \wedge \neg S'T_1)
\end{array}$$

```
unify0 :: BooleanAlgebra a => a -> [Var] -> (Subst a, a)
unify0 $t$ $[]$ = $([], t)$
unify0 $t$ $(x : xs)$ = $(s_e \circ [x := t_0 \vee (x \wedge \neg t_1)], cc)$
  where
    $(s_e, cc) =$ unify0 $e\ xs$
    $e = t_1 \wedge t_0$
    $t_0 = t\ [x := 0]$
    $t_1 = t\ [x := 1]$
```

Figure 2.5: Boolean unification

(continued from the previous page)

$$
\begin{aligned}
& \quad \{S'x \wedge \neg S'x \approx 0\} \\
\approx \quad & \left(S'T_0 \vee (S'x \wedge \neg S'T_1)\right) \vee \left(S'x \wedge \neg S'x\right) \\
& \quad \{\text{ Associativity }\} \\
\approx \quad & S'T_0 \vee \left((S'x \wedge \neg S'T_1) \vee (S'x \wedge \neg S'x)\right) \\
& \quad \{(a \wedge b) \vee (a \wedge c) \approx a \wedge (b \vee c)\} \\
\approx \quad & S'T_0 \vee \left(S'x \wedge (\neg S'T_1 \vee \neg S'x)\right) \\
& \quad \{\text{De Morgan}\} \\
\approx \quad & S'T_0 \vee \left(S'x \wedge \neg S'(T_1 \wedge x)\right) \\
& \quad \{S'(x \wedge T_1) \approx 0 \text{ (see above)}\} \\
\approx \quad & S'T_0 \vee S'x \\
& \quad \{S'x \vee \neg S'x \approx 1\} \\
\approx \quad & \left(S'T_0 \vee S'x\right) \wedge \left(S'x \vee \neg S'x\right) \\
& \quad \{(a \vee b) \wedge (a \vee c) \approx a \vee (b \wedge c)\} \\
\approx \quad & S'x \vee (S'T_0 \wedge \neg S'x) \\
& \quad \{S'(\neg x \wedge T_0) \approx 0 \text{ (see above)}\} \\
\approx \quad & S'x \qquad \square
\end{aligned}
$$

The algorithm is implemented by unify$_0$, shown in Figure 2.5, which takes a term T in a boolean algebra a and the list of free variables in T as input, and returns a substitution and the "consistency condition", which will be zero if unification succeeded.

2.3 Explicitly typed lambda calculus

The untyped lambda calculus was introduced in Chapter 1. We saw that every term in the core p. 11 lambda calculus can be evaluated, and that this is no longer true when we introduce constants. For example, when we add natural numbers and booleans with associated operations, we can write nonsensical terms such as the logical conjunction of two natural numbers. The purpose of a type system is to reject programs that contain these nonsensical terms. There are many type systems for the lambda calculus; in this section, we will introduce various important calculi, guided by a taxonomy known as the *Barendregt cube* (Barendregt, 1991).

The Barendregt cube and its mirror-image under Curry-Howard

$$\begin{tikzcd}
& \lambda\omega \arrow[rr] & & \lambda C \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
& \lambda\underline{\omega} \arrow[uu, dashed] \arrow[rr, dashed] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[uu] \arrow[ur, dashed] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}
\qquad
\begin{tikzcd}
& \mathrm{PROP}\omega \arrow[rr] & & \mathrm{PRED}\omega \\
\mathrm{PROP2} \arrow[ur] \arrow[rr] & & \mathrm{PRED2} \arrow[ur] & \\
& \mathrm{PROP}\underline{\omega} \arrow[uu, dashed] \arrow[rr, dashed] & & \mathrm{PRED}\underline{\omega} \arrow[uu] \\
\mathrm{PROP} \arrow[uu] \arrow[ur, dashed] \arrow[rr] & & \mathrm{PRED} \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Term language

e ::=		expression
	x	variable
	c	constant
	$e\ e$	application
	$\lambda x : e \cdot e$	abstraction
	$\Pi x : e \cdot e$	cartesian product

General typing rules
Let s range over $\{*, \square\}$.

$$\frac{}{\vdash * : \square}\ \textsc{Axiom} \qquad \frac{x : e \in \Gamma \qquad \Gamma \vdash e : s}{\Gamma \vdash x : e}\ \textsc{Var}$$

$$\frac{\Gamma \vdash e : e' \qquad \Gamma \vdash e'' : s}{\Gamma, x : e'' \vdash e : e'}\ \textsc{Weak}$$

$$\frac{\Gamma \vdash e : \Pi x : e'' \cdot e''' \qquad \Gamma \vdash e' : e''}{\Gamma \vdash e\ e' : e'''[x := e']}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : e' \qquad \Gamma \vdash e'' : s \qquad e' =_\beta e''}{\Gamma \vdash e : e''}\ \textsc{Conv}$$

Specific rules
The specific rules are parametrized by two sorts s_1 and s_2. The permissible instantiations of s_1 and s_2 define the individual systems of the taxonomy and are listed at the bottom of this figure.

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma, x : e \vdash e' : s_2}{\Gamma \vdash (\Pi x : e \cdot e') : s_2}\ \textsc{Forall}$$

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma, x : e \vdash e' : e'' \qquad \Gamma, x : e \vdash e'' : s_2}{\Gamma \vdash (\lambda x : e \cdot e') : (\Pi x : e \cdot e'')}\ \textsc{Abs}$$

Rule pairs per system

$\lambda{\to}$	$(*,*)$				*with*	$(*,*)$	Terms depending on terms
$\lambda 2$	$(*,*)$	$(\square,*)$				$(\square,*)$	Terms depending on types
$\lambda\underline{\omega}$	$(*,*)$		$(\square,\square)$			$(*,\square)$	Types depending on terms
$\lambda\omega$	$(*,*)$	$(\square,*)$	$(\square,\square)$			$(\square,\square)$	Types depending on types
λP	$(*,*)$			$(*,\square)$			
$\lambda P2$	$(*,*)$	$(\square,*)$		$(*,\square)$			
$\lambda P\underline{\omega}$	$(*,*)$		$(\square,\square)$	$(*,\square)$			
λC	$(*,*)$	$(\square,*)$	$(\square,\square)$	$(*,\square)$			

Figure 2.6: The type systems of the Barendregt cube

2.3.1 Explicit versus implicit typing

There are two main styles of lambda calculus type systems: implicit and explicit. In an explicit type system ("Church-style"), every variable binder is annotated with a type. For example, the identity function on natural numbers is defined as

$$\lambda(x : \mathbb{N}) \cdot x$$

In an implicit type system ("Curry-style"), the types are not written down but are "implied". The identity function on natural numbers is now simply written as

$$\lambda x \cdot x$$

Of course, this term can equally be interpreted as the identity function on booleans or on any other type. This does not (necessarily) mean that it has more than one type, only that there is more than one type interpretation. To make this point clear, consider

$$(\lambda f \cdot (f\ 1, f\ \texttt{True}))\ (\lambda x \cdot x)$$

Since f is applied to both a natural number and a boolean, this program is rejected whether we choose to interpret the identity function as the identity function on natural numbers or on booleans. It is possible to assign a *polymorphic* type to the identity function, but we will delay a discussion of that possibility until Section 2.5 on implicit type systems. In the next section, we will consider p. 39 how this problem can be solved in an explicit type system.

2.3.2 Hierarchy of types

In the previous section we saw how to define the identity function on natural numbers. We can define an identity function that can be applied to *any* type as follows:

$$\lambda(a : *) \cdot \lambda(x : a) \cdot x$$

Now the identity takes *two* arguments: a type a and then an element x of type a. We can rewrite the example from the previous section as[1]

$$(\lambda f \cdot (f\ \mathbb{N}\ 1, f\ \mathbb{B}\ \texttt{True}))\ (\lambda(a : *) \cdot \lambda(x : a) \cdot x)$$

The type annotation on a $(a : *)$ deserves an explanation. To denote that "5 is a natural number", we write $5 : \mathbb{N}$. A type can be interpreted as a set, and the statement above can be interpreted as $5 \in \mathbb{N}$ (indeed, according to folklore the ":" symbol is derived from the "$\in$" symbol, which in turn is a stylized version of the first character of ἐστί, Greek for "it is".) Given that we can interpret a type as a set, it is natural to consider the type of types. A set of types is known as a *universe*, and the universe of small types (the types of terms) is often denoted by $*$. So, we have

$$\mathbb{N} : *$$

[1] For simplicity, we do not show the (necessary) type annotation on f; the type of f will be discussed in Section 2.3.4.

In words, the type of $\mathbb{N}$ is $*$. The type of a type is sometimes also referred to as the *kind* of a type (Jones, 1993). Thus, the *type* of "5" is $\mathbb{N}$, and the *kind* of "$\mathbb{N}$" is $*$. The type annotation on the identity function now makes sense: we want an argument a of kind $*$ (a small type), and an argument x of type a.

In the context of this section the distinction between types and kinds is somewhat artificial as we can continue the hierarchy and introduce the next universe: the set of the types of types (or, if you prefer, the set of kinds). Barendregt denotes this universe by $\Box$. Thus, we have

$$* : \Box$$

(You could read that statement as "star is a kind"). Although Barendregt does not continue the hierarchy further than $\Box$, it is not hard to see that the hierarchy can be continued *ad infinitum* (see p. 34 also Section 2.3.5).

An alternative approach is to collapse the hierarchy into a single universe $*$ (or `Type`), and assume that $*$ is a member of itself:

$$* : *$$

This should remind the reader of Russell's paradox, and indeed assuming $* : *$ leads to logical inconsistency: all propositions are provable; or, from a computational point of view, non-termination can no longer be guaranteed (Coquand and Herbelin, 1994). However, if the language is intended for programming rather than theorem proving and supports general recursion so that the system is logically inconsistent regardless[1], assuming $* : *$ simplifies the type system (without sacrificing computational soundness) and may thus be a valid choice (Altenkirch and Oury, 2008).

2.3.3 Dependency

When we apply the identity function on natural numbers $\lambda(x : \mathbb{N}) \cdot x$ to a term (say, 5) we obtain another term; hence, this is a term that depends on a term. Similarly, when we apply the generalized identity function $\lambda(a : *) \cdot \lambda(x : a) \cdot x$ to a type (say, $\mathbb{N}$), we obtain a term $\lambda(x : \mathbb{N}) \cdot x$. Hence, the generalized identity function is a term that depends on a type.

Likewise, we can construct types that depend on types or types that depend on terms. E.g.,

$$\lambda(a : *) \cdot a \times a$$

construct the type of pairs of elements of type a given a type a. Hence, this is a type that depends on a type. Finally, consider the proposition

$$n =_{\mathbb{N}} m$$

for two natural numbers n and m. Under the Curry-Howard isomorphism this proposition is considered to be a type; a proof of the proposition is a term of the corresponding type. Now

$$\lambda(n : \mathbb{N}) \cdot n =_{\mathbb{N}} 0$$

constructs the type of proofs that some natural number n equals 0. Hence, this is a type that depends on on a term.

[1] The term $\mu x \cdot x$ has type A for any A, including $A = \bot$.

The type systems in the Barendregt cube are classified by how many of these dependencies they support. All systems support terms depending on terms, so there are three dimensions left: support for terms depending on types, types depending on types and types depending on terms (types depending on terms are also simply known as *dependent* types).

2.3.4 The expression language

In most languages, there is a separate term language (the language of programs) and type language (the language of types of programs). As we saw, however, no such distinction is made in the Barendregt cube. Instead, the distinction is made by the type system. An expression e (say, 5) that would traditionally be thought of as a term (a program) will have a type σ ($\mathbb{N}$) where the kind of σ is $*$. A *type* (say, $\mathbb{N}$) on the other hand will have a type σ ($*$) where the type of σ is $\Box$. More succinctly, e is a program when $e : \sigma : *$ and a type when $e : \sigma : \Box$.

The expression language is shown in Figure 2.6. It is the core lambda calculus (Chapter 1), p. 30, p. 11 except that variables bound by a lambda abstraction are annotated with the type of the variable, and the language is extended with a "cartesian product", which we will look at now. In the previous sections, we considered four different functions. The simplest was the identity function on natural numbers

$$\lambda(x : \mathbb{N}) \cdot x$$

What is the type of this function? The type of functions (the *function space*) from a type a to a type b is usually denoted by $a \longrightarrow b$; hence, the type of the identity function above is

$$\mathbb{N} \longrightarrow \mathbb{N}$$

Similarly, we have

$$(\lambda(a : *) \cdot a \times a) : * \longrightarrow *$$

and

$$(\lambda(n : \mathbb{N}) \cdot n = 0) : \mathbb{N} \longrightarrow *$$

But what is the type of the generalized identity function?

$$\lambda(a : *) \cdot \lambda(x : a) \cdot x$$

Given a type a, this will be a function from a to a; hence, we need a binder at the type level, which we will denote by Π.[1] Thus, the type of the generalized identity function is

$$\Pi(a : *) \cdot a \longrightarrow a$$

This construct is known as the *cartesian product* or the *dependent function space* and generalizes the "ordinary" function space; we can read $a \longrightarrow b$ as shorthand for $\Pi(x : a) \cdot b$ for some variable x that does not appear free in b.

[1] In the concrete syntax of Coq, this is denoted by $\forall$: $\forall(a : *) \cdot a \longrightarrow a$. However, we want to reserve this notation for the "implicit polymorphism" introduced in Section 2.5.

2.3.5 Calculus of Constructions

All the proofs in this dissertation are written in the language of *Coq*, a proof assistant developed in Inria (Bertot and Casteran, 2004). The logic of Coq is an extension of the calculus of constructions (λC in the Barendregt cube or higher order predicate logic). It supports an infinite hierarchy of universes; $*$ is called `Set`, $\square$ is called `Type(0)`, and the hierarchy continues

$$\texttt{Type}(i) : \texttt{Type}(i+1)$$

In addition, Coq has a universe `Prop` at the same level as `Set` (that is, both $\texttt{Prop} : \texttt{Type}(0)$ and $\texttt{Set} : \texttt{Type}(0)$). The difference between `Set` and `Prop` is not important for our current purposes; briefly, elements of a type in `Prop` represent proofs and can be removed when a program is compiled, whereas elements of a type in `Set` represent computational content and need to be retained.

Furthermore, Coq supports inductive and co-inductive data types. In the remainder of this section, we will discuss inductive data types (an explanation of co-induction is beyond the scope of this dissertation). For a more formal exposition, the reader is referred to (Paulin-Mohring, 1993) or (Pfenning and Paulin-Mohring, 1990).

So far, we have assumed the existence of a type of natural numbers and a type of booleans, and we have seen the (dependent) function space constructor which, given two types a and b, constructs the type of the functions from a to b. Inductive data types provide a mechanism to build up new types from scratch. We will demonstrate this by example, using the concrete syntax from Coq.

The simplest type possible is the type that has no inhabitants (the empty set). This type is called `False` in Coq since the set of proofs of the proposition *False* is the empty set (by definition). It is defined as follows:

```
Inductive False : Prop := .
```

The fact that there is nothing in between the "`:=`" and the "`.`" indicates that `False` is the empty type. After the empty type, we can introduce the singleton or unit type (the type that has exactly one inhabitant). In Coq, this type is called `True`.

```
Inductive True : Prop :=
  | I : True.
```

There is one *constructor* for type `True`, and it is called `I`. That is, we have

$$\texttt{I} : \texttt{True}$$

Given two types A and B, we can construct the type of pairs of elements A and B:

```
Inductive prod (A : Type) (B : Type) : Type :=
  | pair : A → B → prod A B.
```

The `pair` constructor takes an argument of type A and an argument of type B, and returns an element of type `prod` A B. For example, we have

$$\texttt{pair I I} : \texttt{prod True True}$$

We can introduce some syntactic sugar and rewrite this as

$$(\texttt{I,I}) : \texttt{True} \times \texttt{True}$$

Similarly, we can construct the sum of two types (this type is called `Either` in Haskell):

```
Inductive sum (A : Type) (B : Type) : Type :=
  | inl : A → sum A B
  | inr : B → sum A B.
```

Inductive data types also give us (restricted) recursion at the type level. For example, we can construct the type of natural numbers as follows:

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

This is the Peano encoding of the natural numbers; we can of course introduce syntactic sugar and simply write n for $\texttt{S}^n$ `O`. Finally, we can define the type of lists of elements of a type A as

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A → list A → list A.
```

Here too we will introduce some syntactic sugar: we will denote the type `list` A as $[A]$, and we will use $[1,2,3]$ to denote the list

```
cons 1 (cons 2 (cons 3 nil))
```

Inductive data types must be positive: no negative occurrences of the data type that is being defined may occur inside its definition. The exact definition of positivity is rather involved and we refer to the Coq reference manual instead (Coq Development Team, 2006, Section 4.5.3, *Well-formed inductive definitions*). The purpose of the positivity requirement is that all programs in Coq must terminate; we note here only that X occurs negatively in $X \longrightarrow A$. The data type that allows the implicit recursion trick from Section 2.9.1 therefore cannot be defined in Coq. p. 70

2.4 System F$_\text{A}$

As we will see in Section 2.5, the type annotations in the explicitly typed lambda calculus make p. 39
programs laborious to write and difficult to read. We prefer to omit them, but without loosing the expressive power of the original type system. Of course, this is easier or harder depending on what type system we start with; the current focus in functional language research is System F$_\text{A}$.

Unfortunately, the literature gets a bit sloppy in its use of terminology here and it is often claimed that System F is the reference calculus (for example Botlan and Rémy, 2003; Leijen, 2008a). In this section, we will explain what System F is and why it is unsuitable as a reference calculus for modern functional languages, and then introduce System F$_\text{A}$.

Note that the name "System F$_\text{A}$" is non-standard as far as we are aware (partly because most people simply refer to "System F" but actually mean a larger calculus); it was coined by Sulzmann et al. (2007).

Term and type language

e ::=		term	σ ::=		type
	x	variable		a	type variable
	$\lambda(x:\sigma)\cdot e$	lambda abstraction		$\sigma \to \sigma$	function space
	$e\,e$	application		$\Pi(a:*)\cdot\sigma$	universal type
	$\Lambda(a:*)\cdot e$	type abstraction			
	$e\;\{\sigma\}$	type application			

Typing rules

$$\frac{x:\sigma\in\Gamma \qquad \Gamma\vdash\sigma:*}{\Gamma\vdash x:\sigma}\;\textsc{Var}$$

$$\frac{\Gamma,x:\sigma\vdash e:\sigma' \qquad \Gamma\vdash\sigma:*}{\Gamma\vdash\lambda(x:\sigma)\cdot e:\sigma\to\sigma'}\;\textsc{Abs} \qquad \frac{\Gamma\vdash e:\sigma\to\sigma' \qquad \Gamma\vdash e':\sigma}{\Gamma\vdash e\,e':\sigma'}\;\textsc{App}$$

$$\frac{\Gamma,a:*\vdash e:\sigma}{\Gamma\vdash\Lambda(a:*)\cdot e:\Pi(a:*)\cdot\sigma}\;\textsc{TAbs} \qquad \frac{\Gamma\vdash e:\Pi(a:*)\cdot\sigma \qquad \Gamma\vdash\sigma':*}{\Gamma\vdash e\;\{\sigma'\}:\sigma[a:=\sigma']}\;\textsc{TApp}$$

Kinding rules

$$\frac{a:*\in\Gamma}{\Gamma\vdash a:*}\;\textsc{TVar} \qquad \frac{\Gamma\vdash\sigma:* \qquad \Gamma\vdash\sigma':*}{\Gamma\vdash\sigma\to\sigma':*}\;\textsc{Fun} \qquad \frac{\Gamma,a:*\vdash\sigma:*}{\Gamma\vdash\Pi(a:*)\cdot\sigma:*}\;\textsc{Univ}$$

Figure 2.7: System F

2.4.1 System F

System F[1] was introduced by Jean-Yves Girard in his 1972 dissertation (in French; the first English description of System F by Girard is Girard, 1986) and by Reynolds (1974). It is effectively system $\lambda 2$ from the Barendregt cube, which supports terms depending on terms $(*,*)$ and terms depending on types $(\Box,*)$, but no types depending on types $(\Box,\Box)$ or types depending on terms $(*,\Box)$. In System F, terms and types are distinguished syntactically rather than through the typing p. 33 relation (see also Section 2.3.4). This means that:

1. In the syntax of terms, we need different syntax for abstraction over terms ($\lambda x\cdot e$) and over types ($\Lambda a\cdot e$), and different syntax for term application ($e\;e'$) and type application ($e\;\{\sigma\}$)[2].

2. We must specialize the dependent function space to its degenerate case ($\sigma\longrightarrow\sigma'$) and the case for terms depending on types ($\Pi(a:*)\cdot\sigma$).[3]

3. We need a separate typing relation (relating terms to types) and kinding relation (relating types to kinds).

The full system is shown in Figure 2.7. All types in System F must have kind $*$; we nevertheless (perhaps somewhat pedantically) specify a kinding relation to make sure that all type variables that occur in terms and types are bound.

[1] ... called *F* by chance ... (Girard, 1986)

[2] To avoid confusion with the type of lists $[a]$, we use Girard's notation for type application rather than the more common $e\;[\sigma]$, which is due to Reynolds.

[3] Girard overloads the Λ operator for both type abstraction and the universal type, so he would denote $\Pi(a:*)\cdot\sigma$ as $\Lambda a\cdot\sigma$; Reynolds uses $\Delta a\cdot\sigma$ instead. Neither Girard nor Reynolds give the kind of a explicitly, as they do not consider types of kind other than $*$. We use Π to be consistent with the notation in Section 2.3, and give the kinds to make the fact that System F only supports types of kind $*$ explicit.

2.4.2 Algebraic data types

Languages such as Clean and Haskell support user-defined types through *algebraic data types*. The definition of an algebraic data type takes the form

```
data T a1 a2 ... ap
  = C1 T11 T12 ... T1m1
  | C2 T21 T22 ... T2m2
  ⋮
  | Cn Tn1 Tn2 ... Tnmn
```

This can (almost) be interpreted as an inductive data type

```
Inductive T (a1 : κ1) (a2 : κ2) ... (ap : κp) : ⋆ :=
  | C1 : T11 → T12 → ... → T1m1 → T a1 a2 ... ap
  | C2 : T21 → T22 → ... → T2m2 → T a1 a2 ... ap
  ⋮
  | Cn : Tn1 → Tn2 → ... → Tnmn → T a1 a2 ... ap
```

except that the positivity requirement is dropped. Note that the syntax precludes specializing the arguments in the codomain of the constructors: each constructor must return the exact same type. Moreover, since these languages are not dependently typed, algebraic data types can be parametrized by types only (and not by terms). They *can* however be parametrized by types of arbitrary kind. For example, here is a Haskell definition of generalized trees:

```
data GTree (f :: * -> *) (a :: *) = MkTree a (f (GTree f a))
```

The Haskell type of `MkTree` as reported by `ghc` is

```
∀ (a :: *) (f :: * → *). a → f (GTree f a) → GTree f a
```

This makes it clear that we cannot give a System F type to `MkTree`, because we need to quantify over types other than $*$ (in this case, a type of kind $* \to *$).

2.4.3 System F_A

Given the observation from the previous section, it may seem that system $\lambda\omega$ (also known as System F_ω) would be a better choice as a reference calculus. After all, it supports types of arbitrary kind. However, System F_ω is *too* powerful: the presence of arbitrary type level functions makes type *inference* (the process of recovering type information from a program written in an implicitly typed language) undecidable. We can therefore use System F_ω as a target language for translation, but we cannot hope to use it as a reference calculus for an implicitly typed language.

Hence, we need something in between System F and System F_ω, which is where System F_A comes in: System F extended with algebraic data types and pattern matching where type variables can be of higher kind and type constructor applications can be partial (Sulzmann et al., 2007). From the *History of Haskell* paper (Hudak et al., 2007):

> “Type inference for a system involving higher kinds seems at first to require higher-order unification, which is both much harder than traditional first-order unification and lacks most general unifiers [...]. However, by treating higher-kinded type constructors as uninterpreted functions and not allowing lambda [abstraction] at the type level, Jones's paper (Jones, 1993) shows that ordinary first-order unification suffices.

2.4.4 Fragments of System F

There are two important ways in which we can restrict the types in System F (or System $\mathrm{F_A}$): we can limit the *rank* of the types and we can make the system predicative. In System F proper, polymorphic functions are *first-class*: we can abstract over polymorphic functions (require them as arguments) and we can pass them as arguments to other polymorphic functions. As we shall see, in the finite-rank fragment of System F we lose the first ability; in the predicative fragment the second.

We define the set $R(n)$ of types of rank n (Kfoury and Tiuryn, 1992) as follows. $R(0)$, the types of rank 0, are the types without any occurrences of Π. $R(n+1)$ is the smallest set such that

$$R(n+1) \supseteq R(n) \cup \Big\{(\Pi(a:\kappa)\cdot\sigma | \sigma \in R(n+1)\Big\} \cup \Big\{\sigma \rightarrow \sigma' | \sigma \in R(n), \sigma' \in R(n+1)\Big\}$$

The rank-n fragment of System F is System F where the set of permissible types is $R(n)$. In particular, the rank-0 fragment is the simply typed lambda calculus ($\lambda_{\rightarrow}$ in the Barendregt cube).

The rank-1 fragment is not particularly more useful than the rank-0 fragment: although we can now abstract over types and define the polymorphic identity function we saw before

$$\texttt{id} = \lambda(a:*)\cdot\lambda(x:a)\cdot x \tag{2.6}$$

we cannot abstract over such terms

$$\lambda(id:\Pi(a:*)\cdot a \rightarrow a)\cdot\ldots \tag{2.7}$$

because that would require a rank-2 type. We will come back to this point when we introduce
p. 41 let-polymorphism in Section 2.5.1.

In the predicative fragment of System F, we can instantiate type variables only by types of rank 0: type variables cannot be instantiated by polymorphic functions. For example, consider the function that creates a singleton list:

$$\texttt{single} : \Pi(b:*)\cdot b \rightarrow [b]$$

If we pass (2.6) as an argument to `single`, we will get a list of type

$$\texttt{single}\ \{\Pi(a:*)\cdot a \rightarrow a\}\ \texttt{id} : [\Pi(a:*)\cdot a \rightarrow a]$$

That is, a list of polymorphic identity functions. In the predicative fragment of System F, we cannot apply `single` to `id` directly. The best we can do is

$$\lambda(a:*)\cdot\texttt{single}\ \{a \rightarrow a\}(\texttt{id}\ a) : \Pi(a:*)\cdot[a \rightarrow a]$$

p. 51 We will revisit this example when discussing HMF in Section 2.5.4.

Term and type language

e	$::=$	term	σ	$::=$	type
	x	variable		a	type variable
	$\lambda x \cdot e$	lambda abstraction		$\sigma \rightarrow \sigma$	function space
	$e\ e$	application		$\forall (a : *) \cdot \sigma$	type scheme

Typing rules

$$\frac{x : \sigma \in \Gamma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash x : \sigma}\ \textsc{Var}$$

$$\frac{\Gamma, x : \sigma \vdash e : \sigma' \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \lambda x \cdot e : \sigma \rightarrow \sigma'}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e : \sigma \rightarrow \sigma' \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash e\ e' : \sigma'}\ \textsc{App}$$

$$\frac{\Gamma, a : * \vdash e : \sigma}{\Gamma \vdash e : \forall (a : *) \cdot \sigma}\ \textsc{Gen} \qquad \frac{\Gamma \vdash e : \forall (a : *) \cdot \sigma \qquad \Gamma \vdash \sigma' : *}{\Gamma \vdash e : \sigma[a := \sigma']}\ \textsc{Inst}$$

Kinding rules

$$\frac{a : * \in \Gamma}{\Gamma \vdash a : *}\ \textsc{TVar} \qquad \frac{\Gamma \vdash \sigma : * \qquad \Gamma \vdash \sigma' : *}{\Gamma \vdash \sigma \rightarrow \sigma' : *}\ \textsc{Fun} \qquad \frac{\Gamma, a : * \vdash \sigma : *}{\Gamma \vdash \forall (a : *) \cdot \sigma : *}\ \textsc{Univ}$$

Figure 2.8: Type assignment system for System F (System F2)

2.5 Implicitly typed lambda calculus

Consider again the following example:

$$(\lambda (f : \Pi (a : *) \cdot a \rightarrow a) \cdot (f\ \mathbb{N}\ 1, f\ \mathbb{B}\ \texttt{True}))\ (\lambda (a : *) \cdot \lambda (x : a) \cdot x) \tag{2.8}$$

This term looks rather involved because of all the type annotations on the variables. Erasing the type annotations gives the following simpler term:

$$(\lambda f \cdot (f\ \mathbb{N}\ 1, f\ \mathbb{B}\ \texttt{True}))\ (\lambda a \cdot \lambda x \cdot x) \tag{2.9}$$

We can simplify this term further and erase all type information:

$$(\lambda f \cdot (f\ 1, f\ \texttt{True}))\ (\lambda x \cdot x) \tag{2.10}$$

For every system in the Barendregt cube we can define a *domain free* type system (Barthe and Sørensen, 2000) that erases type annotations in the style of (2.9) and a *type assignment* system (van Bakel et al., 1997) that erases all type information in the style of (2.10). A discussion of these systems in their full generality is beyond the scope of this dissertation; instead, we will concentrate on the type assignment system for System F, called F2 in (van Bakel et al., 1997)[1], shown in Figure 2.8.

When we compare System F (Figure 2.7) to System F2, we notice that the type annotation on p. 36 the variable bound by a lambda abstraction has disappeared. In addition, the term language no longer includes type abstraction and type application. The corresponding typing rules TAbs and

[1] Presumably, the name System F2 is based on System F_2, which is the name some authors use for System F, based on a hierarchy where System F_1 is the simply typed lambda calculus and going up to System F_ω in the limit (for example Pierce, 2002, Section 30.4, *Fragments of* F_ω). To add to the confusion, Girard himself presents a similar hierarchy (Girard, 1986, Appendix A.1), but starts with System F, which he terms System F_0.

`TApp` have been replaced by two new typing rules `Gen` and `Inst`. These new typing rules have the same purpose as `TAbs` and `TApp`, but can now be applied to *any* term (they are *not syntax directed*). We use $\forall$ instead of Π to signify this change in semantics.

The conceptual difference between $\Pi(a : *) \cdot a \to a$ and $\forall(a : *) \cdot a \to a$ is subtle. A function of the first type is a function that, given a type a, returns a function from $a \to a$ *for that* a. A function of the second type is a function from $a \to a$, *for any* a: it is a function that has an infinite family of types. For this reason, $\forall(a : *) \cdot a \to a$ is often referred to as a *type scheme*; type schemes more closely match our intuition that $\lambda x \cdot x$ can be interpreted to have any of a large
p. 31 range of types (Section 2.3.1).

Although both systems are logically equivalent, there is a catch. For a type system to be useful, we need a *type checking* algorithm that can verify whether a program is type correct. For an explicitly typed language, writing such an algorithm is straightforward; it is slightly harder when considering dependent types, but not significantly so (Löh et al., 2008). The corresponding algorithm for an implicitly typed language is called *type inference*, since the algorithm will need to infer which types were "implied" by the programmer. Unfortunately, type inference for System F2 is undecidable. As it turns out, even the potentially simpler problem of deciding whether a program in System F2 can be typed at all (without specifying exactly what the type of the program is) turns out to be equivalent to the type inference problem, and hence equally undecidable (Wells, 1999).

Part of the problem is that there are no *principal types* in System F2. In System F (indeed, in any explicitly typed type system) every term has exactly one type.[1] But in an implicitly typed language, terms can have more than one type. For example in System F2,[2]

$$\frac{\dfrac{}{x : \mathbb{N} \vdash x : \mathbb{N}}\,\text{\textsc{Var}} \qquad \dfrac{}{\vdash \mathbb{N} : *}}{\vdash \lambda x \cdot x : \mathbb{N} \to \mathbb{N}}\,\text{\textsc{Abs}}$$

or

$$\frac{\dfrac{}{x : \mathbb{B} \vdash x : \mathbb{B}}\,\text{\textsc{Var}} \qquad \dfrac{}{\vdash \mathbb{B} : *}}{\vdash \lambda x \cdot x : \mathbb{B} \to \mathbb{B}}\,\text{\textsc{Abs}}$$

or

$$\frac{\dfrac{\dfrac{}{a : *, x : a \vdash x : a}\,\text{\textsc{Var}} \qquad \dfrac{}{a : * \vdash a : *}\,\text{\textsc{TVar}}}{a : * \vdash \lambda x \cdot x : a \to a}\,\text{\textsc{Abs}}}{\vdash \lambda x \cdot x : \forall(a \cdot *) \cdot a \to a}\,\text{\textsc{Gen}}$$

However, of these three types for the identity function, the third is more general than the first two: $\forall(a : *) \cdot a \to a$ can be instantiated (using rule `Inst`) to $\mathbb{N} \to \mathbb{N}$ or $\mathbb{B} \to \mathbb{B}$. In fact, $\forall(a : *) \cdot a \to a$ is the *most general* or *principal*[3] type of the identity function.

Unfortunately, not all terms in System F2 have a principal type. For instance, the example from the start of this section

$$\lambda f \cdot (f\ 1, f\ \texttt{True}))$$

has type

$$(\forall(a : *) \cdot a \to a) \to (\mathbb{N}, \mathbb{B})$$

[1] Technically, in a dependent type system, a term can have more than one type, but they must all be β–equivalent.

[2] We leave out parts of the proof trees for readability.

[3] **principal**, *adj.*: Of a number of things or persons, or one of their number: belonging to the first rank; among the most important; prominent, leading, main (Oxford English Dictionary)

or

$$\forall(b : *) \cdot (\forall(a : *) \cdot a \to b) \to (b, b)$$

and there is no System F2 type that is more general than (can be instantiated to) both.

In summary, although System F2 appears to give us the expressive power of System F, it does so at too large a cost: type correctness for System F2 is undecidable. However, we do not want to use System F either: the abundance of type annotations obfuscates our programs. The goal therefore is to try and define a calculus which is sufficiently powerful (ideally, as powerful as System F) but that allows us to omit as many type annotations as possible.

The most traditional solution is to restrict types to rank 1 and impose predicativity, but introduce let-polymorphism which allows limited abstraction over polymorphic types (see also Section 2.4.4); such a system is commonly referred to as a Hindley/Milner style type system, and p. 38 will be introduced in Section 2.5.1. This has long been the standard type system in functional languages. No type annotations are necessary, but the expressive power of the system is limited.

More recent proposals are based on an attempt to combine System F with System F2, so that we have to write *some* type information but not all. The proposals differ in how much of the power of System F they recover, how much type information the user needs to write down, and how "obvious" it is when this type information is required.

The first well-known proposal in this category is the type system by Odersky and Läufer. We will discuss their type system briefly in Section 2.5.2, but we will not look at type inference. p. 45 Instead, Section 2.5.3 will discuss a type system proposed by Simon Peyton Jones *et al.* which p. 46 is based on the Odersky/Läufer type system, but simplifies it. Finally, Section 2.5.4 will discuss p. 51 HMF by Daan Leijen; while the type systems of Odersky/Läufer and Peyton Jones *et al.* support arbitrary rank types, they are still limited to the predicative fragment of System F; HMF truly supports first-class polymorphism.

There are various other proposals in this category, such as *boxy types* (Vytiniotis et al., 2006), Rigid MLF (Leijen, 2007a) and the very recent FPH (Vytiniotis et al., 2008). In addition, here are some proposals that go *beyond* System F types (or even System F_ω types) to attempt to solve the problem of principal types; MLF (Botlan and Rémy, 2003), HML (Leijen, 2008b) and the type system of Lushman (2007) fall into this category. A discussion of these systems is beyond the scope of this dissertation.

2.5.1 Let polymorphism

The most successful type system to date that supports polymorphism without requiring any type annotations is the one introduced by Milner for ML (Milner, 1978). It is often referred to as the Hindley/Milner type system; Hindley was a logician who defined a similar type system for combinatory logic ten years earlier (Hindley, 1969). Milner specified a type inference algorithm in his original paper (thus proving that type inference for his type system was decidable) but left open whether this algorithm produced principal types. A few years later, Damas and Milner (1982) published another paper in which they answered this question affirmatively. The latter paper is probably the more accessible and more well-known of the two, and the term Damas/Milner type system sometimes used as synonymous with Hindley/Milner (or let polymorphism).

A Hindley/Milner type system can be described as the predicative rank-1 fragment of System F, in which all types must be in *prenex form.* That is, a type of the form

Term and type language

e	$::=$		term	σ	$::=$		polytype
	x	variable		$\forall a : * \cdot \sigma$	universal quantification
	$\lambda x \cdot e$	lambda abstraction		τ	monotype
	$e\ e$	application			
	$\text{let } x = e \text{ in } e$	local definition	τ $::=$		monotype
				a	type variable
				$\tau \to \tau$	function space

Typing rules

$$\frac{x : \sigma \in \Gamma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash x : \sigma}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau \qquad \Gamma \vdash \tau' : *}{\Gamma \vdash \lambda x \cdot e : \tau' \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau}\ \textsc{App}$$

$$\frac{\Gamma, a : * \vdash e : \sigma}{\Gamma \vdash e : \forall (a : *) \cdot \sigma}\ \textsc{Gen} \qquad \frac{\Gamma \vdash e : \forall (a : *) \cdot \sigma \qquad \Gamma \vdash \tau : *}{\Gamma \vdash e : \sigma[a := \tau]}\ \textsc{Inst}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau}\ \textsc{Let}$$

Figure 2.9: The Hindley/Milner type system

$$\forall a \cdot a \to \forall b \cdot b \to b$$

must be rewritten as[1]

$$\forall ab \cdot a \to b \to b$$

Since all universal quantifiers must therefore be at the top-level, we can define rank-1 types syntactically rather than through an equation: we distinguish between *monotypes* which cannot contain any universal quantifiers, and *polytypes* which consist of a (possibly empty) list of universal quantifiers followed by a monotype. The type system is shown fully in Figure 2.9.

We have presented the type system in a slightly different style than in the original paper to clarify the relation to the type systems we presented so far. In particular, we are very explicit about kinds, even though only types of kind $*$ are permitted; as for System F, this formally ensures that all type variables are properly bound.

Since the domain of the function space must be a monotype, we cannot type

$$(\lambda f \cdot (f\ [1,2,3], f\ [\texttt{True,False}]))\ \texttt{reverse}$$

However, the distinguishing feature of the Hindley/Milner type system is the introduction of the `let` construct, which re-introduces a limited form of abstraction over terms with a universally quantified type. Thus, we *can* type

$$\begin{aligned} &\text{let } f = \texttt{reverse} \\ &\text{in } (f\ [1,2,3], f\ [\texttt{True,False}]) \end{aligned}$$

[1] We use $\forall ab \cdot \tau$ as short-hand for $\forall a \cdot \forall b \cdot \tau$.

since `let`-bound variables may be assigned a polytype (the typing rule for `let` is the only rule that introduces polytypes into the typing environment).

As mentioned in Section 2.4.4, the rank-1 fragment of System F is not much more useful the p. 38 rank-0 fragment since although we can *define* polymorphic functions, we cannot *abstract* over them. Intuitively, the Hindley/Milner type system can be seen as a type system that allows to introduce a series of polymorphic values one by one, where each definition can be used in the following definitions. It follows, however, that we have lost compositionality (Wells, 2002): the type of a `let` expression cannot be given in terms of the types of both subexpressions. When type-checking (`let` $x = e$ `in` e'), e must be type-checked before e'. This is a regrettable consequence, but to avoid it we need to introduce more powerful type systems such as intersection types (Wells, 2002, Section 3.3). A discussion of intersection types is however beyond the scope of this dissertation.

The type system as presented in Figure 2.9 is not *syntax directed*: although most rules have a p. 42 distinct syntactic form in their conclusion, rules `Inst` and `Gen` can be applied to any term at any point. From (Peyton Jones et al., 2007, Section 4.3, *The syntax-directed Damas-Milner system*),

> “ This flexibility makes it hard to turn the rules into a type-inference *algorithm*. For example, given a term, say $\lambda x \cdot x$, it is not clear which rules to use, in which order, to derive a judgement $\vdash \lambda x \cdot x : \sigma$ for some σ.
>
> If *all* the rules had a distinct syntactic form in their conclusions, the rules would be in so-called *syntax-directed form*, and that would, in turn, fully determine the shape of the derivation tree for any particular term t. This is a very desirable state of affairs, because it means that the steps of a type inference algorithm can be driven by the syntax of the term, rather than having to search for a valid typing derivation.
>
> (emphasis in original)

We noted before that the only rule that introduces a polytype into the typing environment is LET. Moreover, the premises to all the other rules are of the form $\Gamma \vdash e : \tau$ for a monotype τ: hence, the conclusion of a rule should be a monotype. These two observations lead to the syntax-directed presentation of the type system in Figure 2.10. The rule for variables now fully instantiates the p. 44 type scheme found in the environment, and the rule for `let` expressions includes an explicit generalization step.

To generalize τ, we find all free type variables in τ that do not occur in the domain of Γ and quantify over them (rule GEN). The reason for subtracting the type variables in the domain of Γ is that the rule for generalization in the original system (Figure 2.9) has an implicit side-condition p. 42 that $a \notin \Gamma$ (otherwise we would violate the invariant that every variable occurs at most once inside an environment).

This side condition is crucial. For example, consider this (invalid) derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a : *, f : a \to a, a : * \vdash f : a \to a}\,\textsc{Var}}{a : *, f : a \to a \vdash f : \forall(a : *) \cdot a \to a}\,\textsc{Gen}^{\ddagger}}{a : *, f : a \to a \vdash f : \mathbb{N} \to \mathbb{N}}\,\textsc{Inst} \qquad \dfrac{}{\ldots \vdash 0 : \mathbb{N}}}{a : *, f : a \to a \vdash f\,0 : \mathbb{N}}\,\textsc{App}}{a : * \vdash \lambda f \cdot f\,0 : (a \to a) \to \mathbb{N}}\,\textsc{Abs}}{\vdash \lambda f \cdot f\,0 : \forall(a : *) \cdot (a \to a) \to \mathbb{N}}\,\textsc{Gen}$$

Typing rules

$$\frac{x : \sigma \in \Gamma \qquad \vdash^{\text{inst}} \sigma \preceq \tau}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau' \qquad \Gamma \vdash \tau : *}{\Gamma \vdash \lambda x : e : \tau \to \tau'}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e : \tau \to \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \tau'}\ \textsc{App}$$

$$\frac{\Gamma \vdash^{\text{gen}} e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash \text{let}\ x = e\ \text{in}\ e' : \tau}\ \textsc{Let}$$

Generalization and instantiation

$$\frac{\Gamma \vdash e : \tau \qquad \overline{a} = \text{ftv}(\tau) - \text{ftv}(\Gamma)}{\Gamma \vdash^{\text{gen}} e : \forall \overline{a} \cdot \tau}\ \textsc{Gen} \qquad \frac{}{\vdash^{\text{inst}} \forall \overline{a} \cdot \tau \preceq \tau[\overline{a := \tau'}]}\ \textsc{Inst}$$

Figure 2.10: Syntax directed presentation of the Hindley/Milner type system

This typing derivation *must* be invalid: it claims to be able to apply *any* function f to a natural number and produce another natural number. Nevertheless, it is *almost* correct, except for a violation of said side condition. In the application of `Gen` marked (‡), we bind a twice in the environment, allowing us to pretend that f has the right type. We effectively assumed that the a bound by the universal quantifier is the same a that was already bound in the environment.

Many presentations of the Hindley/Milner type system do not explicitly record the kinds of type variables. The rule for generalization is then given as

$$\frac{\Gamma \vdash e : \sigma \qquad a \notin \text{fv}(\Gamma)}{\Gamma \vdash e : \forall (a : *) \cdot \sigma}\ \textsc{Gen}$$

If presented this way, the kinds of type variables are not mentioned in Γ (as we have done) but are left implicit. This rule is almost equivalent to the one we have presented (assuming the implicit side condition): it is slightly more permissive, since it allows to (implicitly) reintroduce the same type variable into the environment as long as it is not used anywhere.

The type inference algorithm presented by Damas and Milner is called algorithm $\mathcal{W}$ and is
p. 45 shown in Figure 2.11. It follows rather directly from the syntax-directed typing rules; it relies on
p. 24 the unification algorithm described in Section 2.2.3 to compare two types, and needs to be able to
generate "fresh" type variable. A fresh type variable is one that is not used so far; we will leave a
p. 161 more formal description of freshness until Section 7.2.1.

Algorithm $\mathcal{W}$ returns principal types, but not many proofs of the original algorithm exist in the literature. The original paper is often cited (Damas and Milner, 1982), but that merely states the lemma and refers to the "author's forthcoming Ph.D. thesis" for the proofs, which unfortunately is no longer easily accessible. There are however some recent papers about a formal (machine verified) proof of the correctness of $\mathcal{W}$ (Dubois and Ménissier-Morain, 1999, or Naraschewski and Nipkow, 1999, for instance). We do not reproduce a proof here, but do note that it relies on the unification algorithm to return most general unifiers.

There are also other type inference algorithms for the Hindley/Milner type system; algorithm $\mathcal{M}$ is another well-known one (Lee and Yi, 1998). These algorithms all produce the same result for type correct programs, but have different behaviour for type-incorrect programs; the reader is referred to (Heeren, 2005) for a more in-depth discussion of this issue.

$\mathcal{W}(\Gamma, e)$ returns a pair of a substitution and a type (S, τ) so that $S\Gamma \vdash e : \tau$.

```
$\mathcal{W}((\Gamma, x : \forall a_1 \cdots a_n \cdot \tau), x) =$
    $(\emptyset, \tau[a_1 := b_1, \ldots, a_n := b_n)$                    fresh $b_1, \ldots, b_n$

$\mathcal{W}(\Gamma, f\ e) =$
    let $(S_1, \tau_1) = \mathcal{W}(\Gamma, f)$
        $(S_2, \tau_2) = \mathcal{W}(S_1\Gamma, e)$
        $S_3 = \text{mgu}(S_2\tau_1 \overset{?}{\approx} \tau_2 \to b)$                    fresh $b$
    in $(S_3S_2S_1, S_3b)$

$\mathcal{W}(\Gamma, \lambda x \cdot e) =$
    let $(S, \tau) = \mathcal{W}((\Gamma, x : b), e)$                    fresh $b$
    in $(S, Sb \to \tau)$

$\mathcal{W}(\Gamma, \text{let } x = e_1 \text{ in } e_2) =$
    let $(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$
        $(S_2, \tau_2) = \mathcal{W}(S_1\Gamma, x : \text{generalize}_{(S_1\Gamma)}\tau_1, e_2)$
    in $(S_2S_1, \tau_2)$
```

Figure 2.11: Algorithm $\mathcal{W}$

2.5.2 Odersky/Läufer

The Odersky/Läufer system is shown in Figure 2.12. When compared to the Hindley/Milner system in Figure 2.9, we notice two important differences. First, there is a new syntactic construct, $\lambda(x :: \sigma) \cdot e$, with associated typing rule ANNABS. This construct allows to introduce functions of a higher rank type (abstract over polymorphic functions): rule ANNABS is the second rule after LET that introduces polytypes into the environment. This is the core idea behind this and another type systems, one that we mentioned before: without annotations, types are restricted to rank-1 (rule ABS still restricts the domain and codomain of the function space to be monotypes), but we relax this restriction if the user provides an annotation. Thus, we are combining the implicitly typed lambda calculus with the explicitly typed lambda calculus. p. 47 p. 42

The second difference is that rule INST is replaced by rule SUB. In the Hindley/Milner type system, if $e : \sigma$, we can use rule INST to deduce that $e : \tau$ for any instantiation τ of σ.[1] In the higher rank type system the situation is more complicated: if $e : \sigma$, we can use rule SUB to deduce that $e : \sigma'$ for any σ' that is "less polymorphic" than σ: formally, for any σ' such that $\sigma \preceq \sigma'$ (as a memory aid, consider that the set of expressions of type σ is contained in the set of expressions of type σ' or that σ is "less specific" than σ').

The $(\preceq)$ relation is called subsumption[2]. Its definition is shown in Figure 2.12; here we will consider a few examples only, due to Peyton Jones et al. (2007). The following examples are all valid in the Hindley/Milner type system as well as in the more general Odersky/Läufer type system: p. 47

[1] Technically, for any *generic* instantiation (Damas and Milner, 1982).

[2] **subsume**, *v.*: To bring (one idea, principle, term, etc.) *under* another, (a case, instance) under a rule, to take up *into*, or include *in*, something larger or higher (Oxford English Dictionary)

$$\begin{aligned}
\mathbb{N} &\preceq \mathbb{N} \\
\mathbb{N} \to \mathbb{B} &\preceq \mathbb{N} \to \mathbb{B} \\
\forall a \cdot a \to a &\preceq \mathbb{N} \to \mathbb{N} \qquad (2.11) \\
\forall a \cdot a \to a &\preceq \forall b \cdot [b] \to [b] \\
\forall a \cdot a \to a &\preceq \forall bc \cdot (b,c) \to (b,c) \\
\forall ab \cdot (a,b) \to (b,a) &\preceq \forall c \cdot (c,c) \to (c,c) \qquad (2.12)
\end{aligned}$$

Valid in the Odersky/Läufer system but not in Hindley/Milner (due to higher rank types):

$$\begin{aligned}
\mathbb{B} \to (\forall a \cdot a \to a) &\preceq \mathbb{B} \to \mathbb{N} \to \mathbb{N} \\
(\mathbb{N} \to \mathbb{N}) \to \mathbb{B} &\preceq (\forall a \cdot a \to a) \to \mathbb{B} \qquad (2.13) \\
(\forall b \cdot [b] \to [b]) \to \mathbb{B} &\preceq (\forall a \cdot a \to a) \to \mathbb{B}
\end{aligned}$$

These examples are clarified when we regard these types as System F types, reading Π for $\forall$ p. 36 (System F was defined in Figure 2.7). We can then give *evidence* of $\sigma \preceq \sigma'$ by providing a System F term that produces an element of type σ' given an element e of type σ. For example, evidence of (2.11) is simple:

$$\lambda(e : \Pi(a : *) \cdot a \to a) \cdot e\ \{\mathbb{N}\}$$

Similarly, evidence of (2.12) is

$$\lambda(e : \Pi(a : *) \cdot \Pi(b : *) \cdot (a,b) \to (a,b)) \cdot \Lambda(c : *) \cdot e\ \{c\}\ \{c\}$$

Finally, evidence of (2.13) is

$$\lambda(e : (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}) \cdot \lambda(f : \Pi(a : *) \cdot a \to a) \cdot e\ (f\ \{\mathbb{N}\})$$

Note the contravariance in this last example: the subsumption relation is reversed for the domain of the function space (rule FUN).

Obviously, the tricky part is to come up with a syntax directed presentation of the type system which is suitable for type inference. However, rather than considering Odersky and Läufer's solution to this problem, we will move straight to the solution proposed by Simon Peyton Jones *et al.* in the next section.

2.5.3 Practical type inference for arbitrary rank types

Peyton Jones et al. (2007) give two syntax directed presentations of the Odersky/Läufer system. p. 47 The first, shown in Figure 2.13, is a simplified version of the system given in the original paper (Odersky and Läufer, 1996).

p. 44 Compared to the syntax directed Hindley/Milner system (Figure 2.10), we see two differences. First, there is the new rule for annotated lambda abstractions ANNABS that we already discussed in Section 2.5.2. Second, the rule for application $(e\ e')$ is adapted. Since e may require e' to have a polymorphic type, we must generalize the type of e', and then do a subsumption check to make sure that the type of e' is at least as polymorphic as the type expected by e.

Term and type language

$e ::=$		term	$\sigma ::=$		polytype
	x	variable		$\forall a \cdot \sigma$	universal quantification
	$\lambda x \cdot e$	lambda abstraction		$\sigma \rightarrow \sigma$	higher rank type
	$\lambda(x :: \sigma) \cdot e$	annotated lambda		τ	monotype
	$e\ e$	application			
	$\text{let } x = e \text{ in } e$	local definition	$\tau ::=$		monotype
				a	type variable
				$\tau \rightarrow \tau$	function space

Typing rules

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x \cdot e : \tau \rightarrow \sigma}\ \textsc{Abs} \qquad \frac{\Gamma, x : \sigma \vdash e : \sigma'}{\Gamma \vdash \lambda(x :: \sigma) \cdot e : \sigma \rightarrow \sigma'}\ \textsc{AnnAbs}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \sigma' \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash e\ e' : \sigma'}\ \textsc{App} \qquad \frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \sigma'}{\Gamma \vdash \text{let } x = e \text{ in } e' : \sigma'}\ \textsc{Let}$$

$$\frac{\Gamma \vdash e : \sigma \qquad a \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall a \cdot \sigma}\ \textsc{Gen} \qquad \frac{\Gamma \vdash e : \sigma \qquad \vdash^{\text{subs}} \sigma \preceq \sigma'}{\Gamma \vdash e : \sigma'}\ \textsc{Sub}$$

Subsumption

$$\frac{\vdash^{\text{subs}} \sigma \preceq \sigma' \qquad a \notin \text{ftv}(\sigma)}{\vdash^{\text{subs}} \sigma \preceq \forall a \cdot \sigma'}\ \textsc{Skol} \qquad \frac{\vdash^{\text{subs}} \sigma[a := \tau] \preceq \sigma'}{\vdash^{\text{subs}} \forall a \cdot \sigma \preceq \sigma}\ \textsc{Spec}$$

$$\frac{\vdash^{\text{subs}} \sigma_1' \preceq \sigma_1 \qquad \vdash^{\text{subs}} \sigma_2 \preceq \sigma_2'}{\vdash^{\text{subs}} \sigma_1 \rightarrow \sigma_2 \preceq \sigma_1' \rightarrow \sigma_2'}\ \textsc{Fun} \qquad \frac{}{\vdash^{\text{subs}} \tau \preceq \tau}\ \textsc{Mono}$$

Figure 2.12: Odersky/Läufer type system

Type language

$\sigma ::=$		qualified type	$\rho ::=$		type without top-level quantifiers
	$\forall a \cdot \sigma$	universal quantification		τ	monotype (τ defined as before)
	ρ	type without top-level quantifiers		$\sigma \rightarrow \rho$	higher rank type

Typing rules

$$\frac{x : \sigma \in \Gamma \qquad \vdash^{\text{inst}} \sigma \preceq \rho}{\Gamma \vdash x : \rho}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau \vdash e : \rho}{\Gamma \vdash \lambda x \cdot e : \tau \rightarrow \rho}\ \textsc{Abs} \qquad \frac{\Gamma, x : \sigma \vdash e : \rho}{\Gamma \vdash \lambda(x :: \sigma) \cdot e : \sigma \rightarrow \rho}\ \textsc{AnnAbs}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \rho \qquad \Gamma \vdash^{\text{gen}} e' : \sigma' \qquad \vdash^{\text{subs}} \sigma' \preceq \sigma}{\Gamma \vdash e\ e' : \rho}\ \textsc{App}$$

$$\frac{\Gamma \vdash^{\text{gen}} e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \rho}{\Gamma \vdash \text{let } x = e \text{ in } e' : \rho}\ \textsc{Let}$$

Generalization and instantiation

$$\frac{\Gamma \vdash e : \rho \qquad \overline{a} = \text{ftv}(\rho) - \text{ftv}(\Gamma)}{\Gamma \vdash^{\text{gen}} e : \forall \overline{a} \cdot \rho}\ \textsc{Gen} \qquad \frac{}{\vdash^{\text{inst}} \forall \overline{a} \cdot \rho \preceq \rho[\overline{a := \tau}]}\ \textsc{Inst}$$

Figure 2.13: Syntax directed presentation of the Odersky/Läufer type system

Term and type language

e	$::=$	term
	x	variable
	$\lambda x \cdot e$	lambda abstraction
	$e\,e'$	application
	$(e :: \sigma)$	annotated expression
	let $x = e$ in e	local definition

Typing rules

$$\frac{x : \sigma \in \Gamma \qquad \vdash^{\text{inst}\sigma}_{\delta} \sigma \preceq \rho}{\Gamma \vdash_{\delta} x : \rho}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau \vdash_{\Uparrow} e : \rho}{\Gamma \vdash_{\Uparrow} \lambda x \cdot e : \tau \to \rho}\ \textsc{Abs}_{\Uparrow} \qquad \frac{\Gamma, x : \sigma \vdash_{\Downarrow} e : \rho}{\Gamma \vdash_{\Downarrow} \lambda x \cdot e : \sigma \to \rho}\ \textsc{Abs}_{\Downarrow}$$

$$\frac{\Gamma \vdash_{\Uparrow} e : \sigma \to \rho' \qquad \Gamma \vdash^{\text{gen}}_{\Downarrow} e' : \sigma \qquad \vdash^{\text{inst}\sigma}_{\delta} \rho' \preceq \rho}{\Gamma_{\delta} \vdash e\,e' : \rho}\ \textsc{App}$$

$$\frac{\Gamma \vdash^{\text{gen}}_{\Downarrow} : \sigma \qquad \vdash^{\text{inst}\sigma}_{\delta} \sigma \preceq \rho}{\Gamma \vdash_{\delta} (e :: \sigma) : \rho}\ \textsc{Annot} \qquad \frac{\Gamma \vdash^{\text{gen}}_{\Uparrow} e : \sigma \qquad \Gamma, x : \sigma \vdash_{\delta} e' : \rho}{\Gamma \vdash_{\delta} \text{let } x = e \text{ in } e' : \rho}\ \textsc{Let}$$

Generalization and instantiation

$$\frac{\Gamma \vdash_{\delta} e : \rho \qquad \overline{a} = \text{ftv}(\rho) - \text{ftv}(\Gamma)}{\Gamma \vdash^{\text{gen}}_{\delta} e : \forall \overline{a} \cdot \rho}\ \textsc{Gen} \qquad \frac{\vdash^{\text{inst}\rho}_{\delta} \rho[\overline{a := \tau}] \preceq \rho'}{\vdash^{\text{inst}\sigma}_{\delta} \forall \overline{a} \cdot \rho \preceq \rho'}\ \textsc{Inst}^{\sigma}$$

$$\frac{}{\vdash^{\text{inst}\rho}_{\Uparrow} \rho \preceq \rho}\ \textsc{Inst}\rho_{\Uparrow} \qquad \frac{\vdash^{\text{subs}} \rho \preceq \rho'}{\vdash^{\text{inst}\rho}_{\Downarrow} \rho \preceq \rho'}\ \textsc{Inst}\rho_{\Downarrow}$$

Figure 2.14: PTI

There is one technical detail that makes it possible to keep the presentation so close to the Hindley/Milner type system and the original Odersky/Läufer system. Although we allow polytypes in the *domain* of the function space (thus introducing higher rank types), we restrict the
p. 41 *codomain* to monotypes: we require types to be in prenex form (Section 2.5.1). This was used in older versions of *Practical type inference for arbitrary rank types*, but not in the version that was eventually published. The system without the restriction is slightly more expressive but also more complicated; we prefer the system with the prenex types and will not discuss the alternative. See (Peyton Jones et al., 2007, Section 4.6.2) of the published paper for some details.

Although the syntax directed version given by Simon Peyton Jones *et al.* is slightly simpler than the syntax directed system given by Odersky and Läufer, the real contribution of (Peyton Jones et al., 2007) is a *bidirectional* version of the same type system, shown in Figure 2.14. We will refer to this type system as PTI for *Practical Type Inference* (our term).

The main difference between PTI and the first syntax directed Odersky/Läufer system is that type information is propagated into terms so that we can write $(\lambda x \cdot e) :: (\sigma \to \rho)$ instead of $\lambda(x :: \sigma) \cdot e$. This is common in functional languages where we might write

```
f :: σ → ρ
f x = e
```

rather than

```
f (x :: σ) = e
```

The syntax of terms is modified to allow a type annotation on arbitrary expressions ($e :: \sigma$). For conciseness, we have left out the rules for annotated lambdas (any term $\lambda(x : \sigma) \cdot e$ can be replaced by $(\lambda x \cdot e) :: \sigma \rightarrow \rho$ for a suitable ρ); see (Peyton Jones et al., 2007, Figure 7).

The typing rules now include a directionality; we combine type inference ($\Gamma \vdash_{\Uparrow} e : \rho$) with type checking ($\Gamma \vdash_{\Downarrow} e : \rho$). The difference is most evident in the rule for abstraction: when inferring a type for a lambda abstraction, the domain of the function space must be a monotype (we cannot infer higher rank types). However, when checking that an abstraction has a type $\sigma \rightarrow \rho$, we allow for higher rank types. Many of the other rules are polymorphic in their directionality; this is indicated by the use of a variable ($\Gamma \vdash_{\delta} e : \rho$).

Perhaps surprisingly, type inference for PTI is not much more difficult than type inference for the basic Hindley/Milner type system. For a detailed discussion we refer to Peyton Jones et al. (2007), which shows how to modify a type inference algorithm for Hindley/Milner to support arbitrary rank types. Here we want to highlight only one aspect of this algorithm: the use of skolemization.

The rule for application in the original Hindley/Milner type system is

$$\frac{\Gamma \vdash e : \tau' \rightarrow \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau}\ \textsc{App}$$

In the inference algorithm, we can infer a type for e and infer a type for e', and then call unify to make sure that the type in the function domain matches the type of the argument. In the higher rank system, however, the rule for application becomes

$$\frac{\Gamma \vdash e : \sigma \rightarrow \rho \qquad \Gamma \vdash^{\mathrm{gen}} e' : \sigma' \qquad \vdash^{\mathrm{subs}} \sigma' \preceq \sigma}{\Gamma \vdash e\, e' : \rho}\ \textsc{App}$$

(We are referring to the syntax directed system in Figure 2.13 rather than the bidirectional system p. 47
in Figure 2.14, because it makes the issues slightly clearer. The same problems arise in the p. 48
bidirectional version.) In the inference algorithm, we can no longer use unification to make sure that the argument type matches the expected type; instead, we need to do a *subsumption check* to make sure that the argument is polymorphic enough:

$$\sigma' \preceq \sigma$$

In other words, we need to implement the $\vdash^{\mathrm{subs}}$ relation (Figure 2.12). How can we check whether p. 47
one type scheme is an instance of another? For example, consider

$$\forall ab \cdot a \rightarrow b \rightarrow b \overset{?}{\preceq} \forall c \cdot c \rightarrow c \rightarrow c \tag{2.14}$$

As a first attempt, we might try to instantiate a, b, c with fresh meta-variables a, b, c and then unify $a \rightarrow b \rightarrow b \overset{?}{\approx} c \rightarrow c \rightarrow c$. This unification equation has a solution $[a := c, b := c]$ so that we conclude that the subsumption check is successful. However, we would end up with the same unification equation, the same solution and the same conclusion for

$$\forall c \cdot c \rightarrow c \rightarrow c \overset{?}{\preceq} \forall ab \cdot a \rightarrow b \rightarrow b \tag{2.15}$$

However, (2.14) is true whereas (2.15) is not. Hence, we will need to be more sophisticated. Instead of instantiating the quantified variables by meta-variables, we instantiate the quantified variables of the right-hand type scheme by *skolem constants*. A skolem constant denotes a constant but unknown type. In particular, two different skolem constants $\mathfrak{a}$ and $\mathfrak{b}$ can never unify with each other (as far as unification is concerned, skolem constants are constants and not variables). So, to check (2.14), we will try to unify

$$a \to b \to b \overset{?}{\approx} \mathfrak{c} \to \mathfrak{c} \to \mathfrak{c}$$

which has a solution $[a := \mathfrak{c}, b := \mathfrak{c}]$, but when checking (2.15), we will try to unify

$$\mathfrak{a} \to \mathfrak{b} \to \mathfrak{b} \overset{?}{\approx} c \to c \to c$$

which will fail because $\mathfrak{a}$ and $\mathfrak{b}$ cannot be unified. For nested type schemes we apply the same technique, making sure to take the contravariant behaviour of the function space into account.

The original Odersky and Läufer paper refers to this technique as "unification under a mixed prefix" and cites (Miller, 1992). The "mixed prefix" refers to the fact that some variables are universally quantified (the skolem constants) while others are existentially quantified (the meta-variables). Note however that the problem discussed by Miller is much more general than necessary in this context (indeed, is undecidable in general).

Term and type language

$e ::=$	term	$\sigma ::=$	qualified type
$\quad x$	variable	$\quad \forall a \cdot \sigma$	universal quantification
$\quad \lambda x \cdot e$	lambda abstraction	$\quad \rho$	type without top-level quantifiers
$\quad \lambda(x : \sigma) \cdot e$	annotated lambda		
$\quad e\ e$	application	$\rho ::=$	type without top-level quantifiers
$\quad \text{let } x = e \text{ in } e$	local definition	$\quad a$	type variable
		$\quad c\ \sigma_1 \ldots \sigma_n$	type application
		$\tau ::=$	monotype
		$\quad a$	type variable
		$\quad c\ \tau_1 \ldots \tau_n$	type application

Typing rules

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau \vdash e : \rho}{\Gamma \vdash \lambda x \cdot e : \tau \to \rho}\ \textsc{Abs} \qquad \frac{\Gamma, x : \sigma \vdash e : \rho}{\Gamma \vdash \lambda(x : \sigma) \cdot e : \sigma \to \rho}\ \textsc{AnnAbs}$$

$$\frac{\Gamma \vdash e : \sigma \to \sigma' \qquad \Gamma \vdash e' : \sigma \qquad \text{minimal}(\sigma \to \sigma')}{\Gamma \vdash e\ e' : \sigma'}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \sigma' \qquad \text{most general}(\sigma)}{\Gamma \vdash \text{let } x = e \text{ in } e' : \sigma'}\ \textsc{Let}$$

$$\frac{\Gamma \vdash e : \sigma \qquad a \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall a \cdot \sigma}\ \textsc{Gen} \qquad \frac{\Gamma \vdash e : \forall(a : *) \cdot \sigma}{\Gamma \vdash e : \sigma[a := \sigma']}\ \textsc{Inst}$$

Figure 2.15: HMF

2.5.4 HMF

The typing rules for HMF (Leijen, 2008a) are shown in Figure 2.15 (we discuss only the "Plain HMF" system without extensions). They are almost identical to the Odersky and Läufer typing rules (Figure 2.12), with two important exceptions.

First, HMF is invariant. Instead of the general subsumption relation (rule SUB in Figure 2.12) we can only generalize and instantiate the outermost variables of a type scheme (rules GEN and INST).[1] When comparing two types, any nested type schemes must be identical (up to alpha-renaming): there is no notion of "at least as polymorphic as". In particular, this means that we do not have to worry about the contravariant/covariant behaviour of the function space constructor (as all type constructors are invariant in their arguments).

The second exception is the important one. Rule INST allows to instantiate type variables with type schemes: HMF is impredicative. This is a major departure from PTI and the Odersky and Läufer type system. It is also the reason for the two side conditions added to the rules for APP and LET. Consider

```
id :: ∀ a. a → a
id x = x

single :: ∀ a. a → [a]
single x = [x]
```

Ignoring the side conditions, we can derive two different types for `single id`:

$$\dfrac{\dfrac{\dfrac{\vdash \texttt{single} : \forall a \cdot a \to [a]}{\vdash \texttt{single} : (b \to b) \to [b \to b]}\,\textsc{Inst} \qquad \dfrac{\vdash \texttt{id} : \forall a \cdot a \to a}{\vdash \texttt{id} : b \to b}\,\textsc{Inst}}{\vdash \texttt{single id} : [b \to b]}\,\textsc{App}}{\vdash \texttt{single id} : \forall b \cdot [b \to b]}\,\textsc{Gen}$$

or

$$\dfrac{\dfrac{\vdash \texttt{single} : \forall a \cdot a \to [a]}{\vdash \texttt{single} : (\forall b \cdot b \to b) \to [\forall b \cdot b \to b]}\,\textsc{Inst} \qquad \vdash \texttt{id} : \forall b \cdot b \to b}{\vdash \texttt{single id} : [\forall b \cdot b \to b]}\,\textsc{App}$$

The second derivation requires impredicativity (we are instantiating a with $\forall b \cdot b \to b$) and is therefore not permissible in the Odersky and Läufer type system or in PTI. In System F both derivations of course are allowed, but since System F is explicitly typed this causes no difficulties: the programmer will always make clear which of the two types was intended.

In an implicitly typed language, we have two options. We can make the language of types more expressive so that there is a type which is more general than both. This is essentially the approach taken by MLF (Botlan and Rémy, 2003), but at quite a large cost: type inference for MLF is rather involved.

The alternative is to build a "bias" into the type system, and this is exactly what the side conditions in the HMF typing rules ensure (for a formal definition of these side conditions, see the paper). In the absence of a type annotation, HMF will always prefer predicative instantiation (the first derivation) over impredicative instantiation (the second). The second derivation is only available by using an explicit type annotation.

[1] The HMF paper presents the type system using the System F instance relation, which is a combination of the GEN and INST rules. Our presentation is equivalent but more in line with the other systems presented in this chapter.

For the implementation of type inference, the side conditions do not explicitly need to be verified: type inference for HMF is not difficult exactly because the built-in bias in the typing rules facilitates type inference. We will not reproduce the type inference algorithm here, and refer
p. 41 the reader to the HMF paper instead. The algorithm is very similar to algorithm $\mathcal{W}$ (Section 2.5.1) with a modified unification algorithm which relies on the skolemization technique described in
p. 46 Section 2.5.3.

2.6 Substructural logics

Consider the following three derivations in the simply typed lambda calculus. The first demonstrates functions do not need to use their arguments in the order they are passed.

$$\dfrac{\dfrac{\dfrac{\dfrac{f : a \to b \in x : a, f : a \to b}{x : a, f : a \to b \vdash f : a \to b}\textsc{Var} \qquad \dfrac{x : a \in x : a, f : a \to b}{x : a, f : a \to b \vdash x : a}\textsc{Var}}{x : a, f : a \to b \vdash f x : b}\textsc{App}}{x : a \vdash \lambda f \cdot f x : b}\textsc{Abs}}{\vdash \lambda x \cdot \lambda f \cdot f x : b}\textsc{Abs} \tag{2.16}$$

The second demonstrates that we can use variables more than once.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : a \to a \to b \in f : a \to a \to b}{f : a \to a \to b, x : a \vdash f : a \to a \to b}\textsc{Var} \qquad \dfrac{x : a \in f : \ldots, x : a}{f : \ldots, x : a \vdash x : a}\textsc{Var}}{f : a \to a \to b, x : a \vdash f x : b}\textsc{App} \qquad \dfrac{x : a \in f : \ldots, x : a}{f : \ldots, x : a \vdash x : a}\textsc{Var}}{f : a \to a \to b, x : a \vdash f x x : b}\textsc{App}^{\ddagger}}{f : a \to a \to b \vdash \lambda x \cdot f x x : b}\textsc{Abs}}{\vdash \lambda f \cdot \lambda x \cdot f x x : b}\textsc{Abs} \tag{2.17}$$

The final derivation shows that not all variables have to be used.

$$\dfrac{\dfrac{\dfrac{x : a \in x : a, y : b}{x : a, y : b \vdash x : a}\textsc{Var}}{x : a \vdash \lambda y \cdot x : b \to a}\textsc{Abs}}{\vdash \lambda x \cdot \lambda y \cdot x : a \to b \to a}\textsc{Abs} \tag{2.18}$$

Logical rules

$$\dfrac{}{x : \tau \vdash x : \tau}\textsc{Var}$$

$$\dfrac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x \cdot e : \tau \to \tau'}\textsc{Abs} \qquad \dfrac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Delta \vdash e_2 : \tau}{\Gamma, \Delta \vdash e_1\, e_2 : \tau'}\textsc{App}$$

Structural rules

$$\dfrac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau}\textsc{Weak} \qquad \dfrac{\Gamma, y : \tau', z : \tau' \vdash e : \tau}{\Gamma, x : \tau' \vdash e[y := x, z := x] : \tau}\textsc{Contr} \qquad \dfrac{\Gamma, \Delta, \Theta \vdash e : \tau}{\Gamma, \Theta, \Delta \vdash e : \tau}\textsc{Exch}$$

Figure 2.16: Structural presentation of the simply typed lambda calculus

Such environment manipulations (using assumptions out of order, reusing some assumptions and ignoring others) are implicitly allowed in the type systems we have seen thus far. It is however possible to give a *structural* presentation of a type system in which these operations are made explicit. Figure 2.16 gives such a presentation for the simply typed lambda calculus.

The differences between a structural presentation and a non-structural presentation are subtle. The rule for application (APP) now uses a typing environment Γ to type the function, another environment Δ to type the argument, and uses the combined environment (Γ, Δ) in the conclusion of the rule. This has two consequences.

Recall from Section 2.1.2 that we regard the typing environment as a list of assumptions, and that every variable must occur at most once inside an environment. Suppose we need an assumption $x : a$ both to type the function and to type the argument, as in the rule marked ‡ in derivation (2.17). That means that we must have $x : a \in \Gamma$ and $x : a \in \Delta$. But then x will appear twice in (Γ, Δ), which violates the invariant. Thus, using different environments in the branches of a derivation has the non-obvious consequence that we can no longer use variables more than once. Unless, that is, we apply rule CONTR first, which explicitly duplicates an assumption in the environment, renaming variables to make sure no variable occurs more than once. p. 20

The second consequence is that we can no longer use assumptions in the environment out of order, as list concatenation is not commutative. If the environment is $x : a, f : a \rightarrow b$ as in derivation (2.16), we are forced to use $x : a$ for the first premise and $f : a \rightarrow b$ for the second—unless we apply rule EXCH first to explicitly reorder the assumptions.[1]

The other modification to the typing rules is the rule for variables. Rather than simply requiring that $x : \sigma$ must occur somewhere in the environment Γ, the rule now states that $x : \sigma$ can be typed only in the *singleton* environment $x : \sigma$. This means that every variable must be used; derivation (2.18) is no longer valid. Unless, that is, we apply rule WEAK first, which explicitly allows us to throw away assumptions in the environment.

We have thus also introduced three new rules: exchange (EXCH), weakening (WEAK) and contraction (CONTR). Collectively, these are known as the *structural* rules. In the type systems we saw before this section, these rules are admissible (can be proven from the other rules). The real advantage of making the rules explicit, of course, is that we can restrict the use of these rules to exert control over environment manipulation.

For example, we can remove rule WEAK to obtain *relevant logic* (all assumptions must be used), rule CONTR to obtain *affine logic* (assumptions can be used at most once), rules WEAK and CONTR to obtain *linear logic* (all assumptions must be used exactly once) or rule EXCH to obtain *ordered logic* (assumptions must be used in order). Logics or type systems in which the use of the structural rules is restricted are known as *substructural* logics or type systems[2].

Looking ahead, uniqueness typing can (as a first approximation) be described as a logic in which contraction can only be applied to variables of a non-unique type (that is, only variables of a non-unique type can be used more than once). We will therefore not be interested in limiting weakening, and we do not care about the ordering of the assumptions in the environment. Section 7.2.2 in the chapter on formalization discusses how this can be formalized elegantly. p. 163

[1] This rule is often presented as

$$\frac{\Gamma, \Delta \vdash e : \tau}{\Delta, \Gamma \vdash e : \tau}\ \textsc{Exch}'$$

However, EXCH′ is not powerful enough. In particular, EXCH cannot be proven from EXCH′.

[2] Restall (2000) accredits the name to Schröder-Heister and Došen.

p. 77

We will not further investigate any substructural logics in this chapter, and we will delay a description of substructural *type systems* to the next chapter on related work (Chapter 3). For a more in-depth discussion of substructural logics we refer the reader to (Restall, 2000). For linear logic specifically, refer to the excellent tutorial introduction by Wadler (1993), or the textbook by Troelstra (1992). Note, however, that the textbooks by Restall and Troelstra go well beyond what we will need in this dissertation.

2.7 Operational semantics and soundness

p. 11

In the introduction (Chapter 1) we mentioned that "executing" a program in a functional language means evaluating or simplifying it, and that a type system can be used to make sure that evaluation will not get stuck. In this section we will see how to make these notions precise.

The fundamental evaluation step in the lambda calculus is known as beta-reduction, and is defined as

$$(\lambda x \cdot e)\ e' \rightarrow e[x := e'] \qquad (\beta\text{−reduction})$$

In words, we can reduce the application of a function to an argument to the function body, substituting the argument for the formal parameter of the function. This rule however only lets us simplify an expression of the form $((\lambda x \cdot e)\ e')$. For example, we cannot apply beta-reduction to

$$\Big((\lambda x \cdot x)\ (\lambda y \cdot y\ y)\Big)\ \Big((\lambda x \cdot x)\ (\lambda z \cdot z)\Big) \tag{2.19}$$

since the term does not have the right shape. Instead, we need a rule that allows us to "look inside" this term and apply beta-reduction to a subterm first. The most general such rule is

$$\frac{e \rightarrow e'}{C[e] \rightarrow C[e']}\ \textsc{Eval}$$

The C in this rule denotes a *context*, which is a term with a "hole". Formally:

$$C ::= [] | x | CC | \lambda x \cdot C$$

p. 167

This mirrors the syntax for lambda terms, with the addition of a "hole". If C is a context and T is a term, then $C[T]$ denotes the lambda term obtained from plugging T into the hole in C (section 7.3.3 will give a more formal description). Rule EVAL is sometimes (e.g., Girard, 1986, Definition 3.3) referred to as the *compatible closure* of the beta-reduction relation (with respect to contexts C).

p. 55

As we shall see, however, the order in which we apply beta-reduction to the subterms matters and so programming languages often specify an *evaluation strategy* which defines which subterms can be reduced and in which order that should happen. This is formalized by restricting the locations of the holes in the context, and (in some cases) by restricting the class of terms to which we can apply beta-reduction. In the next three sections, we will consider three such strategies: call-by-value, call-by-name and call-by-need. Their formal definitions are shown in Figure 2.17.

General reduction

$$\frac{}{(\lambda x \cdot e)\ e' \longrightarrow e[x := e']}\ \textsc{Beta} \qquad \frac{e \longrightarrow e'}{C[e] \longrightarrow C[e']}\ \textsc{Eval}$$

Call-by-Value

$$\frac{}{(\lambda x \cdot e)\ V \longrightarrow e[x := V]}\ \textsc{Beta} \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\ \textsc{Eval}$$

where

V ::=		value	E ::=		evaluation context
	x	variable		$[]$	
	$\lambda x \cdot e$	abstraction		$E\ e$	
				$(\lambda x \cdot e)\ E$	

Call-by-Name

$$\frac{}{(\lambda x \cdot e)\ e' \longrightarrow e[x := e']}\ \textsc{Beta} \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\ \textsc{Eval}$$

where

V ::=		value	E ::=		evaluation context
	x	variable		$[]$	
	$\lambda x \cdot e$	abstraction		$E\ e$	

Call-by-Need

$$\frac{}{\text{let } x = \lambda y \cdot e \text{ in } E[x] \longrightarrow (E[x])[x := \lambda y \cdot e]}\ \textsc{Value}$$

$$\frac{}{(\text{let } x = e \text{ in } V)\ e' \longrightarrow \text{let } x = e \text{ in } V\ e'}\ \textsc{Commute}$$

$$\frac{}{\text{let } y = (\text{let } x = e \text{ in } V) \text{ in } E[y] \longrightarrow \text{let } x = e \text{ in let } y = V \text{ in } E[y]}\ \textsc{Assoc}$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\ \textsc{Eval}$$

where let $x = e'$ in e is syntactic sugar for $(\lambda x \cdot e)\ e'$ and

V ::=		value	E ::=		evaluation context
	$\lambda x \cdot e$	abstraction		$[]$	
	let $x = e$ in V	sharing		$E\ e$	
				let $x = e$ in E	
				let $x = E$ in $E'[x]$	

Figure 2.17: Operational semantics

2.7.1 Call-by-value

Call-by-value evaluation is the evaluation strategy implemented in most languages, where the argument to a function must be evaluated to a value before function call is made (variables and lambda abstractions are considered values). Evaluation stops when the outermost term is a value. The call-by-value evaluation of (2.19) is

$$\begin{array}{ll}
& \underline{\big((\lambda x \cdot x)\ (\lambda y \cdot y\, y)\big)}\ \big((\lambda x \cdot x)\ (\lambda z \cdot z)\big) \\
\to & (\lambda y \cdot y\, y)\ \underline{\big((\lambda x \cdot x)\ (\lambda z \cdot z)\big)} \\
\to & \underline{(\lambda y \cdot y\, y)\ (\lambda z \cdot z)} \\
\to & \underline{(\lambda z \cdot z)\ (\lambda z \cdot z)} \\
\to & (\lambda z \cdot z)
\end{array}$$

2.7.2 Call-by-name

In call-by-name the function argument is not evaluated before the function is called; rather, the unevaluated term is passed as an argument and terms will only be evaluated when they are required. The call-by-name evaluation of (2.19) is

$$\begin{array}{ll}
& \underline{\big((\lambda x \cdot x)\ (\lambda y \cdot y\, y)\big)}\ \big((\lambda x \cdot x)\ (\lambda z \cdot z)\big) \\
\to & \underline{(\lambda y \cdot y\, y)\ \big((\lambda x \cdot x)\ (\lambda z \cdot z)\big)} \\
\to & \underline{\big((\lambda x \cdot x)\ (\lambda z \cdot z)\big)}\ \big((\lambda x \cdot x)\ (\lambda z \cdot z)\big) \\
\to & \underline{(\lambda z \cdot z)\ \big((\lambda x \cdot x)\ (\lambda z \cdot z)\big)} \\
\to & \underline{(\lambda x \cdot x)\ (\lambda z \cdot z)} \\
\to & \lambda z \cdot z
\end{array}$$

Call-by-name evaluation opens up new possibilities. For example, in Clean, we can define a function that returns the list of all natural numbers greater or equal than n as

```
nat :: Int -> [Int]
nat n = [n : nat (n + 1)]
```

Since this function returns an infinite list of natural numbers, evaluation of an expression such as

```
length (nat 0)
```

will never terminate. However, evaluation of an expression such as

```
take 5 (nat 0)
```

(where `take` n returns the first n elements of a list) will. Thus, call-by-name evaluation allows programming idioms that are not available in a call-by-value setting (Hughes, 1989; Leijen and Meijer, 2001). Moreover, although it is more difficult to understand the time and space behaviour of call-by-name programs (for instance, see Maessen, 2002), there are many data structures that can only be implemented efficiently in a call-by-name language (Okasaki, 1998; Bird et al., 1997). p. 64 Finally, we will argue in Section 2.8.5 that the use of a substructural type system is more suited to a call-by-name language than a call-by-value language.

2.7.3 Call-by-need

Despite its advantages, call-by-name evaluation has an important downside too: since arguments are substituted for formal arguments in unevaluated form, they may be evaluated more than once. We saw this in the derivation in the previous section, which evaluated $(\lambda x \cdot x)(\lambda z \cdot z)$ twice and therefore took one step more than the call-by-value evaluation.

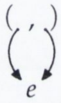
The call-by-need or *lazy* evaluation strategy implements the observational behaviour of call-by-name in a way that requires no more substitution steps than call-by-value reduction (Maraist et al., 1998). Sharing is indicated by a term of the form

$$\text{let } x = e \text{ in } (x, x)$$

which is syntactic sugar for

$$(\lambda x \cdot (x, x))\ e$$

and can be thought of as a graph



The call-by-name evaluation of (2.19) is

$$\begin{array}{ll}
 & \underline{\Big((\lambda x \cdot x)\ (\lambda y \cdot y\ y)\Big)}\ \Big((\lambda x \cdot x)\ (\lambda z \cdot z)\Big) \\
= & \underline{\Big(\text{let } x = \lambda y \cdot y\ y \text{ in } x\Big)}\ \Big((\lambda x \cdot x)\ (\lambda z \cdot z)\Big) \\
\rightarrow_{\text{V}} & \underline{(\lambda y \cdot y\ y)\ \Big((\lambda x \cdot x)\ (\lambda z \cdot z)\Big)} \\
= & \text{let } y = \underline{(\lambda x \cdot x)\ (\lambda z \cdot z)} \text{ in } y\ y \\
= & \text{let } y = \underline{(\text{let } x = \lambda z \cdot z \text{ in } x)} \text{ in } y\ y \\
\rightarrow_{\text{V}} & \underline{\text{let } y = \lambda z \cdot z \text{ in } y\ y} \\
\rightarrow_{\text{V}} & \underline{(\lambda z \cdot z)\ (\lambda z \cdot z)} \\
= & \text{let } z = \lambda z \cdot z \text{ in } z \\
\rightarrow_{\text{V}} & \lambda z \cdot z
\end{array}$$

where all the $(=)$ lines are simply rewriting some terms using the "let" syntax to make the sharing more explicit. Notice that the term $(\lambda x \cdot x)(\lambda z \cdot z)$ is reduced only once.

Since the topic of this dissertation is the design of a type system to allow side effects in a pure functional language without losing properties such as definiteness, it will come as no surprise that an evaluation strategy that may evaluate terms more than once is not suitable for our purposes. Since we are interested in lazy evaluation, we will opt to use the call-by-need semantics.

Unfortunately, although call-by-name and call-by-value are entirely standard, there is no universally agreed definition of call-by-need evaluation. The step definition of call-by-need we have shown are the "standard" reduction rules from (Maraist et al., 1998), where rule VALUE is slightly modified so that all variables are substituted at once (see also Maraist et al., 1998, Section 8, *On types and logic*).

2.7.4 Soundness

In the evaluation strategies discussed in this section, evaluation completes when the program has become a value.[1] A *soundness* proof that the type system guarantees that evaluation does not get stuck then takes the form of two theorems. The first states that if an expression e is well-typed, then either e is a value or e reduces to another term e':

Theorem 2 (Progress) *If* $\vdash e : \sigma$ *for some type* σ*, then either* e *is a value or there exists an expression* e' *such that* $e \longrightarrow e'$.

The second theorem states that evaluation preserves types:

Theorem 3 (Preservation *or* Subject Reduction) *If* $\Gamma \vdash e : \sigma$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e : \sigma$

Taken together, these two theorems prove that if a term is well-typed, it will either reduce to a value or reduce indefinitely.

2.7.5 On small-step semantics

The evaluation relations we specified in this section are known as *small-step* semantics: we specify what a single step in the relation looks like. The use of small-step semantics for soundness proofs was pioneered by Wright and Felleisen (1994).

Given a small-step semantics, we can characterize the evaluation of a term to a value by defining a relation $(\Rightarrow)$ such that $e \Rightarrow v$ for some value v if there exists a sequence of single reduction steps such that

$$e \longrightarrow e_0 \longrightarrow e_1 \longrightarrow \cdots \longrightarrow v$$

Alternatively, we can give evaluation rules that define $\Rightarrow$ directly. Such a definition of an evaluation strategy is known as a *big-step* semantics, *natural* semantics or *Kahn-style* semantics (after the scientist who first proposed it). Big-step semantics have some advantages over small-step semantics; for example, the well-known big-step semantics for call-by-need by Launchbury (1993) is much smaller and (arguably) more elegant than the small-step semantics we showed above.

Nevertheless, although big-step semantics can be used for soundness proofs (Harper, 1994, 1996), they are not quite as well suited. The problem is that we cannot state an equivalent of the progress lemma for languages that allow non-terminating (diverging) programs. For a non-terminating program e, we may not have $e \Rightarrow v$ for any v, even if e is well-typed. Thus, we cannot distinguish between non-terminating programs and programs that get stuck.

The standard solution is to introduce a new term `wrong` or `err`, and add new reduction rule
p. 12 for every possible ill-typed term. For example, in PCF (Section 1.2) we might have a rule that the application of the increment function to a boolean yields a type error:

$$\texttt{inc tt} \Rightarrow \texttt{wrong}$$

In this setup, the preservation lemma suffices and we do not need a separate progress lemma: since `wrong` does not have any type, preservation will prove that well-typed terms do not evaluate to `wrong`. However, as Leroy (2006) observes

[1] Maraist et al. refer to an "answer" instead of a "value"; the change in terminology is irrelevant for our current purposes and we will stick to "value" in this section. In later chapters where we will deal exclusively with the call-by-need semantics, we will refer to an "answer" as in the original article.

> This approach is unsatisfactory for two reasons: (1) extra rules must be provided to define $e \Rightarrow \texttt{err}$, which increases the size of the semantics; (2) there is a risk that the rules for $e \Rightarrow \texttt{err}$ are incomplete and miss some cases of "going wrong", in which case the type soundness statement does not guarantee that well-typed terms either evaluate to a value or diverge.

Leroy's paper contains a good discussion of the benefits of either approach, as well as suggesting some other solutions.

2.8 Side effects

An often cited advantage of functional programming languages such as *Clean* or *Haskell* is that they are *pure* or *referentially transparent*. In this section we will take a closer look at what we mean by purity, referential transparency and related properties, why they are easily lost when we add side effects, and how that can be avoided by using substructural logics or monads. We will not investigate the specifics of individual substructural logics (including uniqueness typing); this will be the subject of the next chapter.

2.8.1 Referential transparency

Referential transparency is a concept from the philosophy of language introduced by Quine:

> One of the fundamental principles governing identity is that of *substitutivity*—or, as it might well be called, that of *indiscernibility of identicals*. It provides that, *given a true statement of identity, one of its two terms may be substituted for the other in any true statement and the result will be true*. It is easy to find cases contrary to this principle.
>
> (Quine, 1953, *Reference and Modality*, emphasis in original)

He then proceeds to give a number of counterexamples. For example, it is true that

$$\text{Tegucigalpa} = \text{the capital of Honduras}$$

Now suppose that it is also true that

Philip believes that Tegucigalpa is in Nicaragua

Presumably, it is *not* true that

Philip believes that the capital of Honduras is in Nicaragua

Quine explains:

> The principle of substitutivity should not be extended to contexts in which the name to be supplanted occurs without referring simply to the object. Failure of substitutivity reveals merely that the occurrence to be supplanted is not *purely referential*, that is, that the statement depends not only on the object but on the form of the name [...] we may speak of the context "believe that" as *referentially opaque*.

So, a context is referentially transparent if it preserves referentiality (preserves substitution of identity) or referentially opaque if it does not. We call a formal language (such as a programming language) referentially transparent if every context in the language is referentially transparent.

We can paraphrase and state that referential transparency asserts the validity of the closure p. 23, p. 24 rule of equational logic (Section 2.2.1 or Figure 2.3), which justifies the claim that referential transparency makes equational reasoning possible. From this perspective we can clearly see the problem with the definition of referential transparency: when we take the set of identities that we want to reason with to be the empty set, every language is vacuously referentially transparent.

For example, do we have the identity `i` $\approx$ `i++` in C? Although the latter expression increments the value of i after being evaluated, the *value* of `i++` is defined to be `i`. Generally, when we allow for $e_1 \approx e_2$ when their values are the same then C is certainly not referentially transparent. However, when we allow $e_1 \approx e_2$ only when their values *and effects* are the same, then perhaps we should conclude that C *is* (non-trivially) referentially transparent.

However, consider trying to prove that addition on integers is commutative in C:

$$\forall (n\, m : \texttt{int}), n + m = m + n$$

Proving this lemma means allowing the identity $n + m \approx m + n$; but if we are doing equational reasoning, then by rule SUBST that means that we should be able to substitute any term for n and m (as long as they are of type `int`) and the rule should still hold. This will clearly fail; the expressions we substitute for n and m may have side effects, so that $n + m$ and $m + n$ may evaluate to different values. Hence, we will be unable to prove this lemma.

One might object that the lemma will hold for expressions without a side effect; i.e., restate the lemma so that it holds for "pure" integers only. That is a valid point of view, but unfortunately "purity" is hard to express in C. In particular, the type system of C does not differentiate between `i` and `i++` (both have type `int`). The monadic approach to side effects in a pure functional p. 65 language (Section 2.8.6) can be regarded as a solution to this problem: a "pure" integer will have type `Int`, but the construction of an integer that has a side-effect will have type m `Int` for some monadic type m (for instance, `IO Int`).

Ultimately, we are not interested in referential transparency *per se* but in reasoning about and proving properties of programs. As we saw, referential transparency does not guarantee that this is possible: every language is vacuously referential transparent when we take the empty set of identities. One might argue however that some languages are more "usefully" referentially transparent than others: if we cannot prove basic arithmetic laws such as commutativity of addition in some language, then terming that language referentially transparent is misleading at best. We want a language to be usefully referentially transparent so that we can prove these mathematical laws, essential for reasoning, and yet have side effects.

2.8.2 Leibniz' Law, extensionality, definiteness and unfoldability

In the previous section, we defined a context to be referentially transparent when it preserves substitution of identity. The principle of indiscernibility of identicals, as mentioned by Quine, is known as *Leibniz' Law*. Thus, a language is referentially transparent when Leibniz's Law holds. Of course, the problem is again which terms we consider as identical. (For an interesting application of Leibniz' Law in functional programming, see Baars and Swierstra, 2002).

An expression is *extensional* if the only thing we need to know about its subexpressions is their value (Søndergaard and Sestoft, 1990) (as opposed to how these values were constructed). For example, it does not matter whether we evaluate $f(1+5)$ or $f(6)$. It is claimed that extensionality is a consequence of referential transparency (Strachey, 2000/1967), but it is not clear to me in what sense that is true; it will certainly be difficult to make that statement precise.

A variable is called *definite* if every occurrence of the variable refers to the same value. For example, in mathematics we assume that every x in $3x^3 + 2x^2 + 17x$ refers to the same value. In programming languages with side effects, however, this is often not the case. For example, in C, if x is a global value, then it may well be that the two occurrences of x in

```
x + f() + x
```

refer to a different value (if f modifies it). Clearly, we want all variables to be definite if we want to be able to reason about programs. That is, we do not want variables to actually vary! It should be noted that many papers seem to use the term "referential transparency" to denote definiteness.

Finally, a language has the *unfoldability* property if we can always replace a function call by its definition. I.e., if

$$f\,x = e$$

then we should be able to replace $f a$ by $[x \mapsto a]e$. Unfoldability means that we can take a function definition such as the above and treat it as an actual identity $f x \approx e$ which we can then use for equational reasoning. To a large extent, unfoldability depends on evaluation strategies (Section 2.7). For example, if $\perp$ denotes a non-terminating computation, then given p. 54

$$g\,x = 5$$

in a strict language $g\perp$ will not terminate, but unfolding g will simply give 5. In a lazy language, both $g\perp$ and its unfolding will evaluate to 5, but even in a lazy language we need to be careful with non-termination; we cannot avoid strictness completely. For example, given a "lazy" conjunction operator, which evaluates its second argument only if the first argument is not equal to `False`, then

$$\texttt{False} \wedge \perp$$

will evaluate to `False`, but

$$\perp \wedge \texttt{False}$$

will evaluate to $\perp$. Hence, we cannot prove the lemma

$$\forall (a\,b : \mathbb{B}), a \wedge b = b \wedge a$$

but can only prove

$$\forall (a\,b : \mathbb{B}), a \neq \perp \rightarrow b \neq \perp \rightarrow a \wedge b = b \wedge a$$

Some theorem provers such as Sparkle (de Mol et al., 2002) are rather fastidious about termination; other people argue that it is okay (with some reservations) to ignore this issue (Danielsson et al., 2006). We will not consider non-termination further in relation to side effects and purity or equational reasoning.

2.8.3 Purity

The running theme of this section is that we want to be able to reason about programs; that is, we want to be able to prove basic lemmas such as mathematical arithmetic identities; we want extensionality, variables should be definite, unfoldability should apply. We want the language to be *pure*, and yet have side effects.

In *What is a Purely Functional Language*, Sabry (1998) considers various ways to characterize purity. He rejects referential transparency as a possibility because it does not have a universally agreed definition, and notes that the simple fact that equational reasoning is possible gives few guarantees. As we noted previously, *every* language permits equational reasoning; the question is how many identities we will be able to prove. After reviewing various other possibilities, he eventually finds one suitable candidate. He defines a language to be *purely functional* if

1. it is a conservative extension of the simply typed lambda-calculus,

2. it has well-defined call-by-value, call-by-need, and call-by-name evaluation functions, and

3. all three evaluation functions are weakly equivalent

A language is a conservative extension of the simply typed language calculus (STLC) when it includes every well-typed term of the STLC, and the semantics coincide on those terms; two semantics are weakly equivalent if they both evaluate a term t to the same value, or one of the two semantics is undefined on t. We will consider whether this characterization of a purely functional language is appropriate for our purposes in the next section.

2.8.4 Substructural logics

It is clear that the core lambda calculus is purely functional following Sabry's definition (previous section). Moreover, it is not hard to see that extensionality, definiteness and unfoldability apply. However, when we add side effects to the lambda calculus, unless we take precautions we quickly loose many of these properties.

For example, suppose we add support for arrays to the language, and an operation that modifies an array in-place:

```
update_array :: Array → Int → Int → Array
update_array array index new_value = ..
```

We have not added "global" state to the language, but have lost definiteness. For example, in

```
f :: Array → (Array, Array, Array)
f arr = (arr, update_array arr 0 1, arr)
```

the first occurrence of `arr` will have a different value than the last (assuming left-to-right evaluation of the arguments). Or, perhaps even worse, if the `update_array` happens to be evaluated first, all occurrences of `arr` will have the same value: the updated array!

Similarly, we loose definiteness when we add a function to read an integer from the user. The two occurrences of `read_int` in this function will not have the same value:

```
g = read_int + read_int
```

Moreover, we have lost the identity $2 * n = n + n$, because the above program is not equivalent to

```
h = 2 * read_int
```

p. 11 As explained in the introduction (Chapter 1), substructural logics such as uniqueness typing can be used to guarantee that a variable will be used at most (or exactly) once. The type of `update_array` is modified to

```
update_array :: Array• → Int → Int → Array•
```

where the bullets (•) in the type indicate the uniqueness of the array: the array must be used only once (the specifics of individual logics will be considered in the next chapter). This means that the definition of f, above, will be rejected; in fact, we can safely add `update_array` without loosing definiteness, because the type system guarantees that any array that is passed to `update_array` will be used only once. Definiteness holds therefore trivially: arrays that are modified can only occur once.

"True" side-effecting functions such as `read_int` refer to an implicit state (the "world state"). In order to add such functions to a purely functional language we have to make this state explicit. The type of functions such as `read_int` is thus modified to

```
read_int :: World• → (Int, World•)
```

The difference between g and h is now more explicit:

```
g world = let (i, world')  = read_int world
              (j, world'') = read_int world'
          in (i + j, world'')

h world = let (i, world') = read_int world
          in (2 * i, world')
```

Again, the type system will guarantee that every `world` object is used exactly once.

However, does this approach allow us to prove useful lemmas? For example, can we prove that

$$\forall (n : \mathbb{N}), 2 \cdot n = n + n \tag{2.20}$$

Consider a function that finds the minimal element in an array and then discards the array. For efficiency purposes, the function does a partial (in-place) sort of the array to find the minimal element. Hence, the function will have type

```
find_minimal :: Array• → Int
```

We can use this function to multiply the smallest element of the array by two:

```
twice_minimal arr = 2 * find_minimal arr
```

By equation (2.20), this should be equal to

```
twice_minimal arr = find_minimal arr + find_minimal arr
```

but in fact that is not even a valid program, much less equivalent to the previous program. So, it seems that we have lost ("useful") equational reasoning.[1] Nor does this language seem to meet Sabry's definition of a purely functional language. For example, consider the following term

```
(\x -> (x, x)) (find_minimal arr)
```

[1] Note that the "observable sharing" extension to Haskell (Claessen and Sands, 1999)—another calculus that is sensitive to sharing—has similar problems. Diviánszky (2006) notes that purity may be recovered in such a calculus using a uniqueness type system, but unfortunately does not show how. An investigation into this topic might be interesting future work.

for some array `arr`. Under call-by-need semantics, this term will be in normal form after the evaluation of `find_minimal arr`, but under call-by-name semantics this term reduces to

```
(find_minimal arr, find_minimal arr)
```

which is certainly not equivalent (in fact, will be rejected by the type checker).

However, if we regard equation (2.20) as having an implicit side-condition that the left hand side and the right hand side of the equation must be well-typed then we *can* use equational reasoning. Similarly, it should be possible to adjust Sabry's definition so that the various semantics p. 159 take typing into account. We will prove in Chapter 7 that the call-by-need semantics preserves types, but that will not be true for the call-by-name semantics (although it may be true for the call-by-value semantics). If we modify the weak equivalence to include that the types must remain well-typed at every step during evaluation, we should be able to prove that a language that uses a substructural logic can be purely functional. However, at this stage this is only a conjecture and we must unfortunately leave a formal treatment of this issue to future work.

2.8.5 Laziness

In this section we consider one more, slightly subtler, issue related to substructural logics and equational reasoning. By definition, the unit type $\mathbb{1}$ (denoted as `()` in Haskell or `UNIT` in Clean) has a single element, also denoted by $\mathbb{1}$. Therefore, we should be able to prove that

$$\forall (u : \mathbb{1}), u = \mathbb{1} \tag{2.21}$$

Will we be able to prove this in a functional language that uses substructural logics to add side effects in a "safe" manner? Suppose we have a function `hello` that prints "hello world" to the console:

```
hello :: World• → World•
```

Given `hello`, we can easily define a derived function `hello'`:

```
hello' :: World• → 𝟙
hello' world = let world' = hello world in 𝟙
```

Now we want to consider the following problem: is the program

```
λworld. hello' world
```

equivalent to the program

```
λworld. 𝟙
```

Equation (2.21) tells us that these two programs are indeed equivalent, but at first sight they seem not to be equivalent at all: we have lost the side effect (the printing of "hello world") in the second example. However, in a lazy language, `hello'` does not have a side effect either! Since the result of the call to `hello` is ignored in the body of `hello'`, laziness means that `hello` will never be invoked at all by `hello'`. Hence, these two programs *are* equivalent, but only in a lazy language. In a strict language, these two programs are not equivalent, and it is not clear how to modify equation (2.21) in a similar vein to equation (2.20) to make it hold.

2.8.6 Monads

The monadic approach abstracts over *functions* from one world state to another as follows:

```
data IO a = IO (World → (a, World))

unIO :: IO a → World → (a, World)
unIO (IO f) = f
```

For example, the functions that open and close files have type

```
openFile :: FilePath → IOMode → IO Handle
hClose   :: Handle → IO ()
```

which, when expanded, are approximately the same as

```
openFile :: FilePath → IOMode → World → (Handle, World)
hClose   :: Handle → World → ((), World)
```

The crucial difference between the monadic approach and the approach in Clean, however, is that the `IO` type is abstract (and the `unIO` function not exported). Programmers never manipulate `world` objects directly. Instead, the standard library offers numerous primitive functions on the world state, and an interface to string these functions together.

This interface consists of two functions, called `return` and »= (also called `bind`), which are defined as follows.

```
return :: a → IO a
return x = IO (λworld → (x, world))

(>>=) :: IO a → (a → IO b) → IO b
x >>= f = IO (λworld → let (a, world') = unIO x world
                        in unIO (f a) world')
```

In fact, any type *m* that supports two operations

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

which satisfy a few laws (Section 2.10.3) is known as a *monad*. There are many such structures, p. 75
but a general discussion of monads is beyond the scope of this dissertation. Wadler (1995) provides a good introduction.

Using the monadic interface and assuming a function

```
readInt :: IO Int
```

we can write the program that reads two integers from the user and adds them together as

```
readTwo :: IO Int
readTwo = readInt >>= λa →
          readInt >>= λb →
          return (a + b)
```

This is exactly the same program as function *g* in Section 2.8.4, except that the threading of the p. 62
world state is now hidden inside the monad. As a consequence, re-using the same world state twice is impossible.

Since monads are a very useful abstraction, Haskell provides a special syntax to write monadic programs. Using this syntax, the above program can be rewritten as

```
readTwo :: IO Int
readTwo = do a ← readInt
             b ← readInt
             return (a + b)
```

Note however that this is a change in syntax only; this program can be "desugared" into the previous version. Although this is superficially very similar to an imperative program (reading ":=" for "←"), this is still a functional program with all the associated benefits. See (Peyton Jones, 2001, Section 7) for a discussion of this point.

2.8.7 Uniqueness typing versus Monads

At the time of writing, the world's most popular purely functional programming language is undoubtedly Haskell, which uses monads for I/O. Although it is largely orthogonal to the subject of this dissertation, the reader is likely to wonder about the relative merits of uniqueness typing and the monadic approach. We will therefore briefly discuss the advantages and disadvantages of both approaches in this section.

Before we begin the comparison it must be pointed out that monads are a very general functional design pattern, which can be used for much more than just modelling I/O—this applies equally to languages with and without uniqueness typing. Even Clean programmers use monads when they are found to provide a useful interface (e.g., Plasmeijer et al., 2007). Conversely, uniqueness typing can be used for purposes other than I/O too; for instance, the compiler can use uniqueness information to generate more efficient code. Besides, the choice is ultimately a matter of taste: *de gustibus perpetue disputandum*!

We are not aware of any serious studies that compare the two, except for one paper that compares proving properties over programs written in monadic style and programs written using a uniqueness type system. That paper comes to the conclusion that, at least as far as proofs are concerned, the difference is negligible:

> “ In (Butterfield and Strong, 2002) we explored a program which manipulated a small part of the world file system component, namely a single file. This lead us to a tentative conclusion that the explicit environment passing style of Clean programs was easier to reason about because (i) we could confine our attention to the small portion of the world state under consideration, and (ii) because we did not have the small overhead of unwrapping the monad.
>
> However, the case study presented here differs crucially in that (i) the program involves the state of the entire file system at every step, and (ii) we have developed a set of operators which simplify the proofs in both paradigms. We can conclude that for this case study, the differences in reasoning overhead between the two paradigms are too small to be of any concern.
>
> (Dowse et al., 2003)

Nevertheless, some general statements can and have been made about the differences between both approaches. In the paper that introduces Haskell's monadic I/O system, Peyton Jones and Wadler (1993) make the following comparison, which sums up the most commonly heard arguments from either side:

> “Compared to the monad approach, [uniqueness typing] suffers from a number of drawbacks: programs become more cluttered; the linear type system has to be explained to the programmer and implemented in the compiler; and code-improving transformations need to be re-examined to ensure they preserve linearity. The latter problem may be important; Wakeling found that some standard transformations could not be performed in the presence of linearity (Wakeling, 1990).
>
> The big advantage of a linear type system is that it enables us to write programs which manipulate more than one piece of updatable state at a time. The monadic [presentations of arrays passes] the array around implicitly, and hence can only easily handle one at a time. This is an important area for future work.

We will have a closer look at the claims made in this comparison.

Programs become more cluttered

Consider the Haskell function that writes the characters ‘a’ and ‘b’ to a file:

```
ab :: Handle → IO ()
ab f = do hPutChar f 'a'
          hPutChar f 'b'
```

The Clean equivalent is

```
ab :: (File• → File•)
ab = fwritec 'a' o fwritec 'b'
```

I will claim that the Clean version is less “cluttered” than the Haskell version, and certainly has a more functional feel. The objection is nonetheless fair: the example above is one that actually benefits from uniqueness typing. The most important disadvantage of uniqueness typing is that it affects all types, even those of function that do not deal with I/O. For example, the identity function has the following type in Clean:

`id ::` $t^u \xrightarrow{\times} t^u$

These types get increasingly more complicated with multiple arguments and higher-order functions. This is an important issue even when types can be inferred: functional programmers rightly like to write down the types of their top-level definitions. However, part of our thesis is that these types can be simplified. Moreover, although we have left this issue for future work (Section 8.2.4), p. 213 adding syntactic conventions can go a long way towards removing all or nearly all “clutter”.

Linear type system has to be explained to the programmer

In our experience, the concept of a monad is significantly more difficult to explain than the concept of uniqueness typing. Although monads are simple to define, they are a very general concept making them difficult to comprehend. Whether because of a lack of good teaching materials or intrinsic complexity, monads are a stumbling block for most new Haskell programmers. When teaching object oriented programming, we do not teach design patterns until the programmer is comfortable with the paradigm and the language. Unfortunately, for a beginning student wishing to write his *hello world* example in Haskell, he will immediately be faced with monads. One might claim that this is the wrong example to start with, but then again *I/O is the raison d’être of every program* (Peyton Jones, 2001).

In a language with uniqueness typing writing *hello world* is no more difficult than writing a function to add two numbers, or requires any concepts that are not required when writing the addition function. The types of the functions can be inferred by the compiler and can therefore be ignored by the student if he wishes. Some people even claim that programmers do not really need to understand a type system:

> “ The [Boxy] type system is arguably too complicated for Joe Programmer to understand, but that is true of many type systems, and perhaps it does not matter too much: in practice, Joe Programmer usually works by running the compiler repeatedly, treating the compiler as the specification of the type system.
>
> (Vytiniotis et al., 2006)

While I certainly do not want to argue that uniqueness typing is too complicated to understand, I *will* argue that getting started with I/O in a programming language with uniqueness typing is easier than getting started with monadic I/O, and moreover, the programmer does not even need to understand the type system before he can do so.

Linear type system has to be implemented

While this statement is obviously true, this dissertation aims to show that with a suitable formalization of uniqueness typing, the cost of implementation is very low (in particular, see Section 6.4). p. 149

Code-improving transformations need to be re-examined

While this is listed as a disadvantage in the original citation, I maintain that this is actually an advantage (a feature, not a bug!). In Section 2.8 of *Tackling the Awkward Squad*, Peyton Jones (2001) discusses that in ghc, IO is internally defined as

```
type IO a = World → (a, World)
```

This definition is not available to the programmer, but in ghc’s internal representation, the definition of the IO monad is expanded. This is dangerous: the fact that the World must be kept single-threaded is lost, and “optimization” passes in the compiler might duplicate side effects. Peyton Jones claims that this will not happen because of certain properties of the compiler, but these properties cannot be verified. Contrast this to the following quote from the *History of Haskell* paper:

> “ GHC appears to be the first compiler to use System F as a typed intermediate language [...]. Several years later, we added a “Core Lint” typechecker that checked that the output of each pass remained well-typed. If the compiler is correct, this check will always succeed, but it provides a surprisingly strong consistency check—many, perhaps most bugs in the optimizer produce type-incorrect code. Furthermore, catching compiler bugs this way is vastly cheaper than generating incorrect code, running it, getting a segmentation fault, debugging it with gdb, and gradually tracing the problem back to its original cause. Core Lint often nails the error immediately. This consistency checking turned out to be one of the biggest benefits of a typed intermediate language, although it took us a remarkably long time to recognize this fact.
>
> (Hudak et al., 2007, Section 9.1)

Even if one feels that monadic I/O is easier to work with than uniqueness typing, we can build a monadic interface to I/O on top of uniqueness typing (see Jones, 1995c, for one worked out example). For example, we can define `IO` as

```
type IO a = World• → (a, World•)
```

If the uniqueness type system is then maintained throughout the compiler (that is, for the internal representation as well as the source language) expanding the definition of `IO` is safe: optimization passes in the compiler that might inadvertently duplicate side effects will be caught by the type checker.

Granularity

The most important objection to monadic I/O (other than that monads are difficult to explain to beginning students) is the following:

> If we compare the monad approach to [uniqueness typing] then there are some striking differences. The environment that is manipulated in the monad approach is *implicit* and "appears" only in the `IO` type. As a result programming in the system creates one single spine of I/O operations and therefore over determines order of evaluation [...].
>
> (Achten and Plasmeijer, 1995, emphasis in original)

For example, consider the following function in Clean:

```
f :: ({Char}•, {Char}•)• → ({Char}•, {Char}•)•
f (arr1, arr2) = (update arr1 0 'x', update arr2 0 'x')
```

Function *f* takes a pair of arrays of characters as arguments, and updates both arrays (sets the first element in the array to "x"). Compare this to the Haskell implementation:

```
f :: (IOArray Int Char, IOArray Int Char) → IO ()
f (arr1, arr2) = do writeArray arr1 0 'x'
                    writeArray arr2 0 'x'
```

Even apart from the fact that the Haskell programs looks like an imperative program rather than a functional one, there is one important difference between both programs: in the Haskell program, the first array must be updated before the second. In the Clean program the order of evaluation is unspecified; in fact, both updates could even be done in parallel. Conceptually, the problem is that in Haskell the whole "world" is updated when the array is updated (albeit implicitly through the monadic interface), whereas in the Clean program only part of the world is updated. This "splitting up" of the world state is an important concept in uniqueness typing (Plasmeijer and van Eekelen, 1999; see also Achten and Plasmeijer, 1995):

> [Using monads to implement side-effects] is simple but has the disadvantage that all objects to be destructively updated must be maintained by the system in a single state which is kept hidden from the programmer. Clean does not have this restriction. One can have arbitrary states which can be passed around explicitly. Such a state can be fractured into independent parts (e.g., distinct variables for the file system and the event queue).

This problem is acknowledged by proponents of monadic I/O (Wadler, 1997; Peyton Jones, 2003). One proposed solution is to introduce the concept of commutative monads, which have the property that the following two expressions are equivalent:

```
do a ← f x          do b ← g y
   b ← g y             a ← f x
   h a b               h a b
```

Unfortunately the `IO` monad is certainly not commutative, so introducing commutative monads by itself will not solve this problem: there must be a way to split the world.

Of course, one must be careful when splitting the world. Clean for instance allows to "split off" a file from the world state; writes to that file affect the file, but do not affect the world:

```
writeFile :: Char → File• → File•
```

However, writing to a file obviously does have an effect on the world; for instance, if there is also a library function to query the disk space available,

```
diskSpace :: World• → (Int, World•)
```

then referential transparency may be lost.

A related problem with the granularity of the monadic approach is that it is more difficult to express that *some* parts of a data structure may be destructive modified, but others may not. Using uniqueness typing, it is straightforward to give a type to spine-unique lists ($[a]^\bullet$ for some a); the *elements* of such a list cannot be modified in-place, but the *list* itself can. Of course, it is possible to simulate this using pointers (`Refs`) in the `IO` monad, but not with the same ease and not without having to define new data structures.

2.9 Language extensions

2.9.1 Recursion

In the untyped lambda calculus, we can define non-terminating programs with the help of a *fixed point combinator* (attributed to the logician Curry), defined as

```
Y = λf → (λx → f (x x)) (λx → f (x x))
```

This combinator gives us general recursion, and hence non-termination. For example, we can rewrite the factorial function

```
fac n = if n == 0 then 1 else n * fac (n - 1)
```

as

```
fac = Y (λf λn → if n == 0 then 1 else n * f (n - 1))
```

In most typed languages, however, we cannot type Y due to the self-application of x: since x is applied to an argument, it must have a function type ($\sigma \rightarrow \sigma'$); but since x is applied to *itself*, we must have that $x : \sigma$. Hence, we must have a type σ such that $\sigma \cong \sigma \rightarrow \sigma'$. As observed by Morris (1969), such a type can be defined when we introduce a fixed point operator on the type level ($\mu a \cdot a \rightarrow \sigma'$), but most languages do not support general recursive types because they make p. 37 type checking difficult. However, many languages do support algebraic data types (Section 2.4.2), which provides a controlled recursion on the type level; we can use an algebraic data type to define Y (formulation due to Stefan Holdermans, personal communication):

```
data F a = F {unF :: F a → a}

{-# NOINLINE y #-}

y   :: (a → a) → a
y f =  g  (g . unF)
  where
    g x = f (x (F x))
```

(The pragma is necessary to stop ghc from trying to inline the y combinator, which will make compilation non-terminating.)

Of course, an easier solution is to introduce a fixpoint operator μ into the term language. The typing rule can be presented as

$$\frac{\Gamma, x : \sigma \vdash e : \sigma}{\Gamma \vdash \mu x \cdot e : \sigma}\textsc{Rec}$$

Equivalently, we can introduce a recursive let expression

$$\frac{\Gamma, x : \sigma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \sigma'}{\Gamma \vdash \text{letrec } x = e \text{ in } e' : \sigma'}\textsc{LetRec}$$

To allow for multiple mutually recursive definitions, we can modify the rule to

$$\frac{\Gamma, \overline{x : \sigma} \vdash \overline{e : \sigma} \qquad \Gamma, \overline{x : \sigma} \vdash e' : \sigma'}{\Gamma \vdash \text{letrec } \overline{x = e} \text{ in } e' : \sigma'}\textsc{LetRec}$$

Alternatively, an encoding of mutual recursion using pairs is also possible. For example, we can define odd as

```
odd = letrec even = λn → if n == 0 then True  else odd  (n - 1)
             odd  = λn → if n == 0 then False else even (n - 1)
      in odd
```

where we define even and odd as two mutually recursive functions, or we can define a *pair* of the functions even and odd as a single recursive definition:

```
odd = letrec e_o = ( λn → if n == 0 then True  else snd e_o (n - 1)
                   , λn → if n == 0 then False else fst e_o (n - 1)
                   )
      in snd e_o
```

As we have seen briefly in Section 2.6 (and will see in more detail in Chapter 3), some languages impose restrictions on variables that are used more than once. However, these restrictions will also apply to variables that are defined recursively (Section 3.2.5). So although the encoding of mutual recursion using pairs requires more than once reference to e_o in its definition, this will not impose any additional restrictions on the use of the recursively defined terms. p. 52, p. 77 p. 83

Finally, it should be mentioned that in a language such as Coq that does not support generic recursion but does support co-inductive data types, generic recursion can be recovered by embedding it into a special monad known as the *Partiality monad* (Capretta, 2005).

2.9.2 Qualified types

The theory of qualified types was introduced by Jones (1994, 1995b,a). We will not make use of qualified types in this thesis, but it has been suggested that we should (we will address this in the p. 201, p. 77 conclusions, Chapter 8) and some of the related work does (Chapter 3). We will therefore present a brief description here. For a more thorough introduction, please refer to (Jones, 1995a).

Qualified types introduce an "intermediate" between monomorphic types and polymorphic types. For example, a type $\forall(a : *) \cdot \pi(a) \Rightarrow a \rightarrow a \rightarrow \mathbb{B}$ describes the set of types

$$\{a \rightarrow a \rightarrow \mathbb{B} \mid a : \star, \pi(a)\}$$

The exact nature of the predicates (π) is not important, but we must have an entailment relation $\Vdash$ between (finite) sets of predicates, satisfying the following conditions:

- $P \Vdash P'$ whenever $P \supseteq P'$ (monotonicity)
- If $P \Vdash Q$ and $Q \Vdash R$ then $P \Vdash R$ (transitivity)
- If $P \Vdash Q$ then $SP \Vdash SQ$ for some substitution S (closure)

The typing rules are shown in Figure 2.18. The typing relation takes the form

$$P \mid \Gamma \vdash e : \sigma$$

which states that e has type σ in environment Γ assuming the set of constraints P. The system we have shown is the extension of System F with qualified types; other presentations based on System F2 or the Hindley/Milner type system are also possible. Since we are showing an explicitly typed calculus, we are using Π instead of $\forall$ in line with the notation elsewhere in this chapter.

Type schemes $\forall \overline{a} \cdot P \Rightarrow \sigma$ are usually restricted to be unambiguous: $\overline{a} \cup \mathrm{fv}(P) \subseteq \mathrm{fv}(\sigma)$. This restriction can be relaxed in some cases, but one must be careful; see (Jones, 1995a) for details.

Type language

σ ::=		type
	a	type variable
	$\sigma \rightarrow \sigma$	function space
	$\Pi(a : *) \cdot \sigma$	universal type
	$\pi \Rightarrow \sigma$	qualified type (unambiguous π)

Typing rules

$$\frac{x : \sigma \in \Gamma}{P \mid \Gamma \vdash x : \sigma}\ \textsc{Var}$$

$$\frac{P \mid \Gamma, x : \sigma \vdash e : \sigma'}{P \mid \Gamma \vdash \lambda(x : \sigma) \cdot e : \sigma \rightarrow \sigma'}\ \textsc{Abs} \qquad \frac{P \mid \Gamma \vdash e : \sigma \rightarrow \sigma' \qquad P \mid \Gamma \vdash e' : \sigma}{P \mid \Gamma \vdash e\, e' : \sigma'}\ \textsc{App}$$

$$\frac{P \mid \Gamma \vdash t : \sigma \qquad a \notin \mathrm{fv}(\Gamma) \cup \mathrm{fv}(P)}{P \mid \Gamma \vdash \Lambda(a : *) \cdot t : \Pi(a : *) \cdot \sigma}\ \textsc{TAbs} \qquad \frac{P \mid \Gamma \vdash e : \Pi(a : *) \cdot \sigma}{P \mid \Gamma \vdash e\ \{\sigma'\} : \sigma[a := \sigma']}\ \textsc{TApp}$$

$$\frac{P, \pi \mid \Gamma \vdash e : \sigma}{P \mid \Gamma \vdash e : \pi \Rightarrow \sigma}\ \textsc{Pred} \qquad \frac{P \mid \Gamma \vdash e : \pi \Rightarrow \sigma \qquad P \Vdash \pi}{P \mid \Gamma \vdash e : \sigma}\ \textsc{Sat}$$

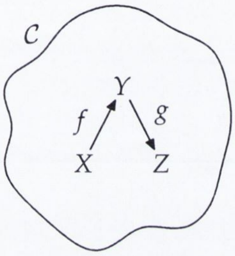
Figure 2.18: System F with qualified types

2.10 Category theory

Category theory is an increasingly important tool in theoretical computer science. However, this is not the place—nor am I the author—to give a detailed exposition of this large subject. We will need category theory only in the discussion of related work in the next chapter; the summary we provide here therefore builds up only to the concepts that we will have a use for. For more details, the reader is referred to one the many texts on category theory such as (Awodey, 2006), (Barr and Wells, 1999) or (Pierce, 1991).

2.10.1 Fundamental concepts

A *category* is a collection of *objects* and *arrows* between them, together with a few axioms. We will often draw a category like this:



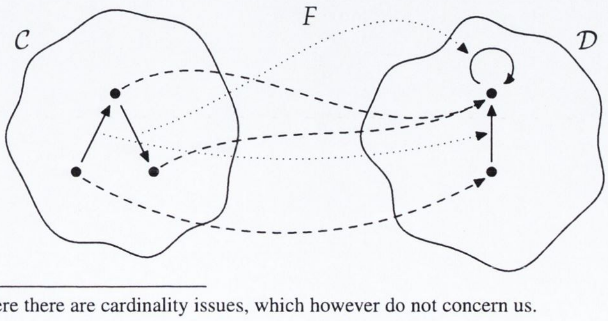
As a notational convention, we will use $\mathcal{C}, \mathcal{D}$ for categories, X, Y, Z for objects in categories, and f, g for arrows in categories.

For all arrows $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ the composition $g \circ f : X \longrightarrow Z$ must be defined, and composition must be associative ($g \circ f$ is sometimes also written as gf; $g \circ g$ can be written as g^2). Moreover, for every object X in the category, there must be an identity arrow 1_X on X satisfying $f \circ 1_X = f = 1_Y \circ f$ for all $f : X \longrightarrow Y$.

When we draw a category the composition and identity arrows are often left implicit (as we did above). Do not be misled by the apparent directionality of the arrows in a category: an arrow from X to Y is not necessarily a way to get from an X to a Y (although it often is); rather, it merely indicates *some* connection between X and Y.

An important example of a category is **Sets**, the category of sets and total functions[1]. Composition in **Sets** is function composition (which we know to be associative); the identity arrow on an object X is the identity function on X.

We can go "one level up" and consider the category **Cat** of all categories and structure preserving maps between categories, called *functors* (denoted by F, G, U). A functor from $\mathcal{C}$ to $\mathcal{D}$ has two components: a map from the objects in $\mathcal{C}$ to objects in $\mathcal{D}$, and a map from the arrows in $\mathcal{C}$ to arrows in $\mathcal{D}$.



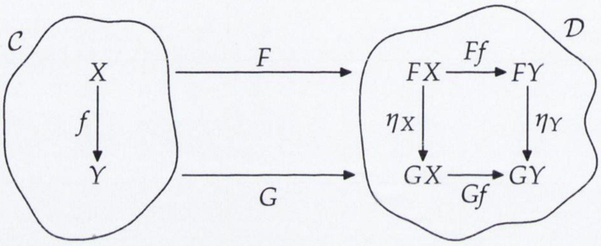
[1] Here and elsewhere there are cardinality issues, which however do not concern us.

A functor must satisfy the following axioms

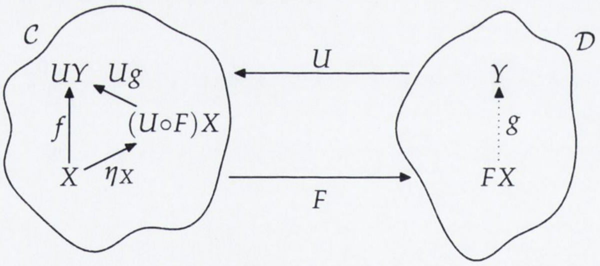
- If $f : X \to Y$ in $\mathcal{C}$, then $F(f) : F(X) \to F(Y)$ in $\mathcal{D}$
- $F(g \circ f) = F(g) \circ F(f)$
- $F(1_X) = 1_{F(X)}$.

For every category $\mathcal{C}$ we can define a functor $\mathrm{Hom}_{\mathcal{C}}(X, -) : \mathcal{C} \to \mathbf{Sets}$ which takes any object Y in $\mathcal{C}$ to the set of arrows from X to Y (in **Sets**).

On the next level up we find the *functor category* $\mathrm{Fun}(\mathcal{C}, \mathcal{D})$ of functors between $\mathcal{C}$ and $\mathcal{D}$ and *natural transformations* between functors. A natural transformation η can be thought of as P. 35 a generalization of a polymorphic function (in the sense of System F, see Section 2.4). Given two functors $F, G : \mathcal{C} \to \mathcal{D}$ and any object X in $\mathcal{C}$, η_X is an arrow $FX \to GX$ in $\mathcal{D}$ such that the following diagram[1] in $\mathcal{D}$ commutes for any $f : X \to Y$:



Finally, functors and natural transformations give rise to the notion of an *adjunction*. An adjunction consists of a pair of adjoint functors $F : \mathcal{C} \to \mathcal{D}$ and $U : \mathcal{D} \to \mathcal{C}$ together with a natural transformation $\eta : 1_{\mathcal{C}} \to U \circ F$ so that for any $X \in \mathcal{C}$, $Y \in \mathcal{D}$, and $f : X \to UY$, there exists a unique $g : FX \to Y$ such that the following diagram in $\mathcal{C}$ commutes:



When F and U form an adjunction, we write $F \dashv U$ and say that F is left adjoint to U and U is right adjoint to F. Adjunctions describe a relation between two categories. This relation is more evident from the following equivalent definition: $F \dashv U$ is an adjunction if there exists an isomorphism between hom-sets which is natural in both X and Y:

$$\phi : \mathrm{Hom}_{\mathcal{D}}(FX, Y) \cong \mathrm{Hom}_{\mathcal{C}}(X, UY) : \psi$$

where ϕ is given by $\phi(g) = U(g) \circ \eta_X$. Alternatively, the unit $\eta : 1_{\mathcal{C}} \to U \circ F$ and the *co-unit* $\epsilon : F \circ U \to 1_{\mathcal{D}}$ can be defined in terms of the witness of the isomorphism:

$$\eta_C = \phi(1_{FC})$$
$$\epsilon_D = \psi(1_{UD})$$

[1] By an abuse of notation we will present categories and diagrams in categories in the same way.

2.10.2 Products, exponentials and currying

The *product* of two objects X and Y in a category $\mathcal{C}$ is an object $X \otimes Y$ together with two projection arrows $\pi_1 : X \otimes Y \to X$ and $\pi_2 : X \otimes Y \to Y$ (often called `fst` and `snd` in functional programming languages) so that for all other objects Z with two arrows $f : Z \to X$ and $g : Z \to Y$, there exists a unique arrow $h : Z \to X \otimes Y$ such that $f = \pi_1 \circ h$ and $g = \pi_2 \circ h$. If $X \otimes Y$ and $X' \otimes Y'$ are product objects, then for every pair of arrows $f : X \to X'$ and $g : Y \to Y'$, $f \otimes g : X \otimes Y \to X' \otimes Y'$ is the arrow $\langle f \circ \pi_1, g \circ \pi_2 \rangle$. In **Sets**, the product of two objects (two sets) is the cartesian product of the sets (not all categories have products).

We can now define a functor $X \otimes - : \mathcal{C} \to \mathcal{C}$ which maps an object $Y \in \mathcal{C}$ to $X \otimes Y$ and an arrow $f : Y \to Z \in \mathcal{C}$ to $1_X \otimes f : (X \otimes Y) \to (X \otimes Z)$. We define the *exponential* $(-)^X$ to be right adjoint to this functor; that is, we must have a natural isomorphism

$$\phi : \mathrm{Hom}_{\mathcal{C}}(X \otimes Y, Z) \cong \mathrm{Hom}_{\mathcal{C}}(Y, Z^X) : \psi$$

In **Sets**, Z^X is the set of all functions from $X \to Z$. Functional programmers will recognize ϕ and ψ as

```
curry   :: ((a, b) → c) → (b → a → c)
uncurry :: (b → a → c) → ((a, b) → c)
```

Semantically, function application can defined in terms of the co-unit $\epsilon : (X \otimes -) \circ (-)^X \to 1_{\mathcal{C}} = (X, (-)^X) \to 1_{\mathcal{C}}$ of this adjunction, that is as $\epsilon_Y = \psi(1_{Y^X}) : (X, Y^X) \to Y$, or in a functional language:

```
uncurry (id :: (a -> b) -> (a -> b)) :: (a, a -> b) -> b
```

2.10.3 Monads

In category theory, a monad on a category $\mathcal{C}$ is defined to be a triple (T, η, μ) of a functor $T : \mathcal{C} \to \mathcal{C}$ and two natural transformation $\eta : 1_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ such that the following diagrams commute:

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[dr, "="'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T\eta"'] \arrow[dl, "="] \\
& T &
\end{tikzcd}
\qquad
\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

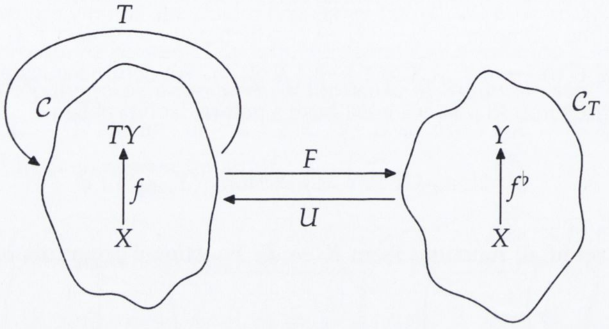
We refer to η as the unit of the monad and to μ as the multiplication; functional programmers usually refer to the unit as the `return` operation, and to the multiplication as the `join` operation. Note that `bind` (Section 2.8.6) can be recovered using $\lambda x \cdot \lambda f \cdot \mu\,((T\,f)\,x)$. p. 65

Two remarks on notation. First, the composition ηT of a natural transformation η with a functor T is defined to be $(\eta T)(X) = \eta(TX)$ (using the component of the functor on objects); similarly, the composition $T\eta$ of a functor with a natural transformation is defined to be $(T\eta)(X) = T(\eta X)$ (using the component of the functor on arrows). Second, these two diagrams are diagrams of natural transformations: for every object $X \in \mathcal{C}$, we can "instantiate" these diagrams to a commutative diagram for X in $\mathcal{C}$.

Every monad gives rise to an adjunction. Define the Kleisli category $\mathcal{C}_T$; the objects in $\mathcal{C}_T$ are the objects in $\mathcal{C}$, but an arrow $f^\flat : A \xrightarrow{\flat} B$ in $\mathcal{C}_T$ corresponds to an arrow $f : A \to TB$ in $\mathcal{C}$. The identity arrow $1^\flat_X$ on an object $X \in \mathcal{C}_T$ is given by η_X. Composition of $f^\flat : A \xrightarrow{\flat} B$ and $g^\flat : B \xrightarrow{\flat} C$ must be an arrow $A \xrightarrow{\flat} C$, corresponding to an arrow $A \to TC$ in $\mathcal{C}$, and is given by

$$A \xrightarrow{f} TB \xrightarrow{Tg} T^2C \xrightarrow{\mu_C} TC$$

To define the adjunction, we must find two functors $F : \mathcal{C} \to \mathcal{C}_T$ and $U : \mathcal{C}_T \to \mathcal{C}$:



Define $FX = X$ for all $X \in \mathcal{C}$. For each arrow $f : X \to Y$ in $\mathcal{C}$ we must get an arrow $F(f) : FX \xrightarrow{\flat} FY = X \xrightarrow{\flat} Y$ in $\mathcal{C}_T$, which is an arrow $X \to TY$ in $\mathcal{C}$; it is given by $Ff = Tf \circ \eta_X$. Going the other way, define $UX = TX$. Then for an arrow $f^\flat : X \xrightarrow{\flat} Y$ in $\mathcal{C}_T$, corresponding to an arrow $f : X \to TY$ in $\mathcal{C}$, we must get an arrow $Uf^\flat : U(X) \to U(Y) = TX \to TY$ in $\mathcal{C}$; it is given by $Uf^\flat = \mu_Y \circ Tf$. The unit of the adjunction is the unit of the monad.

Equivalently, a monad (T, η, μ) gives rise to a natural isomorphism

$$\phi : \mathrm{Hom}_{\mathcal{C}_T}(FX, Y) \cong \mathrm{Hom}_{\mathcal{C}}(X, UY)$$

with $\mathcal{C}_T$, F and U as before. It is useful to spell that out. Unfolding the definition of F and U:

$$\phi : \mathrm{Hom}_{\mathcal{C}_T}(X, Y) \cong \mathrm{Hom}_{\mathcal{C}}(X, TY)$$

That is, every function $f^\flat : X \xrightarrow{\flat} Y$ in $\mathcal{C}_T$ is isomorphic to a function $g : X \to TY$ in $\mathcal{C}$. Of course, that is no surprise, since that is how we defined $\mathcal{C}_T$ in the first place; in fact, this isomorphism is an identity: $\phi(g) = U(g) \circ \eta_X = \mu_Y \circ Tg \circ \eta_X = g$:

$$\begin{tikzcd}
& TX \arrow[r, "Tg"] & T^2Y \arrow[dr, "\mu_Y"] & \\
X \arrow[ur, "\eta_X"] \arrow[rrr, bend right, "g"'] & & & TY
\end{tikzcd}$$

To see this, remember that $\eta : 1_\mathcal{C} \to T$ is a natural transformation:

$$\begin{tikzcd}
X \arrow[r, "\eta_X"] \arrow[d, "g"'] & TX \arrow[d, "Tg"] \\
TY \arrow[r, "\eta_{TY}"'] & T^2Y
\end{tikzcd}$$

Therefore $\phi(g) = \mu_Y \circ (Tg \circ \eta_X) = \mu_Y \circ (\eta_{TY} \circ g) = (\mu_Y \circ \eta_{TY}) \circ g = 1_{TY} \circ g = g$ (the first equality comes from the naturality of η, the second from the triangle equalities of the monad).

Related Work

The motivation for and the explanation of the use of substructural type systems for adding side effects to functional languages without losing purity was explained in Chapters 1 and 2, especially p. 11, p. 17
Section 2.8. In this chapter we look at the specific properties of instances of substructural type p. 59
systems: uniqueness typing, linear logic, uniqueness logic and the single threaded polymorphic lambda calculus (STPLC). We also discuss a number of other related type systems, but the subject area is huge and a full survey is beyond the scope of this dissertation. We will mostly ignore type systems targeted at sharing analysis (but see Section 3.2.8) or static analyses like (Shankar, p. 86
2001) which aim to perform destructive updates *when possible* but cannot be used for "real" side effects such as I/O. We will not discuss more theoretical properties such as relational parametricity (Mogelberg and Simpson, 2007; Bierman et al., 2000), denotational semantics (Wadler, 1992, 1994; Benton and Wadler, 1996; Turner and Wadler, 1999) or program equivalence (Bierman, 2000), nor will we discuss applications of uniqueness typing in the context of imperative languages such as in the papers by Boyland (2001) and Fähndrich and DeLine (2002) or the work on Eiffel* (Eiffel extended with support for uniqueness, Minsky, 1996). A comparison between uniqueness typing in Clean and some of these languages can be found in (Ahmed et al., 2005).

We will not discuss each type system in detail, but rather highlight only their most important aspects. In particular, we will generally ignore algebraic data types and recursion, although we will explain the basic issues in the section on uniqueness typing (Section 3.2). p. 78

3.1 An aside on syntax

All systems in this chapter decorate types to distinguish between "restricted" and "unrestricted" types, for some meaning of restricted such as "has been used at most once" (unique), "will be used exactly once" (linear), or "is written to" (in the STPLC). Rather than using the original syntax of these systems, we have tried to use one uniform syntax. We denote a type τ with an attribute ν as τ^ν. In linear or affine type systems (including uniqueness typing) we denote types for which the application of structural rules (contraction or weakening) is restricted (unique, linear or affine types) as $\tau^\bullet$, and types for which the structural rules can arbitrarily be applied as $\tau^\times$.

Although we hope that a uniform syntax makes this chapter more readable and makes it easier to compare the various systems, it must be pointed out that the chosen syntax can have profound influences on the design of a type system. For example, in the original presentation of linear logic types do not have an attribute. Instead, `Int` is the type of linear (restricted) integers and `!Int` is the type of non-linear (unrestricted) integers (we would denote these two types as $\texttt{Int}^\bullet$ and $\texttt{Int}^\times$, respectively). Likewise, $a \longrightarrow b$ is the type of linear functions from a to b, and $!(a \longrightarrow b)$ is the type of non-linear functions from a to b (here denoted as $a \xrightarrow{\bullet} b$ and $a \xrightarrow{\times} b$).

Although this may seem like a simple matter of syntax, using "!"-style syntax suggests that it is necessary to "unwrap" a non-linear type before it can be used. For example, if f has type $!(a \longrightarrow b)$, that is, a function wrapped in a !-constructor, before we can apply it we must apply a typing rule[1] to unwrap the function to obtain an element of type $a \longrightarrow b$. However, that means that we must have a way to go from a non-linear type to a linear type—and we may not want such a rule (this is in fact the major difference between uniqueness typing and linear logic, as discussed below). Moreover, the "!"-style syntax makes it difficult to see how to denote types with polymorphic linearity, whereas with attributes this can simply be denoted by $\forall u \cdot \texttt{Int}^u$.

Thus, it may be argued that presenting linear logic without the "!"-constructor but using attributes somehow misrepresents the type system, especially if one is interested in the evolution of these systems. Nevertheless, as our primary goal in this chapter is the comparison of these systems, we feel that a uniform syntax is to be preferred.

Finally, a note on the translation from "!"-syntax to the syntax using attributes: when a typing rule in "!"-style mentions a type τ, for some meta-variable τ, this means that we can substitute *any* type for τ. Hence, we will denote this type as τ^ν in this chapter, to emphasize that both the base type and its attribute are arbitrary. (This can be simplified if we treat both types and attributes p. 143 as types of special kinds, but this will be the subject of Chapter 6.) Similarly, a type such as $!\tau$ for a meta-variable τ will be presented here as $\tau^\times$.

3.2 Uniqueness typing

The type system we will develop in this dissertation is directly based on the uniqueness type system of the programming language *Clean* (van Bakel et al., 1992; Smetsers et al., 1994; Barendsen and Smetsers, 1993b, 1995b,a, 1996). Nevertheless, of all the type systems considered in this chapter, the original development of Clean's type system has a distinctly different feel, as it was developed as a type system for a (graph) rewriting system rather than the lambda calculus. This slightly different perspective results in some subtle differences in the typing rules.

For example, at the time of writing, the paper that is probably the best (formal) reference p. 79 for Clean's type system is (Barendsen and Smetsers, 1996); Figure 3.1 shows the typing rules. The term rewriting background of the system is clearly visible in these rules: no rule is given for lambda abstraction. Instead, two special rules are given for currying functions (CURRY) and applying curried functions (CAPP). Although we have tried to keep the syntax as uniform as possible for the type systems in this chapter, we have not attempted to give a "lambda calculus" presentation of Clean's type system here. Indeed, such a presentation was one of the original motivations for our work, and we will present such systems in later chapters.

In the rest of this section we will explain the type system using a series of examples designed to expose the various aspects of the system. This will also set us up for the rest of this chapter, because we will reconsider many of these examples in the context of alternative type systems.

3.2.1 Introduction

Every type in *Clean* comes in a unique version and a non-unique version. Elements of a unique type are guaranteed never to be shared and can hence be updated in place without losing definiteness p. 59 (see Section 2.8 for a more detailed discussion of this idea).

[1] In some cases, the system even requires a special construction at the term level.

Typing rules

$$\frac{}{x : \tau^{\nu} \vdash x : \tau^{\nu}}\ \textsc{Var}$$

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{C} : \overline{\tau'^{\nu'}} \rhd \tau^{\nu} \qquad \Gamma_i \vdash e_i : \tau_i'^{\nu_i'}}{\overline{\Gamma} \vdash \mathbf{C}\overline{e} : \tau^{\nu}}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : \tau^{\nu} \qquad \tau^{\nu} \leq \tau'^{\nu'}}{\Gamma \vdash e : \tau'^{\nu'}}\ \textsc{Sub}$$

$$\frac{\Gamma : e : \tau^{\nu}}{\Gamma, x : \tau'^{\nu'} \vdash e : \tau^{\nu}}\ \textsc{Weak} \qquad \frac{\Gamma, y : \llcorner \tau'^{\nu'} \lrcorner, z : \llcorner \tau'^{\nu'} \lrcorner \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\nu'} \vdash e[y := x, z := x] : \tau^{\nu}}\ \textsc{Contr}$$

$$\frac{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_{k+1} : (\overline{\tau^{\nu}}, \tau'^{\nu'}) \rhd \tau''^{\nu''} \qquad \nu_f \leq \Pi\overline{\nu}}{\mathcal{F}, \mathcal{A} \vdash \mathbf{F}_k : \overline{\tau^{\nu}} \rhd (\tau'^{\nu'} \xrightarrow{\nu_f} \tau''^{\nu''})}\ \textsc{Curry}$$

$$\frac{\Gamma \vdash e : \tau^{\nu} \xrightarrow{\nu_f} \tau'^{\nu'} \qquad \Delta \vdash e' : \tau^{\nu}}{\Gamma, \Delta \vdash \mathbf{Ap}(e, e') : \tau'^{\nu'}}\ \textsc{CApp}$$

$\Pi\overline{u}$ is • if any of $\overline{u}$ is.

Subtyping relation

$$t^{\nu} \leq t^{\nu'} \Leftrightarrow \nu = \nu' \qquad \text{(type variable } t\text{)}$$

$$\mathbf{C}^{\nu_a}\overline{\tau^{\nu}} \leq \mathbf{C}^{\nu_b}\overline{\tau'^{\nu'}} \Leftrightarrow \nu_a \leq \nu_b \text{ and } \overline{\tau^{\nu}} \leq^{\mathrm{sign}(\mathbf{C})} \overline{\tau'^{\nu'}} \qquad \text{(type constructor } \mathbf{C}\text{)}$$

$$\tau_a^{\nu_a} \xrightarrow{\nu} \tau_b^{\nu_b} \leq \tau_c^{\nu_c} \xrightarrow{\nu'} \tau_d^{\nu_d} \Leftrightarrow \nu = \nu' \text{ and } \tau_a^{\nu_a} \leq^{\ominus} \tau_c^{\nu_c} \text{ and } \tau_b^{\nu_b} \leq^{\oplus} \tau_d^{\nu_d}$$

where $\mathrm{sign}(\mathbf{T})$ denotes the variance of $\mathbf{T}$ (covariant $\oplus$, contravariant $\ominus$, or invariant). For details see (Barendsen and Smetsers, 1996).

Uniqueness correction

Change the top-level attribute of a type to be non-unique. Note that $\llcorner \cdot \lrcorner$ is a partial function; in particular, $\llcorner t^{\bullet} \lrcorner$ (for some type variable t) and $\llcorner \tau_a^{\nu_a} \xrightarrow{\bullet} \tau_b^{\nu_b} \lrcorner$ are undefined.

$$\llcorner t^{\times} \lrcorner = t^{\times}$$

$$\llcorner \mathbf{C}^{u}\overline{\tau^{\nu}} \lrcorner = \mathbf{C}^{\times}\overline{\tau^{\nu}}$$

$$\llcorner S \xrightarrow{\times} T \lrcorner = S \xrightarrow{\times} T$$

Figure 3.1: Uniqueness typing rules (Barendsen and Smetsers, 1996) (see comments in text)

For example, *Clean* supports arrays which, unlike lists, can be accessed and updated in $O(1)$ time. The function that overwrites an element at some index in an array of characters has type

```
cUpdate :: {Char}• Int Char → {Char}•
```

(`{Char}` is Clean syntax for an array of characters; in the ASCII syntax for Clean, `{Char}`$^\bullet$ is written as `*{Char}`.) As mentioned before, functions in Clean can have more than one argument; currying is treated separately (see Section 3.2.4). p. 82

Likewise, a Clean *program* is a function from an old to a new world state:

```
Start :: World• → World•
```

For example, here is a simple Clean program to output "Hello world" to the console:

```
Start :: World• → World•
Start world = let (console, world')   = stdio world
                  console'            = fwrites "Hello world" console
                  (success, world'') = fclose console' world'
              in world''
```

This program uses the following functions from the Clean standard library, which open the console, write a string to a file handle, and close a file handle, respectively.

```
stdio   :: World• → (File•, World•)
fwrites :: {Char} File• → File•
fclose  :: File• World• → (Bool, World•)
```

The Clean syntax has various conventions for denoting common patterns of types with uniqueness attributes, and a special scoping rule to avoid having to invent a new name for each new `world` object. These conventions make code easier to write, but to the inexperienced Clean programmer will obscure the details. Hence, we will not use any of these conventions in this dissertation, with the exception of one: in the context of Clean types, a type without an attribute, such as `Int`, denotes the *non-unique* version of the type (`Int`$^\times$). Thus, the type of `fwrites` above is really

```
fwrites :: {Char}× File• ×→ File•
```

That is, `fwrites` is a non-unique function which takes a non-unique string (character array) and a unique file and returns a unique file.

3.2.2 Subtyping

A term of a unique type is guaranteed not to be shared, whereas no such guarantees are given for terms of a non-unique type. However, suppose a function (such as `fwrites`, above) expects a non-unique string. That is, it expects a string for which no sharing guarantees are given; should we be able to pass a unique string to `fwrites`? Intuitively, the answer should be affirmative; the additional guarantees about the string are not relevant to the operation of `fwrites`, but neither are they harmful.

For this reason Clean introduces a subtyping relation, where a unique type is considered a subtype of its non-unique counterpart. Hence, we will be able to pass a unique string to `fwrites`, or indeed write a function that duplicates unique arrays:

```
dupArray :: {Char}• → ({Char}×, {Char}×)
dupArray arr = (arr, arr)
```

Only elements of a non-unique type can be duplicated, so in the body of dupArray we are expecting a non-unique array. Due to subtyping, however, arr can be treated as if its type is a non-unique array. The subtyping relation is not uniform, however. In particular, a unique *function* cannot be treated as a non-unique function. Hence, it is impossible to write a function

```
// Illegal example
dupFn :: (Int •→ Int) → (Int ×→ Int, Int ×→ Int)
dupFn fn = (fn, fn)
```

It is therefore also impossible to write a generic "duplication" function of type

```
// Illegal example
dupAny :: a• → (a×, a×)
dupAny x = (x, x)
```

We *can* write this function, but its type is different:

```
dup :: a× → (a×, a×)
dup x = (x, x)
```

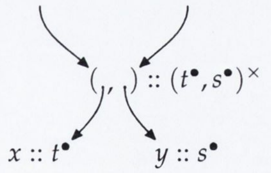
Due to Clean's subtyping relation, dup can be used to duplicate unique arrays, but cannot be used to duplicate unique functions. The reason for this non-uniformity will be given in the section on partial application (Section 3.2.4).

3.2.3 Uniqueness propagation and polymorphism

Uniqueness types often have constraints (implications between uniqueness attributes) associated with them. In *Clean*, for example, *fst* (the function which returns the first element of a pair) has type

fst :: $(t^u, s^v)^w \to t^u, [w \le u, w \le v]$

The type uses *uniqueness variables* to denote unknown uniqueness attributes, just like type variables are used to denote unknown types. The constraint $[w \le u]$ denotes that w must be unique if u is (u implies w)[1]. To understand the need for this constraint, suppose we have a pair with elements $x :: t$ and $y :: s$. The only references to these elements are from this pair, so t and s get a unique (•) attribute. Further, suppose that there are two references to the pair, making the attribute of the type of the pair non-unique (×). Visually:



If we could extract a unique element from a non-unique pair, we could extract x from the pair and modify it. But then the value of x as seen through the second reference will also change, and referential transparency is lost. So, we can only extract a unique element from a container if the container is unique itself $(w \le u)$. In section 3.6.3 on Mercury (which does not currently enforce p. 108
this), we will give an example program that duplicates arbitrary unique objects by extracting unique elements from non-unique pairs.

[1] Perhaps the choice of the symbol $\le$ is unfortunate. In logic $a \le b$ denotes a implies b, whereas here $u \le v$ denotes v implies u. Usage here conforms to *Clean* conventions.

3.2.4 Partial application

The problem of partial application is probably the most subtle in the design of substructural type systems, and it is important that we understand it fully. Consider the function that returns the first of its two arguments:

```
const = λx → λy → x
```

Temporarily ignoring the attributes on arrows, *const* has type

const :: $t^u \rightarrow s^v \rightarrow t^u$

Given *const*, what would be the type of

```
sneakyDup = λx → let f = const x
                 in (f 1, f 2)
```

It would seem that since f has type $s^v \rightarrow t^u$, this term has type

sneakyDup :: $t^u \rightarrow (t^u, t^u)$

but this is clearly wrong: the elements in the result pair are shared, so the attribute on their types must be non-unique.

Recall from the previous section that if we want to extract a unique element from a container, the container must be unique itself. When we execute a function, the function can extract elements from its closure (the environment which binds the free variables in the function body). If any of those elements is unique, executing the function will involve extracting unique elements from a container (the closure), which must therefore be unique itself. Since we do not distinguish between a function and its closure in the lambda calculus, this means that the function must be unique. Thus a function needs to be unique on application (that is, a function can be applied only once) if the function can access unique elements from its closure.

There are various ways in which this problem can be solved and we will propose two new ways in Chapters 4 and 6. p. 115, p. 143 The solution adopted in Clean is that the result of a partial application is a *unique* function if the supplied argument was unique. Moreover, the subtyping relation is defined such that a unique function is not considered a subtype of a non-unique function, and therefore must *remain* unique. We say that the function is *necessarily* unique or *essentially* unique.

In the example above, `f` is duplicated (used twice), so that its type must be non-unique. Hence, the argument supplied to `const` had to be non-unique, and the type of `sneakyDup` is therefore the same as the type of `dup`, as expected:

```
sneakyDup :: t× → (t×, t×)
```

p. 79 In the typing rules (Figure 3.1) this is formalized in rule CURRY and the definition of uniqueness correction[1]. CURRY says that if we partially apply a function the result is a unique function if any of the supplied arguments is unique. Then the rule for contraction (CONTR), which duplicates a variable so that it can be used more than once, uses a uniqueness correction on the type of the variable to make it non-unique. This correction however is partial, and will fail on unique functions.

[1] Note that Barendsen and Smetsers use $[\cdot]$ instead of $\llcorner\cdot\lrcorner$ to denote uniqueness correction; we do not use this notation to avoid confusion with the syntax for lists.

3.2.5 Recursion

A recursively defined object must be non-unique. To see why, consider the following definition of an infinite list of 1s:

```
ones = [1 : ones]
```

Suppose that we would allow the type for `ones` to be a unique list of integers, $[\texttt{Int}]^\bullet$. Then we would be able to pass `ones` as argument to a function that changes the head of a list in-place:

`update_head ::` $[t^u]^\bullet \to t^u \to [t^u]^\bullet$

Since `update_head` only modifies the *head* of a list, we would expect the result of

```
update_head ones 2
```

to be the (infinite) list `[2, 1, 1, 1, ...]`. However, since `ones` is recursively defined, the result would actually be `[2, 2, 2, 2, ...]`. In fact, there is more than one reference to `ones` in `update_head ones 2`; one from the call to `update_head`, and one from within `ones` itself. Hence, a recursively defined object must always be non-unique[1].

It is possible to define a fixpoint operator in Clean, for instance using a recursive data type (Section 2.9.1) or simply using term-level recursion as p. 70

`y ::` $(t^u \xrightarrow{\times} t^u) \xrightarrow{\times} t^u$
`y f = f (y f)`

The type of `y` may seem slightly worrying, since it does not require the generated object to be non-unique. In particular, we can define

`ones' ::` $[\texttt{Int}]^\bullet$
`ones' = y (`λ`tail` $\to$ `[1:tail])`

However, `ones` and `ones'` are not the same object. In `ones`, the tail of the list *is* `ones`, whereas in `ones'` the tail is a function application which when evaluated will produce the next element of the list. Hence, the list nodes themselves will not be shared in `ones'` and it is sound to give `ones'` a unique type.[2]

3.2.6 Read-only access

One of the challenges that type systems such as uniqueness typing and linear logic face is to support multiple read-only accesses before a write access. The canonical example is the definition of a function `swap` which swaps two elements in an array. It can be defined in Clean as

`swap ::` $\{\texttt{a}\}^\bullet$ `Int Int` $\to$ $\{\texttt{a}\}^\bullet$

```
swap arr i j
    #! a = select arr i
       b = select arr j
    = update (update arr i b) j a
```

[1] Clean does in fact allow to give a type annotation to `ones` to make it a unique list of integers, $[\texttt{Int}]^\bullet$. However, this changes the semantics of `ones`: it now becomes a function that returns an infinite list of 1s, much like `ones'`. Thanks to John van Groningen for pointing this out.

[2] Note that since the argument to `y` must be a non-unique function, it cannot contain any unique elements in its closure (Section 3.2.4). Hence, we cannot use `y` to create a list with "shared but unique" elements.

This example is type correct and modifies a (unique) array in-place, even though there are three references to `arr` in the body of `swap`. The reason that the program is nevertheless accepted is that the two `select` operations must be evaluated before the calls to `update` (the `#!` syntax denotes strict evaluation). Since `select` only inspects the array, and since the calls to `select` are guaranteed to be evaluated before the call to `update`, we can allow such definitions without sacrificing properties such as definiteness (Section 2.8).

p. 59

We have to be careful however: it is imperative to avoid aliasing. For example, consider[1]

```
strictDup :: {#Char}• -> ({#Char}•, {#Char}×)•
strictDup arr
    #! arr` = arr
    = (arr, arr`)
```

This is yet another variant of `dup`, but one that should be rejected: the type of `strictDup` tells us that it returns one unique and one non-unique array; but since both arrays are the same, they should never be given a unique type. The problem is that although access to `arr` in the `#!` line is indeed read-only, it also creates an alias to `arr`. To make the problem even more obvious, if we do not prohibit aliasing we could define

```
disaster :: {#Char}• -> ({#Char}•, {#Char})•
disaster arr
    #! arr` = arr
    = (update arr 0 'a', arr`)
```

Since `arr` and `arr'` are both references to the same array, the result of `disaster "xyz"` will be `("ayz", "ayz")`, and we have lost definiteness.

This problem is difficult to avoid, and both uniqueness typing and linear logic use a heuristic: severe restrictions are placed on the types of the variables that are assigned in the read-only accesses: essentially, only basic types such as integers are allowed (Plasmeijer and van Eekelen, 2002, Section 9.4). This is however difficult to get right; in fact, it turns out that the Clean compiler accepts `strictDup` and `disaster` (de Vries, 2007)! This bug has since been solved, but it illustrates the difficulty of the problem.

Unfortunately, it seems that a more principled solution to this problem is not possible without making the system significantly more complicated. We will discuss one proposed solution (in the context of linear logic) in Section 3.3.3. The single threaded polymorphic lambda calculus also solves this problem to some extent (Section 3.5.3). Finally, we will briefly come back to this issue when we list future work (Section 8.2.9).

p. 90
p. 100
p. 221

3.2.7 Reference count analysis[2]

As discussed in the section on substructural logics in the previous chapter (Section 2.6), we must be careful to split the typing environment in rules such as `App` with more than one premise. This is however a non-deterministic operation: when the type inferencer, which is trying to find a proof that a term is well-typed, needs to apply a rule such as `App` (bottom-up), it does not know how to split the typing environment.

p. 52

[1] `{#Char}` is the type of arrays of "unboxed" characters; that is, actual characters as opposed to pointers to thunks of code for yet-to-be-evaluated characters.

[2] Reference count analysis is sometimes also referred to as *sharing analysis*; however, we want to reserve this term for a different analysis, discussed in Section 3.2.8.

One solution is to annotate the *terms* with local sharing information. Every use of a variable x in a term is annotated as $x^{\otimes}$ if it occurs more than once in its range (or if it is recursively defined) and as $x^{\odot}$ otherwise. We consider a few examples. The identity function is marked as

$\texttt{id x = x}^{\odot}$

since there is only one reference to x in the body of *id*. Note that even when a variable is used only once, that does not automatically make its type unique: when a shared term is passed to `id`, it will still be shared when it is returned from `id`.

In the definition of *dup*, however, there are two references to the same variable, which must therefore be marked as shared:

$\texttt{dup x = (x}^{\otimes}\texttt{, x}^{\otimes}\texttt{)}$

The reference count analysis does not make a distinction between variables that correspond to functions and variables that correspond to function arguments. For example, the function *twice* is marked as

$\texttt{twice f x = f}^{\otimes}\texttt{ (f}^{\otimes}\texttt{ x}^{\odot}\texttt{)}$

We can now redefine the typing rules over annotated terms. We no longer need to split typing environments, nor do we need explicit structural rules. In fact, the type system starts to look like an ordinary structural type system, which makes type inference much easier. Of course, the rule for variables must now be split into two rules: one for variables marked as $\otimes$ (for which we need to apply the uniqueness correction shown in Figure 3.1) and one for variables marked p. 79 as $\odot$ (for which we do not). We do not show the redefined typing rules here, because they do not illustrate any new properties of the type system (see instead (Barendsen and Smetsers, 1996, Section *Uniqueness Typing Inference*), which also contains a theorem that the two presentations are equivalent). We will however make extensive use of this style of presentation in the chapters to come.

An alternative solution proposed by Hage et al. (see next section; personal communication with Stefan Holdermans) is to pass the same environment to both branches of a typing derivation, but keep track of which variables in the typing environment are referenced. When typing an application $(f\ e)$, we type both f and e in an environment Γ; variables that are used in both f and e will be corrected *when typechecking the application* to have a non-unique type. This approach only works when the type of the variable at the usage site cannot differ from the type of the variable at the binding site. For example, in a type system where variables can lose their uniqueness (such as the one we propose in Chapter 4), we can define a function `dup` of type p. 115

$\texttt{dup :: } t^{u} \longrightarrow (t^{\times}, t^{\times})^{v}$

`dup x = (x, x)`

In such a type system, we cannot correct the type of x when we typecheck the application (of the pair constructor), since the type of x in the typing environment has not changed, even if the type of x at the usage site is different. A similar problem occurs in a type system where variables cannot lose their uniqueness, but with support for first-class uniqueness polymorphism (Chapter 6): p. 143

$\texttt{dup :: } (\forall u.t^{u}) \longrightarrow (t^{\times}, t^{\times})^{v}$

`dup x = (x, x)`

The alternative approach to reference count analysis is feasible in Hage et al. (2007) because variables cannot loose their uniqueness in that system and the type of the second `dup` variation cannot even be represented (see next section).

3.2.8 Uniqueness typing and sharing analysis

Call-by-need languages such as Haskell are often implemented by updatable *thunks*. A thunk is a data structure in memory that represents a suspended computation. For example, in

```
let x = f 1 ; y = f 2 in y + x
```

two thunks are created for $(f\ 1)$ and $(f\ 2)$; no computation takes place until the result of $(f\ 2)$ is required by the addition. Once evaluated, a thunk is updated with the result of the computation to avoid needless future re-evaluation. This is of course only possible in pure languages where the p. 59 result of evaluation depends solely on the expression and not on global state (Section 2.8). See (Harris and Singh, 2007, section 2) for a good introduction to thunks.

According to Marlow (1993), however, 70% of the time the result of a thunk is only required once and updating the thunk is unnecessary. *Sharing analysis* (Turner et al., 1995; Wansbrough and Peyton Jones, 1999) attempts to find out which thunks do and do not need to be updated. Like uniqueness typing, sharing analysis therefore distinguishes between expressions that are used only once and expressions that are used more than once. For example, the function `double` is assigned the type

$$\texttt{double :: Int}^{\bullet} \xrightarrow{\times} \texttt{Int}^{\times}$$
$$\texttt{double x = 2 * x}$$

The annotation on the domain of the function means that `double` will evaluate its argument at most once. However, passing an argument that is used more than once will not cause any problems:

```
let x = .. in (double x) * x
```

Thus, there is a coercion

$$\texttt{coerce ::}\ t^{\times} \rightarrow t^{\bullet}$$

Conversely, given

$$\texttt{square :: Int}^{\times} \xrightarrow{\times} \texttt{Int}^{\times}$$
$$\texttt{square x = x * x}$$

if we pass an expression of type `Int`$^\bullet$, that is, an argument for which we have inferred that it will be accessed only once, then this argument will be re-evaluated twice within `square`; worse, it might be that the argument thunk will be garbage collected after the first use resulting in an abnormal termination of the program. Thus, we do *not* want to allow a coercion of the form

$$\texttt{invalid\_coerce ::}\ t^{\bullet} \rightarrow t^{\times}$$

Hage, Holdermans, and Middelkoop (2007) observe that this subtyping behaviour is dual to that p. 80 of uniqueness typing (Section 3.2.2). They present a type system which is parametrized by the direction of the subtyping relation and encompasses both uniqueness typing and sharing analysis.[1]

The authors present two type systems, an explicitly typed "target" language in the style of p. 35, p. 41 System F (Section 2.4) and an implicitly typed Hindley-Milner style (Section 2.5.1) "source" p. 72 language. Both systems are presented using the qualified types framework (Section 2.9.2) to deal with inequality constraints between uniqueness attributes. Like Clean's type system, the system supports both uniqueness polymorphism (referred to as *polyvariance*) and subtyping.

[1] Linear logic is dual to uniqueness typing in the same way as sharing analysis and one might argue that sharing analysis and linear logic are two sides of the same coin. For example, we will discuss a linear logic (Turner et al., 1995) in Section 3.3 that is also often cited as a paper on sharing analysis.

To the best of our knowledge, Hage et al.'s type system is the only uniqueness system supporting first class polymorphism other than the one we define in this dissertation (although an *inference* algorithm is only given for the Hindley/Milner fragment). Regrettably, the type system as presented does not guarantee that unique elements are not shared, as it does not deal with the partial application problem (Section 3.2.4). The authors acknowledge the problem, and mention p. 82 that the solution we proposed in an earlier paper (de Vries et al., 2007b, or Chapter 4 in this p. 115 dissertation) might solve the problem; they adopt this approach in (Hage and Holdermans, 2008).

Although the target language supports first-class polymorphism, the top-level attribute of a type cannot be universally quantified (Wansbrough and Peyton Jones (1999) adopt the same approach). The typing relation takes the form[1]

$$P \,|\, \Gamma \vdash e : \sigma^u$$

for some type σ, which can be a type scheme. This means that we can infer

$$(\lambda x \cdot x) : (\forall t\, u \cdot t^u \to t^u)^\times$$

However, the *body* of the type scheme (the type on the dots in $\forall a \cdot \ldots$) must be a type without an attribute. We will argue later in this dissertation that the type

$$(\lambda x \cdot x) : \forall t\, u \cdot (t^u \to t^u)^\times$$

is more intuitive (and will enforce this structure using a kind system; Chapter 6); but even if p. 143 the first type seems reasonable, this representation does have an important disadvantage: since the assumptions in the typing environment take the same form ($x : \sigma^u$), top-level uniqueness attributes on assumptions cannot be universally quantified and hence

$$\lambda(x :: \forall u.\mathtt{Int}^u) \cdot (x, x)$$

cannot be typed (indeed, the term itself cannot even be represented). Depending on whether the language supports subtyping or not, terms of this shape may be useful (Section 6.3). p. 147

3.3 Linear logic

Linear logic was introduced by Jean-Yves Girard (Girard, 1987, 1991); its use as a type system was pioneered by Philip Wadler. It is probably both the most well-known substructural logic and the most well-known substructural type system, and many people seem to equate substructural logic and linear logic. As O'Hearn and Pym (1999) put it:

> “ Perhaps the most lasting impression of linear logic, much more than the formal system itself, will be its revealing of the computational significance of the structural rules of Weakening and Contraction.

Indeed, an often heard comment at conferences after a presentation on uniqueness typing is: “is this not just linear logic?” Before presenting some linear type systems, we will address this question first.

[1] This relation is denoted as $\Gamma \vdash e :^u \sigma$ in the paper, where no distinction is made between the typing environment and the predicate environment. See also Section 2.9.2 on qualified types.

3.3.1 Linear logic versus uniqueness typing

Like uniqueness typing, linear typing distinguishes between two classes of types: linear types and non-linear types. Like terms of unique types, terms of linear types cannot be duplicated. One relatively minor difference is that linear type systems also restrict *disregarding* elements of a linear type (in that respect, uniqueness typing is closer to affine logic than to linear logic).

However, there is a more important difference. In its original form, linear logic was designed for a very different application domain than uniqueness typing. Rather than reasoning about sharing, linear logic reasons about "supply" and "consumption". Linear types model resources that can be used (consumed) only once; non-linear types model resources of which there is an infinite supply. For example, a "linear" banknote can be regarded as a single banknote (and can be expended only once), whereas a "non-linear" banknote can be regarded as an infinite stack of banknotes. From this perspective, it is reasonable to allow the terms

$$\begin{aligned} & \lambda x \cdot x :: t^{\times} \xrightarrow{\bullet} t^{\bullet} \\ \text{or} \quad & \lambda x \cdot (x, x) :: t^{\times} \xrightarrow{\bullet} (t^{\bullet}, t^{\bullet})^{\bullet} \end{aligned}$$

To continue the analogy, these terms peel off one and two banknotes from our infinite stack. This coercion from a non-linear type to a linear type is known as *dereliction* in linear logic.

From a uniqueness perspective, however, we certainly do not want to admit these terms. The first changes a term for which there are no sharing guarantees into a term that is guaranteed not to be shared; worse, the second duplicates a term and guarantees that the two terms in the result will be unique (guaranteed not to be shared). While we may want to allow these operations for some types such as arrays (where we can implement the term by copying the entire array), we certainly do not want to allow this operation for all types (especially in the case of the `world` state).

Indeed, in a uniqueness type system we may want to allow

$$\lambda x \cdot x :: t^{\bullet} \xrightarrow{\bullet} t^{\times}$$

From the perspective of uniqueness typing this term simply loses the uniqueness guarantee.[1] When considered from the perspective of linear logic, however, this term should definitely not be allowed: it changes a single bank note into an infinite supply. Harrington (2001, Chapter 5, *Conclusions*) phrases it well: in linear logic, "linear" means "will not be duplicated" whereas in uniqueness typing, "unique" means "has not been duplicated". In a paper on a type system based on linear logic, Wadler (1991) argues

> Does this mean that linearity is useless for practical purposes? Not completely. Dereliction means we cannot guarantee *a priori* that a variable of linear type has exactly one pointer to it. But if we know this by other means, then linearity guarantees that the pointer will not be duplicated or discarded.

Nevertheless, we consider this a serious weakness of the use of linear logic for reasoning about sharing. In fact, Wadler appears to agree: three out of the four type systems he proposes do not include dereliction. These type systems are therefore closer in spirit to uniqueness typing than to linear logic, although no mention of uniqueness typing is made in any of the papers.

[1] We must however be careful with partially applied functions; see Section 3.2.4.

$$\frac{}{x : \tau^{\nu} \vdash x : \tau^{\nu}}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau^{\nu} \vdash e : \tau'^{\nu'}}{\Gamma \vdash \text{¡}\lambda x \cdot e : \tau^{\nu} \xrightarrow{\bullet} \tau'^{\nu'}}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau^{\nu} \xrightarrow{\bullet} \tau'^{\nu'} \qquad \Delta \vdash e_2 : \tau^{\nu}}{\Gamma, \Delta \vdash (\text{¡}e_1\, e_2) : \tau'^{\nu'}}\ \textsc{App}$$

$$\frac{\Gamma, x : \tau^{\nu} \vdash e : \tau'^{\nu'} \qquad \text{non-linear } \Gamma}{\Gamma \vdash \lambda x \cdot e : \tau^{\nu} \xrightarrow{\times} \tau'^{\nu'}}\ \textsc{Abs}^{\times} \qquad \frac{\Gamma \vdash e_1 : \tau^{\nu} \xrightarrow{\times} \tau'^{\nu'} \qquad \Delta \vdash e_2 : \tau^{\nu}}{\Gamma, \Delta \vdash (e_1\, e_2) : \tau'^{\nu'}}\ \textsc{App}^{\times}$$

$$\frac{\Gamma \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\ \textsc{Weak} \qquad \frac{\Gamma, x : \tau'^{\times}, x : \tau'^{\times} \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\ \textsc{Contr}$$

Figure 3.2: Typing rules from (Wadler, 1990)

$$\frac{}{x : \tau^{\nu} \vdash x : \tau^{\nu}}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau^{\nu} \vdash e : \tau'^{\nu'}}{\Gamma \vdash \lambda x \cdot e : \tau^{\nu} \xrightarrow{\bullet} \tau'^{\nu'}}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau^{\nu} \xrightarrow{\bullet} \tau'^{\nu'} \qquad \Delta \vdash e_2 : \tau^{\nu}}{\Gamma, \Delta \vdash e_1\, e_2 : \tau'^{\nu'}}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\ \textsc{Weak} \qquad \frac{\Gamma, x : \tau'^{\times}, x : \tau'^{\times} \vdash e : t^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\ \textsc{Contr}$$

$$\frac{\Gamma \vdash e : \tau^{\nu} \qquad \text{non-linear } \Gamma}{\Gamma \vdash e : \tau^{\times}}\ \textsc{Promotion} \qquad \frac{\Gamma, x : \tau'^{\nu'} \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\ \textsc{Dereliction}$$

Figure 3.3: Typing rules from (Wadler, 1991, Section 3)

3.3.2 Wadler's type systems

In the type system presented in Wadler's first paper (Wadler, 1990), the linear and non-linear types are kept completely separate, *even on the term level.* There is a different syntax for defining linear functions ($\text{¡}\lambda x \cdot e$) and non-linear functions ($\lambda x \cdot e$), and likewise there is a different syntax for applying linear functions ($\text{¡}e_1\, e_2$) and applying non-linear functions ($e_1\, e_2$).

The typing rules, presented in Figure 3.2, are based on the simply typed lambda calculus where the rules for contraction and weakening (see Section 2.6) apply to non-linear types only. p. 52
Linear and non-linear types are kept completely separate: coercion from a non-linear type to a linear type (dereliction) is not allowed (nor is coercion in the other direction).

Since the type system distinguishes *syntactically* between linear and non-linear functions, we need separate typing rules for linear and non-linear functions. However, rules App$^{\bullet}$ and App$^{\times}$ are identical, and the only difference between rules Abs$^{\bullet}$ and Abs$^{\times}$ is that non-linear functions cannot contain any linear elements in their closure (side-condition "non-linear Γ"; see Section 3.2.4 for a discussion on the necessity of this condition). p. 82

In his next paper, Wadler (1991) presents a series of type systems, starting with an earlier linear type system presented by Abramsky (1993) (see also Mackie, 1994). This type system is based directly on linear logic (including dereliction) and is shown in Figure 3.3.

Wadler proceeds to give a version of the typing rules that supports "use variables" and constraints (inequalities) between them. These can be compared to uniqueness variables and constraints between uniqueness variables (see Section 3.2), except that in the original presentation p. 78
the syntax is rather awkward; since Wadler uses the !–style syntax (see Section 3.1), he is led to p. 77
put the use variable on the !. That is, $\texttt{Int}^{u}$ would be denoted by $!^{u}\texttt{Int}$.

$$\frac{}{x : \tau^{\nu} \vdash x : \tau^{\nu}\ []}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau^{\nu} \vdash e : \tau'^{\nu'}\ [\mathcal{C}]}{\Gamma \vdash \lambda x \cdot e : \tau^{\nu} \xrightarrow{\nu_f} \tau'^{\nu'}\ [\mathcal{C} \cup \{\nu_f \leq \nu_c \mid \tau_c^{\nu_c} \in \Gamma\}]}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau^{\nu} \xrightarrow{\nu_f} \tau'^{\nu'} \qquad \Delta \vdash e_2 : \tau^{\nu}}{\Gamma, \Delta \vdash e_1\, e_2 : \tau'^{\nu'}}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\ \textsc{Weak} \qquad \frac{\Gamma, x : \tau'^{\times}, x : \tau'^{\times} \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\ \textsc{Contr}$$

Figure 3.4: “Steadfast” typing rules from (Wadler, 1991, Section 7)

The next variation presented uses “standard types” (a canonical form of types) and presents the typing rules with rules DERELICTION and PROMOTION “embedded” in the other rules. Neither of these changes affect the properties of the type system in any fundamental way and are irrelevant for our current purposes.

Finally, he presents a version with “steadfast” linear types that cannot be coerced to non-linear types (i.e., dereliction is removed). These rules are shown in Figure 3.4.

If the steadfast type system looks like uniqueness typing, the system presented by Turner, Wadler, and Mossin (1995) is even more similar. It it almost identical to the steadfast type system, except that weakening is now unrestricted. In other words, variables of a linear type no longer *have* to be used, but if they are used they can be used at most once. The only feature distinguishing uniqueness typing from (Turner et al., 1995) is the absence of a coercion from linear (unique) types to non-linear (non-unique types) in the latter.

For a further discussion of linear type systems and their relation to the purity question we
p. 59 considered in Section 2.8 the reader might like to refer to (Cooper, 1997), but note that Cooper claims (Chapter 6) that the steadfast typing rules do *not* guarantee that terms of a linear type are not shared. He creates the following term:

use_twice :: $(t^u \xrightarrow{\bullet} s^v) \xrightarrow{\bullet} (t^u \xrightarrow{\bullet} r^w) \xrightarrow{\bullet} (\mathbb{1}^{\times} \xrightarrow{\times} t^u) \xrightarrow{\bullet} (s^v, r^w)^{u_p}$
use_twice f g x = (f (x $\perp$), g (x $\perp$))

Since we can instantiate this type to

use_twice :: $(t^{\bullet} \xrightarrow{\bullet} s^v) \xrightarrow{\bullet} (t^{\bullet} \xrightarrow{\bullet} r^w) \xrightarrow{\bullet} (\mathbb{1}^{\times} \xrightarrow{\times} t^{\bullet}) \xrightarrow{\bullet} (s^v, r^w)^{u_p}$

Cooper concludes that this means that we can use `use_twice` to use unique elements twice. This conclusion however is not justified. *If* we pass a function (x) to `use_twice` that can return a unique element whenever it is applied to an element of type unit, then indeed we can invoke that function twice to get two unique elements. However, such a function cannot be defined in the core calculus: the elements in the closure of a non-linear function must be non-linear (see also the
p. 82 discussion of partial application in Section 3.2.4).

3.3.3 Observable linear types

Both uniqueness typing and linear logic struggle with the read-only access problem first described
p. 83 in Section 3.2.6. Wadler presents a solution similar to the solution adopted in Clean (for example, see Wadler, 1990, Section 4, *Read-only access*), with the same problems.

Odersky (1992) proposes to solve this problem in a more principled fashion by distinguishing between non-linear, linear and "observer" types. Variables that are assigned a linear type in an expression can be assigned an observer type in the body of a strict-let expression. By checking that the types of the variables assigned in the strict-let expression do not contain any observer types, the type system can guarantee that these variables do not escape.

The typing rules are based on the type system from (Turner et al., 1995), the final system we described in the previous section with steadfast linear types and unrestricted weakening. Variables of a linear type cannot be coerced to a non-linear type (no subtyping), but they *can* be coerced to observer types, and thus be read from (but not written to) arbitrarily many times. The system correctly deals with the partial application problem: lambda expressions with linear elements in their closure must be linear themselves, and can thus be executed only once. The rule for application has the important side-condition that a function to be executed is linear or non-linear, but not of an observer type. This is important to prohibit executing a function with linear elements in its closure more than once by coercing it to an observer type in the body of a strict-let expression.

As long as there are only two possible attribute values (linear or non-linear) we can express linearity requirements simply by adorning type variables with the required linearity attribute. An unfortunate consequence of introducing a third kind of attribute (observer) is that we need to introduce constraints instead.[1] A constraint on a type τ can be of the form "τ must be linear or non-linear" (but not of an observer type), "τ must be non-linear or observer" (but not linear) or "τ must be linear or observer" (but not non-linear). Odersky uses $\nu \leq \bar{0}$, $\nu \leq \bar{1}$ and $\nu \leq \bar{2}$ to denote these constraints, where ν is the attribute on τ. The system uses qualified types (Section 2.9.2) p. 72
as a constraint framework; we do not show the full typing rules as they provide little additional insight.

3.4 Uniqueness logic

Uniqueness logic (Harrington, 2001, 2006) is an attempt to define the logic that corresponds to uniqueness typing through the Curry-Howard isomorphism. That said, while the logic is "inspired by" Clean's uniqueness type system, there are significant differences, especially regarding the treatment of partial application. The rules are defined in Figure 3.5. The type system is presented p. 92
as a sequent calculus (Section 2.1.3) rather than in natural deduction style. This makes the p. 22
comparison between uniqueness logic and the other systems in this chapter slightly more difficult, but does not otherwise change the properties of the type system.

3.4.1 Affinity

At a first glance it may seem that the type system is linear rather than affine: rule WEAK allows to discard variables of a non-unique type only. However, rule WEAK′ is admissible (can be derived from the other rules):

$$\frac{\Gamma \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\nu'} \vdash e : \tau^{\nu}} \textsc{Weak}'$$

[1] Note that it is *not* correct to introduce a subtyping relation so that an observer type is considered a subtype of a non-linear type: this would mean that we can forget that a term has an observer type, and thus escape from from a strict-let expression. It is thus important to use constrained polymorphism rather than subtyping.

$$\dfrac{}{x : \tau^{\nu} \vdash x : \tau^{\nu}}\,\textsc{Var}$$

$$\dfrac{\Gamma \vdash e' : \tau'^{\nu'} \qquad \Delta, x : \tau'^{\nu'} \vdash e : \tau^{\nu}}{\Gamma, \Delta \vdash e[x := e'] : \tau^{\nu}}\,\textsc{Cut}$$

$$\dfrac{\Gamma \vdash e' : \tau^{\nu} \qquad \Delta, x : \tau'^{\nu'} \vdash e : \tau''^{\nu''}}{\Gamma, \Delta, y : \tau^{\nu} \xrightarrow{\bullet} \tau'^{\nu'} \vdash e[x := y\, e'] : \tau''^{\nu''}}\,{\to}\textsc{Left} \qquad \dfrac{\Gamma, x : \tau^{\nu} \vdash e : \tau'^{\nu'}}{\Gamma \vdash \lambda x \cdot e : \tau^{\nu} \xrightarrow{\bullet} \tau'^{\nu'}}\,{\to}\textsc{Right}$$

$$\dfrac{\Gamma, x : \tau'^{\nu'} \vdash e : \tau^{\times}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\times}}\,\textsc{Dereliction} \qquad \dfrac{\Gamma \vdash e : \tau^{\nu}}{\Gamma \vdash e : \tau^{\times}}\,\textsc{Promotion}$$

$$\dfrac{\Gamma \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\,\textsc{Weak} \qquad \dfrac{\Gamma, x : \tau'^{\times}, x : \tau'^{\times} \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}\,\textsc{Contr}$$

Figure 3.5: Uniqueness Logic (Harrington, 2001)

which means that the type system is in fact affine. To show that WEAK$'$ is admissible, we first show that REDEMPTION is admissible (Harrington, 2006, Section 2)

$$\dfrac{\Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\nu'} \vdash e : \tau^{\nu}}\,\textsc{Redemption}$$

using the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \tau'^{\nu'} \vdash x : \tau'^{\nu'}}\,\textsc{Var}}{x : \tau'^{\nu'} \vdash x : \tau'^{\times}}\,\textsc{Promotion} \qquad \Gamma, x : \tau'^{\times} \vdash e : \tau^{\nu}}{\Gamma, x : \tau'^{\nu'} \vdash e[x := x] : \tau^{\nu}}\,\textsc{Cut}}{\Gamma, x : \tau'^{\nu'} \vdash e : \tau^{\nu}}\,=$$

Notice that the admissibility of REDEMPTION together with the requirement that the conclusion of DERELICTION must be non-unique implies the duality of the subtyping relation when compared to linear logic.

Given REDEMPTION, admissibility of WEAK$'$ is trivial:

$$\dfrac{\dfrac{\Gamma \vdash e : t^{u}}{\Gamma, x : s^{\times} \vdash e : t^{u}}\,\textsc{Weak}}{\Gamma, x : s^{v} \vdash e : t^{u}}\,\textsc{Redemption}$$

3.4.2 Partial application

The rules for left and right introduction of × (non-unique) are more interesting. Rule PROMOTION is essentially subtyping: every unique term can be coerced into a non-unique term. This is similar to Clean's subtyping relation, but unlike in Clean, the subtyping relation applies even to functions.

To avoid problems with partial application, the rules for left and right introduction of → deal with *unique* functions only. The only way to apply a non-unique function is to apply rule p. 22 DERELICTION (see equation (2.1) for an explanation of application in sequent calculi); but DERELICTION can only be applied if the result is non-unique. Put another way, functions can always be applied multiple times (even if they have unique elements in their closure), but the result of applying a shared function is always non-unique.

This solves the dup problem from Section 3.2.4, but we nevertheless consider the proposed p. 82 solution inadequate. Consider the function that closes a file and returns a boolean indicating whether the operation was successful:

$$\texttt{closeFile :: File}^{\bullet} \xrightarrow{\times} \texttt{Bool}^{\times}$$

In Harrington's system, the following program would be accepted

```
closeFileTwice file = (\g. g ⊥, g ⊥) (\x. closeFile file)
```

even though it is not referentially transparent (in Clean, this program would be rejected by the type checker). In fact, a function such as closeFile, which returns a non-unique result, can even be applied to non-unique files using rule Dereliction. Harrington discusses the implementation of this rule:

> The [DERELICTION] rule can be implemented in two ways. We can either change all destructive updates on that value to equivalent non-destructive updates, or [...] we can make a copy of the value to a sufficient depth to guarantee that any structure which might be modified is unshared.
>
> (Harrington, 2006, Section 2.1.1)

Both these options can be implemented for destructive updates on arrays, for example. The destructive update can be replaced by a non-destructive update, or the array can be copied before the destructive update is applied. However, for "truly" destructive updates (such as operations on files) neither option is possible.

One might argue that this is a library design issue: it is the responsibility of the library designer to type foreign (side-effecting) functions in such a way that they always return a unique result (so that rule DERELICTION does not apply). This may seem reasonable: after all, it is already the responsibility of the library designer to require unique inputs to side-effecting functions. However, even when the library is designed as follows

$$\begin{array}{ll} \texttt{closeFile} & \texttt{:: File}^{\bullet} \xrightarrow{\times} \texttt{File}^{\bullet} \\ \texttt{isClosed} & \texttt{:: File}^{\times} \xrightarrow{\times} \texttt{Bool}^{\times} \end{array}$$

a user can easily define closeFile'

$$\begin{array}{l} \texttt{closeFile' :: File}^{\bullet} \xrightarrow{\times} \texttt{Bool}^{\times} \\ \texttt{closeFile' = isClosed . closeFile} \end{array}$$

with exactly the same problems as closeFile (there is no rule in the type system that forces the result of a function to be unique when its input is unique).

3.4.3 Exponentials in a non-unique context

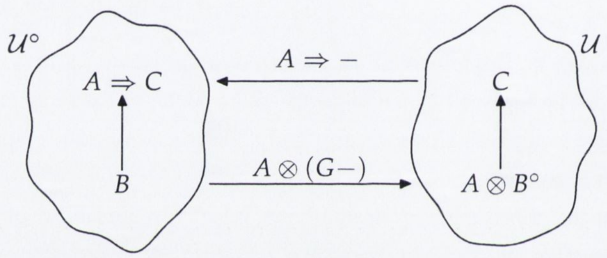
The result of applying a shared function in the logic hitherto described is always non-unique, *even when that function has no unique objects in its closure*. This is overly restrictive, and the final section of (Harrington, 2006) describes an extension to the type system to overcome this deficiency. Unfortunately, only a categorical characterization of this new kind of function is given (without the corresponding logical rules); so rather than attempting to give an "equivalent" set of logical rules in the syntax we use throughout this dissertation, we will give the presentation as it is in the paper.

Most of Harrington's dissertation is concerned with a categorical semantics for the logic; a discussion of this semantics is beyond the scope of this chapter, but we do need a few of its definitions. The semantics is given with respect to a "uniqueness category" $\mathcal{U}$. An object X in this category denotes a type X in the type system, and an arrow from $X \longrightarrow Y$ is a term (a program) of type $X \longrightarrow Y$. The exact definition of the category and its properties does not concern us here.

The category comes equipped with a monad $(\circ, \eta, \mu)$ where $\circ$ is Harrington's symbol for non-unique; $\circ$ is not *quite* equivalent to $\times$ in our notation, since Harrington treats $\circ$ as a type constructor so that `Int` is the type of unique integers (which we would denote $\texttt{Int}^{\bullet}$), and both $\texttt{Int}^{\circ}$ and $(\texttt{Int}^{\circ})^{\circ}$ describe the type of non-unique integers (which we would denote as $\texttt{Int}^{\times}$; p. 77 see also the discussion of syntax in Section 3.1). The first component of the monad $\eta_X : X \longrightarrow X^{\circ}$ is essentially subtyping; the second component $\mu_X : (X^{\circ})^{\circ} \longrightarrow X^{\circ}$ says that $\circ$ is idempotent.

The new function space (or "exponential") is denoted by $A \Rrightarrow C$. That is, a term of type $A \Rrightarrow C$ is a function from A to C that does not have any unique elements in its closure and can therefore be shared freely: Harrington defines $A \Rrightarrow B$ and $(A \Rrightarrow B)^{\circ}$ to be isomorphic.

Denoting the Kleisli category corresponding to $\mathcal{U}$ by $\mathcal{U}^{\circ}$ and the right adjoint of the Kleisli p. 75 adjunction for $\circ$ by G (Section 2.10.3), $A \Rrightarrow B$ is defined to be the right adjoint of $A \otimes (G-)$:



That is, there is a natural bijection[1]

$$\texttt{curry}_{\circ} : \mathrm{Hom}_{\mathcal{U}}(A \otimes B^{\circ}, C) \cong \mathrm{Hom}_{\mathcal{U}^{\circ}}(B, A \Rrightarrow C)$$

which by the definition of the Kleisli category and the fact that $(A \Rrightarrow C)^{\circ}$ is isomorphic to $A \Rrightarrow B$ is equivalent to:

$$\texttt{curry}_{\circ} : \mathrm{Hom}_{\mathcal{U}}(A \otimes B^{\circ}, C) \cong \mathrm{Hom}_{\mathcal{U}}(B, A \Rrightarrow C)$$

In other words, a function from $B \longrightarrow A \Rrightarrow C$ is (isomorphic to) a function from $A \otimes B^{\circ} \longrightarrow C$ so that the B in the closure of $A \Rrightarrow C$ must be non-unique. Evaluation of $A \Rrightarrow C$ is given as usual by the co-unit of the adjunction (Section 2.10.2).

Like in linear logic, there is a choice between introducing a new syntax for abstraction and p. 89 application for terms of type $a \Rrightarrow b$ (as in Figure 3.2) or allowing the same syntax for both terms of type $a \longrightarrow b$ and terms of $a \Rrightarrow b$ (as in Figure 3.3). Unfortunately, neither choice is completely satisfactory. Consider the function `apply` $(\lambda f \cdot \lambda x \cdot f\ x)$. When using a different syntax for both constructs, the user must choose whether `apply` applies functions of type $a \longrightarrow b$ or functions of type $a \Rrightarrow b$. When using the same syntax, the type system must choose; unfortunately, $a \longrightarrow b$ and $a \Rrightarrow b$ are incomparable (neither is more general than the other) so we lose principal types. Harrington does not discuss this issue.

[1] In the paper, the bijection is given as $\texttt{curry}_{\circ} : \mathrm{Hom}_{\mathcal{U}}(A \otimes B^{\circ}, C) \cong \mathrm{Hom}_{\mathcal{U}^{\circ}}(B^{\circ}, A \Rrightarrow C)$, but I believe that to be a mistake since that does not follow from the definition of $A \Rrightarrow -$ as right adjoint to $A \otimes (G-)$.

3.5 Single-threaded polymorphic lambda calculus

The single-threaded polymorphic lambda calculus, or STPLC for short, by Guzmán (Guzmán and Hudak, 1990; Guzmán, 1993) is similar to uniqueness typing in spirit but significantly different in implementation. It is also considerably more complex than the other systems we consider in this chapter, despite repeated claims by Guzmán that the system is "simple and intuitive".

Our review here is based mainly on Guzmán's PhD thesis. There are some subtle differences between the PhD thesis and the paper (which predates the thesis by approximately three years); some discussion of the differences can be found in the conclusions to the thesis (Chapter 8).

3.5.1 Intuition

An important difference between uniqueness typing (or linear logic) and the STPLC is that in the STPLC an attribute denotes how a variable is *used*, as opposed to an innate property of the variable:

> A static type system characterizes the type of values used in programs; i.e., it is concerned with properties *satisfied* by the values in the programs. A liability [attribution] system on the other hand, is concerned with *how* the values are *used* in the program: i.e., how often they are used, whether they are mutated, etc.
>
> (Guzmán, 1993, Section 4.2; emphasis in original)

This is a fundamental difference and one that makes understanding the STPLC difficult to someone accustomed to uniqueness or linear typing. For example, it means that a function f cannot make any demands about the type of term that is passed as argument to f. If f destructively updates its argument, all it can say is that it writes to its argument (in particular, it can *not* demand that the argument must be single-threaded). Instead, when the type checker finds that a multi-threaded variable (one that is used more than once) is passed to a function that writes to its argument, it signals a type error.

Instead of a single attribute (such as "uniqueness"), Guzmán considers three: *read-only* (R), *free* (F; not "captured" as part of another object such as a list), and *single-threaded* (linearity, S). Of these three, single-threaded corresponds most closely to uniqueness and indicates if a variable is used once or more than once within its scope. He then considers all possible conjunctions and disjunctions of these attributes, which form a lattice (Figure 3.6). However, many of the p. 96 resulting 20 properties are indistinguishable, and Guzmán reduces the set of 20 properties down to 8 *abstract uses* (also shown in the same figure). These uses are:

$\bot$	No use at all
rs	Read-only, free, and single-threaded
rm	Read-only and free, but multi-threaded
cs	Captured and single-threaded
cm	Captured and multi-threaded
ws	Written to and single-threaded
$ws \vee cm$	Disjunction of cm and ws
wm	Written to and multi-threaded (error)

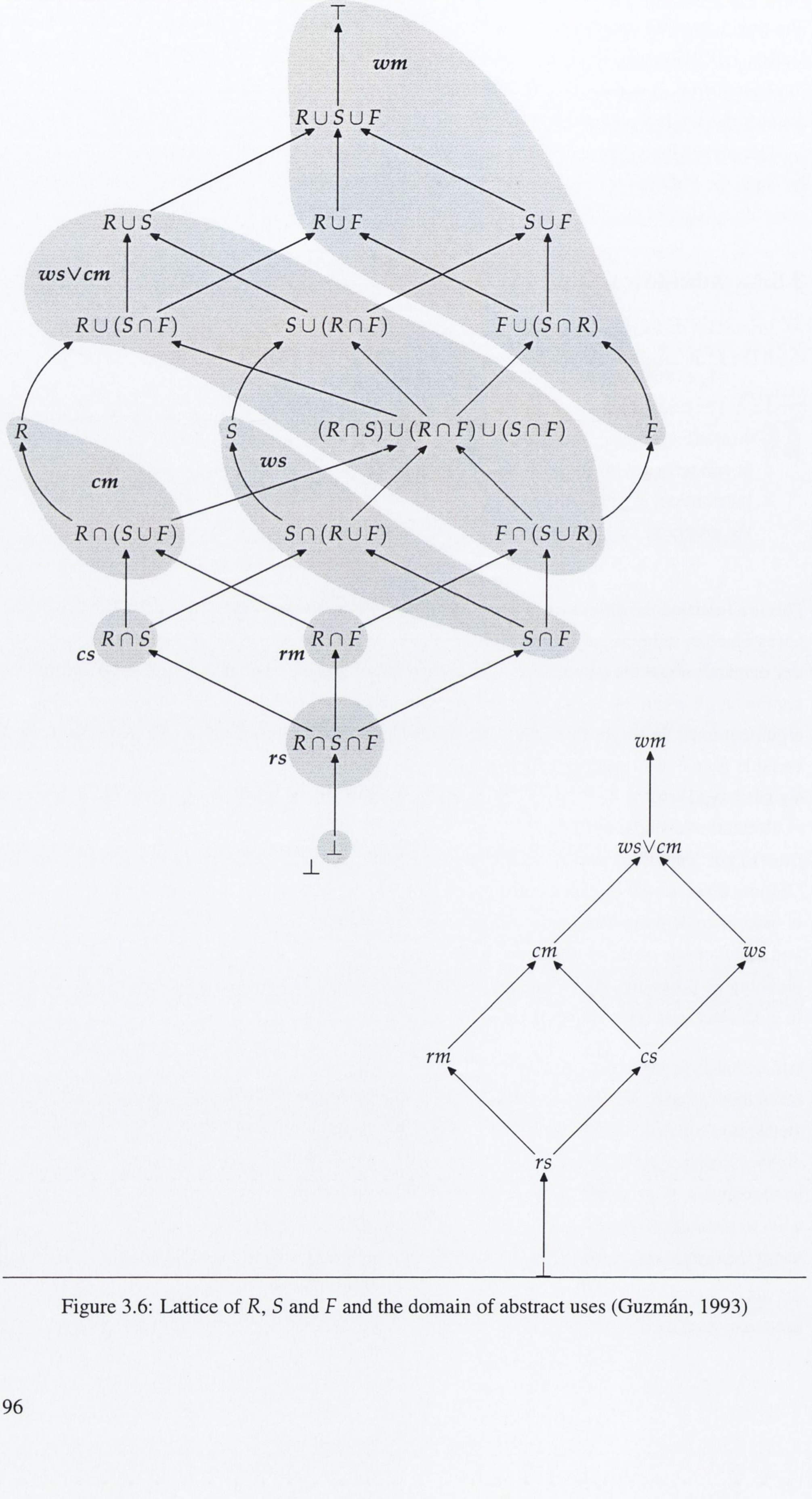


Figure 3.6: Lattice of *R*, *S* and *F* and the domain of abstract uses (Guzmán, 1993)

The abstract use of x in

$$x$$

is cs (independent of the type of x): it is used once, not written to, and captured in the closure of the enclosing lambda expression. However, the abstract use of x in

$$f\ x$$

depends on the use of x by f. Therefore, the function space is decorated with an abstract use, which indicates how applying the function to an argument changes the abstract use of that argument. For example, if the type of f is $a \xrightarrow{ws} b$, then the abstract use of x in $(f\ x)$ is $ws \cdot cs = ws$.

It is important to realize however that attributes are *not* considered part of the type in the STPLC; the types a and b in the domain and codomain of f are "base" types without an attribute. The attribute on the arrow does *not* specify the attribute of the function, but rather how applying the function modifies the attribute on the argument to the function.

A somewhat unusual design decision of the STPLC is that attributes are associated only with variables (as opposed to with arbitrary terms). The *liability* of a term is an environment mapping all the free variables in the term to an attribute (abstract use). That is, attributes can only be used to reason about how *variables* are used.

Some terms however generate objects that are not associated with any variable. For example, there is an "anonymous" object created by

$$\text{mkArray } 10\ 0$$

(which creates an array of 10 integers initialized to 0). Of course, this array is only anonymous insofar as there is no *variable* that is associated with the array; we *can* name the array (indeed, this is exactly the expression mkArray 10 0). However, since the STPLC associates abstract uses only with variables, but still wants to be able to reason about the abstract use of this "anonymous object", a distinguished variable ξ is introduced which describes the use of "the" anonymous object. The function space constructor is modified again to get a second attribute; when a function of type $a \xrightarrow[u_2]{u_1} b$ is applied to an argument, the abstract use of the argument is modified based on u_1, and the abstract use of the anonymous object is modified based on u_2.

There could be more than one anonymous object in an expression, so that the abstract use associated with ξ must be a conservative estimate across all anonymous objects. This is one of the most obscure aspects of the STPLC and makes it quite difficult to understand exactly what a liability of an expression means.

3.5.2 Typing rules

The most important typing rules are shown in Figure 3.7. They take the form p. 98

$$C, \Gamma \vdash e : \langle \tau, L \rangle$$

C is a set of subtyping constraints which we will not consider further (the STPLC uses a subtyping relation between liabilities based on the lattice of abstract uses). Γ is a typing environment and maps identifiers to types as usual; τ is the type of e and L is the liability of e. It is instructive to consider some of our running examples again, `dup`, `sneakyDup` and `strictDup`.

Typing rules

$$\frac{}{C,\Gamma \vdash k : \mathcal{K}(k)}\ \textsc{Con}$$

(for some mapping $\mathcal{K}$ from constants to types and liabilities)

$$\frac{}{C,(\Gamma, x : \sigma) \vdash x : \langle \sigma, [x \mapsto cs] \rangle}\ \textsc{Var}$$

$$\frac{C,(\Gamma, x : \tau_1) \vdash e : \langle \tau_2, [L, x : u_1, \xi : u_2] \rangle}{C,\Gamma \vdash \lambda x \cdot e : \langle \tau_1 \xrightarrow[u_2]{u_1} \tau_2, L \rangle}\ \textsc{Abs}$$

$$\frac{C,\Gamma \vdash e_1 : \langle \tau_1 \xrightarrow[u_2]{u_1} \tau_2, L_1 \rangle \qquad C,\Gamma \vdash e_2 : \langle \tau_1, L_2 \rangle}{C,\Gamma \vdash e_1\, e_2 : \langle \tau_2, (L_1 \overset{\text{par}}{\odot} (u_1 \cdot L_2)) \sqcup [\xi \mapsto u_2] \rangle}\ \textsc{App}$$

$$\frac{C,\Gamma \vdash e_1 : \langle \tilde{\tau}_1 \xrightarrow[u_2]{u_1} \tau_2, L_1 \rangle \qquad C,\Gamma \vdash e_2 : \langle \tilde{\tau}_1, L_2 \rangle}{C,\Gamma \vdash e_1\, e_2 : \langle \tau_2, ((u_1 \cdot L_2) \overset{\text{seq}}{\odot} L_1) \sqcup [\xi \mapsto u_2] \rangle}\ \textsc{StrictApp}$$

(where $\tilde{\tau}$ is the set of all types excluding the functions)

$$\frac{C,\Gamma \vdash e_c : \langle \text{Bool}, L_c \rangle \qquad C,\Gamma \vdash e_t : \langle \tau, L_t \rangle \qquad C,\Gamma \vdash e_f : \langle \tau, L_f \rangle}{C,\Gamma \vdash \text{if}\ e_c\ e_t\ e_f : \langle \tau, (L_c \overset{\text{seq}}{\odot} (L_t \overset{\text{alt}}{\odot} L_f)) \rangle}\ \textsc{If}$$

Combining abstract uses

$$\begin{array}{ll}
u_1 \overset{\text{alt}}{\odot} u_2 = u_1 \sqcup u_2 & u_1 \overset{\text{seq}}{\odot} \bot = u_1 \\
 & \bot \overset{\text{seq}}{\odot} u_2 = u_2 \\
u_1 \overset{\text{par}}{\odot} \bot = u_1 & rs \overset{\text{seq}}{\odot} u_2 = u_2 \\
\bot \overset{\text{par}}{\odot} u_2 = u_2 & rm \overset{\text{seq}}{\odot} u_2 = rm \sqcup u_2 \\
u_1 \overset{\text{par}}{\odot} u_2 = (\uparrow u_1) \sqcup (\uparrow u_2) & u_1 \overset{\text{seq}}{\odot} u_2 = (\uparrow u_1) \sqcup (\uparrow u_2)
\end{array}$$

$$\begin{array}{llll}
u \cdot \bot = \bot & & \uparrow \bot = \bot & \\
u \cdot rs = rs & & \uparrow u = rm & (u \sqsubseteq rm) \\
\bot \cdot u = u & (ws \sqsubseteq u) & \uparrow u = cm & (u \sqsubseteq cm) \\
\bot \cdot u = rs & & \uparrow u = wm & \\
cs \cdot u = u & & & \\
u \cdot cs = u & & & \\
cm \cdot cm = cm & & & \\
ws \cdot ws = ws & & & \\
u_1 \cdot u_2 = wm & & &
\end{array}$$

Combining Liabilities

$$(L_1 \odot L_2)\ \xi = (L_1\ \xi) \sqcup (L_2\ \xi)$$
$$(L_1 \odot L_2)\ x = (L_1\ x) \odot (L_2\ x)$$

$$(u \cdot L)\ x = u \cdot (L\ x)$$

Figure 3.7: Typing rules in the STPLC

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathrm{p} : \langle a \xrightarrow{cs} (a \xrightarrow[cs]{cs} \mathrm{P}\ a\ a), []\rangle}\,\textsc{Con} \qquad \dfrac{}{x : a \vdash x : \langle a, [x : cs]\rangle}\,\textsc{Var}}{x : a \vdash \mathrm{p}\ x : \langle a \xrightarrow[cs]{cs} \mathrm{P}\ a\ a, [x : cs]\rangle}\,\textsc{App} \qquad \dfrac{}{x : a \vdash x : \langle a, [x : cs]\rangle}\,\textsc{Var}}{x : a \vdash \mathrm{p}\ x\ x : \langle \mathrm{P}\ a\ a, [x : cm, \xi : cs\rangle}\,\textsc{App}}{\vdash \lambda x \cdot \mathrm{p}\ x\ x : \langle a \xrightarrow[cs]{cm} \mathrm{P}\ a\ a, []\rangle}\,\textsc{Abs}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathrm{p} : \langle a \xrightarrow{cs} (a \xrightarrow[cs]{cs} \mathrm{I}), []\rangle}\,\textsc{Con} \qquad \dfrac{\dfrac{}{f : \mathrm{I} \to a \vdash f : \langle \mathrm{I} \to a, [f : cs]\rangle}\,\textsc{Var} \qquad \dfrac{}{\vdash 1 : \langle \mathrm{I}, [\xi : cs]\rangle}\,\textsc{Con}}{f : \mathrm{I} \to a \vdash f_1 : \langle a, [f : cs, \xi : rs\rangle}\,\textsc{App}}{f : \mathrm{I} \to a \vdash \mathrm{p}\ f_1 : \langle a \xrightarrow[cs]{cs} \mathrm{P}\ a\ a, [f : cs, \xi : rs]\rangle}\,\textsc{App} \qquad \dfrac{\dfrac{}{f : \mathrm{I} \to a \vdash f : \langle \mathrm{I} \to a, [f : cs]\rangle}\,\textsc{Var} \qquad \dfrac{}{\vdash 2 : \langle \mathrm{I}, [\xi : cs]\rangle}\,\textsc{Con}}{f : \mathrm{I} \to a \vdash f_2 : \langle a, [f : cs, \xi : rs\rangle}\,\textsc{App}}{\dfrac{f : I \to a \vdash \mathrm{p}\ f_1\ f_2 : \langle \mathrm{P}\ a\ a, [f : cm, \xi : cs]\rangle}{\vdash \lambda f \cdot \mathrm{p}\ f_1\ f_2 : \langle (\mathrm{I} \to a) \xrightarrow[cs]{cm} \mathrm{P}\ a\ a, []\rangle}\,\textsc{Abs}}\,\textsc{App} \qquad \dfrac{\dfrac{}{x : a \vdash x : \langle a, [x : cs]\rangle}\,\textsc{Var}}{x : a \vdash \lambda y \cdot x : \langle \mathrm{I} \to a, [x : cs]\rangle}\,\textsc{Abs}}{\dfrac{x : a \vdash (\lambda f \cdot \mathrm{p}\ f_1\ f_2)(\lambda y \cdot x) : \langle \mathrm{P}\ a\ a, [x : cm, \xi : cs]\rangle}{\vdash \lambda x \cdot (\lambda f \cdot \mathrm{p}\ f_1\ f_2)(\lambda y \cdot x) : \langle a \xrightarrow[cs]{cm} \mathrm{P}\ a\ a, []\rangle}\,\textsc{Abs}}\,\textsc{App}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathrm{p} : a \xrightarrow{cs} (a \xrightarrow[cs]{cs} \mathrm{P}\ a\ a), []\rangle}\,\textsc{Con} \qquad \dfrac{}{x : a \vdash x : \langle a, [x : cs]\rangle}\,\textsc{Var}}{x : a \vdash \mathrm{p}\ x : \langle a \xrightarrow[cs]{cs} \mathrm{P}\ a\ a, [x : cs]\rangle}\,\textsc{App} \qquad \dfrac{}{y : a \vdash y : \langle a, [y : cs]\rangle}\,\textsc{Var}}{x : a, y : a \vdash \mathrm{p}\ x\ y : \langle \mathrm{P}\ x\ y, [x : cs, y : cs, \xi : cs]\rangle}\,\textsc{App}}{x : a \vdash \lambda y \cdot \mathrm{p}\ x\ y : \langle a \xrightarrow[cs]{cs} \mathrm{P}\ a\ a, [x : cs]\rangle}\,\textsc{Abs} \qquad \dfrac{}{x : a \vdash x : \langle a, [x : cs]\rangle}\,\textsc{Var}}{\dfrac{x : a \vdash (\lambda y \cdot \mathrm{p}\ x\ y)\ x : \langle \mathrm{P}\ a\ a, [x : cm, \xi : cs]\rangle}{x : a \vdash \mathrm{let^*}\ y = x\ \mathrm{in}\ \mathrm{p}\ x\ y : \langle \mathrm{P}\ a\ a, [x : cm, \xi : cs]\rangle}\,=}\,\textsc{StrictApp}}{\vdash \lambda x \cdot \mathrm{let^*}\ y = x\ \mathrm{in}\ \mathrm{p}\ x\ y : \langle a \xrightarrow[cs]{cm} \mathrm{P}\ a\ a, []\rangle}\,\textsc{Abs}$$

Abbreviations: $\mathrm{I} \equiv \mathrm{Int}, \mathrm{p} \equiv \mathrm{pair}, \mathrm{P} \equiv \mathrm{Pair}, \to \equiv\ \xrightarrow[\perp]{\perp}, f_1 \equiv f\ 1$

Only relevant entries in the environments are shown to the left of the turnstile.

Figure 3.8: Derivation for `dup`, `sneakyDup` and `strictDup`

p. 99 All three terms have type $a \xrightarrow[cs]{cm}$ `Pair` a a; Figure 3.8 shows the derivations. The attribute cm above the arrow means that applying `dup` to an argument will cause that argument to be multi-threaded (we would say, "non-unique"). As alluded to before, it is harder to give an intuition for the attribute below the arrow; it is cs because the partial application (`pair` x) in the body of `dup` "captures" x in a closure.

The interesting aspect of this example is that the type system somehow notices that x is duplicated in `sneakyDup`, even though the system does not reason about the abstract use of the closure $\lambda y \cdot x$ (nor could it, since abstract uses are only associated with variables). In particular, functions can always be applied multiple times, and the rule for application does not take the abstract use of the function that we apply into account. The duplication of x is nevertheless noticed since the *liability* associated with $\lambda y \cdot x$ lists the abstract uses of all free variables in $\lambda y \cdot x$; that is, the abstract uses of all elements in the closure of $\lambda y \cdot x$!

The function `closeFileTwice` is nearly identical to the derivation of `sneakyDup`. The type inferred for $\lambda y \cdot$ `closeFile` x, where we assume

$$\texttt{closeFile :: File} \xrightarrow{ws} \texttt{File}$$

is `Int` → `File` as before, but the inferred liability is $[x : ws]$. Then the inferred liability of the body of `closeFileTwice` is $[x : wm]$. That is, x is written to but not single-threaded, and the program is rejected by the type checker.

3.5.3 Strict application

The single-threaded polymorphic lambda calculus is considerably more complicated than uniqueness typing and linear logic, and so far we have not yet seen any obvious benefits. But its real chance to shine is the rule for strict application.

p. 78 We have seen in the introduction to uniqueness typing (Section 3.2) that it may be advantageous to introduce a "strict let" construct (denoted by `let⋆`) which allows one or more read-only accesses to a term followed by one potentially destructive update (`swap` is the canonical example). We have also seen that adding this rule to uniqueness typing or linear logic is less than trivial. The STPLC however can properly deal with this rule due to the distinction between functions that merely read their arguments and functions that "capture" their arguments. Compare

```
strictDup = λx. let⋆ y = x in Pair x y
```

to

```
f = λx. let⋆ y = g x in Pair x y
```

p. 99 for some function $g :: a \xrightarrow{rs} \mathrm{I}$. The derivation of `strictDup` is shown in Figure 3.8; the derivation of f is shown below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{(as for }\texttt{strictDup}\text{)}}{x : a \vdash \lambda y \cdot \mathrm{p}\; x\; y : \langle \mathrm{I} \xrightarrow[cs]{cs} \mathrm{P}\; a\; \mathrm{I}, [x : cs]\rangle}\;\textsc{Abs} \qquad \dfrac{\dfrac{}{\vdash g : \langle a \xrightarrow{rs} \mathrm{I}, []\rangle}\;\textsc{Con} \quad \dfrac{}{x : a \vdash x : \langle a, [x : xs]\rangle}\;\textsc{App}}{x : a \vdash g\; x : \langle \mathrm{I}, [x : rs]\rangle}\;\textsc{App}}{x : a \vdash (\lambda y \cdot \mathrm{p}\; x\; y)\; (g\; x) : \langle \mathrm{P}\; a\; \mathrm{I}, [x : cs, \xi : cs]\rangle}\;\textsc{StrictApp}}{x : a \vdash \text{let* } y = g\; x \text{ in p } x\; y : \langle \mathrm{P}\; a\; \mathrm{I}, [x : cs, \xi : cs]\rangle}\;=}{\vdash \lambda x \cdot \text{let* } y = g\; x \text{ in p } x\; y : \langle a \xrightarrow[cs]{cs} \mathrm{P}\; a\; \mathrm{I}, []\rangle}\;\textsc{Abs}$$

The fact that g only reads its argument (but does not capture it) means that applying f to an argument does *not* cause that argument to be multi-threaded (whereas applying `strictDup` *does*). Since there is no distinction between $a \xrightarrow{cs} b$ and $a \xrightarrow{rs} b$ in uniqueness typing or linear logic, this difference between `strictDup` and f is more difficult to formalize in those systems.

In the `swap` example,

```
swap! arr i j =
  let* x = lookup arr i
       y = lookup arr j
  in update! (update! arr i y) j x
```

there are two read-only accesses to `arr` before the array is updated; it is thus crucial that `lookup` does not capture the array[1]:

`lookup ::` $\texttt{Array}\ a \xrightarrow{rs} \texttt{Int} \xrightarrow{rs} a$

But there is a subtle point to be made here. This type of `update` also allows us to define the following function:

`fst ::` $\texttt{Array}\ a \xrightarrow{cs} \texttt{Pair}\ (\texttt{Array}\ a)\ a$

```
fst arr = let* elem = lookup arr 0 in pair arr elem
```

The overlap between the array and the array element in the result of `fst` is lost to the type system, and it is therefore important to maintain referential transparency that the elements of the array are immutable. Arrays are therefore not "first-class": for example, arrays of arrays are not permitted. Chapter 8 of Guzmán's thesis briefly discusses how the STPLC must be extended to remove this restriction, at the cost of additional complexity.

3.5.4 Polymorphic liabilities

So far the inferred abstract uses and liabilities are all monomorphic: there is no support for polymorphism on the abstract use level. Guzmán extends the type system to support polymorphic liabilities in his thesis, and this is where the system gets *truly* complicated. A detailed discussion of this extended type system is beyond the scope of this dissertation; we show a few example types only.

A simple example is `apply`:

`apply ::` $(a \xrightarrow[v_2]{v_1} b) \xrightarrow{cs} (a \xrightarrow[v_2]{v_1} b)$

```
apply f x = f x
```

The real complications start appearing when dealing with recursive function definitions, since the types will now need to include *recursive equations between liabilities* to maintain principal types. Here is a relatively simple example:

`many ::` $(a \xrightarrow{v_1} a) \xrightarrow{cm} \texttt{Int} \xrightarrow{rs} a \xrightarrow{v_2}$ **where** $v_2 = cs \overset{\texttt{alt}}{\odot} v_2 \cdot v_1$

```
many f n a = if (n == 0) a (many f (n - 1) (f a))
```

We conclude the discussion of STPLC with the principal type of `folda`, a version of the left fold operation `foldl` specialized to arrays. The definition is

```
folda v i n inc f = if (i == n) v (folda (f i v) (inc i) n inc f)
```

[1] Unfortunately, the type of `lookup` is listed as Array $a \xrightarrow{cs}$ Int $\xrightarrow{rs} a$ in Guzmán's thesis; however, it is clear from the discussion in Chapter 8 that this must be a mistake.

The principal type of `folda` is (Guzmán, 1993, Section 5.4)

$$\texttt{folda ::}\ \tau_1 \xrightarrow{\nu_7} \tau_2 \xrightarrow{\nu_8} \tau_2 \xrightarrow{rs} (\tau_2 \xrightarrow[\nu_2]{\nu_1} \tau_2) \xrightarrow{\nu_9} (\tau_2 \xrightarrow[\nu_4]{\nu_3} \tau_1 \xrightarrow[\nu_5]{\nu_6} \tau_1) \xrightarrow[\nu_{11}]{\nu_{10}} \tau_1$$

$$\begin{aligned}
\textbf{where}\ \nu_7 &= cs \overset{\texttt{alt}}{\odot} (\nu_7 \cdot (\nu_6 \cdot cs)) \\
\nu_8 &= rs \overset{\texttt{seq}}{\odot} ((\nu_7 \cdot (\nu_3 \cdot cs)) \overset{\texttt{par}}{\odot} (\nu_8 \cdot (\nu_1 \cdot cs))) \\
\nu_9 &= (\nu_8 \cdot cs) \overset{\texttt{par}}{\odot} (\nu_9 \cdot cs) \\
\nu_{10} &= (\nu_7 \cdot cs) \overset{\texttt{par}}{\odot} (\nu_{10} \cdot cs) \\
\nu_{11} &= (\nu_7 \cdot (\nu_4 \overset{\texttt{alt}}{\odot} \nu_5)) \overset{\texttt{alt}}{\odot} (\nu_8 \cdot \nu_2) \overset{\texttt{alt}}{\odot} \nu_{11}
\end{aligned}$$

where it must be noted that the `where` clauses are all part of the type of `folda`. It appears that the small additional power of the single-threaded polymorphic lambda calculus over uniqueness typing comes at a steep price indeed.

3.5.5 "How to make destructive updates less destructive"

The type system presented by Odersky in (Odersky, 1991) is a close cousin of the STPLC. It makes a number of simplifying assumptions, the most important of which is that it considers a first-order language, thereby sidestepping a number of difficult issues (not the least of which is partial application). On the other hand, he considers a more complex abstract use domain, and associates a *pair* of abstract uses with each variable in a liability. The domain of abstract
p. 104 uses, along with the parallel and sequential combination operations, is shown in Figure 3.9. The system is presented as an abstract interpretation system rather than a type system, so discussing the technical details of the system is beyond the scope of this dissertation.

The reason for associating a pair of abstract uses with each variable is to distinguish between the abstract use of the variable and the abstract use of the *elements* of the variable. For example, the function `head` has type

$$\texttt{head ::}\ [a] \xrightarrow{rd.sh} a$$

This means that `head` only reads (*rd*) the list (but the list is not part of the result), but the result of `head` is a potential alias (*sh*) of the elements of the list. This makes the analysis more precise (and types more complicated), but at the same time the distinction is somewhat arbitrary: the two-level split may not be sufficient (arrays of arrays) or may be unnatural (what are the elements of an integer?).

3.5.6 Relevance of linearity

Odersky makes the following observation (without justification):

> We simplify [the approach of the STPLC] in that we are able to drop without loss of precision their distinction between single- and multiple-threaded accesses.
>
> (Odersky, 1991, Section 1, *Related Work*)

This is surprising because intuitively the distinction between single-threaded and multiple-threaded seems to be the most important distinction in the STPLC; after all, a variable may be destructively updated only when it is single-threaded. Nevertheless, even in the STPLC the distinction between single-threaded and multiple-threaded is irrelevant for most operators; if there is a "write" use of a variable parallel with a "read" use, it makes no difference whether that read is once or more than once—both cases constitute a type error (two parallel writes is always a type error).

A careful analysis of the binary operators shows that $u_1 \sqcup u_2$ (and thus $u_1 \overset{\text{alt}}{\odot} u_2$) never introduces wm (the "erroneous" use); $u_1 \overset{\text{par}}{\odot} u_2$ is wm if either operand is a "write" use, and $u_1 \overset{\text{seq}}{\odot} u_2$ is wm if the first operand is a "write" use; in either case, the linearity of the operands is irrelevant. The only operator for which single-threaded or multiple-threaded makes a difference is the projection operator $(u \cdot v)$, which is used to "project" how a function modifies (u) the abstract use (v) of the argument to the function. The following table shows for which operands an error is thrown only if one of the operands is multiple-threaded:

Accepted (single-threaded)	**Rejected (multiple-threaded)**
$rs \cdot rs$	$rs \cdot rm$, $rm \cdot rm$
$rs \cdot cs$	$rs \cdot cm$, $rm \cdot cm$
$rm \cdot cs$	$rm \cdot cm$
$cs \cdot ws$	$cm \cdot ws$
$cs \cdot ws \vee cm$	$cm \cdot ws \vee cm$

An analysis of whether the pairs of abstract uses in the right column of this table constitute actual errors would involve re-doing the referential transparency proofs in Guzmán's thesis, introducing those pairs one by one as acceptable (or alternatively, rejecting the corresponding pair in the left column as unacceptable). This is a significant amount of work and beyond the scope of this dissertation.

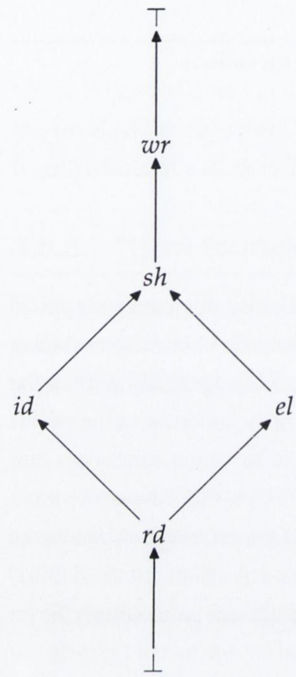
A casual inspection of the table seems to indicate that the STPLC is indeed conservative. It is unclear why any of the combinations on the first three lines of the table should cause an error—why should an expression $f\ e$ be rejected if f only reads (rs) its argument and the liability of e includes $[x : rm]$? On the other hand, it may be that these abstract use pairs simply never arise.

However, it is not difficult to construct an example for the fourth line of the table. For example, given some function $f : a \xrightarrow{ws} a$ which writes to its argument, the following definition is rejected in the STPLC:

$$\dfrac{\dfrac{\dfrac{\text{(Figure 3.8)}}{\vdash \mathtt{dup} : a \xrightarrow[cs]{cm} \mathtt{Pair}\ a\ a} \qquad \dfrac{\dfrac{}{\vdash f : \langle a \xrightarrow{ws} a, []\rangle} \quad \dfrac{}{x : a \vdash x : \langle a, [x : cs]\rangle}\ \textsc{Var}}{x : a \vdash f\ x : \langle a, [x : ws]\rangle}\ \textsc{App}}{x : a \vdash \mathtt{dup}\ (f\ x) : \langle \mathtt{Pair}\ a\ a, [x : wm, \xi : cs]}\ \textsc{App}}{\vdash \lambda x \cdot \mathtt{dup}\ (f\ x) : \langle a \xrightarrow[cs]{wm} \mathtt{Pair}\ a\ a, []\rangle}\ \textsc{Abs}$$

Thus it seems that in the STPLC, a variable cannot be duplicated once written to. However, while we may want to allow this particular program, it is not possible to simply assign a non-error value to $cm \cdot ws$, because that would mean that the system would also accept the (higher-order) `closeFileTwice` example (Section 3.5.2). Thus, the distinction between single-threaded P. 97
and multiple-threaded is in fact important; Odersky can only ignore the distinction because his language is first-order.

Domain lattice and binary operators



$$\begin{array}{lll l}
\top & \overset{\text{par}}{\odot}\, y & = \top & \\
x & \overset{\text{par}}{\odot}\, \top & = \top & \\
\bot & \overset{\text{par}}{\odot}\, y & = y & \\
x & \overset{\text{par}}{\odot}\, \bot & = x & \\
x & \overset{\text{par}}{\odot}\, y & = \top & (x = wr \vee y = wr) \\
x & \overset{\text{par}}{\odot}\, y & = x \sqcup y & \\
 & & & \\
\top & \overset{\text{seq}}{\odot}\, y & = \top & \\
x & \overset{\text{seq}}{\odot}\, \top & = \top & \\
\bot & \overset{\text{seq}}{\odot}\, y & = y & \\
x & \overset{\text{seq}}{\odot}\, \bot & = wr & (x = wr) \\
x & \overset{\text{seq}}{\odot}\, \bot & = rd & (x \neq wr) \\
x & \overset{\text{seq}}{\odot}\, y & = \top & (x = wr) \\
x & \overset{\text{seq}}{\odot}\, y & = y & (x \neq wr) \\
 & & & \\
\top & \cdot\ x.y & = \top & \\
u_1.u_2 & \cdot\ \top & = \top & \\
u_1.u_2 & \cdot\ x.y & = (u_1.u_2 \cdot x).(u_1.u_2 \cdot y) & \\
 & & & \\
\bot.\bot & \cdot\ x & = \bot & \\
u_1.u_2 & \cdot\ x & = x & (x \in \{\bot, rd, wr\}) \\
u_1.u_2 & \cdot\ id & = u_1 & \\
u_1.u_2 & \cdot\ el & = u_2 & \\
u_1.u_2 & \cdot\ sh & = u_1 \sqcup u_2 &
\end{array}$$

For consistency's sake we used Guzmán's notation for the binary operators.
The notation used by Odersky is $\overset{\text{par}}{\odot} \equiv ||$, $\overset{\text{seq}}{\odot} \equiv ;$; and $\cdot \equiv :$

Explanation of the abstract uses

The effect on an entity x of evaluating an expression e:

$\bot$	x is not used
rs	x is read, but is not captured by e (not part of e's normal form)
id	x is a potential alias of e's value
el	x is a potential element of e's value
sh	x is potentially an alias or an element of e's value
wr	x is potentially updated during evaluation of e
$\top$	conflicting access to x (error)

Figure 3.9: Abstract use domain from (Odersky, 1991)

3.6 Other related work

3.6.1 LFPL

LFPL (Hofmann, 2000) is an extremely simple functional programming language: it is strict, monomorphic, first-order (functions cannot be stored inside data structures, nor be passed as arguments or returned as results). Hence, it does not allow currying (functions cannot be partially applied) and it thus avoids all of the problems of partial application that we have been emphasizing in this chapter, but at the cost of limiting expressiveness severely. The author describes it as a "design pattern for writing C-code in a functional style".

Algebraic data type constructors in LFPL come with an extra argument which essentially denotes the heap space to be used by the constructor. These arguments can be used to be precise about memory re-use. For example,

```
append (l, m) =
  case l of
    nil          → m
    cons (d, h, t) → cons (d, h, append (t, m))
```

The extra argument d of `cons` denotes the heap space used by the node; since the same argument d is used on the right hand side to construct a new node, that new node must re-use the memory space of the old node. Obviously, memory space can only be used once and in the original paper LFPL is equipped with an affine linear type system to guarantee this.

In (Aspinall et al., 2008) (an extended version of Aspinall and Hofmann, 2002) this type system is redefined to deal with read-only access in the style of the single-threaded polymorphic lambda calculus (Section 3.5). Three "aspects" are introduced (Guzmán would call them "abstract uses"): p. 95

1 Modifying use
2 Non-modifying use, but shared with result
3 Non-modifying use, not shared with result

Aspect 2 closely corresponds to *cs* in the STPLC; like the domain of abstract uses in the STPLC, the domain of aspects is ordered with $1 \leq 2 \leq 3$. Surprisingly, the authors claim that this property of their type system is novel and that something analogous to aspect 2 does not appear in the functional programming literature (Aspinall et al., 2008, Section 6.1)—Guzmán's work on the STPLC is not even cited. Of course, the type system is much simpler than that of the STPLC due to the simplicity of the language.

One interesting feature of LFPL is that it makes use of the two kinds of products of linear logic. The tensor product, which allows simultaneous access to both components using a case analysis-like construct, allows sharing between both components only if they are accessed read-only; if either component is written to (updated destructively), the components of the product cannot be shared (since they can both be accessed individually). The cartesian product, which comes with the two projection functions (`fst` and `snd`) but crucially does *not* allow simultaneous access to both components of the pair, *does* allow sharing between the component even if one of the components is later updated destructively, since only one component of the product can be accessed (the rules are actually slightly relaxed: access to both components is allowed, but as soon as one is modified the other can no longer be referenced).

Note that Hage and Holdermans (2008) present a similar system (in which users can be explicit about memory re-use) but with support for higher-order functions and call-by-need evaluation, and which is therefore much more expressive. The type system used is the uniqueness type system the authors presented previously (Hage et al., 2007), corrected to deal with the partial application p. 86 problem. We discussed this type system in Section 3.2.8.

3.6.2 SAC

SAC (Scholz, 2003; Grelck and Scholz, 1995) is a domain specific purely functional programming language; its primary focus is high performance numerical array computation. We will mostly ignore this aspect of SAC, however, and discuss only its use of uniqueness typing for input/output. Here is a simple example:

```
use StdIO: all;
use Array: all;

int read_int()
{
    success, result = scanint();
    return(result);
}

int main()
{
    print(reshape([3], [ read_int()
                       , read_int()
                       , read_int()
                       ] ));
    return(0);
}
```

If this looks like C, then that is intentional: the syntax of SAC (Single Assignment C) is designed to make it accessible to programmers with an imperative background. SAC is nevertheless purely p. 59 functional (in the sense described in Section 2.8), and the SAC compiler makes heavy use of properties such as definiteness to optimize and parallelize programs.

Definiteness *appears* not to hold in the example, however. The `print` statement creates a single-dimensional array with three elements whose values are determined by reading from standard input. Given the input

```
1
2
3
```

the output of the program is

```
Dimension:  1
Shape     : <   3>
< 3  2  1 >
```

The contents of the array are shown in the third line (the first two lines describe its shape). As expected, the various calls to `read_int` returned different results (although the order in which they were invoked might be surprising)—a clear violation of definiteness.

The reason that SAC is nevertheless pure is that the program above is considered syntactic sugar for

```
use StdIO: all;
use Array: all;

int read_int(File& stdin)
{
    success, result = scanint(stdin);
    return(result);
}

int main()
{
    print(reshape([3], [ read_int(stdin)
                       , read_int(stdin)
                       , read_int(stdin)
                       ] ));
    return(0);
}
```

where we've made the `stdin` parameter argument explicit; `stdin` is known as a *global object* in SAC. Of course, this program still looks like it has a problem with definiteness: we now have three occurrences of `read_int(stdin)` rather than three occurrences of `read_int()`. Consider however the type of the `stdin` parameter to `read_int`: it looks like a C-style *pass-by-reference* parameter. In SAC, this is considered syntactic sugar for

```
use StdIO: all;
use Array: all;

File, int read_int(File stdin)
{
    stdin, success, result = scanint(stdin);
    return(stdin, result);
}

int main(File stdin)
{
    stdin, a = read_int(stdin);
    stdin, b = read_int(stdin);
    stdin, c = read_int(stdin);

    print(reshape([3], [c,b,a]));
    return(0);
}
```

This now *almost* looks like a functional program, except for the apparent re-assignment of `stdin`. However, when a variable is "re-assigned", this is considered yet more syntactic sugar for a scoping rule which introduces a new variable that shadows the previous (Clean supports a similar rule). Thus, the definition of `main` *really* means

```
int main(File stdin0)
{
    stdin1, a = read_int(stdin0);
    stdin2, b = read_int(stdin1);
    stdin3, c = read_int(stdin2);

    print(reshape([3], [c,b,a]));
    return(0);
}
```

where we have finally recovered a purely functional program.

Syntactic sugar aside, the uniqueness type system of SAC is rather simple and reminiscent of p. 89 Wadler's first type system (Section 3.3.2). SAC introduces a strict separation between special data types called *classes*, instances of which must always be unique, and ordinary data types, instances of which can never be unique. There is therefore no need for uniqueness attributes on types.

Since SAC does not support polymorphism, functions such as

$$\lambda x \cdot (x, x)$$

will be typeable if the type of x is an ordinary data type, and not typeable if the type of x is a class. Moreover, SAC is first-order: functions are not first-class and cannot be passed as arguments p. 82 or returned as results, avoiding the partial application problem (Section 3.2.4). Under these restrictions, uniqueness typing reduces to a syntactic check (after type inference and elimination of syntactic sugar) that every variable with a unique (class) type is used at most once within its p. 84 scope (i.e., uniqueness typing reduces to reference count analysis, Section 3.2.7).

3.6.3 Mercury

Mercury comes from the Prolog family of languages which are based on predicate logic rather than the lambda calculus. A detailed discussion of programming in these languages is beyond the scope of this dissertation; there are many tutorials on Prolog available online (such as Blackburn et al., 2006) which will mostly be applicable to Mercury too. The main difference between Mercury and Prolog is that Mercury is strongly typed (Prolog is untyped) and *strongly moded.*

In Prolog-like languages, there is no distinction between "input" parameters and "output" parameters. Instead, a predicate describes *relations* between parameters; which parameters are input and which are output is determined by how the predicate is used. For example, given the definition of append

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

we can use

```
append([1,2,3],[4,5,6],X)
```

to append the lists $[1,2,3]$ and $[4,5,6]$, but equally we can use

```
append(X,[3,4],[1,2,3,4])
```

which will bind X to the list $[1,2]$, or

```
append(X,Y,[1,2,3])
```

to find all possible ways to split the list [1,2,3]. The way a predicate is used is known as its *mode*. The fact that predicates can be used in more than one mode is exciting, but designing predicates in such a way that they can actually be used in every mode imaginable is difficult and an active research area (for example, see Kiselyov et al., 2008).

It is also one of the reasons that optimizing Prolog code is hard, and a source of programmer errors when predicates are inadvertently used in the wrong mode. For these reasons, Mercury uses a *mode system* where all permissible modes for a predicate must be declared explicitly. For example, the mode declarations that describe the three example uses of `append` are

```
:- mode append(ground >> ground,
               ground >> ground,
               free   >> ground)
               is det.
:- mode append(free   >> ground,
               ground >> ground,
               ground >> ground)
               is nondet.
:- mode append(free   >> ground,
               free   >> ground,
               ground >> ground)
               is multi.
```

A mode declaration describes for each parameter its *instantiation* before and after the evaluation of the predicate. For the first mode, the first two parameters must be instantiated when the predicate is "called", and the last must be free. After the predicate has been evaluated, the third argument will be instantiated (ground). The mode declaration also lists the determinism of the predicate; for example, `append` is deterministic when used in the first mode (will always succeed with exactly one result), non-deterministic when used in the second mode (will either fail or else succeed with exactly one result), and might return more than one answer in the third mode.

As well as a strong mode system, Mercury uses a strong type system based on the Hindley/-Milner system extended with algebraic data types. For example, the type of `append` is

```
:- pred append(list(T), list(T), list(T)).
```

We are interested in Mercury's mode system because it is used to reason about uniqueness as well as "instantiatedness". Mercury's uniqueness system is largely based on the work of Henderson (1992), and formalized and extended in the doctoral thesis of Overton (2003).

Since modes describe many aspects of a predicate, it is not surprising that Mercury's mode system is rather complex. However, if we abstract away from the details the uniqueness system is conceptually very similar to Clean's and used in similar ways. For example, the predicate that writes a string to the standard output has type

```
:- pred write_string(string, io, io).
:- mode write_string(ground >> ground,
                     unique >> clobbered,
                     free   >> unique) is det.
```

Here, `io` corresponds to the `World` type in Clean; `write_string` takes a world state to a new world state. The input world state must be unique on input and not referenced again (*clobbered*) after the predicate has been evaluated; the output world state must be free on input, and will be instantiated and unique on output.

The type and mode annotation for a predicate are given separately. When algebraic data types are involved, the structure of the mode annotation follows the definition of the data type. For example, the function that returns the first non-unique element from a non-unique pair is given by

```
:- pred fst(pair(A,B),A).
:- mode fst(bound(mk_pair(ground, ground)) >> ground,
            free                           >> ground) is det.
fst(mk_pair(X,_),X).
```

Here, "`bound`" means that the attribute on the pair is non-unique. Unfortunately, this concrete syntax of Mercury limits expressiveness. For example, it does not seem possible to define a function which acts on algebraic data types with a polymorphic spine uniqueness (something of the form $\forall u \cdot (\ldots,\ldots)^u \to \ldots$) because the only concrete syntax offered by Mercury is

```
bound(...)
```

or

```
unique(...)
```

which, according to (Overton, 2003, Table 3.1) denote

$$\text{bound}(\text{shared}, \ldots)$$

and

$$\text{bound}(\text{unique}, \ldots)$$

in Overton's abstract syntax; but there is no concrete counterpart to the abstract syntax

$$\text{bound}(u, \ldots)$$

Moreover, it seems that some parts of the mode system described by Overton have not yet been implemented in the compiler (de Vries, 2008b). Although Overton (2003, Section 3.2.3, *Unique modes*, p. 50) prescribes that a subterm cannot be more unique than its parent terms (see p. 81 also Section 3.2.3 in this dissertation) and describes a polymorphic mode system with constraints to enforce this (much like Clean's), uniqueness propagation is not enforced by the compiler (latest stable release as of July 2008). This makes it possible to write a predicate that duplicates arbitrary p. 111 unique objects (including the world state), as shown in Figure 3.10; the code takes advantage of the fact that we can extract a unique element from a non-unique pair. Finally, the standard library does not make use of uniqueness typing at all (except for I/O), even though it does allow for destructive updates to objects such as arrays (and is therefore not pure).

Mercury's uniqueness type diverges from Clean's when it comes to curried functions, however. p. 82 We saw in Section 3.2.4 that we must be careful with partial applications of functions with unique elements in their closure (in Clean, such functions must be "necessarily unique"). Mercury adopts an arguably simpler solution: every variable in a function closure is considered non-unique Overton (2003, Section 3.2.4, *Higher-Order Modes*, p. 58). This is an interesting design decision, p. 201 and we will come back to it in the conclusions (Chapter 8).

The following code shows that due to an incomplete implementation of uniqueness typing in Mercury, we can write a predicate that duplicates arbitrary unique objects. For comparison, the comments describe the purpose and type of each predicate in a Clean-style syntax.

```
% dup_unique(x,y,z) is approximately the same as
% let (y :: a•, z :: a•) = dup_unique(x :: a•) in ..
:- pred dup_unique(A,A,A).
:- mode dup_unique(
        unique
    >> clobbered,
        free
    >> unique,
        free
    >> unique) is det.
dup_unique(X,Y,Z) :- dup(mk_pair(X,1),Ps),
                     split(Ps,P1,P2),
                     fst(P1,Y),
                     fst(P2,Z).

% dup :: (a•,Int×)× → ((a•,Int×)×,(a•,Int×)×)×
:- pred dup(pair(A,int),pair(pair(A,int),pair(A,int))).
:- mode dup(
        bound(mk_pair(unique, ground))
    >> clobbered,
        free
    >> bound(mk_pair(bound(mk_pair(unique, ground)),
                     bound(mk_pair(unique, ground))))) is det.
dup(X,mk_pair(X,X)).

% split(p,a,b) is approximately the same as
% let (a :: (a•,Int×)×, b :: (a•,Int×)×) = p :: ((a•,Int×)×,(a•,Int×)×)×
:- pred split(pair(pair(A,B),pair(A,B)),pair(A,B),pair(A,B)).
:- mode split(
        bound(mk_pair(bound(mk_pair(unique, ground)),
                      bound(mk_pair(unique, ground))))
    >> clobbered,
        free
    >> bound(mk_pair(unique, ground)),
        free
    >> bound(mk_pair(unique, ground))) is det.
split(mk_pair(X,Y),X,Y).

% fst :: (a•,b×)× → a•
:- pred fst(pair(A,B),A).
:- mode fst(
        bound(mk_pair(unique, ground))
    >> clobbered,
        free
    >> unique) is det.
fst(mk_pair(X,_),X).
```

Figure 3.10: Duplication of unique objects in Mercury

3.6.4 Bunched Implications

The logic of Bunched Implications (BI), introduced in (O'Hearn and Pym, 1999), allows reasoning about *local* sharing. Its use as a type system is explored in (O'Hearn, 2003) and a PhD thesis (Armelín, 2002), although the latter focuses exclusively on its use as a type system for a *logic* (rather than functional) programming language.

BI distinguishes between two kinds of function spaces (denoted by $a \longrightarrow b$ and $a \mathrel{-\!*} b$). There are no restrictions on applying "additive" functions of type $a \longrightarrow b$, but a "multiplicative" (linear) function f of type $a \mathrel{-\!*} b$ can only be applied to arguments disjoint from the elements in its closure (thus, the system guarantees that there is no sharing between f and e in $f\ e$). Similarly, the system distinguishes between additive pairs ($a \wedge b$) and multiplicative pairs ($a * b$). For a multiplicative pair $(e_1, e_2) :: a * b$, BI guarantees that there is no sharing between e_1 and e_2.

However, this local sharing is not sufficient to allow destructive updates in a language, unless that language is in CPS form (Berdine and O'Hearn, 2006) (not unlike linear logic; see quote p. 88 from Wadler in Section 3.3.1). For example, the following function is type correct in BI:

```
first :: (a → b) -* a → a ∧ b
first f x = (x, f x)
```

but if the f passed to `first` destructively modifies its argument, use of `first` means loosing referentially transparency. Although the type of `first` indicates that there is sharing between the components of the result pair, we are unable to express in BI that the arguments to a function must not be shared; thus, we cannot prohibit passing in functions that modify their argument to `first`.

3.6.5 Separation logic

Separation logic is a general class of logics which can be used to "describe the separation of storage into disjoint components" in imperative languages (Reynolds, 2002). This can be used for example to verify that the parallel composition of two assignment statements is well-formed. That is, we want to allow the parallel assignment to two disjoint variables:

```
x := 5 || y := 10
```

but disallow parallel assignment to the same variable:

```
x := 5 || x := 10
```

These statements are said to *interfere*. Interference can be controlled using a substructural type system without contraction where the rule for parallel composition splits the typing environment into two. The earliest separation logics were based on linear logic (O'Hearn, 1991); the ! operator of linear logic is used to type *passive* terms that cannot have any side effects, and hence can never interfere. More recent separation logics use the logic of bunched implications instead (Section 3.6.4). The literature on separation logic is large, and a survey is beyond the scope of this dissertation; (Reynolds, 2002) is a good starting point for readers who want to know more.

3.6.6 Type and effect systems

Type and effect systems record the *type* of an expression, as well as the *effect* of the expression in a relation that takes the form

$$\Gamma \vdash e : \sigma, \phi$$

for some type σ and effect ϕ. The exact nature of the effects depends on the specific type system; typically, the language includes some primitive operations with predefined effects. The typing rules for the other constructs then simply collect effects, with the exception of the abstraction rule which is defined as[1]

$$\frac{\Gamma, x : \sigma' \vdash e : \sigma, \phi}{\Gamma \vdash \lambda x \cdot e : \sigma' \xrightarrow{\phi} \sigma, \varnothing}\ \textsc{Abs}$$

The effects in the body of the function are recorded in the type of the function and are delayed until the function is applied:

$$\frac{\Gamma \vdash f : \sigma' \xrightarrow{\phi''} \sigma, \phi \qquad \Gamma \vdash e : \sigma', \phi'}{\Gamma \vdash f\ e : \sigma, \phi \cup \phi' \cup \phi''}\ \textsc{Abs}$$

One common effect is the allocation, modification or deallocation of *regions* (areas of memory) in imperative languages such as Cyclone (Grossman et al., 2002) or functional languages such as SAFE (Peña et al., 2007).

Type and effect types also provide a solution to the well-known problem of polymorphic references. Impure functional languages such as ML or the recent DDC project (Lippmeier, 2008) offer modifiable references. Such a reference is created using

```
let r :: ref Int
    r = new 5
in ..
```

which creates a reference with initial contents "5". This reference can later be modified (in-place). It is clear that adding references violates purity, but less obvious is that it may also violate soundness. For example, consider

```
let r :: ∀ a. ref [a]
    r = new []
in r := [1] ;
   if head (deref r) then .. else ..
```

Since the empty list has type $\forall a \cdot [a]$, this becomes the type of the reference also. Both the update to r and the dereference of r are therefore type correct; but of course, taken together they will cause a program error.

The conclusion is that the types of multiple references should not be generalized, but keeping track of which type variables can and which type variables cannot be generalized is non-trivial. This is especially true for references inside function closures. As Leroy and Weis (1991) observe:

> "The problem is that the types of the free variables of a function do not appear in the type of the function. In this context, it is useful to think of functions as closures. Closures, in contrast with any other data structure, are not adequately described by their (functional) type: we do not know anything about the types of the values contained in the environment part of the closure.
>
> Our solution is to keep track of what is inside a closure. We associate with any function type a set of types, the types of all variables in the function.

[1] We explain only the call-by-value case here; see (Henglein et al., 2005) for the call-by-name case.

By recording modification of references as effects, generalization of the “dangerous” type variables can be avoided:

$$\frac{\Gamma \vdash e : \sigma, \phi \qquad a \notin \mathrm{ftv}(\Gamma) \cup \mathrm{ftv}(\phi)}{\Gamma \vdash e : \forall a \cdot \sigma}\ \textsc{Gen}$$

As an aside, note that in a purely functional programming language this problem does not arise. Even if the language supports arrays that can be modified in-place and even if the language supports “empty” arrays of a polymorphic type, then a modification of that array would return a (conceptually) *new* array of a monomorphic type and the original polymorphic array can no longer be used.

Type and effect systems are usually not used by pure functional programming languages, and are not typically regarded as substructural type systems (although they *can* be, as Fluet (2007) shows in this doctoral thesis). We included this section mainly because we too will have a need to p. 115 be more precise about the types of the elements in a function closure (Chapter 4). A more detailed discussion is therefore beyond the scope of this dissertation; we refer the reader to (Henglein et al., 2005) for more information on type and effect systems, or to (Garrigue, 2004) for a brief survey of solutions to the polymorphic references problem.

Scaling Uniqueness Typing to Arbitrary Rank Types*

We modify *Clean*'s uniqueness type system in two ways. First, while Clean functions that are partially applied to a unique argument are *necessarily* unique (they cannot lose their uniqueness), we just require that they must be unique *when applied*. This ultimately makes subtyping redundant. Second, we extend the type system to allow for higher-rank types. To be able to do this, we explicitly associate type constraints (attribute inequalities) with type schemes. Consequently, types in our system are much more precise about constraint propagation.

Since the typing rules for rank-1 are easier to understand than the typing rules for arbitrary rank, we first present the rank-1 typing rules in Section 4.1 and then extend them to arbitrary rank in Section 4.2. We consider a few examples in Section 4.3, outline a type inference algorithm in p. 118, p. 123
Section 4.4, compare our system to the original Clean type system in Section 4.5, and conclude p. 125, p. 126
this chapter in Section 4.6. p. 128

4.1 Rank-1 typing rules

We will present a uniqueness type system that allows for rank-1 types only, before showing the full type system in Section 4.2. Although both the expression language and the type language p. 118
must be modified to support arbitrary rank types, the typing rules as presented in this section are easier to understand and provide a better way to introduce the type system.

4.1.1 The Language

We define our type system over a core lambda calculus:

e ::=		expression
	$x^{\odot}, x^{\otimes}$	variable (exclusive, shared)
	$\lambda x \cdot e$	abstraction
	$e\ e$	application
	i	integer

The typing rules assign an attributed type τ^{ν} to an expression e, given a type environment Γ and a uniqueness attribute ν_{γ} (explained in Section 4.1.4), denoted p. 117

$$\Gamma, \nu_{\gamma} \vdash e : \tau^{\nu}$$

*The material in this chapter was published as *Uniqueness Typing Redefined* in Proceedings of the International Symposium on the Implementation and Application of Functional Languages (IFL) 2006, Zoltán Horváth, Viktória Zsók and Andrew Butterfield (Eds.), Lecture Notes in Computer Science volume 4449 (de Vries et al., 2007b).

The language of types and uniqueness attributes is defined as

τ	::=		type	ν	::=		uniqueness attribute
		t, s	type variable			u, v	variable
		$\tau^{\nu} \underset{\nu_a}{\longrightarrow} \tau'^{\nu'}$	function			$\bullet$	unique
		Int	constant type			$\times$	non-unique

The syntax for arrows (function space constructor) warrants a closer look. The domain and codomain of the arrow are two attributed types τ^{ν} and $\tau'^{\nu'}$. The arrow itself has an *additional* attribute ν_a, whose role will become apparent when we discuss the rule for abstractions. We will adopt the notational convention of writing $(\tau^{\nu} \underset{\nu_a}{\longrightarrow} \tau'^{\nu'})^{\nu_f}$, where ν_f is "ordinary" uniqueness attribute of the arrow, as $(\tau^{\nu} \overset{\nu_f}{\underset{\nu_a}{\longrightarrow}} \tau'^{\nu'})$.

As is customary, all type and attribute variables in an attributed type τ^{ν} are implicitly universally quantified at the outermost level (of course, this will not be true for the arbitrary rank system). In this section, a type environment maps variable names to attributed types (in Section
p. 118 4.2, it will map variable names to type schemes).

4.1.2 Integers

We can choose between three possible rules for integers; we can assume integers are unique (rule INT$^{\bullet}$), non-unique (INT$^{\times}$), or we can leave the uniqueness unspecified (INT):

$$\frac{}{\Gamma, \nu_\gamma \vdash i : \mathrm{Int}^{\bullet}}\ \textsc{Int}^{\bullet} \qquad \frac{}{\Gamma, \nu_\gamma \vdash i : \mathrm{Int}^{\times}}\ \textsc{Int}^{\times} \qquad \frac{}{\Gamma, \nu_\gamma \vdash i : \mathrm{Int}^{\nu}}\ \textsc{Int}$$

A case can certainly be made that unique integers (that is, integers that can be updated in-place) are not particularly useful; moreover, base types *must* be considered non-unique if Clean-style
p. 83 read-only access to unique variables is supported (Section 3.2.6)[1,2]. However, even if unique
p. 117 integers are considered useful, rule INT$^{\bullet}$ is very restrictive, as we shall see in Section 4.1.4. We will therefore assume rule INT in the remainder of this chapter.

4.1.3 Variables

To find the type of the variable, we look up the variable in the environment, correcting the type to be non-unique for shared variables:

$$\frac{}{(\Gamma, x : \tau^{\nu}), \nu_\gamma \vdash x^{\odot} : \tau^{\nu}}\ \textsc{Var}^{\odot} \qquad \frac{}{(\Gamma, x : \tau^{\nu}), \nu_\gamma \vdash x^{\otimes} : \tau^{\times}}\ \textsc{Var}^{\otimes}$$

Note that VAR$^{\otimes}$ leaves the uniqueness attribute of the variable *in the environment* arbitrary. This means that variables can "lose" their uniqueness. For example, the function `dup` defined as $\lambda x \cdot (x^{\otimes}, x^{\otimes})$ has type $t^{u} \longrightarrow (t^{\times}, t^{\times})$ (assuming a product type); in other words, no matter what the uniqueness of *a* on input is, each *a* in the pair will be non-unique.

[1] Clean uses type information as a heuristic to detect aliasing (Section 3.2.6): the result of a strict-let-before must be of primitive type. This heuristic breaks if unique elements of primitives types are supported. See also Section 8.2.9.

[2] The Clean manual specifies that arguments of a basic type are stored on the stack and therefore is does not make sense to make them unique (Plasmeijer and van Eekelen, 2002, Section 9.7, *Destructive updates using uniqueness typing*). One might consider cases however where integers are used to model areas in memory for low-level hardware interfacing. In such circumstances it may be useful to offer unique integers, as it is important to update that area of memory in-place.

4.1.4 Abstractions

Consider the following function that writes two characters to a file:

```
f file = (write_char⊗ file⊗ 'a', write_char⊗ file⊗ 'b')
```

The reference count analysis (Section 3.2.7) has marked the two uses of `file` as shared, which p. 84 will cause its type to be inferred as non-unique by rule VAR$^{\otimes}$. Since `write_char` expects a unique file, the type checker can reject this program. But what happens if we partially apply `write_char`?

```
f file = let g = write_char⊙ file⊙ in (g⊗ 'a', g⊗ 'b')
```

Both programs are semantically equivalent, so the type-checker should reject both. However, the argument `file` to `write_char` is in fact exclusive in the second example, so how can we detect the type error? This is of course the partial application problem we discussed before in Section 3.2.4. To recap, the general principle is p. 82

> *when a function accesses unique objects from its closure, that closure (i.e., the function) must be unique itself* $(*)$

In the example above, `g` accesses a unique file from its closure, and must therefore be unique itself—but is not, resulting in a type error. We can approximate[1] $(*)$ by

> *if a function is partially applied, and the supplied argument is unique, the resulting function must be unique when applied* $(*')$

In the lambda calculus, functions only take a single argument, and the notion of currying translates into lambda abstractions returning new lambda abstractions. Thus, we can rephrase $(*')$ as

> *if a lambda abstraction returns a new lambda abstraction, and the argument to the outer lambda abstraction is unique, the inner lambda abstraction must be unique when applied* $(*'')$

In our type language, the additional "closure" attribute ν_a in the arrow type $\tau_1^{\nu_1} \xrightarrow[\nu_a]{} \tau_2^{\nu_2}$ indicates whether the function is required to be "unique when applied". It is essentially a form of closure typing (Leroy and Weis, 1991; Garrigue, 2004), except that we do not need to be quite so precise: we do not need to know the types of all the variables in the closure of the function, but only require to know whether any of those variables is unique.

The purpose of ν_γ in the typing rules is to indicate whether we are currently in the body of an (outer) lambda abstraction whose argument must be unique. Thus we arrive at rule ABS:

$$\frac{(\Gamma, x : \tau^{\nu}), \nu_{\gamma'} \vdash e : \tau'^{\nu'} \qquad \nu_a \leq \nu_\gamma, \nu_{\gamma'} \leq \nu, \nu_{\gamma'} \leq \nu_\gamma}{\Gamma, \nu_\gamma \vdash \lambda x \cdot e : \tau^{\nu} \xrightarrow[\nu_a]{\nu_f} \tau'^{\nu'}} \ \textsc{Abs}$$

This rule is very similar to the conventional rule for abstractions in a Hindley/Milner type system, with the exception of the attribute inequalities in the premise of the rule. The $u \leq v$ operator can be read as an implication: if v is unique, then u must be unique (v implies u, $u \leftarrow v$).

[1] This is an approximation since the function may not use the curried argument. In $\lambda x \cdot \lambda y \cdot y^{\odot}$, x is not used in the body of the function, so its uniqueness need not affect the type of the function.

The first constraint establishes the conclusion of $(*'')$: if we are in the body of an outer lambda abstraction whose argument must be unique (ν_γ), then the inner lambda abstraction must be unique when applied (ν_a). The second constraint $\nu_{\gamma'} \le \nu_1$ is a near direct translation of the premise of $(*'')$. Finally, $\nu_{\gamma'} \le \nu_\gamma$ simply propagates ν_γ: if the premise of $(*'')$ already holds (ν_γ), it will continue to do so in the body of the abstraction ($\nu_{\gamma'}$). ABS is the only rule that changes the value of ν_γ; all the other rules simply propagate it. When typing an expression, ν_γ is initially assumed to be non-unique.

It is instructive to consider an example at this point. We show the type derivation for $\lambda x \cdot \lambda y \cdot x^{\odot}$, the function that returns the first of its two arguments:

$$\dfrac{\dfrac{\dfrac{}{(x : t^u, y : s^v), u_{\gamma''} \vdash x^{\odot} : t^u \qquad u_{a'} \le u_{\gamma'}, u_{\gamma''} \le v, u_{\gamma''} \le u_{\gamma'}}\textsc{Var}^{\odot}}{(x : t^u), u_{\gamma'} \vdash \lambda y \cdot x^{\odot} : s^v \xrightarrow[u_{a'}]{u_{f'}} t^u \qquad u_a \le \times, u_{\gamma'} \le u, u_{\gamma'} \le \times}\textsc{Abs}}{\emptyset, \times \vdash \lambda x \cdot \lambda y \cdot x^{\odot} : t^u \xrightarrow[u_a]{u_f} (s^v \xrightarrow[u_{a'}]{u_{f'}} t^u)}\textsc{Abs}$$

Since $u_a \le \times$ and $u_{\gamma'} \le \times$ are vacuously true, $u_{\gamma''} \le v$ and $u_{\gamma''} \le u_{\gamma'}$ are irrelevant as $u_{\gamma''}$ does not constrain any other attributes, and $u_{a'} \le u_{\gamma'}$ and $u_{\gamma'} \le u$ imply that $u_{a'} \le u$ (by transitivity), we can simplify this type to

$$\lambda x \cdot \lambda y \cdot x^{\odot} : t^u \xrightarrow[u_a]{u_f} (s^v \xrightarrow[u_{a'}]{u_{f'}} t^u) \qquad u_{a'} \le u$$

where the constraint $u_{a'} \le u$ says that if we curry the function (specify x but not y), and x happens to be unique, the result function must be unique on application ($u_{a'}$ must be $\bullet$).

If we now consider rule INT$^\bullet$, which says that integers are always unique, this definition of ABS would imply that if we curry a function by passing in an integer, the result function must be unique on application, which is unnecessary. For example, we want the following expression to be type correct:

$$\text{let fst} = \lambda x \cdot \lambda y \cdot x \text{ in let one} = \text{fst } 1 \text{ in } (\text{one } 2, \text{one } 3)$$

For the same reason, nothing in ABS constrains ν_f, and the actual uniqueness of the function is left free. In summary, assigning a unique type to an expression that does not need one is unnecessarily restrictive.

4.1.5 Application

The rule for function application is relatively straightforward. The only difference between the rule as presented here and the usual definition is that APP enforces the constraint that functions that must be unique when applied, are unique when applied ($\nu_f \le \nu_a$):

$$\dfrac{\Gamma, \nu_\gamma \vdash e : \tau^\nu \xrightarrow[\nu_a]{\nu_f} \tau'^{\nu'} \qquad \Gamma, \nu_\gamma \vdash e' : \tau^\nu \qquad \nu_f \le \nu_a}{\Gamma, \nu_\gamma \vdash e\, e' : \tau'^{\nu'}}\textsc{App}$$

4.2 Arbitrary Rank Types

p. 38 As explained in Section 2.4.4, the rank of a type is the depth at which universal quantifiers appear in the domain of functions. In most cases, universal quantifiers appear only at the outermost level.

For example,

$$\text{id} : \forall a.a \rightarrow a$$

is a type of rank 1. In higher-rank types, we have nested universal quantifiers. For example (Peyton Jones et al., 2007),

$$g : (\forall a.[a] \rightarrow [a]) \rightarrow ([\text{Bool}], [\text{Int}]) = \lambda f.\, (f\ [\text{True}, \text{False}], f\ [1,2,3])$$

In this example, g requires a function f that works on lists of type $[a]$ for all a (the rank of the type of g is 2). Type *inference* is undecidable for types with rank $n > 2$ (Wells, 1999), but we can support type inference by combining type inference with type checking. Thus, higher-rank types are only supported when function arguments are given an explicit type signature. We extend the expression language with annotated lambda expressions[1]:

$$\begin{array}{lll} e & \mathrel{+}= & & \text{expression } \textit{(ctd.)} \\ & & \lambda(x :: \sigma) \cdot e & \text{annotated abstraction} \end{array}$$

In the rank-1 system presented in Section 4.1 (as well as in Clean's system), constraints are never p. 115
explicitly associated with types but are left implicit in the typing rules. This makes the types simpler but this approach does not scale to higher rank types. When we generalize a type τ^ν to a type scheme σ, τ^ν may be constrained by a set of constraints $\mathcal{C}$. Those constraints should be associated with the type scheme σ, because if at a later stage we instantiate σ to get a type $\tau'^{\nu'}$, the same set of constraints should apply to $\tau'^{\nu'}$. This makes the types more complicated, but it also makes them more precise (see Sections 4.5 and 4.6). So, we define a type scheme as p. 126, p. 128

$$\begin{array}{lll} \sigma & ::= & \forall \overline{a}.\tau^\nu, \mathcal{C} & \text{type scheme} \end{array}$$

where $\overline{a}$ is a set of type and uniqueness variables, and $\mathcal{C}$ is set of constraints or a constraint variable. We modify the type language to allow for type schemes in the domain of the arrow. We follow (Peyton Jones et al., 2007) and do not allow for type schemes in the codomain:

$$\begin{array}{llll} \tau & ::= & & \text{type} \\ & & t, s & \text{type variable} \\ & & \sigma \xrightarrow[\nu_a]{} \tau^\nu & \text{arrow type (functions)} \\ & & \text{Int} & \text{constant type} \end{array}$$

Typing derivations now have the structure

$$\Gamma, \nu_\gamma \vdash e : \tau^\nu \mid \mathcal{C}$$

which says that e has type τ^ν, given an environment Γ and uniqueness attribute ν_γ (see Section 4.1.4), provided constraints $\mathcal{C}$ are satisfied (where environments now map variable names to p. 117
type schemes). The full typing rules are listed in Fig. 4.1; we will explain them separately below. p. 120

[1] The paper also gives a rule for `let` expressions, but unfortunately the rule as shown is wrong. Since the `let` expression introduces a new variable into a function closure, the variable bound by the `let` expression must be taken into account when determining ν_γ (in a similar vein to the rule for abstraction). However, rather than correcting the rule, we will delay a discussion of `let` expressions until the next chapter.

$$\frac{}{\Gamma, \nu_\gamma \vdash i : \mathrm{Int}^\nu \mid \emptyset} \qquad \textsc{Int}$$

$$\frac{\vdash^{\mathrm{inst}} \sigma \preceq \tau^\nu \mid \mathcal{C}}{(\Gamma, x : \sigma), \nu_\gamma \vdash x^{\odot} : \tau^\nu \mid \mathcal{C}} \qquad \textsc{Var}^{\odot}$$

$$\frac{\vdash^{\mathrm{inst}} \sigma \preceq \tau^\nu \mid \mathcal{C}}{(\Gamma, x : \sigma), \nu_\gamma \vdash x^{\otimes} : \tau^{\times} \mid \mathcal{C}} \qquad \textsc{Var}^{\otimes}$$

$$\frac{(\Gamma, x : \forall.\tau^\nu, \mathcal{C}_1), \nu_{\gamma'} \vdash e : \tau'^{\nu'} \mid \mathcal{C}_2}{\Gamma, \nu_\gamma \vdash \lambda x \cdot e : (\forall.\tau^\nu, \mathcal{C}_1) \xrightarrow[\nu_a]{\nu_f} \tau'^{\nu'} \mid \mathcal{C}_2, \nu_a \leq \nu_\gamma, \nu_{\gamma'} \leq \nu_\gamma, \nu_{\gamma'} \leq \nu} \qquad \textsc{Abs}$$

$$\frac{\Gamma, \nu_\gamma \vdash e : \sigma_1 \xrightarrow[\nu_a]{\nu_f} \tau^\nu \mid \mathcal{C} \qquad \Gamma, \nu_\gamma \vdash^{\mathrm{gen}} e' : \sigma_2 \qquad \vdash^{\mathrm{subs}} \sigma_2 \preceq \sigma_1}{\Gamma, \nu_\gamma \vdash e\, e' : \tau^\nu \mid \mathcal{C}, \nu_f \leq \nu_a} \qquad \textsc{App}$$

$$\frac{(\Gamma, x : \sigma), \nu_{\gamma'} \vdash e : \tau^\nu \mid \mathcal{C}}{\Gamma, \nu_\gamma \vdash \lambda(x :: \sigma) \cdot e : \sigma \xrightarrow[\nu_a]{\nu_f} \tau^\nu \mid \mathcal{C}, \nu_a \leq \nu_\gamma, \nu_{\gamma'} \leq \nu_\gamma, \nu_{\gamma'} \leq \lceil \sigma \rceil} \qquad \textsc{Annot}$$

$$\frac{\Gamma, \nu_\gamma \vdash e : \tau^\nu \mid \mathcal{C} \qquad \overline{a} = \mathrm{ftuv}(\tau^\nu) - \mathrm{ftuv}(\Gamma)}{\Gamma, \nu_\gamma \vdash^{\mathrm{gen}} e : \forall \overline{a}.\tau^\nu, \mathcal{C}} \qquad \textsc{Gen}$$

$$\frac{}{\vdash^{\mathrm{inst}} \forall \overline{a}.\tau^\nu, \mathcal{C} \preceq \mathcal{S}_x \tau^\nu \mid \mathcal{S}_x \mathcal{C}} \qquad \textsc{Inst}$$

$$\frac{\overline{b} \notin \mathrm{ftuv}(\forall \overline{a}.\tau^\nu) \qquad \vdash^{\mathrm{subs}} S_x \tau^\nu \preceq \tau'^{\nu'} \qquad C_2 \vDash \mathcal{S}_x C_1}{\vdash^{\mathrm{subs}} \forall \overline{a}.\tau^\nu, \mathcal{C}_1 \preceq \forall \overline{b}.\tau'^{\nu'}, \mathcal{C}_2} \qquad \textsc{Subs}^{\sigma}$$

$$\frac{\vdash^{\mathrm{subs}} \sigma_2 \preceq \sigma_1 \qquad \vdash^{\mathrm{subs}} \forall.\tau_1^{\nu_1}, \emptyset \preceq \forall.\tau_2^{\nu_2}, \emptyset}{\vdash^{\mathrm{subs}} \sigma_1 \rightarrow \tau_1^{\nu_1} \preceq \sigma_2 \rightarrow \tau_2^{\nu_2}} \qquad \textsc{Subs}^{\rightarrow}$$

$$\frac{}{\vdash^{\mathrm{subs}} \tau^\nu \preceq \tau^\nu} \qquad \textsc{Subs}^{\tau}$$

Figure 4.1: Uniqueness Typing Rules

4.2.1 Variables

Because the type environment now associates variable names with type schemes rather than types, to find the type of a variable we must look up the associated type scheme in the environment, and instantiate it. Instantiation is defined as

$$\frac{}{\vdash^{\text{inst}} \forall\overline{a}.\tau^{\nu}, \mathcal{C} \preceq \mathcal{S}_x\tau^{\nu} \mid \mathcal{S}_x\mathcal{C}} \;\textsc{Inst}$$

where S_x is some substitution $[\overline{a} \mapsto \ldots]$ mapping all variables $\overline{a}$ to fresh variables. Since we associate a set of constraints $\mathcal{C}$ with a type scheme, a type $S_x\tau^{\nu}$ is only an instance of a type scheme σ if those constraints are satisfied.

4.2.2 Abstraction

The rule for abstraction remains unchanged except for the domain of the arrow operator which is now a type scheme. However, since we can only infer rank-1 types, the type scheme for unannotated lambda expressions must be a "degenerate" type scheme with no quantified variables $(\forall.\tau^{\nu}, \mathcal{C})$—in other words, a type.[1]

4.2.3 Application

The rule for application looks slightly different from the rank-1 version. Previously, with App the type of the actual parameter had to equal the type of the formal parameter of the function:

$$\frac{\Gamma, \nu_\gamma \vdash e : \tau^{\nu} \xrightarrow[\nu_a]{\nu_f} \tau'^{\nu'} \qquad \Gamma, \nu_\gamma \vdash e' : \tau^{\nu} \qquad \nu_f \leq \nu_a}{\Gamma, \nu_\gamma \vdash e\ e' : \tau'^{\nu'}} \;\textsc{App}_1$$

In the rank-n case, the only requirement is that the type of the actual parameter is an instance of the type of the formal parameter. To this end, we infer a type scheme for the actual parameter, and do a subsumption check:

$$\frac{\Gamma, \nu_\gamma \vdash e : \sigma_1 \xrightarrow[\nu_a]{\nu_f} \tau^{\nu} \mid \mathcal{C} \qquad \Gamma, \nu_\gamma \vdash^{\text{gen}} e' : \sigma_2 \qquad \vdash^{\text{subs}} \sigma_2 \preceq \sigma_1}{\Gamma, \nu_\gamma \vdash e\ e' : \tau^{\nu} \mid \mathcal{C}, \nu_f \leq \nu_a} \;\textsc{App}$$

(We will explain subsumption separately in Section 4.2.5.) To infer a type scheme, we first infer a p. 122
type, and then generalize over all the free variables in the type, excluding the free variables in the environment:

$$\frac{\Gamma, \nu_\gamma \vdash e : \tau^{\nu} \mid \mathcal{C} \qquad \overline{a} = \text{ftuv}(\tau^{\nu}) - \text{ftuv}(\Gamma)}{\Gamma, \nu_\gamma \vdash^{\text{gen}} e : \forall\overline{a}.\tau^{\nu}, \mathcal{C}} \;\textsc{Gen}$$

[1] In (Peyton Jones et al., 2007) the arrow $\longrightarrow$ is overloaded; there is an arrow $\tau \longrightarrow \tau$ and an arrow $\sigma \longrightarrow \tau$. Since we do not use the notion of ρ–types, our arrows always have type $\sigma \longrightarrow \tau^{\nu}$. Strictly speaking this makes our inference algorithm incomplete (though not unsound): while the typing rules allow to instantiate a type variable by a type of the form $\sigma \longrightarrow \tau$ for a non-degenerate type scheme σ, the type inferencer will never do this. We show the type system without ρ-types only to simplify the presentation.

4.2.4 Annotated Lambda Abstractions

The rule for annotated lambda abstractions is similar to the rule for "ordinary" lambda abstractions, except that programmers can now specify a type scheme manually, allowing for higher-rank types:

$$\frac{(\Gamma, x : \sigma), \nu_{\gamma'} \vdash e : \tau^{\nu} \mid C}{\Gamma, \nu_{\gamma} \vdash \lambda(x :: \sigma) \cdot e : \sigma \xrightarrow[\nu_a]{\nu_f} \tau^{\nu} \mid C, \nu_a \leq \nu_{\gamma}, \nu_{\gamma'} \leq \nu_{\gamma}, \nu_{\gamma'} \leq \lceil \sigma \rceil} \text{ ANNOT}$$

We have to be careful defining $\lceil \forall \overline{a}.\tau^{\nu} \rceil$, used to constrain $\nu_{\gamma'}$. The obvious answer (ν) is only correct if ν is not itself universally quantified. For example, consider the rank-2 type

$$\lambda(x :: \forall u.t^u) \cdot \lambda y \cdot x^{\odot} : (\forall u.t^u) \xrightarrow[u_a]{u_f} s^v \xrightarrow[u_{a'}]{u_{f'}} t^u, ?$$

What should the constraint at the question mark be? One possible solution is

$$\forall u \cdot u_{a'} \leq u$$

but that is equivalent to saying

$$u_{a'} \leq \bullet$$

So, to avoid unnecessary complication by introducing universal quantification into the constraint language, we define $\lceil\ \rceil$ as

$$\lceil \forall \overline{a}.\tau^{\nu} \rceil = \begin{cases} \nu & \text{if } \nu \notin \overline{a} \\ \bullet & \text{otherwise} \end{cases}$$

4.2.5 Subsumption

The rules for subsumption are defined as in (Peyton Jones et al., 2007), except that we have collapsed rules SKOL and SPEC into one rule (SUBS$^{\sigma}$) and added one additional premise. SUBS$^{\sigma}$ is the main rule that checks whether one type scheme is a (generic) instance of another.

$$\frac{\overline{b} \notin \mathrm{ftuv}(\forall \overline{a}.\tau^{\nu}) \qquad \vdash^{\mathrm{subs}} S_x \tau^{\nu} \preceq \tau'^{\nu'} \qquad C_2 \vDash S_x C_1}{\vdash^{\mathrm{subs}} \forall \overline{a}.\tau^{\nu}, C_1 \preceq \forall \overline{b}.\tau'^{\nu'}, C_2} \text{ SUBS}^{\sigma}$$

In a standard type system, as here, a type scheme $\sigma_1 = \forall \overline{a}.\tau_1$ is at least as polymorphic as another type scheme $\sigma_2 = \forall \overline{b}.\tau_2$ if a unifier S_x can be found that instantiates τ_1 to an *arbitrary* instantiation of τ_2 (guaranteed by $\overline{b} \notin \mathrm{ftuv}(\forall \overline{a}.\tau_1^{\nu_1})$). In our system, however, we need an additional constraint $C_2 \vDash S_x C_1$, which is best explained by example. Consider

$$f : (\forall u, v \cdot t^u \xrightarrow[u_a]{u_f} s^v) \rightarrow \ldots$$

$$g : t^u \xrightarrow[u_a]{u_f} s^v, [u \leq v]$$

Should the application $f\ g$ type-check? Intuitively, f expects to be able to use the function it is passed to obtain an s with uniqueness v (say, a unique s), independent of the uniqueness of t. However, g only promises to return a unique s if t is also unique! Thus, the application $f\ g$ should be disallowed.

Conversely, if we instead define f' and g' as

$$f' : (\forall u, v \cdot t^u \xrightarrow[u_a]{u_f} s^v, [u \leq v]) \rightarrow \ldots$$
$$g' : t^u \xrightarrow[u_a]{u_f} s^v$$

the application $f' \, g'$ *should* be allowed because the type of g' is more general than the type expected by f'. The condition $\mathcal{C}_2 \vDash S_x\mathcal{C}_1$, where the $\vDash$ symbol stands for logical entailment from propositional logic, means that if constraints $\mathcal{C}_2$ are satisfied, constraints $\mathcal{C}_1$ must also be satisfied[1]. In other words, the constraints of the offered type must be the same or less restrictive than the constraints of the requested type.

4.3 Examples

In this section we consider a few example expressions and their associated types. We start with very simple expressions and slowly build up from there. First, we consider a single integer:

$$5 : \forall u.\mathrm{Int}^u, \emptyset$$

Rule INT says that integers have type Int with an arbitrary uniqueness, hence the universally quantified u. Next we consider the identity function `id`:

$$\lambda x.x^{\odot} : \forall t, u, u_f, u_a, c.(\forall.t^u, c) \xrightarrow[u_a]{u_f} t^u, c$$

This type may appear more complicated than it really is, because we show top-level attributes and degenerate type schemes; we can be slightly less formal:

$$\lambda x.x^{\odot} : (t^u, c) \xrightarrow[u_a]{u_f} t^u, c$$

Either way, this is the type one would expect an identity function to have. Note that this function is polymorphic in the constraints of its argument: if the argument has type t^u under constraints c, then the result has type t^u only if the same set of constraints is satisfied.

The function `apply` ($ in Haskell) behaves like `id` restricted to function types:

$$\lambda f.\lambda x.f^{\odot}\, x^{\odot} :: \left((t^u, c_1) \xrightarrow[u_{a''}]{u_{f''}} s^v, c_2\right) \xrightarrow[u_a]{u_f} \left((t^u, c_1) \xrightarrow[u_{a'}]{u_{f'}} s^v\right), [c_2, \\ u_{a'} \leq u_{a''}, u_{a'} \leq u_{f''}, u_{f''} \leq u_{a''}]$$

This is a complicated type, and in later chapters we will consider how to simplify the type system. For now, we consider each constraint in turn:

c_2 — If f has type $(t^u, c_1) \xrightarrow[u_{a''}]{u_{f''}} s^v$ only when constraints c_2 are satisfied, then `apply` f also has that type only when those constraints are satisfied (*cf.* the constraint c in the type of `id`).

[1] If either $\mathcal{C}_1$ or $\mathcal{C}_2$ in $\mathcal{C}_1 \vDash \mathcal{C}_2$ is a constraint variable, we apply unification instead of the entailment check.

$u_{a'} \leq u_{a''}$ If f can only be executed once (in other words, if f must be unique on application, if $u_{a''}$ is unique), then `apply` f can also only be executed once.

$u_{a'} \leq u_{f''}$ If f is unique, then `apply` f can only be executed once; this is a direct consequence of the "partial application" from Section 4.1.4.

$u_{f''} \leq u_{a''}$ Finally, `apply` f applies f, so if f must be unique on application, we require that it is unique.

The next example emphasizes a point with respect to the reference count analysis. Suppose that we have a primitive type Array and two functions *resize* to (destructively) resize the array, and *size* to return the current size of the array:

$$\begin{aligned} &\mathit{resize} : \mathrm{Array}^{\bullet} \xrightarrow[u_a]{u_f} \mathrm{Int}^{v} \xrightarrow[\bullet]{u_{f'}} \mathrm{Array}^{\bullet} \\ &\mathit{size} : \mathrm{Array}^{u} \xrightarrow[u_a]{u_f} \mathrm{Int}^{v} \end{aligned}$$

Then the following expression is correctly marked and type correct:

$$\lambda \mathit{arr} \cdot \text{if } \mathit{size}^{\odot}\ \mathit{arr}^{\otimes} < 10 \text{ then } \mathit{resize}^{\otimes}\ \mathit{arr}^{\odot}\ 20 \text{ else } \mathit{resize}^{\otimes}\ \mathit{arr}^{\odot}\ 30$$

This expression is marked correctly, because only one of the two branches of the conditional expression will be executed, and the shared mark $\mathit{arr}^{\otimes}$ in the condition guarantees that the p. 83 condition cannot modify *arr* (we are taking advantage of read-only access; see Section 3.2.6).

To conclude this section, we consider two examples that contain a type error, which in both cases will be detected in the subsumption check (although for different reasons). The first example shows a simple case of an argument not being polymorphic enough:

$$\begin{aligned} &\text{let } \mathit{id}_{\mathrm{f}} = \lambda(f :: \forall u.t^{u} \xrightarrow[u_a]{u_f} t^{u}) \cdot f^{\odot} \\ &\text{in let } \mathit{id}_{\mathrm{int}} = \lambda(i :: \mathrm{Int}^{\bullet}) \cdot i^{\odot} \\ &\text{in } \mathit{id}_{\mathrm{f}}^{\odot}\ \mathit{id}_{\mathrm{int}}^{\odot} \end{aligned}$$

Here, id_{f} demands that its argument is polymorphic in u, but $\mathit{id}_{\mathrm{int}}$ is not (it works only on unique integers). The problem is detected when we do the subsumption check

$$\forall.\mathrm{Int}^{\bullet} \xrightarrow[u_a]{u_f} \mathrm{Int}^{\bullet} \overset{?}{\preceq} \forall u.t^{u} \xrightarrow[u_{a'}]{u_{f'}} t^{u}$$

We have to check that we can unify $\mathrm{Int}^{\bullet}$ and t^{u} for *an arbitrary instantiation* of u, but that will clearly fail[1]. The second "incorrect" example that we consider fails due to the entailment check p. 122 explained in Section 4.2.5:

$$\begin{aligned} &\text{let } \mathit{first} = \lambda(f :: t^{u} \xrightarrow[u_a]{u_f} s^{v} \xrightarrow[u_{a'}]{u_{f'}} t^{u}) \cdot \lambda x \cdot \lambda y \cdot f^{\odot}\ x^{\odot}\ y^{\odot} \\ &\text{in } \mathit{first}^{\odot}\ (\lambda x \cdot \lambda y \cdot x^{\odot}) \end{aligned}$$

[1] The implementation of $\textsc{Subs}^{\sigma}$ will have instantiated u with a fresh "skolem constant": an unknown, but fixed, uniqueness attribute. These skolem constants are the "rigid variables" known from, for example, `ghc`, and the type error the user will get is `Cannot unify rigid attribute` u `and` $\bullet$.

The function that is passed as an argument to *first* has type[1]

$$\lambda x \cdot \lambda y \cdot x^{\odot} : t^u \xrightarrow[u_a]{u_f} s^v \xrightarrow[u_{a'}]{u_{f'}} t^u, [u_{a'} \leq u]$$

whereas the type specified for the argument f of *first* does not allow for the constraint $u_{a'} \leq u$; so, the type-checker will fail with

`[] does not entail` $[u_{a'} \leq u]$

4.4 Type Inference

We have written a prototype implementation of the type system presented in this chapter. The typing rules as presented in Fig. 4.1 allow for a relatively straightforward translation to an p. 120 algorithm $\mathcal{W}$ style (Figure 2.11) type-checker (our prototype is just under a thousand lines long) p. 45 once the following subtleties have been observed.

When doing unification, a unification goal $\tau^{\nu} \overset{?}{\approx} \tau'^{\nu'}$ should be expanded into two subgoals $\tau \overset{?}{\approx} \tau'$ and $\nu \overset{?}{\approx} \nu'$. In other words, the base types and the uniqueness attributes should be unified independently.

Unification should not be used to unify functions because, as far as unification is concerned, $\sigma_1 \to \tau_1^{\nu_1} \overset{?}{\approx} \sigma_2 \to \tau_2^{\nu_2}$ is the same as $\sigma_2 \to \tau_2^{\nu_2} \overset{?}{\approx} \sigma_1 \to \tau_1^{\nu_1}$, but to compare two type schemes we need to use subsumption, which clearly gives different answers for $\vdash^{\text{subs}} \sigma_1 \preceq \sigma_2$ and $\vdash^{\text{subs}} \sigma_2 \preceq \sigma_1$. However, when properly implemented, by the time we need unification, the subsumption rules (in particular, $\text{SUBS}^{\to}$) will have taken care of all arrows[2].

To implement the subsumption check, the technique suggested by Peyton Jones (Peyton Jones et al., 2007) of using skolem constants can be applied, introducing skolem constants both type and uniqueness variables (see Section 2.5.3).

Logical entailment of two sets of constraints $\mathcal{C}_1$ and $\mathcal{C}_2$ can be implemented as a validity check for the propositional logic formula $\text{C}_1 \to \text{C}_2$, where the $u \leq v$ operator is regarded as an implication $v \to u$. Although the complexity of checking the validity of functions in propositional logic is exponential, that will not matter much in practice since the formulae generated by the type-checker will be small (most type schemes will not have many associated constraints). A simple algorithm (the one we have implemented) to check the validity of a formula in propositional logic is to convert the formula to conjunctive normal form, inspect every conjunct and search for atoms in the conjunct such that the conjunct contains the atom and its negation. If such a match is found for all conjuncts, the formula is valid (see Huth and Ryan, 2004, Section 1.5, for details).

Finally, when generalizing a type τ^{ν} with respect to a set of constraints $\mathcal{C}$, the set should be checked for inconsistencies; these should be reported as type errors. For improved readability of types, it is also useful to take the transitive closure of $\mathcal{C}$ instead of $\mathcal{C}$ itself, and add only the "relevant" inequalities to the type scheme (rule ABS might generate unnecessary constraints $[\nu_{\gamma'} \leq \nu_{\gamma}, \nu_{\gamma'} \leq \nu_1]$ if $\nu_{\gamma'}$ is never used to constrain other attributes); this is demonstrated in the example in Section 4.1.4. p. 117

[1] There are additional "polymorphic" constraint variables in these types that we are leaving out for conciseness.

[2] In (Peyton Jones et al., 2007), due to the distinction between ρ functions and τ functions, unification must still deal with arrows $\tau \to \tau$; since we only have one arrow type, this is unnecessary in our approach.

4.5 Comparison with Clean

The uniqueness type system presented here is based on that of the programming language *Clean* (Barendsen and Smetsers, 1993a, 1996), which is in turn strongly related to substructural logics (see (Wadler, 1993) for an accessible introduction to linear logic; (Walker, 2005) is a good introduction to substructural type systems). However, there are a number of important differences, one being that Clean's system is defined over graph rewrite rules rather than the lambda calculus; this gives the type system a very different "feel".

A rather more important difference is the treatment of partially applied functions. In Clean, a function that is (partially) applied to a unique argument, is itself unique. Moreover, unique functions are *necessarily unique*: they cannot lose their uniqueness. In the curry example in Section 4.1.4, there are two references to the partially applied function (`g`), which is therefore marked as $\otimes$. The type correction in rule VAR$^{\otimes}$ (a trivial operation in our system) must check whether the variable represents a function, and if so, reject the program. While this solves the curried function problem, it has far reaching consequences for the type system.

The first is that type variables, as well as functions, are not allowed to lose their uniqueness, since a type variable can be instantiated to a function type. In Clean, for example, the function `dup` has type

$$\lambda x \cdot (x^{\otimes}, x^{\otimes}) : t^{\times} \rightarrow (t^{\times}, t^{\times})$$

and not

$$\lambda x \cdot (x^{\otimes}, x^{\otimes}) : t^{u} \rightarrow (t^{\times}, t^{\times})$$

The type assigned by Clean is not as restrictive at is seems, however, due to Clean's subtyping relation: a unique type is considered to be *subtype* of its non-unique counterpart. For example, the following is a correct Clean program:

```
five :: Int•
five = 5

dup :: t^× → (t^×,t^×)
dup x = (x, x)

Start = dup five
```

where `Start` is assigned the type (Int$^{\times}$, Int$^{\times}$). Of course, the subtyping relation is adapted for
p. 80 arrows (Section 3.2.2):

$$\tau_a^{\nu_a} \xrightarrow{\nu} \tau_b^{\nu_b} \leq \tau_c^{\nu_c} \xrightarrow{\nu'} \tau_d^{\nu_d} \qquad \Leftrightarrow \nu = \nu' \text{ and } \tau_a^{\nu_a} \leq^{\ominus} \tau_c^{\nu_c} \text{ and } \tau_b^{\nu_b} \leq^{\oplus} \tau_d^{\nu_d}$$

There are two things to note about this definition: a unique function is never a subtype of its non-unique version (condition $\nu = \nu'$), since functions are not allowed to lose their uniqueness (a similar restriction applies to type variables); and subtyping is contravariant in the function argument. Although this is not surprising, it complicates the type system—especially in the
p. 128 presence of algebraic data types. We have not discussed ADTs in this chapter (see Section 4.6), but they are easy to add to our system. However, algebraic data constructors can include arrows, for example

```
data Fun a b = Fun (a → b)
```

which means that arguments to constructors must be analyzed to check whether they have covariant, contravariant or invariant subtyping behaviour. An additional complication is that Clean supports abstract data types, where the types of the constructors are hidden. Unfortunately, the contravariant/covariant behaviour of the data type cannot be independently specified, breaking the abstraction boundary.

By contrast, in our system we do not have the notion of "necessarily unique"; instead we add a single additional attribute ν_a as explained before, and the condition that (some) curried functions can only be executed once becomes a local constraint $\nu_f \leq \nu_a$ in the rule for function application. There are no global effects (for example, type variables are unaffected) and we do not need subtyping[1].

That last point is worth emphasizing. The subtyping relation in Clean is very shallow. The only advantage of subtyping is that we can pass in a unique object to a function that expects a non-unique object. So, in Clean, marking a formal parameter as non-unique really means, "I do not care about the uniqueness of this parameter". However, in our system, we can always use an attribute variable to mean the same thing.[2] That is not always possible in Clean, since type variables are not allowed to lose their uniqueness (the type we assign to the function `dup` above would be illegal in Clean).

Since we do not have subtyping, functions can specify that their arguments must be unique ($t^\bullet$), non-unique ($t^\times$), or indicate that the uniqueness of the input does not matter (t^u). In Clean, it is only possible to specify that arguments must be unique ($t^\bullet$) or that the uniqueness of an argument does not matter (t^u or, due to subtyping, $t^\times$). Experience will tell whether this extra functionality is useful.

Another consequence is mentioned in (Barendsen and Smetsers, 1996, Section *Uniqueness Type Inference*):

> *However, because of our treatment of higher-order functions (involving a restriction on the subtype relation w.r.t. variables), it might be the case that lifting this most general solution fails, whereas some specific instance is attributable. [...] Consequently, there is no "Principal Uniqueness Type Theorem".*

Since we can correct the type of a shared variable to be non-unique, no matter what the type of that variable is, our approach does not suffer from this drawback. This also facilitates type inference: when we see a variable marked as shared, we can immediately return the correct type for that variable. In Clean, we might not yet know what that type is, as the type of the variable may still be a meta-variable. If that meta-variable is later instantiated with a base type (such as an array of integers), then the uniqueness correction will succeed, but if it is bound to a function type or if the variable is universally quantified (when generalization a type), the uniqueness correction must fail.

[1] One might argue that subsumption introduces subtyping between type schemes; however, due to the predicative nature of our type system, this does not have an effect on algebraic data type arguments; see the discussion in (Peyton Jones et al., 2007, Section 7.3).

[2] Note that while the use of type variables to model subtyping has precedent in the literature (e.g., Leijen, 2004, 2007b), the situation in Clean is particularly simple due to the shallow depth of the subtyping relation.

4.6 Notes

We have designed a uniqueness type system for the lambda calculus that can be used to add side effects to a pure functional language without losing referential transparency. This type system is based on the type system of the functional programming language *Clean*, but modifies it in a number of ways. First, it is defined over the lambda calculus rather than a graph rewrite system. Second, our treatment of curried functions is completely different and makes the type system much simpler; in particular, there is no need for subtyping. Third, our system supports arbitrary rank types, associating constraints with type schemes.

The system as presented in this chapter deals only with the core lambda calculus; however, extensions to deal with algebraic data types and recursive definitions are straightforward. For recursive definitions $\mu \cdot e$, the type of e is corrected to be non-unique (this is the same approach as taken in (Barendsen and Smetsers, 1996) for `letrec` expressions). The main principle in dealing with algebraic data types is that if a unique object is extracted from an enclosing container, the enclosing container must in turn be unique (this is a slightly more permissive definition than the one used in Clean, which requires that a container must be unique *when it is constructed* if any of
p. 206 its elements are unique; we will come back to this point in Section 8.1.4).

p. 125 The inference algorithm described briefly in Section 4.4 is based on algorithm $\mathcal{W}$ and inherits its associated problems, in particular unhelpful error messages. An investigation into better approaches is future work; the constraint based algorithm proposed by Heeren looks promising (Heeren et al., 2002).

The formalization of the constraint language in this chapter is not as precise as it could be. A better approach might be to recast the type system in terms of the qualified types framework
p. 72 (Section 2.9.2). However, that will still not help to give precise definitions of the constraint
p. 117 simplification step shown in Section 4.1.4. Moreover, the constraints significantly complicate the types. In the next two chapters will we will see how to remove the need for constraints completely.

In the explanation of the rule for abstractions ABS in Section 4.1.4 we mentioned that our method of constraining v_a is conservative. For example, the constraint $u_{a'} \leq u$ in

$$\lambda x \cdot \lambda y \cdot y^{\odot} : (t^u, c_1) \xrightarrow[u_a]{u_f} (s^v, c_2) \xrightarrow[u_{a'}]{u_{f'}} s^v, [c_2, u_{a'} \leq u]$$

is not actually necessary since x is not referenced in $\lambda y \cdot y^{\odot}$. Hence, it may be possible to relax the rules to be less conservative. This would only affect how v_a in ABS is established; it would not change the type language. This too will be considered in the next chapter.

Removing Inequality Constraints[*]

Uniqueness types often involve implications between uniqueness attributes, which complicates type inference and incorporating modern extensions such as arbitrary rank types. In this chapter we show how to avoid these difficulties by recoding attribute inequalities as attribute equalities.

5.1 Typing the core $\lambda-$calculus

We present a uniqueness type system for the lambda calculus which does not involve implications. The expression and type language are defined in Figure 5.1. The expression language is the p. 130
standard lambda calculus, except that variables are marked as exclusive ($x^{\odot}$) or shared ($x^{\otimes}$). The type language includes base types, type variables and the function space, all of which get a uniqueness attribute indicating whether there is more than one reference to a term. The domain and codomain of the arrow (function space constructor) are both attributed types, and the arrow itself gets *two* attributes: the "normal" uniqueness attribute ν (indicating whether there is more than one reference to the function) and an additional "closure attribute" ν_c, which is the disjunction of all attributes on the types of the elements in the closure of the function (Section 5.1.2).

We treat a uniqueness attribute as a boolean expression, reading True (unique) for "$\bullet$" and False (not unique) for "$\times$", and allow for arbitrary boolean expressions involving variables, negation, conjunction and disjunction[1]. This definition of attributes is different from their definition in Chapter 4 where (like in *Clean*, Section 3.2) we only allow for unique, non-unique and variables. p. 115, p. 78
It may not be immediately obvious why this is useful, but the improvements of the system as presented in this chapter over the previous are made possible by this one change.

The typing relation itself takes the form

$$\Gamma \vdash e : \tau^{\nu}|_{fv}$$

which reads as "in environment Γ, expression e has attributed type τ^{ν}; the attributes on the types of the free variables in e are *fv*". We represent *fv* as a relation $Var \times Attribute$; its purpose will become clear when we discuss the rule for abstraction in Section 5.1.2. The environment maps p. 130
expression variables to attributed types.

Note the conspicuous absence of constraints in the type language. We will explain how we deal with this when we discuss the individual typing rules.

[1] Although the typing rules only introduce disjunctions, unification may introduce more complicated types also involving conjunctions and negation; we will come back to this point in Section 8.2.1.

[*] The material in this chapter was presented as *Equality-Based Uniqueness Typing* at the Eighth Symposium on Trends in Functional Programming, 2007 (de Vries et al., 2007a).

$e ::=$	expression	$\nu ::=$	attribute
$x^{\odot}$	variable (exclusive)	u	attribute variable
$x^{\otimes}$	variable (shared)	$\bullet$	unique
$\lambda x \cdot e$	abstraction	$\times$	non-unique
$e\,e$	application	$\neg\nu$	negation
		$\nu_1 \wedge \nu_2$	conjunction
$\tau ::=$	base type	$\nu_1 \vee \nu_2$	disjunction
B	constant type		
t, s	type variable		
$\tau^{\nu} \underset{\nu_c}{\longrightarrow} \tau'^{\nu'}$	function space		

Figure 5.1: Expression and type language for the core system

5.1.1 Variables

To check that a variable x marked as exclusive has attributed type τ^{ν}, we simply look up the variable in the environment[1]. For shared variables, we need to correct the type found in the environment to be non-unique. In both cases we also record the uniqueness attribute of the type of the variable (see Section 5.1.2).

$$\frac{}{\Gamma, x : \tau^{\nu} \vdash x^{\odot} : \tau^{\nu}|_{(x,\nu)}}\ \textsc{Var}^{\odot} \qquad \frac{}{\Gamma, x : \tau^{\nu} \vdash x^{\otimes} : \tau^{\times}|_{(x,\times)}}\ \textsc{Var}^{\otimes}$$

p. 116 VAR$^{\otimes}$ does not require the type *in the environment* to be non-unique. As we saw in Section 4.1.3 in the previous chapter, this effectively means that variables can lose their uniqueness. Consider again the function *dup* $= \lambda x \cdot (x^{\otimes}, x^{\otimes})$. Both components of the pair point to the same element, which is therefore non-unique by definition. Thus the type of *dup* is

$$\text{dup} :: t^{u} \xrightarrow[\times]{u_f} (t^{\times}, t^{\times})^{v}$$

The attributes on the arrow will be explained in the next section.

5.1.2 Abstraction

p. 82, p. 117 Once again (Section 3.2.4, Section 4.1.4) we must consider the problem of partial application. Consider the function that returns the first of its two arguments:

$$\text{const} = \lambda x \cdot \lambda y \cdot x^{\odot}$$

Temporarily ignoring the attributes on arrows, *const* has type

$$\text{const} :: t^{u} \longrightarrow s^{v} \longrightarrow t^{u}$$

Given *const*, what would be the type of

$$\text{sneakyDup} = \lambda x \cdot \text{ let } f = \text{const } x^{\odot} \text{ in } (f^{\otimes}\,1, f^{\otimes}\,2)$$

[1] When a variable usage is marked as exclusive, that does not automatically make its type unique; for example, the identity function $\lambda x \cdot x^{\odot}$ has type $t^{u} \longrightarrow t^{u}$, not $t^{\bullet} \longrightarrow t^{\bullet}$ (or even $t^{u} \longrightarrow t^{\bullet}$). In other words, reference count analysis just notes that there is only one reference to x in the body of the identity function; however, when a non-unique argument is passed to *id*, it will still be non-unique when it is returned again.

It would seem that since f has type $b^v \longrightarrow a^u$, this term has type

$$\text{sneakyDup} :: t^u \longrightarrow (t^u, t^u)^w$$

but this is clearly wrong: the elements in the result pair are shared and should be non-unique.

Recall from the Section 3.2.3 that if we want to extract a unique element from a container, the container must be unique itself. When we execute a function, the function can extract elements from its closure (the environment which binds the free variables in the function body). If any of those elements is unique, executing the function will involve extracting unique elements from a container (the closure), which must therefore be unique itself. Since we do not distinguish between a function and its closure in the lambda calculus, this means that the function must be unique. Thus a function needs to be unique on application (that is, a function can be applied only once) if the function can access unique elements from its closure. p. 81

The function type must therefore be modified to indicate whether there are any unique elements in the closure of the function; this is the purpose of the second uniqueness attribute on arrows (ν_c). We described this in the previous chapter (Section 4.1.4), but needed to use constraints to define the closure attribute. Using boolean attributes, however, we can simply use the disjunction of all the attributes on the types of the (used) elements in the closure of the function. p. 117

Going back to the example, the full type of f in the definition of *sneakyDup* is therefore

$$f :: s^v \xrightarrow[u]{u_f} t^u$$

One way to read this type is that if you want a unique a to be returned from f, f must be unique on application. In the definition of *sneakyDup*, f is not unique ($u_f = \times$) when applied since it is marked as shared, so the actual type of *sneakyDup* is (see also Section 5.1.5): p. 132

$$\text{sneakyDup} :: t^\times \xrightarrow[\times]{u_f} (t^\times, t^\times)^w$$

It should now be clear why the typing rules record the attributes on the free variables in an expression: we need this information to determine ν_c. Using $\bigvee \mathit{fv}$ to denote the disjunction of all attributes in the range of fv, and $⩥_x\, \mathit{fv}$ (domain subtraction) to denote fv with x removed from its domain, we obtain

$$\frac{\Gamma, x : \tau^\nu \vdash e : \tau'^{\nu'}|_{\mathit{fv}}}{\Gamma \vdash \lambda x \cdot e : \tau^\nu \xrightarrow[\bigvee(⩥_x\, \mathit{fv})]{\nu_f} \tau'^{\nu'}|_{⩥_x\, \mathit{fv}}} \quad \textsc{Abs}$$

We must remove x from fv because x is not free in $\lambda x \cdot e^1$.

5.1.3 Application

As we have seen, some functions must be unique on application; this is enforced in the typing rule for application. In Section 4.1.5, we used an inequality constraint: p. 118

$$\frac{\Gamma \vdash e_1 : \tau^\nu \xrightarrow[\nu_c]{\nu_f} \tau'^{\nu'}|_{\mathit{fv}} \quad \Gamma \vdash e_2 : \tau^\nu|_{\mathit{fv}'} \quad \nu_f \leq \nu_c}{\Gamma \vdash e_1\, e_2 : \tau'^{\nu'}|_{\mathit{fv} \cup \mathit{fv}'}} \quad \textsc{ConstrApp}$$

[1] We use the Barendregt convention and assume that all bound variables are distinct.

The implication $[\nu_f \leq \nu_c]$ (ν_c implies ν_f) expresses the requirement that the function must be unique (ν_f) if it has any unique elements in its closure (ν_c). How can we model the requirement $\nu_f \leq \nu_c$ without using constraints? The easiest solution is to require that $\nu_f = \nu_c$:

$$\frac{\Gamma \vdash e_1 : \tau^\nu \xrightarrow[\nu_c]{\nu_c} \tau'^{\nu'}|_{fv} \quad \Gamma \vdash e_2 : \tau^\nu|_{fv'}}{\Gamma \vdash e_1\, e_2 : \tau'^{\nu'}|_{fv \cup fv'}} \; \textsc{App}$$

While this rule is technically more restrictive than ConstrApp, in practice the programmer will not notice the difference. We will discuss this issue in more depth in Section 5.1.5.

5.1.4 Examples

We discuss two examples. First, we consider the type of *apply* $= \lambda f \cdot \lambda x \cdot f^{\odot} x^{\odot}$:

$$\text{apply} :: (t^u \xrightarrow[u_c]{u_c} s^v) \xrightarrow[\times]{u_f} t^u \xrightarrow[u_c]{u_{f'}} s^v$$

Unsurprisingly, *apply* takes a function f from a to b, and a term of type a, and returns a term of type b. Since *apply* f applies f, if f must be unique on application, it must be unique when passed as an argument to *apply* (in the type of *apply*, this requirement is encoded by specifying that f must have the same attribute below and above the arrow). Finally, if f is unique, then *apply* f must be unique on application, since it can extract a unique element from its closure (to wit, f).

As a second example, consider the definition of *split* ($\triangle$)

$$f \triangle g = \lambda x \cdot (f\,x, g\,x)$$

The type of $\triangle$ is

$$\triangle :: (t^\times \xrightarrow[u_1]{u_1} s^v) \xrightarrow[\times]{u_f} (t^\times \xrightarrow[u_2]{u_2} r^w) \xrightarrow[u_1]{u_{f'}} t^u \xrightarrow[u_1 \vee u_2]{u_{f''}} (s^v, t^w)^z$$

In words, $\triangle$ wants two functions f and g which return a b^v and a c^w given a non-unique a, and returns a pair of type $(b^v, c^w)^z$. If either f or g must be unique on application, they must be unique when they are passed as arguments to $\triangle$ as $\triangle$ applies them. Finally, $f \triangle g$ must itself be unique on application when either f or g is unique, because $f \triangle g$ will then be able to extract unique elements from its closure (i.e., f and g) when it is applied. A function such as

$$\text{clearArray} :: \text{Array}^\bullet \xrightarrow[\times]{u_f} \text{Array}^\bullet$$

cannot be passed as an argument to $\triangle$ since Array$^\bullet$ does not unify with $a^\times$.

5.1.5 Reflection on the core system

In general, we can always recode a type of the form

$$\ldots \Box^u \quad \ldots \Box^v \ldots, [u \leq v]$$

using a disjunction

$$\ldots \Box^{u \vee v} \ldots \Box^v \ldots$$

This faithfully models the implication: when v is unique, $u \vee v$ reduces to unique, but when v is non-unique, $u \vee v$ reduces to u. For example, in Clean the function `fst` that extracts the first element of a pair has the type

$$\begin{array}{l}\texttt{fst :: } (t^u,s^v)^w \longrightarrow t^u, [w \leq u] \\ \texttt{fst (x, y) = x}\end{array}$$

which we can recode as

$$\texttt{fst :: } (t^u,s^v)^{w \vee u} \longrightarrow t^u$$

However, in many cases we can do slightly better. For example, suppose the typing rule for pairs is

$$\frac{\Gamma \vdash e_1 : \tau^{\nu}|_{fv} \qquad \Gamma \vdash e_2 : \tau'^{\nu'}|_{fv'}}{\Gamma \vdash (e_1, e_2) : (\tau^{\nu}, \tau'^{\nu'})^{\nu''}|_{fv \cup fv'}} \quad \text{PAIR}$$

then for every derivation of $e :: (\tau^{\nu}, \tau'^{\nu'})^{\bullet}$, there is also a derivation of $e :: (\tau^{\nu}, \tau'^{\nu'})^{\times}$ (because the typing rule leaves the attribute on the pair free). That means that we can simplify the type of `fst` to

$$\texttt{fst' :: } (t^u,s^v)^u \longrightarrow t^u$$

The only pairs accepted by `fst` but rejected by `fst'` are unique pairs, but since the type checker will never infer a pair to be unique (but always either non-unique or polymorphic in its uniqueness), that situation will never arise.

In Section 5.1.3, we removed the implication from CONSTRABS by replacing the implication $[\nu_f \leq \nu_c]$ by $[\nu_f = \nu_c]$. Again, it is possible to remove the implication *without* giving a more restrictive rule by using disjunction with a free variable: p. 131

$$\frac{\Gamma \vdash e_1 : \tau_1^{\nu_1} \xrightarrow[\nu_c]{\nu_f \vee \nu_c} \tau_2^{\nu_2}|_{fv} \quad \Gamma \vdash e_2 : \tau_1^{\nu_1}|_{fv'}}{\Gamma \vdash e_1\, e_2 : \tau_2^{\nu_2}|_{fv \cup fv'}} \quad \text{APP}'$$

We nevertheless prefer rule APP (requiring that $\nu_f = \nu_c$), since it leads to more readable types. For example, based on rule APP′, *split* would have the type

$$\Delta :: (t^{\times} \xrightarrow[u_{t_1}]{u_{f_1} \vee u_{t_1}} s^v) \xrightarrow[\times]{u_f} (t^{\times} \xrightarrow[u_{t_2}]{u_{f_2} \vee u_{t_2}} r^w) \xrightarrow[u_{f_1} \vee u_{t_1}]{u_{f'}} t^u \xrightarrow[u_{f_1} \vee u_{t_1} \vee u_{f_2} \vee u_{t_2}]{u_{f''}} (s^v, r^w)^z$$

We claimed that rule APP is not as restrictive as it may seem; the argument proceeds along the same lines as the argument for the simplified type of `fst`. An expression will be rejected by APP but allowed by APP′ if and only if the function that we are applying is unique, but does not have any unique elements in its closure; so, if we have an expression $f\ x$ where f has type

$$f :: t^u \xrightarrow[\times]{\bullet} s^v$$

Clearly • does not unify with ×, so rule APP will reject this application. The corresponding error message will be a bit mystifying: "*The function you are applying is too unique. Use it more often.*" Of course, this is a consequence of replacing the implication by an equality. However, barring type annotations, that type will never be assigned to a term. Instead, the following type would be inferred:

$$f :: t^u \xrightarrow[\times]{u_f} s^v$$

That is, the function will be polymorphic in its uniqueness rather than actually be unique; none of the typing rules even mention • anywhere. The typing rules force terms to be non-unique if they are shared, but they never force them to be unique. Given the latter type of f, rule App has no difficulty typing the application, since u_f trivially unifies with $\times$.

The reader might wonder why it is useful to distinguish between the uniqueness of the function and the (disjunction of) the uniqueness of the elements in the closure of the function, if we insist that they must be the same on application. It is possible to collapse these two attributes, and use the uniqueness attribute of the function for both. Then a function must be unique if it has any unique elements in its closure, and must *remain* unique. This is the approach taken in *Clean*, but it p. 130 complicates the type system. For example, the function *dup* from Section 5.1.1 must be assigned the type $t^\times \xrightarrow[\times]{u_f} (t^\times, t^\times)^v$ instead of $t^u \xrightarrow[\times]{u_f} (t^\times, t^\times)^v$, because the latter type would allow us to duplicate a function with unique elements in its closure (and then apply that function twice). To make this type less restrictive, *Clean* then introduces subtyping, but that brings complications p. 126 of its own (this is discussed in more detail in Section 4.5). By distinguishing between the two attributes (which really do represent different properties of the function), the requirement that a function with unique elements in its closure must be unique on application becomes a local requirement in the rule for application, and does not complicate the rest of the type system.

Finally, the reader may have expected the type of *sneakyDup* to be the same as the type of *dup*. The reason that it is not (but is more restrictive) is due to rule $\textsc{Var}^\odot$. When a variable usage is marked as exclusive, rule $\textsc{Var}^\odot$ states that the type of the variable is equal to the type listed for the variable in the environment. It is possible to relax this rule to

$$\frac{}{\Gamma, x : \tau^{\nu \vee \nu'} \vdash x^\odot : \tau^{\nu}|_{(x,\nu)}} \quad \textsc{Var}'^{\odot}$$

With this rule a variable whose type is listed as non-unique in the environment must be non-unique (as before), but when the type of the variable in the environment is listed as unique, this rule places no restrictions on the attribute of the type derived for x (if $\nu \vee \nu' = \bullet$, either ν or ν' must be unique, but they do not both have to be unique). With $\textsc{Var}'^\odot$ *sneakyDup* will indeed have the same type as *dup*, but at a cost: types become more complicated. For example, the identity function would have the type

$$\lambda x \cdot x^\odot :: t^{u \vee v} \xrightarrow[\times]{u_f} t^u$$

which is correct, but perhaps more difficult to understand than the type derived for the identity function using rule $\textsc{Var}^\odot$ ($t^u \xrightarrow[\times]{u_f} t^u$).

p. 143 In Chapter 6 we will consider an alternative approach which enables us to use a simple rule for variables and still get identical types for *dup* and *sneakyDup*, and which will make it possible to remove the closure attribute on arrows without reintroducing the need for subtyping.

5.1.6 Type inference

One advantage of removing constraints from the type language is that standard inference algorithms such as algorithm $\mathcal{W}$ (Damas and Milner, 1982) can be applied without any modifications. The inference algorithm uses a unification algorithm, which must be modified in two ways.

- It must be adapted to deal with boolean expressions (boolean unification was explained in p. 25 Section 2.2.4).

- It must treat a unification goal $\tau_1^{\nu_1} \overset{?}{\approx} \tau_2^{\nu_2}$ as two separate goals $\tau_1 \overset{?}{\approx} \tau_2$ and $\nu_1 \overset{?}{\approx} \nu_2$ (in other words, base types and their attributes must be unified independently).

In Chapter 6 we will remove the distinction between types and uniqueness attributes so that we p. 143 regard an attributed type τ^ν as syntactic sugar for the application of a special type constant `Attr` to two variables (`Attr` τ ν). This will make the second modification redundant.

5.2 Arbitrary Rank Types

We claim that our core uniqueness system is sufficiently similar to a standard Hindley/Milner type systems that modern extensions can be added without much difficulty. To substantiate this claim we show in this section how to extend the core type system to support arbitrary rank types using the techniques described in a recent paper by Peyton Jones *et al.* (Peyton Jones et al., 2007); in Section 5.3, we will show how to support GADTs too. p. 140

Section 5.2.1 recalls what arbitrary rank types are and outlines how they are dealt with. Section 5.2.2 explains how we must modify the typing rules from (Peyton Jones et al., 2007) to deal with p. 136 uniqueness, and Section 5.2.4 explains why these modifications are much simpler in a system p. 138 without inequalities than in a system with inequalities.

5.2.1 Arbitrary rank types

The core type system described in Section 5.1 does not have an explicit notion of universal p. 129 quantification. When we say that the identity function has the type

$$\mathrm{id} = \lambda x \cdot x :: t^u \xrightarrow[\times]{u_f} t^u$$

what we mean is that for any instantiation of t, u and u_f, the identity function has that (instantiated) type. But this is a meta-level notion: the type language defined in Figure 5.1 does not allow p. 130 universal quantification over type or uniqueness variables.

We can make universal quantification an object-level notion by introducing "type schemes": (attributed) types together with a list of universally quantified type (and uniqueness) variables.

$\sigma ::= \forall \overline{t}, \overline{u}.\tau^\nu$ type scheme

The typing rules can then be modified to assign a type scheme, rather than a type, to an expression. For example, the type scheme assigned to *id* would be

$$\lambda x \cdot x :: \forall t\, u\, u_f.t^u \xrightarrow[\times]{u_f} t^u$$

So far we have not gained much by introducing type schemes, but we can go one step further. We can modify the type language so that the domain of the function type constructor becomes a type scheme σ, rather than an (attributed) type τ^ν:

$\tau ::=$	base type
B	constant type
t	type variable
$\sigma \xrightarrow[\nu_c]{} \tau^\nu$	function space

$$\frac{\vdash^{\mathrm{inst}\sigma}_{\delta} \sigma \leq \tau^{\nu}}{\Gamma, x : \sigma \vdash_{\delta} x^{\odot} : \tau^{\nu}|_{(x,\nu)}}\ \textsc{Var}^{\odot} \qquad \frac{\vdash^{\mathrm{inst}\sigma}_{\delta} \sigma \leq \tau^{\nu}}{\Gamma, x : \sigma \vdash_{\delta} x^{\otimes} : \tau^{\times}|_{(x,\times)}}\ \textsc{Var}^{\otimes}$$

$$\frac{\Gamma, x : \forall.\tau^{\nu} \vdash_{\Uparrow} e : \tau'^{\nu'}|_{\mathit{fv}}}{\Gamma \vdash_{\Uparrow} \lambda x \cdot e : \forall.\tau^{\nu} \xrightarrow[\vee \mathrel{|\!\rhd}_x \mathit{fv}]{u_f} \tau'^{\nu'}|_{\mathrel{|\!\rhd}_x \mathit{fv}}}\ \textsc{Abs}^{\tau} \qquad \frac{\Gamma, x : \sigma \vdash_{\Downarrow} e : \tau^{\nu}|_{\mathit{fv}}}{\Gamma \vdash_{\Downarrow} \lambda x : \sigma \cdot e \xrightarrow[\vee \mathrel{|\!\rhd}_x \mathit{fv}]{u_f} \tau^{\nu}|_{\mathrel{|\!\rhd}_x \mathit{fv}}}\ \textsc{Abs}^{\sigma}$$

$$\frac{\Gamma \vdash_{\Uparrow} e : \sigma \xrightarrow[\nu_c]{\nu_c} \tau'^{\nu'}|_{\mathit{fv}} \quad \Gamma \vdash^{\mathrm{gen}}_{\Downarrow} e' : \sigma|_{\mathit{fv}'} \quad \vdash^{\mathrm{inst}\tau}_{\delta} \tau'^{\nu'} \leq \tau^{\nu}}{\Gamma \vdash_{\delta} e\ e' : \tau^{\nu}|_{\mathit{fv} \cup \mathit{fv}'}}\ \textsc{App}$$

$$\frac{\Gamma \vdash^{\mathrm{gen}}_{\Downarrow} e : \sigma|_{\mathit{fv}} \quad \vdash^{\mathrm{inst}\sigma}_{\delta} \sigma \leq \tau^{\nu}}{\Gamma \vdash_{\delta} (e :: \sigma) : \tau^{\nu}|_{\mathit{fv}}}\ \textsc{Ann} \qquad \frac{\Gamma \vdash^{\mathrm{gen}}_{\Uparrow} e : \sigma|_{\mathit{fv}} \quad \Gamma, x : \sigma \vdash_{\delta} e' : \tau^{\nu}|_{\mathit{fv}'}}{\Gamma \vdash_{\delta} \mathrm{let}\ x = e\ \mathrm{in}\ e' : \tau^{\nu}|_{\mathit{fv} \cup (\mathrel{|\!\rhd}_x \mathit{fv}')}}\ \textsc{Let}$$

$$\frac{(\overline{t}, \overline{u}) = \mathrm{ftuv}(\tau^{\nu}) - \mathrm{ftuv}(\Gamma, fv) \qquad \Gamma \vdash_{\delta} e : \tau^{\nu}|_{\mathit{fv}}}{\Gamma \vdash^{\mathrm{gen}}_{\delta} e : \forall \overline{t}, \overline{u}.\tau^{\nu}|_{\mathit{fv}}}\ \textsc{Gen}$$

$$\frac{\Gamma \vdash_{\delta} e : \tau^{\nu}|_{\mathit{fv} \cup (x,u)}}{\Gamma \vdash_{\delta} e : \tau^{\nu}|_{\mathit{fv} \cup (x,\bullet)}}\ \textsc{Fv} \qquad \frac{\vdash^{\mathrm{inst}\rho}_{\delta} [\overline{t \mapsto \tau}][\overline{u \mapsto v}]\tau^{\nu} \leq \tau'^{\nu'}}{\vdash^{\mathrm{inst}\sigma}_{\delta} \forall \overline{t}, \overline{u}.\tau^{\nu} \leq \tau'^{\nu'}}\ \textsc{Inst}^{\sigma}$$

$$\frac{}{\vdash^{\mathrm{inst}\rho}_{\Uparrow} \tau^{\nu} \leq \tau^{\nu}}\ \textsc{Inst}^{\Uparrow} \qquad \frac{\vdash^{\mathrm{subs}} \tau^{\nu} \leq \tau'^{\nu'}}{\vdash^{\mathrm{inst}\rho}_{\Downarrow} \tau^{\nu} \leq \tau'^{\nu'}}\ \textsc{Inst}^{\Downarrow}$$

$$\frac{(\overline{t}, \overline{u}) \notin \mathrm{ftuv}(\sigma) \quad \vdash^{\mathrm{subs}} \sigma \leq \tau^{\nu}}{\vdash^{\mathrm{subs}} \sigma \leq \forall \overline{t}, \overline{u}.\tau^{\nu}}\ \textsc{Skol} \qquad \frac{\vdash^{\mathrm{subs}} [\overline{t \mapsto \tau}][\overline{u \mapsto v}]\tau_1^{\nu_1} \leq \tau_2^{\nu_2}}{\vdash^{\mathrm{subs}} \forall \overline{t}, \overline{u}.\tau_1^{\nu_1} \leq \rho_2^{\nu_2}}\ \textsc{Spec}$$

$$\frac{\vdash^{\mathrm{subs}} \sigma_2 \leq \sigma_1 \quad \vdash^{\mathrm{subs}} \tau_1^{\nu_1} \leq \tau_2^{\nu_2}}{\vdash^{\mathrm{subs}} (\sigma_1 \xrightarrow[\nu_c]{\nu_f} \tau_1^{\nu_1}) \leq (\sigma_2 \xrightarrow[\nu_c]{\nu_f} \tau_2^{\nu_2})}\ \textsc{Fun} \qquad \frac{}{\vdash^{\mathrm{subs}} \tau^{\nu} \leq \tau^{\nu}}\ \textsc{Mono}$$

Figure 5.2: Arbitrary rank typing rules

p. 118 As explained in Section 4.2, this change gives us a lot more expressive power in the type system. We saw that type inference for higher rank types (types with nested universal quantifiers) is undecidable, but we can support higher rank types by combining type inference with type checking. We therefore use a slightly modified typing relation

$$\Gamma \vdash_{\delta} e : \tau^{\nu}|_{\mathit{fv}}$$

p. 129 This is different from the typing relation in Section 5.1 in two ways: the environment Γ now maps expression variables to type schemes (not types), and we introduce a "typing mode" δ: $\Uparrow$ for type inference or $\Downarrow$ for type checking. The full typing rules are shown in Figure 5.2. A detailed p. 46 discussion of the arbitrary rank typing rules was given in Section 2.5.3. Here we emphasize only that the typing rules shown in Figure 5.2 (which support uniqueness) closely resemble the original typing rules (which do not). We highlight the differences in the next section.

5.2.2 Modifications to deal with uniqueness

Our starting point is the "bidirectional" typing rules from (Peyton Jones et al., 2007, Fig. 8), except for simplicity of presentation we do not show the rules for annotated lambda abstractions (rules AABS1 and AABS2), and we require types to be in *prenex form*: we allow for type schemes in the domain of the function type but not in the codomain. This choice is discussed in (Peyton Jones et al., 2007, Section 4.6.2), and simplifies the type system (skolemization in particular).

We make the following modifications:

1. We refer to an attributed type τ^ν wherever the original rules refer to a type τ.
2. We add a rule VAR$^\otimes$ to deal with sharing.
3. We record *fv*, the attributes on the free variables in a term, and remember to remove a variable from *fv* at all binding sites (rules ABS$^\tau$, ABS$^\sigma$ and LET).
4. Any rule that mentions the function space constructor ($\xrightarrow[\nu_c]{\nu_f}$) is modified to deal with ν_f and ν_c. The modifications to rules ABS$^\tau$, ABS$^\sigma$ and APP follow directly from the core system. Rule FUN compares two attributed types to check if one is at least as polymorphic as the other. For attributed types this is generally only true if both are equal (rule MONO), except rule FUN deals with the type schemes in the domain of the function type. However, the attributes on the arrow are simply part of the attributed types, and must therefore be equal (like in rule MONO).
5. All the rules that deal with type schemes are modified to allow for universal quantification of attribute variables in addition to universal quantification over type variables (rules GEN, INST$^\sigma$, SKOL and SPEC). The function ftuv returns the free type and uniqueness variables in its argument.
6. The rule for generalization must be adapted so that it does not allow generalization over variables in *fv*.
7. The only new rule we add is rule FV, which we discuss in Section 5.2.3.

We argue that these modifications (with the exception of the final one) follow in a straightforward way given the core system we presented in Section 5.1 and do not change the type system presented p. 129
by Peyton Jones et al. (2007) in any essential way: the structure of the type system (with the exception of rule VAR$^\otimes$) is still the exact same. Moreover, the implementation of the type system, including techniques such as skolemization as described in (Peyton Jones et al., 2007, Sections 5 and 6) can be applied without any major modifications.

The typing rules do not include a rule for recursive let expressions. It is possible to add such a rule, but the current presentation of the rules makes it a bit awkward to express. It is not difficult to reorganize the rules to solve that problem, but that would make a superficial comparison between the type system presented in this chapter and the original type system in (Peyton Jones et al., 2007) more difficult, so we opted not to. Either way, the rule for recursive let expressions must make sure that a term which is defined recursively gets a non-unique type (Section 3.2.5). p. 83

5.2.3 Polymorphic uniqueness and closure typing

In this section we explain the need for and purpose of rule FV. Suppose we define a function

$$\texttt{f (x ::}\ \forall u \cdot t^u\texttt{) = ...}$$

Function f requires that its argument must have a polymorphic uniqueness; this makes it possible to use x at two different uniqueness levels in the body of f (we will see an example of when this would be useful in Section 6.3). Let us suppose that f treats x as unique (instantiates the type of x p. 147
to $t^\bullet$)—possibly by modifying x in-place.

Now suppose we define another function g which calls f:

$$\texttt{g (x ::}\ \forall u \cdot t^{u}\texttt{) y = f x}$$

The difference between f and g is that g does not instantiate the type of x, but uses x *at a polymorphic uniqueness*. When typing the body of g, rule VAR will have instantiated the type of x to a monomorphic type t^{v}, but without rule FV we cannot generalize this type back to $\forall u \cdot t^{u}$ since v will be mentioned in fv. This is important, because otherwise we could give g the type

$$\texttt{g ::}\ (\forall u \cdot t^{u}) \rightarrow s^{v} \underset{\times}{\rightarrow} \ldots$$

(only the relevant parts of the type are shown). Note that this type neglects to mention that $g\ x$ for some x will have a unique element in its closure. For this reason, rule GEN disallows generalization over type variables that are mentioned in fv, but we can use rule FV to assume for any variable that it is unique. Using this rule, we can give g the (correct) type

$$\texttt{g ::}\ (\forall u \cdot t^{u}) \rightarrow s^{v} \underset{\bullet}{\rightarrow} \ldots$$

As an aside, this is comparable to the definition of the ceiling $\lceil \cdot \rceil$ operator we defined in Section 4.2.4, p. 122 but slightly modified to support arbitrary boolean expressions. Note that simply substituting $\bullet$ in fv for all uniqueness variables that we are generalizing over is not correct. For example, this would give the wrong type to h:

$$\texttt{h (x ::}\ \forall u \cdot t^{\neg u}\texttt{) y = f x}$$

5.2.4 Complications due to inequalities

We have shown in the previous section that it is straight-forward to extend our core system with support for arbitrary rank types. This extension is not so trivial when the type system involves inequalities (constraints). In this section we explain why, and compare the type system in this chapter with the system presented in Chapter 4. p. 115

In *Clean* constraints are never explicitly associated with types in the typing rules. Rather, the typing rules simply list the constraints as additional premises. However, that approach does not scale up to arbitrary rank types. When we generalize a type $\tau_a^{\nu_a}$ to a type scheme σ, $\tau_a^{\nu_a}$ may be constrained by a set of constraints $\mathcal{C}$. Those constraints should be associated with the type scheme σ, because if at a later stage we instantiate σ to get a type $\tau_b^{\nu_b}$, the same set of constraints should apply to $\tau_b^{\nu_b}$ as well. Thus in Section 4.2 p. 118 we defined a type scheme σ as

$$\forall \overline{x}.\tau^{\nu}, \mathcal{C}$$

In other words, a type scheme is an attributed type τ^{ν}, together with a set of universally quantified (type and uniqueness) variables $\overline{x}$, and a set of constraints $\mathcal{C}$. The typing rules then are careful to manipulate constraint sets. For example, the rule for instantiating a type scheme reads

$$\frac{}{\forall \overline{x}.\tau^{\nu}, \mathcal{C} \leq \mathcal{S}_x \tau^{\nu} \mid \mathcal{S}_x \mathcal{C}}\ \textsc{OldInst}$$

With this rule we can instantiate a type scheme to a type using a substitution $\mathcal{S}_x$, but only if the constraints associated with the type scheme are satisfied.

If we want to allow for arbitrary rank types we must modify the domain of the arrow (the function type constructor) to be a type scheme. Unfortunately that means that we now have constraints appearing in multiple places in type schemes. For example, we might have

$$\mathrm{id}' :: \forall\, t\, u\, u_f.(\forall.t^u, \emptyset) \xrightarrow[\times]{u_f} t^u, \emptyset = \lambda x \cdot x$$

We could add some syntactic sugar to make this type more readable (to get $t^u \xrightarrow[\times]{u_f} t^u$ or even $t^u \longrightarrow t^u$), but that hides a more fundamental problem: the type of *id′* only accepts arguments of type t^u, if those arguments have type t^u *under the empty set of constraints*. If a term has type t^u only if a particular set of constraints is satisfied, that term cannot be used as an argument to *id′*. To get around this problem we need to introduce types that are polymorphic in their constraint sets. This is what we did in the previous chapter. The type of *id* would then be

$$\mathrm{id} :: \forall\, t\, u\, u_f\, c.(\forall.t^u, c) \xrightarrow[\times]{u_f} t^u, c$$

which says that *id* accepts terms that have type t^u under the set of constraints c; the result then also has type t^u, if the same set of constraints is satisfied. This becomes particularly cumbersome for functions with many arguments, and especially for higher order functions (functions taking functions as arguments).

The definition of subsumption (checking whether one type scheme is at least as general as another) is also complicated by the presence of the constraint sets and constraint variables associated with type schemes. To check whether a type scheme σ_1 subsumes σ_2, we need to check whether the constraints associated with σ_2 logically entail σ_1. Recall the examples from Section 4.3: p. 123

$$f :: (\forall\, u\, v.t^u \xrightarrow[u_c]{u_f} s^v, \emptyset) \longrightarrow \ldots$$
$$g :: t^u \xrightarrow[u_c]{u_f} s^v, [u \leq v]$$

Should the application $f\,g$ type-check? Intuitively, f expects to be able to use the function it is passed to obtain an s with uniqueness v (say, a unique s), independent of the uniqueness of t. However, g only promises to return a unique s if t is also unique; the application $f\,g$ should therefore be disallowed. Conversely, if we instead define f' and g' as

$$f' :: (\forall\, u\, v.t^u \xrightarrow[u_c]{u_f} s^v, [u \leq v]) \longrightarrow \ldots$$
$$g' :: t^u \xrightarrow[u_c]{u_f} s^v, \emptyset$$

the application $f'\,g'$ *should* be allowed because the type of g' is more general than the type expected by f'. But it is not completely clear how to define subsumption in a completely general fashion. For example, suppose f was defined as

$$f :: (\forall\, u\, v.t^u \xrightarrow[u_c]{u_f} s^v, c_1 \cup c_2) \longrightarrow \ldots$$

(Recall that c_1 and c_2 are constraint sets.) Then should the application $f\,g$ be allowed? Intuitively it should, since we can instantiate c_1 to $u \leq v$ and c_2 to the empty constraint (the constraint that is vacuously satisfied), but it is not easy to define this formally. When constraints are remodelled as boolean expressions, however, this problem is taken care of by boolean unification.

The use of qualified types would have solved some of these problems, but even within a qualified types framework we need to define logical entailment between constraint sets (see p. 122 Section 4.2.5). Moreover, to keep types readable, it is necessary to take the transitive closure of a set of constraints and only add those constraints to a type scheme that constrain some attributes in p. 123 the type scheme (as explained in Section 4.3).

The fact that we do not have to do *anything* special to define subsumption—in particular, we do not need any definition of logical entailment between sets of constraints—in this chapter is interesting, and further evidence for our claim that the core system is sufficiently similar to the Hindley/Milner type system that modern extensions can easily be incorporated. It is instructive to reconsider the last two examples. Recast in the new type system, the types of f and g are

$$f :: (\forall u\, v. t^{u} \xrightarrow[u_c]{u_f} s^{v}) \to \dots$$

$$g :: t^{u \vee v} \xrightarrow[u_c]{u_f} s^{v}$$

where we have remodelled the implication $u \leq v$ as a disjunction $u \vee v$. Of course, by the same argument as the one used above, the application $f\,g$ should still be disallowed. This will be detected by the subsumption check. Part of the subsumption check will try to solve $\mathfrak{u} \overset{?}{\approx} u \vee v$ and $\mathfrak{v} \overset{?}{\approx} v$ (where $\mathfrak{u}$ and $\mathfrak{v}$ are skolem constants, that is, fixed but unknown attributes). Taken individually, each equation can be solved. However, as soon as we solve one, the other becomes insoluble and the subsumption check fails with an error message such as

Cannot unify $\mathfrak{u}$ and $u \vee \mathfrak{v}$

On the other hand, given the types of f' and g'

$$f' :: (\forall u\, v. t^{u \vee v} \xrightarrow[u_c]{u_f} s^{v}) \to \dots$$

$$g' :: t^{u} \xrightarrow[u_c]{u_f} s^{v}$$

subsumption will need to solve the equations $\mathfrak{u} \vee \mathfrak{v} \overset{?}{\approx} u$ and $\mathfrak{v} \overset{?}{\approx} v$, which have a trivial solution $[u \mapsto \mathfrak{u} \vee \mathfrak{v}, v \mapsto \mathfrak{v}]$, and the application $f'\,g'$ is therefore accepted. So, where we needed to check for logical entailment before, the technique of skolemization (which we needed anyway) will suffice in the new system.

5.3 Generalized algebraic data types

The higher rank type system from (Peyton Jones et al., 2007) that we used as a basis in the last section combines easily with the GADT extension proposed in (Peyton Jones et al., 2006). Modifying the typing rules of the combined system to support uniqueness is a straight-forward exercise and follows the same pattern that was explained in the previous section. We therefore do not show the full typing rules, as they are reasonably complicated (even without support for uniqueness) and provide little additional insight.

The only new modification is in the rule that types the branch of a case analysis statement, p. 81 which must be modified to do uniqueness propagation (Section 3.2.3): when extracting elements from a container, the container must be at least as unique as the extracted elements.

As a simple example, the function *fst* that extracts the first element of a pair has type

$$\text{fst} :: (t^u, s^v)^{u \vee w} \xrightarrow[\times]{u_f} t^u$$

Note that we are again using a disjunction to model an implication: the pair itself must be unique when t is unique (when $u = \bullet$), but if t is non-unique, the pair itself may or may not be unique (w). Alternatively, following a similar argument to the one in Section 5.1.5, we can simplify this p. 132 type to

$$\text{fst} :: (t^u, s^v)^{u} \xrightarrow[\times]{u_f} t^u$$

Apart from the usual arguments for GADTs, supporting GADTs has an additional benefit in a uniqueness type system. Consider the algebraic data type *Rose* of trees with an arbitrary number of branches. In *Clean*, this type is defined as

```
:: Rose a = Rose a [Rose a]
```

The problem with this definition is that it is unclear how the uniqueness of the list of rose trees relates to the uniqueness of the overall rose tree. Clean provides some hooks to influence this, but with a GADT, the problem disappears altogether since we can explicitly specify the type of the constructor:

$$\begin{array}{l}
\textbf{data}\ \texttt{Rose} :: \star \rightarrow \star\ \textbf{where} \\
\quad \texttt{Rose} :: t^u \xrightarrow[\times]{u_f} \texttt{List}^v\ (\texttt{Rose}^v\ t^u) \xrightarrow[u]{u'_f} \texttt{Rose}^v\ t^u
\end{array}$$

With the definition as given, the list of rose trees must have the same uniqueness attribute as the overall rose tree (which can be accomplished in Clean by adding a dot, as in `.[Rose a]`), but other options are also possible.

Note that we do not require outwards propagation in the type of the constructor; it is possible to construct a unique rose tree with non-unique elements. This is impossible in Clean where the constructors enforce outwards propagation, but that is unnecessary. It suffices that the case statement enforces outwards propagation, as explained above. We will discuss this in more detail in Section 8.1.4. p. 206

5.4 Notes

p. 115 Uniqueness types in the type system of *Clean* or in the system we proposed in Chapter 4 often involve inequalities (implications) between uniqueness attributes. This complicates type inference and makes incorporating modern extensions such as arbitrary rank types difficult; *Clean* for example does not fully support arbitrary rank types. We have shown how to avoid these difficulties by recoding attribute inequalities as attribute equalities. The new type system is sufficiently similar to the standard Hindley/Milner type system that standard inference algorithms can be applied, and modern extensions such as arbitrary rank types or GADTs can be incorporated using existing techniques (Peyton Jones et al., 2006). In the next chapter we will consider how we can further simplify the type system.

Simplifying the Type System*

So far we have regarded types and uniqueness attributes as separate entities. In this chapter we show that we can regard uniqueness attributes as type constructors of a special kind. This increases the expressive power of the type system and simplifies the presentation and implementation of uniqueness typing. In Chapter 4 we showed how to avoid subtyping by adding a second uniqueness attribute to the function arrow. In this chapter we propose an alternative way to avoid subtyping without requiring this second uniqueness attribute, further simplifying the type system.

We describe our implementation in *Morrow*. Morrow supports higher rank types and impredicativity, but adding support for uniqueness typing to Morrow required only a few changes to the compiler. This provides strong evidence for our claim that retrofitting uniqueness typing to an existing compiler and extending uniqueness typing with advanced features is straightforward using the techniques in this chapter.

Finally, we outline a soundness of our type system with respect to the call-by-need lambda calculus (Maraist et al., 1998). The full formal proof is described in Chapter 7. p. 159

6.1 Attributes Are Types

In this section, we show that we can regard types and attributes as one syntactic category. This simplifies both the presentation and implementation of a uniqueness type system and increases the expressive power of the type system. If we regard types and attributes as distinct, we need type variables and attribute variables, and we need quantification ($\forall$) over type variables and attribute variables. In addition, the status of arguments to algebraic data types (such as `List a`) is unclear: are they types, attributes, or types *with* an attribute?

These issues are clarified when we regard types and attributes as a single syntactic category. Thus `Int` and `Bool` are types, and so are $\bullet$ (unique) and $\times$ (non-unique). We regard $\texttt{Int}^{\times}$ as syntactic sugar for the application of a special type constructor `Attr` to two arguments, `Int` and $\times$. There are no values of type $\times$, nor are there values of type `Int`, because `Int` is lacking a uniqueness attribute (there are however values of type $\texttt{Int}^{\times}$).

Types that do not classify values are not a new concept. For example, they arise in Haskell as *type constructors* such as the list type constructor (`[]`). We can make precise which types do and do not classify values by introducing a kind system (Jones, 1993). Kinds can be regarded as the "types of types". By definition, the kind of types that classify values is denoted by $*$. In Haskell, we have $\texttt{Int} :: \star$, $\texttt{Bool} :: \star$, but $\texttt{[]} :: \star \rightarrow \star$. The idea of letting the language of vanilla types and additional properties coincide is not new either (e.g., Sheard, 2005; Sulzmann et al., 2007), but as far as we are aware it is new in the context of substructural type systems.[1]

[1] Fluet (2007, Section 4.2) mentions the possibility in a footnote, but dismisses it as not useful.

*The material in chapters 5 and 6, with the exception of sections 6.4.2 and 6.6 which were written later, was published as *Uniqueness Typing Simplified* in Proceedings of the International Symposium on the Implementation and Application of Functional Languages (IFL) 2007, Olaf Chitil, Zoltán Horváth and Viktória Zsók (Eds.), Lecture Notes in Computer Science volume 5083 (de Vries et al., 2008).

Kind language

κ	$::=$	kind
	$\mathcal{T}$	base type
	$\mathcal{U}$	uniqueness attribute
	$*$	base type together with a uniqueness attribute
	$\kappa_1 \to \kappa_2$	type constructors

Type constants

$\mathtt{Int}, \mathtt{Bool}$	$::\ \mathcal{T}$	base type
$\to$	$::\ * \to * \to \mathcal{T}$	function space
$\bullet, \times$	$::\ \mathcal{U}$	unique, non-unique
$\vee, \wedge$	$::\ \mathcal{U} \to \mathcal{U} \to \mathcal{U}$	disjunction, conjunction
$\neg$	$::\ \mathcal{U} \to \mathcal{U}$	negation
$\mathtt{Attr}$	$::\ \mathcal{T} \to \mathcal{U} \to *$	combine a base type and attribute

Syntactic conventions

t^u	$\equiv \mathtt{Attr}\ t\ u$
$a \xrightarrow{u} b$	$\equiv \mathtt{Attr}\ (a \to b)\ u$

Figure 6.1: The kind language and some type constructors with their kinds

Since we do not regard Int as a type classifying values, its kind cannot be $*$ in our type system. Instead, we introduce two new kinds, $\mathcal{T}$ and $\mathcal{U}$, classifying "base types" and uniqueness attributes. Since Attr combines a base type and an attribute into a type of kind $*$, its kind is $\mathcal{T} \to \mathcal{U} \to *$. The kind language and some type constructors along with their kinds are listed in Fig. 6.1. At this point it is useful to introduce the following convention.

> **(Syntactic convention.)** Type variables[1] of kind $\mathcal{T}$ and $\mathcal{U}$ will be denoted by t, s and u, v. Type variables of kind $*$ will be denoted by a, b.

One advantage of treating attributes as types is that we can use type variables to range over base types, uniqueness attributes or types with an attribute, simply by varying the kind of the type variable. This gives more expressive power when defining algebraic data types:

$$\textbf{data}\ \mathtt{X\ a\ =\ MkX\ a}$$
$$\textbf{data}\ \mathtt{Y\ t\ =\ MkY\ t}^{\times}$$
$$\textbf{data}\ \mathtt{Z\ u\ =\ MkZ\ Int}^{\mathtt{u}}$$

Since the type of a constructor argument must have kind $*$, the first data type is parametrized by an attributed type (a type of kind $*$), the second by a base type (a type of kind $\mathcal{T}$), and the third by an attribute (a type of kind $\mathcal{U}$). The kinds of X, Y and Z are therefore $* \to \mathcal{T}$, $\mathcal{T} \to \mathcal{T}$ and $\mathcal{U} \to \mathcal{T}$, respectively. The codomain is $\mathcal{T}$ since $\mathtt{X\ Int}^{\times}$ still lacks an attribute; $(\mathtt{X\ Int}^{\times})^{\bullet}$ on the other hand, which we denote by $(\mathtt{X}^{\bullet}\ \mathtt{Int}^{\times})$, is a unique X containing a non-unique Int. So, assuming $(5 :: \mathrm{Int}^{\times})$, we have $(\mathtt{MkX}\ 5 :: \mathtt{X}^u\ \mathtt{Int}^{\times})$, $(\mathtt{MkY}\ 5 :: \mathtt{Y}^u\ \mathtt{Int})$ and $(\mathtt{MkZ}\ 5 :: \mathtt{Z}^u\ \times)$.

In Clean, we can only define the first of these three data types, so we have gained expressive power. Similarly, type synonyms in Clean are restricted to types without an attribute (types of kind $\mathcal{T}$). With the "attributes are types" approach, we can introduce type synonyms for base types, attributes, or types with an attribute. Moreover, although we have used syntactic conventions to give a visual clue about the kinds of the type variables, the kinds of these types can automatically be inferred by the kind checker, so the expressive power comes at no cost to the programmer.

[1] Strictly speaking, these are meta-variables since our core language does not support universal quantification.

$e ::=$	expression	$\tau_k, \nu ::=$	type
$x^{\odot}$	variable (used once)	c_k	constant
$x^{\otimes}$	variable (used more than once)	$\tau_{(k' \to k)}\ \tau_{k'}$	type application
$\lambda x \cdot e$	abstraction		
$e\, e$	application		

Figure 6.2: Expression and type language for the core system

$$\frac{}{\Gamma, x : \tau^{\nu} \vdash x^{\odot} : \tau^{\nu}|_{x:\nu}}\ \textsc{Var}^{\odot} \qquad \frac{}{\Gamma, x : \tau^{\times} \vdash x^{\otimes} : \tau^{\times}|_{x:\times}}\ \textsc{Var}^{\otimes}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'|_{fv} \qquad fv' = \triangleright_x fv}{\Gamma \vdash \lambda x \cdot e : \tau \xrightarrow{\bigvee fv'} \tau'|_{fv'}}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e : \tau \xrightarrow{\nu} \tau'|_{fv_1} \quad \Gamma \vdash e' : \tau|_{fv_2}}{\Gamma \vdash e\, e' : \tau'|_{fv_1 \cup fv_2}}\ \textsc{App}$$

Figure 6.3: Typing rules for the core lambda calculus

There are two possible variations to the kind system we propose. We could treat `Int`$^{\times}$ as the application of (`Int` $:: \mathcal{U} \to *$) to ($\times :: \mathcal{U}$), or as the (reverse) application of ($\times :: \mathcal{T} \to *$) to (`Int` $:: \mathcal{T}$), avoiding the need for `Attr`. We prefer distinguishing between $\mathcal{T}, \mathcal{U}$ and $*$, but if the reader feels otherwise they should feel free to read $\mathcal{T}$ as syntactic sugar for ($\mathcal{U} \to *$), or $\mathcal{U}$ as syntactic sugar for ($\mathcal{T} \to *$). In all three variations only types of kind $*$ are inhabited, as usual.

6.2 The core system

Although the core system we present in this chapter is similar to the one presented in Section 5.1, p. 129
it is subtly different in a number of aspects. We explain the rules in this section, and discuss the
differences in Section 6.3. p. 147

The expression language and type language are defined in Fig. 6.2 (types have been indexed by their kind k). Both are almost entirely standard, except that—as in the previous chapters—we assume that a sharing analysis has annotated variable uses with $\odot$ or $\otimes$. A variable x marked as $x^{\odot}$ is used only once within its scope; a variable marked as $x^{\otimes}$ is used more than once. The typing rules are listed in Fig. 6.3. As in the previous chapter, the typing relation takes the form

$$\Gamma \vdash e : \tau|_{fv}$$

which reads as "in environment Γ, expression e has type τ; the attributes on the types of the free variables in e are fv". Both Γ and fv are mappings from term variables to types; the only difference is that Γ maps variables to types of kind $*$ and fv maps variables to types of kind $\mathcal{U}$ (in other words, to uniqueness attributes).

6.2.1 Variables

We need to distinguish variables that are used once in their scope and variables that are used multiple times. The rule for variables that are used only once (Var$^{\odot}$) is identical to the normal Hindley/Milner rule, and we simply look up the type of the variable in the environment.

Even when a variable is used only once, that does not automatically make its type unique. For example, there is only one use of x in the identity function:

$$\texttt{id x = x}^{\odot}$$

but when a shared term is passed to id, it will still be shared when it is returned from id. On the other hand, if a variable is used more than once (rule $\text{V}\scriptstyle{\text{AR}}^{\otimes}$), its type must be non-unique (shared). Note that we no longer allow variables to lose their uniqueness (Section 6.3).

6.2.2 Partial Application

We will consider the problem of partial application one final time, as the approach in this chapter is subtly different from the approach in the previous two chapters. Temporarily ignoring the attributes on arrows, the type of dup is

$$\texttt{dup :: } t^{\times} \longrightarrow (t^{\times}, t^{\times})^{u}$$
$$\texttt{dup x = (x}^{\otimes}\texttt{, x}^{\otimes}\texttt{)}$$

Since dup duplicates its argument, it only accepts non-unique arguments. The type checker can easily recognize that dup duplicates x because there is more than one use of x in the function body, which is therefore marked as $\otimes$. However, what if we rewrite dup as

$$\texttt{sneakyDup x = (\textbackslash f -> (f}^{\otimes}\ \bot\texttt{, f}^{\otimes}\ \bot\texttt{)) (const x}^{\odot}\texttt{)}$$

Now there is only one reference to x, which is therefore marked as $\odot$. Still ignoring the attributes on arrows, the function const is defined as

$$\texttt{const :: } t^{u} \longrightarrow s^{v} \longrightarrow t^{u}$$
$$\texttt{const x y = x}$$

It would therefore seem that the type of sneakyDup is

$$\texttt{sneakyDup :: } t^{u} \longrightarrow (t^{u}, t^{u})^{v}$$

But that cannot be correct, because this type tells us that if we pass a single unique t to sneakyDup, it will return a pair of two unique ts. However, the full type of const in our type system is

$$\texttt{const :: } t^{u} \xrightarrow{\times} s^{v} \xrightarrow{u} t^{u}$$

If you pass in a unique t, you get a ***unique*** function from s to t: a function that can only be used once. Conversely, *if* you use a partial application of const more than once, the argument to const must be non-unique. The type of sneakyDup is therefore

$$\texttt{sneakyDup :: } t^{\times} \xrightarrow{\times} (t^{\times}, t^{\times})^{u}$$

Reassuringly, this is the same type as the type of dup. In general, a function must be unique (and can be applied only once) if it has any unique elements in its closure (the environment that binds the free variables in the function body).

Section 6.3 will describe the difference between this approach to partial application and the
p. 117, p. 130 approach described in Sections 4.1.4 and 5.1.2.

6.2.3 Abstraction and Application

The rule for abstractions must determine the value of the attribute on the arrow. As discussed in Section 6.2.2, a function must be unique if it has any unique elements in its closure. The closure of a function $\lambda x \cdot e$ consists of the free variables in the body e of the function, minus x. The attributes on the free variables in the body of the function are recorded in *fv*; using $fv' = \ndres_x fv$ (domain subtraction) to denote *fv* with x removed from its domain, we use the disjunction $\bigvee fv'$ of all the attributes in the range of fv' as the uniqueness attribute on the arrow (recall that we treat uniqueness attributes as boolean expressions).

The rule for application is the normal one, except that we collect the free variables. The attribute on the arrow is ignored (we can apply both unique and shared functions).

6.3 On Subtyping

In this section we compare our approach to subtyping with the approach of Clean (Barendsen and Smetsers, 1996) and the approach proposed in Chapter 4. Consider again the function dup: p. 115

$$\texttt{dup ::}\ t^{\times} \xrightarrow{\times} (t^{\times},t^{\times})^{u}$$

```
dup x = (x, x)
```

In Clean dup has the same type, but that type is interpreted differently. Clean's type system uses a subtyping relation: a unique type is considered a subtype of a non-unique type. That is, we can pass in something that is unique (such as a unique Array) to a function that is expecting a non-unique type (such as dup).

The fact that a unique array can become non-unique is an important feature of a uniqueness type system. A non-unique array can no longer be updated, but can still be read from. However, adding subtyping to a type system leads to considerable additional complexity, especially when considering a contravariant/covariant system with support for algebraic data types (such as Clean's). It becomes simpler when considering an invariant subtyping relation, but we feel that subtyping is not necessary at all.

In Chapter 4, we argued that the type of dup should be

$$\texttt{dup ::}\ t^{u} \xrightarrow[\times]{u_f} (t^{\times},t^{\times})^{v}$$

The (free) uniqueness variable on the t in the domain of the function indicates that we can pass unique or non-unique terms to dup. Since it is always possible to use a uniqueness variable in lieu of a non-unique attribute, an explicit subtyping relation is not necessary.

But there is a catch. As we saw in Section 6.2.2, functions with unique elements in their closure must be unique, and must *remain* unique: they should only be applied once. In Clean, this is accomplished by regarding unique functions as *necessarily unique*, and the subtyping is adjusted to deal with this third notion of uniqueness: a necessarily unique type is *not* a subtype of a non-unique type. Hence, we cannot pass functions with unique elements in their closure to dup.

Unfortunately, when dup gets the type from our previous chapter it *can* be used to duplicate functions with unique elements in their closure. Therefore we introduced a second attribute on the function arrow, indicating whether the function had any unique elements in its closure. The typing rule for application enforced that functions with unique elements in their closure (second attribute) were unique (first attribute). That means that functions with unique elements in their closure can be duplicated, but once duplicated can no longer be applied.

This removed the need for subtyping, but that advantage was offset by the additional complexity introduced by the second uniqueness attribute on arrows: the additional attribute made types more difficult to read (especially in the case of higher order functions).

An important contribution of this chapter is the observation that this additional complexity can be avoided if we are careful when assigning types to primitive functions[1]. For example, a function that returns a new empty array should get the type

$$\texttt{newArray :: Int} \xrightarrow{\times} \texttt{Array}^{u}$$

rather than

$$\texttt{newArray :: Int} \xrightarrow{\times} \texttt{Array}^{\bullet}$$

Similarly, the function that clears all elements of an array should get the type

$$\texttt{resetArray :: Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{u}$$

rather than

$$\texttt{resetArray :: Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{\bullet}$$

An `Array` that is polymorphic in its uniqueness can be passed to `resetArray` as easily as it can be passed to `dup` (of course, a shared array still cannot be passed to `resetArray`). If we are careful never to *return* a unique array from a function, we will always be able to share arrays. We still do not have an explicit subtyping relation but we get the same functionality: the subtyping is encoded in the type of `Array`, rather than in the type of `dup`.

Not all functions should be so modified. For example, many functions with side effects in Clean have a type such as

$$\texttt{fun :: } \cdots \rightarrow (\texttt{World}^{\bullet} \rightarrow \texttt{World}^{\bullet})$$

where the `World` is a token object representing the world state. It never makes sense to duplicate the world, which can be enforced by returning a unique `World` (rather than a `World` which is polymorphic in its uniqueness).

It may seem that a disadvantage of our approach is that we can no longer take advantage of more advanced reference count analysis. For example, given

$$\begin{array}{ll} \texttt{isEmpty} & \texttt{:: Array}^{u} \xrightarrow{\times} \texttt{Bool}^{\times} \\ \texttt{shrink, grow} & \texttt{:: Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{u} \end{array}$$

reference count analysis has been applied correctly to the following definition (Barendsen and Smetsers, 1996):

$$\begin{array}{ll} \texttt{f arr =} & \textbf{if } \texttt{isEmpty arr}^{\otimes} \\ & \textbf{then } \texttt{shrink arr}^{\odot} \\ & \textbf{else } \texttt{grow arr}^{\odot} \end{array}$$

Even though there are three uses of `arr` within `f`, only one of the two branches of the `if`-statement will be executed. Moreover, the condition is guaranteed to be evaluated before either of the branches, and the shared (⊗) annotation on `arr` means that the array will not be modified when the condition is evaluated.

[1] Functions that cannot be defined within the language but must be defined within the compiler, or in a different language such as C.

However, this example uses `arr` at two different types: `Array`$^{\times}$ within the condition and `Array`$^{\bullet}$ within both branches. This works in Clean because `Array`$^{\bullet}$ is a subtype of `Array`$^{\times}$. In the systems proposed in Chapters 4 and 5, this works because a unique term can always be considered as a non-unique term. p. 115, p. 129 In our new proposal however, this program would be rejected (since `Array`$^{\bullet}$ does not unify with `Array`$^{\times}$).

However, we can take advantage of the fact that we have embedded our core system in an advanced type system that supports first class polymorphism (Section 6.4). We want to use a polymorphic value (`arr` :: $\forall u.$`File`^{u}) at two different types within a function: the classic example of a higher rank type (Peyton Jones et al., 2007). Our example above typechecks if we provide the following type annotation:

$$\texttt{f :: } \forall v.(\forall u.\texttt{Array}^{u}) \xrightarrow{\times} \texttt{Array}^{v}$$

The function `f` now *demands* that the array that is passed in is polymorphic in its uniqueness. That is reasonable when we consider that we are using the array at two different types in the body. Moreover, since we regard all unique objects as necessarily unique, it is also reasonable that we cannot pass in a truly unique array to `f`.

Of course there is a trade-off here between simplicity (and ease of understanding) of the type system on the one hand and usability on the other. Since the user must provide a type annotation in order for the definition of `f` to typecheck, the type system has arguably become more difficult to use. However, this case is rare enough that the additional burden on the programmer is small, and a case can be made that it is useful to require a type annotation as it is non-obvious why the function definition is accepted.

Moreover, since rank-2 type inference is decidable (Kfoury and Wells, 1994; Lushman, 2007) (although at the cost of losing principal types) it may be possible to modify the inference algorithm so that the type of `f` can be inferred. An investigation into this possibility is future work.

6.4 Implementation in Morrow

We have integrated our type system in *Morrow*, an experimental functional language developed by Daan Leijen.[1] Morrow's type system is HMF (Leijen, 2008a), which is a Hindley/Milner-like type system that supports first class polymorphism (higher rank types and impredicativity). As such, it is an alternative to both Boxy Types (Vytiniotis et al., 2006) and MLF (Botlan and Rémy, 2003). However, unlike boxy types, it is presented as a small logical system which makes it easier to understand, and at the same time it is much simpler than MLF. Although HMF is quite a good fit with our type system, we have also experimented with integrating it into other type systems. For example, we have a prototype implementation of the arbitrary rank type system by Peyton Jones et al. (2007) extended with uniqueness typing.

As it turns out, the implementation of our type system in Morrow is agreeably straightforward. This provides strong evidence for our claim that adding uniqueness typing to an existing compiler, and more importantly, extending uniqueness typing with advanced features such as higher rank types poses little difficulty when using the techniques from this chapter.

6.4.1 Modifying the type system

We outline the most important changes we had to make to Morrow:

[1] Unfortunately we cannot currently make the source available due to licensing issues.

1. We modified the kind checker to do kind inference for our new kind system (mostly a matter of changing the kinds of type constants)

2. We implemented reference count analysis, annotating variables with information on how often they are used within their scope (once or more than once)

3. We modified the rules for variables and abstraction, so that shared variables must be non-unique, and abstractions become unique when they have unique elements in their closure. To be able to do the latter, all the typing rules had to be adapted to return the *fv* structure from Section 6.2. Variables that are used at a polymorphic uniqueness (a type of the form $\forall u.t^u$ for some t) must be treated as if they were unique for the purposes of *fv* (see also Section 5.2.3). p. 137

4. Let bindings had to be adapted to remove the variables bound from *fv*. Moreover, as is standard in a uniqueness type system (Barendsen and Smetsers, 1996), the type of every binding in a recursive binding group must be non-unique.

5. Most of the work was in modifying the types of the built-in functions and the kinds of the built-in types, and adding the appropriate type constants (such as `Attr`) and kind constants ($\mathcal{T}$, $\mathcal{U}$). However, all of these changes were local and did not affect the rest of the type checker.

6. Unification had to be adapted to do boolean unification, as explained in Section 2.2.4. In addition, it is necessary to simplify boolean expressions; for example, $t^{u \vee \times}$ must be simplified to t^u. This is important because if no simplification is used the boolean expressions can quickly get complicated. Fortunately, we can use an independent module for boolean unification and simplification.[1] When unifying $a \overset{?}{\approx} b$, it suffices to check the kinds of a and b, and if they are $\mathcal{U}$, to call the boolean unification module. Therefore, boolean unification does not in any way complicate the unification algorithm of the type checker.

7. Morrow uses System F (with pattern matching) as its typed internal language. Although the "attributes are types" approach of Section 6.1 means that the internal language does not need to change, Morrow also includes a System F type checker to ensure that the various phases of the compiler generate valid code. This type checker had to be adapted in a similar way to the main type checker.

The majority of these changes were local (did not require any significant refactoring of the compiler), and none of the changes were complicated. The fact that we can treat both vanilla types and uniqueness attributes as types (of different kinds) really helped: modifying the kind checker was straightforward, we got the additional expressive power described in Section 6.1 virtually for free, we did not have to introduce an additional universal quantifier for uniqueness attributes (and thus avoided having to modify operations on types such as capture avoiding substitution or pretty-printing), etc.

[1] The complexity of boolean simplification—like that of type checking (Kfoury et al., 1994; Mairson, 1990)—is exponential (Umans et al., 2006) and our implementation is rather slow. However, there has been a lot of work on fast algorithms for boolean optimization in the context of hardware synthesis (for example, see Coudert, 1994, for a survey). It should be possible to adopt this work in the type checker.

6.4.2 Supporting records and variants

Morrow's support for extensible records and variants is mostly based on the work of Gaster (Gaster and Jones, 1996; Gaster, 1998).[1] The core concept is the notion of a *row*, which is a list of types. The empty row is denoted by $(\!|\,|\!)$, and the row with a field x of type a and tail r' is denoted by $(\!|x : a \mid r'|\!)$. As usual, the row $(\!|x : a \mid (\!|y : b \mid (\!|z : c \mid (\!|\,|\!)|\!)|\!)|\!)$ will be denoted by $(\!|x : a, y : b, z : c|\!)$.

A row is a type-level object with kind `row`. Given a row r, we can interpret the fields of the row as a product to obtain the *record* $\{r\}$, or interpret the fields of the row as a sum to obtain the *variant* $\langle r\rangle$. That is, we have two type-level operators

$$\begin{aligned} \{\} &: \texttt{row} \to \star \\ \langle\rangle &: \texttt{row} \to \star \end{aligned}$$

In addition, there are the following basic operations on records:

$$\begin{aligned} (\_.l) &: r \setminus l \Rightarrow \{(\!|l : a \mid r|\!)\} \to a && \text{selection} \\ \{\_ - l\} &: r \setminus l \Rightarrow \{(\!|l : a \mid r|\!)\} \to \{r\} && \text{restriction} \\ \{l = \_ \mid \_\} &: r \setminus l \Rightarrow a \to \{r\} \to \{(\!|l : a \mid r|\!)\} && \text{extension} \end{aligned}$$

The qualifier $(r \setminus l)$ (pronounced "r lacks l") in these types prohibits repeated occurrences of a label l in the same record. Given these operations we can define two derived operators to update a field or rename a field:

$$\begin{aligned} \{l := \_ \mid \_\} &: r \setminus r \Rightarrow a \to \{(\!|l : b \mid r|\!)\} \to \{(\!|l : a \mid r|\!)\} && \text{update} \\ \_[l \leftarrow m] &: r \setminus l, r \setminus m \Rightarrow \{(\!|l : a \mid r|\!)\} \to \{(\!|m : a \mid r|\!)\} && \text{rename} \end{aligned}$$

We have a similar set of basic operations on variants:

$$\begin{aligned} \langle l = \_\rangle &: r \setminus l \Rightarrow a \to \langle(\!|l : a \mid r|\!)\rangle && \text{injection} \\ \langle l \mid \_\rangle &: r \setminus l \Rightarrow \langle r\rangle \to \langle(\!|l : a \mid r|\!)\rangle && \text{embedding} \\ \texttt{case} &: \langle r\rangle \to \{\texttt{to}\ a\ r\} \to a && \text{pattern matching} \end{aligned}$$

This definition of records is much more powerful than the support for records found in most programming languages: record selection can select a record l from *any* record that contains l, and fields can be added to or removed from records at will. Nevertheless, all record operations are fully statically checked and moreover the additional power does not incur any runtime overhead: it will always be known at compile time where in a record a field can be found, so that it can be accessed in constant time. We refer the reader to Gaster's papers for more information.

Most of the types of these basic operations are self-explanatory, with the exception of the type for pattern matching. The function `case` requires a variant for a row r and a record for a row $(\texttt{to}\ a\ r)$. The `to` operator in this type is a type-level operator defined such that

$$\texttt{to}\ a\ (\!|x : b, y : c \mid r|\!)$$

[1] Although Morrow used to support scoped labels (Leijen, 2007b), the version we worked with did not.

evaluates to

$$\llparenthesis x : b \rightarrow a, y : c \rightarrow a \mid r \rrparenthesis$$

In other words, the `case` function requires a record with a function for each of the alternatives of the variant from its type to the type of the result of the pattern match. This means that the body of a pattern becomes a first-class object, since it is simply another record—which can be passed around, modified, etc.

To support these operations in our modified version of Morrow with support for uniqueness, we need to adapt the types of these operations. The kinds of the type operators change to

$$\begin{aligned} \{\} &: \texttt{row} \rightarrow \mathcal{T} \\ \langle\rangle &: \texttt{row} \rightarrow \mathcal{T} \end{aligned}$$

to signify that $\{r\}$ and $\langle r \rangle$ are still missing a top-level uniqueness attribute. The basic idea is that if we select a field from a record, the record must be unique if the field is unique (uniqueness
p. 81 propagation; Section 3.2.3) and that the uniqueness must be invariant along the spine of a record (these requirements are similar to the requirements for algebraic data types in Clean). The modifications for most of the operations are straightforward:[1]

$$\begin{aligned} (\_.l) &: r \setminus l \Rightarrow \{\llparenthesis l : t^u \mid r \rrparenthesis\}^{u \vee v} \xrightarrow{\times} t^u && \text{selection} \\ \{\_ - l\} &: r \setminus l \Rightarrow \{\llparenthesis l : a \mid r \rrparenthesis\}^{u} \xrightarrow{\times} \{r\}^u && \text{restriction} \\ \{l = \_ \mid \_\} &: r \setminus l \Rightarrow t^u \xrightarrow{\times} \{r\}^v \xrightarrow{u} \{\llparenthesis l : t^u \mid r \rrparenthesis\}^v && \text{extension} \\ \langle l = \_ \rangle &: r \setminus l \Rightarrow a \xrightarrow{\times} \langle \llparenthesis l : a \mid r \rrparenthesis \rangle^u && \text{injection} \\ \langle l \mid \_ \rangle &: r \setminus l \Rightarrow \langle r \rangle^u \xrightarrow{\times} \langle \llparenthesis l : a \mid r \rrparenthesis \rangle^u && \text{embedding} \end{aligned}$$

The only surprise here might be that we did not simplify the type for selection to

$$(\_.l) : r \setminus l \Rightarrow \{\llparenthesis l : t^u \mid r \rrparenthesis\}^{u} \xrightarrow{\times} t^u \qquad \text{selection (simplified)}$$

p. 132 which may be the type the reader might have expected based on Section 5.1.5. We will discuss
p. 153 why we need the more general type in Section 6.4.3.

The type for pattern matching is slightly more tricky to modify. As for field selection, we need to express uniqueness propagation. For example:

```
select :: $\langle \llparenthesis l : t^u \rrparenthesis \rangle^{u \vee v} \xrightarrow{\times} t^u$
select v = **case** v {l = λx → x}
```

However, recall that the type for pattern matching is expressed in terms of a "`to`" operator. Hence, we need something of the form

$$\texttt{case} : \langle r \rangle^u \rightarrow \{\texttt{to}\ u\ a\ r\}^v \rightarrow a \tag{6.1}$$

Since the `to` operator acts on the types of the fields in the record, but cannot modify u, it is difficult to see how to modify this type to express outwards uniqueness propagation.

[1] Due to the invariance of Morrow's type system, the type of selection limits the fields of records to monomorphic type. We will come back to this point in Section 8.2.3.

Fortunately, there is an alternative way to express propagation. So far, we have encoded a type

$$\forall t\,u\,v \cdot t^u \to t^v, [u \leq v]$$

as

$$\forall t\,u\,v \cdot t^{u \vee v} \to t^v \tag{6.2}$$

where we have modified the attribute that was the *target* of the implication using a *disjunction*. However, (6.2) is isomorphic to the type

$$\forall t\,u\,v \cdot t^u \to t^{u \wedge v}$$

in the sense that both types subsume each other. In this encoding, we have modified the attribute that was the *source* of the implication using a *conjunction*. We can take advantage of this alternative encoding to replace the type of `select` by the alternative (isomorphic) type

`select ::` $\langle \llparenthesis l : t^{u \wedge v} \rrparenthesis \rangle^v \xrightarrow{\times} t^{u \wedge v}$

We can now define the type of pattern matching as in (6.1), with a modified `to` operator which is defined so that

$$\texttt{to}\; u\; a\; \llparenthesis x : t^v, y : s^w \mid r \rrparenthesis$$

evaluates to

$$\llparenthesis x : t^{v \wedge u} \to a, y : s^{w \wedge u} \to a \mid r \rrparenthesis$$

6.4.3 Multiple field accesses

We mentioned that based on Section 5.1.5 we might have expected to give the following type to p. 132 field selection:

$$(\_.l) : r \setminus l \Rightarrow \{\llparenthesis l : t^u \mid r \rrparenthesis\}^u \xrightarrow{\times} t^u$$

This type for field selection is more restrictive than it needs to be, since it requires the record to be non-unique if the field that we are extracting has a non-unique type—even though it is perfectly sound to extract a non-unique element from a unique container. Nevertheless, since (barring type annotations) records will never be unique, but will either be non-unique or have a polymorphic uniqueness, this will not cause any type errors and result in simpler types.

But there is a catch. Morrow does not offer a special syntax for accessing more than one field of a record. Consider the function that swaps the two components of a pair. A pair in Morrow is syntactic sugar for a record with two fields called `field1` and `field2`. Hence, we can define `swap` as:

```
swap p = (p.field2, p.field1)
```

Provided that the reference count analysis correctly recognizes this as a single access to p (since both fields can be extracted simultaneously) and marks this program as

`swap p = (`$\texttt{p}^{\odot}$`.field2, `$\texttt{p}^{\odot}$`.field1)`

then the inferred type of `swap` will be

`swap ::` $(t^u, s^u)^u \xrightarrow{\times} (s^u, t^u)^v$

This type demands that the two components of the pair have the same uniqueness[1]. This is a consequence of the fact that field selection requires the fields of the record to have the same uniqueness as the record, and therefore as each other. This is unnecessarily restrictive, and in the absence of a special syntax for accessing more than one field of a record, we are therefore forced to relax the type of record selection to

$$(\_.l) : r \setminus l \Rightarrow \{(\!|l : t^u \mid r|\!)\}^{u \vee v} \xrightarrow{\times} t^u$$

where we no longer require the record to be non-unique if we are extracting a non-unique field (although of course the record must still be unique when we are extracting a unique field). The type of `swap` now becomes

`swap ::` $(t^u, s^v)^{u \vee v \vee w} \xrightarrow{\times} (s^v, t^u)^{w'}$

6.5 Soundness

To prove soundness we use a slightly modified but equivalent set of typing rules.[2] Rather than giving different typing rules for variables marked as used once or used more than once, we do not mark variables at all but enforce that unique variables are used at most once by splitting the environment into two in rule APP. Non-unique variables can still be used more than once because the context splitting operation collapses multiple assumptions about non-unique variables (rule SPLIT$^\times$). This presentation of the type system is known as a *substructural* presentation because some of the structural rules (in this case, contraction) do not hold. The presentation style we have used, using a context splitting operation, is based on that given in (Walker, 2005), where it is attributed to (Cervesato and Pfenning, 2002).[3]

The soundness proof for a type system states that when a program is well-typed it will not "go wrong" when evaluated with respect to a given semantics. We are interested in a lazy semantics; often the call-by-name lambda calculus is used as an approximation to the lazy semantics, but it is not hard to see that we will not be able to prove soundness with respect to the call-by-name semantics. For example, consider

```
(\x. (x, x)) (f y)
```

In the call-by-name semantics, this term evaluates to

```
(f y, f y)
```

But when we allow for side effects, these two terms have a different meaning. In the first, we evaluate $f\ y$ once and then duplicate the result; in the second, we evaluate $f\ y$ twice (and so have the potential side effect of f twice).

[1] The uniqueness of the *result* tuple is free as it will be newly created by `swap`.

[2] The syntax-directed presentation using sharing marks is easier to understand and more suitable for type inference. However, it is not usable for a soundness proof. Such a distinction between a syntax-directed and a logical presentation is not uncommon, and has been used before in the context of uniqueness typing (Barendsen and Smetsers, 1996).

[3] One subtle aspect of the definition of the context splitting operation is that it does not allow weakening: when an environment Γ is split into two environments Γ_1 and Γ_2, every assumption in Γ *must* occur in either Γ_1 or Γ_2 (or possibly both for non-unique assumptions). Weakening on the typing environment Γ is implicit in the definition of `Var`, but weakening on *fv* is not allowed; this is important since weakening on *fv* might affect the type of an expression (it could force a function to be unique). A function such as $\lambda x \cdot \lambda y \cdot x$ is nevertheless typeable, since $fv = \rhd_x fv$ if $x \notin fv$: the rule for abstraction does not require to add the variable to environment *fv* when typechecking the function body.

Term language

e	$::=$	$x \mid \lambda x \cdot e \mid e\,e$	term
A	$::=$	$\lambda x \cdot e \mid \text{let } x = e \text{ in } A$	answer
E	$::=$	$[] \mid E\,e \mid \text{let } x = e \text{ in } E \mid \text{let } x = E_0 \text{ in } E_1[x]$	evaluation context

Syntactic convention

$(\text{let } x = e_1 \text{ in } e_2) \equiv (\lambda x \cdot e_2)\, e_1$

Evaluation rules

$\mapsto$ is the smallest relation that contains VALUE, COMMUTE, ASSOC and is closed under the implication $M \mapsto N$ implies $E[M] \mapsto E[N]$.

(VALUE)	$\text{let } x = \lambda y \cdot e \text{ in } E[x]$	$\mapsto$	$\{(\lambda y \cdot e)/x\}\, E[x]$
(COMMUTE)	$(\text{let } x = e_1 \text{ in } A)\, e_2$	$\mapsto$	$\text{let } x = e_1 \text{ in } A\, e_2$
(ASSOC)	$\text{let } y = (\text{let } x = e \text{ in } A) \text{ in } E[y]$	$\mapsto$	$\text{let } x = e \text{ in let } y = A \text{ in } E[y]$

Substructural typing rules

$$\frac{}{\Gamma, x : \tau^\nu \vdash x : \tau^\nu|_{x:\nu}}\ \textsc{Var}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'|_{fv} \qquad fv' = \triangleright_x fv}{\Gamma \vdash \lambda x \cdot e : \tau \xrightarrow{\vee fv'} \tau'|_{fv'}}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e : \tau \xrightarrow{\nu} \tau'|_{fv_1} \qquad \Delta \vdash e' : \tau|_{fv_2}}{\Gamma \circ \Delta \vdash e\, e' : \tau'|_{fv_1 \circ fv_2}}\ \textsc{App}$$

Context splitting

$$\frac{}{\emptyset = \emptyset \circ \emptyset}\ \textsc{Split}^{\emptyset} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \tau^\times = (\Gamma_1, x : \tau^\times) \circ (\Gamma_2, x : \tau^\times)}\ \textsc{Split}^{\times}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \tau^\nu = (\Gamma_1, x : \tau^\nu) \circ \Gamma_2}\ \textsc{Split}^{\nu}_1 \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \tau^\nu = \Gamma_1 \circ (\Gamma_2, x : \tau^\nu)}\ \textsc{Split}^{\nu}_2$$

Figure 6.4: Call-by-Need Semantics

Accordingly, the types of both terms in a uniqueness type system are also different. In the first, f may or may not be unique, and must have a non-unique result (because the result is duplicated). In the second, f cannot be unique (because it is applied twice) and may or may not return a unique result.

Traditionally (Barendsen and Smetsers, 1996) a graph rewriting semantics is used to prove soundness, but this complicates equational reasoning. Fortunately, it is possible to give an algebraic semantics for lazy evaluation. Launchbury's *natural semantics for lazy evaluation* (Launchbury, 1993) is well-known and concise, but is a big-step semantics which makes it less useful for a soundness proof (see Section 2.7.5). The call-by-need semantics by Maraist *et al.* p. 58
(Maraist et al., 1998) is slightly more involved, but is a small-step semantics and fits our needs perfectly. The semantics is shown in Fig. 6.4.

We only give an outline of the proof here; the full formal proof is described in Chapter 7. p. 159

Theorem 4 (Progress) *Suppose e is a closed, well-typed term ($\emptyset \vdash e : \tau|_{fv}$ for some τ and fv). Then either e is an answer or there exists some e' such that $e \mapsto e'$.*

The easiest way to prove progress is to prove a weaker property first: for every term e, e is an answer, there exists some e' such that $e \longmapsto e'$, or $e = E[x]$ for some x. This weaker property can be proven by a complete structural induction on e; the proof is laborious but not difficult. To prove progress using the weak progress property, we just need to rule out the last possibility. However, if $e = E[x]$ for some x, and $\varnothing \vdash e : \tau|_{fv}$, then we must have $x \in \varnothing$, which is impossible. □

The proof of preservation is more involved and we only give a brief outline here. The main lemma that we need to be able to prove preservation is the substitution lemma:

Lemma 1 (Substitution) *If* $\Gamma, x : \tau_a \xrightarrow{\vee fv_2} \tau_b \vdash e_1 : \tau|_{fv_1, x:\vee fv_2}$, *$x$ is free in* e_1, *and* $\Delta \vdash \lambda y \cdot e_2 : \tau_a \xrightarrow{\vee fv_2} \tau_b|_{fv_2}$, *then* $\Gamma \circ \Delta \vdash \{(\lambda y \cdot e_2)/x\}\, e_1 : \tau|_{fv_1 \circ fv_2}$.

The proof is by induction on $\Gamma, x : \tau_a \xrightarrow{\vee fv_2} \tau_b \vdash e_1 : \tau|_{fv_1, x:\vee fv_2}$ and is not trivial. The essence of the proof is that if $(\lambda x \cdot e_1)(\lambda y \cdot e_2)$ is well-typed, then either x occurs once in e_1, in which case we can substitute $\lambda y \cdot e_2$ for x without difficulty, or x occurs more than once in e_1. In that case, x must have a non-unique type, which means that $\lambda y \cdot e_2$ must be non-unique, and therefore the function cannot have any unique elements in its closure—or equivalently, that e_2 be typed in an environment where every variable has a non-unique type. Since $\Delta = \Delta \circ \Delta$ if all assumptions in Δ are non-unique, this means that we can type the result term even when $\lambda y \cdot e_2$ is duplicated.

Armed with the substitution lemma, we can prove preservation:

Theorem 5 (Preservation) *If* $\Gamma \vdash e : \tau|_{fv}$ *and* $e \longmapsto e'$ *then* $\Gamma \vdash e' : \tau|_{fv}$.

By induction on $e \longmapsto e'$. The cases for COMMUTE, ASSOC, and the three closure rules (one for each of the non-trivial evaluation contexts) are reasonably straightforward. The case for VALUE relies on the substitution lemma. □ A full formalization of the calculus extended with (let-bound or first-class) polymorphism is future work.

6.6 Metatheoretical musings

There are some subtle metatheoretical advantages of the approach we advocate in this chapter over
p. 130 the one suggested in the previous chapter. Recall the following definitions from Section 5.1.2:

$$\texttt{dup :: } t^{u} \xrightarrow[\times]{u_f} (t^{\times}, t^{\times})^{v}$$

```
dup x = (x, x)
```

$$\texttt{sneakyDup :: } t^{\times} \xrightarrow[\times]{u_f} (t^{\times}, t^{\times})^{w}$$

```
sneakyDup x = let f = const x in (f 1, f 2)
```

It is unfortunate that `dup` and `sneakyDup` do not get the same type. We mentioned in Sec-
p. 132 tion 5.1.5 that this is a consequence of the rule for variables:

$$\frac{}{\Gamma, x : \tau^{\nu} \vdash x^{\odot} : \tau^{\nu}|_{(x,\nu)}}\ \textsc{Var}^{\odot}$$

This rule says that variables marked as exclusive (not shared) have the type that is listed in the environment; we can relax this rule to one that requires the type in the environment to be at least as unique as the type of the variable:

$$\frac{}{\Gamma, x : \tau^{\nu|\nu'} \vdash x^{\odot} : \tau^{\nu}|_{(x,\nu)}} \; \textsc{Var}'^{\odot}$$

This gives `dup` and `sneakyDup` the same type, but at the cost of complicating other types; for instance, the identity function now has the type

$$\texttt{id :: } t^{u \vee v} \xrightarrow[\times]{u_f} t^{u}$$

```
id x = x
```

We preferred to avoid this additional complexity, which is why we opted for the first rule instead. However, this approach makes it impossible to give an equivalent presentation of the type system with explicit rules weakening and contraction without giving up principal types.

In particular, if we have the rules from the previous chapter

$$\frac{}{\Gamma, x : \tau^{\nu} \vdash x^{\odot} : \tau^{\nu}|_{(x,\nu)}} \; \textsc{Var}^{\odot} \qquad \frac{\Gamma, x : \tau^{\nu} \vdash e : \tau'^{\nu'}|_{fv}}{\Gamma \vdash \lambda x \cdot e : \tau^{\nu} \xrightarrow[\bigvee(\triangleright_x fv)]{\nu_f} \tau'^{\nu'}|_{\triangleright_x fv}} \; \textsc{Abs}$$

together with the substructural rules

$$\frac{\Gamma, y : \tau^{\times}, z : \tau^{\times} \vdash e : \tau'^{\nu'}}{\Gamma, x : \tau^{\nu} \vdash e[y := x, z := x] : \tau'^{\nu'}} \; \textsc{Contr} \qquad \frac{\Gamma \vdash e : \tau'^{\nu'}}{\Gamma, x : \tau^{\nu} \vdash e : \tau'^{\nu'}} \; \textsc{Weak}$$

then we can assign two types to the identity function:

$$\texttt{id :: } a^{u} \xrightarrow[\times]{u_f} a^{u}$$

or

$$\texttt{id :: } a^{u} \xrightarrow[\times]{u_f} a^{\times}$$

(by applying contraction immediately followed by weakening). Unfortunately, these two types are incomparable and with the rule for variables as it stands, there is no other type that is more general than both. Hence, we have lost principal types. They can be recovered by using rule $\textsc{Var}'^{\odot}$, but again at the cost of introducing more complexity.

Whether or not this is an important issue is debatable—in our formalization we make no use of an explicit contraction rule (and prefer to rely on a context splitting operation instead; see Section 6.5) so it does not affect us. However, it is good to know that the type system we propose p. 154 in this chapter (without a coercion from unique to non-unique) does not suffer from this problem.

Another advantage of the type system we propose in this chapter (where coercion from unique to non-unique is disallowed) and the previous (which used an additional closure attribute on arrows) is that we can potentially prove a stronger soundness result. We have proven soundness with respect to the *standard* reduction rules from the call-by-need calculus. From our perspective, the most important difference between the standard reduction rules and the general call-by-need reduction rules is in the treatment of lambda abstractions. In the standard rules, a lambda abstraction will only be duplicated if it is applied. For example,

```
(λf. (f, f)) (λy. ...)
```

will *not* reduce to

```
(λy. ..., λy. ...)
```

because the function is not applied. This is important in a type system such as the one from the previous chapter where functions with unique elements in their closure (that is, variables on the "dots" in $\lambda y \cdot \ldots$) can be duplicated—because in that case, the first program would be typeable but the second would not and we have lost preservation. If an abstraction *does* get applied, as in

```
(λf. (f 1, f 2)) (λy. ...)
```

the function *will* be duplicated, but now the typing rules (in particular, the rule for application) will make sure that there are no unique variables in the function closure and all is well.

In the type system of this chapter, however, a function with unique elements in its closure must be unique itself. Moreover, unique elements cannot be duplicated. Hence, we know that if

```
(λf. (f, f)) (λy. ...)
```

is type correct, there cannot be any variables of a unique type in the function closure ("on the dots") and hence we can duplicate the function without difficulty.

To paraphrase, if the type system verifies that functions with unique elements in their closure p. 115, p. 129 arc unique *when they are applied* (as in Chapters 4 and 5) it is important that the semantics will duplicate functions *only* if they are applied. If on the other hand the type system guarantees that functions with unique elements in their closure are unique *when they are created* (as in this chapter), we can duplicate functions irrespective of how they are used. We nevertheless used the standard reduction rules because it is the more difficult proof, and more easily adapted to the type system of the previous chapter if necessary.

6.7 Notes

By treating uniqueness attributes as types of a special kind $\mathcal{U}$, the presentation and implementation of a uniqueness type system is simplified, and we gain expressiveness in the definition of algebraic data types. Moreover, no explicit subtyping relation is necessary if we are careful when assigning types to primitive functions: we require that unique terms must never be shared, and make sure that functions never return unique terms (but rather terms that are polymorphic in their uniqueness).

Together with observation from the previous chapter that we can recode inequality constraints as equality constraints if we allow for arbitrary boolean expressions as attributes, these observations lead to an expressive yet simple uniqueness type system, which is sound with respect to the call-by-need lambda calculus. The system can easily be extended with advanced features such as higher rank types. We have integrated our type system in Morrow, an experimental programming language with an advanced type system. The implementation required only minor changes to the compiler, providing strong evidence for our claim that retrofitting our type system to existing compilers is straightforward.

7 Formalization[*]

This chapter describes the full formal proof of soundness that was briefly outlined in the previous section. The proof itself is available as Coq sources and can be downloaded from the author's homepage[1].

In sections 7.2 and 7.3 we highlight some of the difficulties we faced when developing the p. 160, p. 165
proof, and discuss some of its more subtle aspects. In Section 7.4 we define the notion of an p. 169
environment, various operations on environments, the kinding and typing relations, and the
operational semantics for our language. Sections 7.5 and 7.6 prove numerous auxiliary lemmas p. 174, p. 192
that will be necessary in the main proof, which is described in Section 7.7. Appendix A.1 finally p. 196, p. 225
gives a formalization of boolean algebra based on Huntington's Postulates (Huntington, 1904).

7.1 Note on the proofs

Every lemma in this chapter is preceded by a brief description of the lemma in informal language (English), followed by a precise statement of the lemma (in the syntax of *Coq*) and a brief description (again in English) of how the lemma can be proven. For most lemmas, this description will begin with "*By induction on...*" or "*By inversion on...*"; many descriptions will also include the most important other lemmas that the proof relies on. *Coq* verifies a proof strictly from top to bottom, so if a lemma B relies on lemma A, A must have been proven before lemma B; this therefore applies equally to the structure of this chapter. When the description of the proof does not mention induction or inversion, then these techniques are not necessary and the lemma can be proven by direct application of other lemmas.

What we do not show is the actual proofs themselves: there would be little point. The proofs have been verified by *Coq*, a widely respected proof assistant. If the reader nevertheless prefers to verify the proofs by hand, he will want to redo them himself; the short description of the proof should provide enough information to get started.

Besides, the proofs are written in the syntax of *Coq*. Coq is based on the calculus of constructions, a powerful version of the dependently typed lambda calculus (Section 2.3.5). As such, a p. 34
proof in *Coq* is a program (a term of the lambda calculus) that, given the premises, constructs a proof of the conclusion. However, in all but the most simple cases, these programs are too difficult to write by hand, and instead the proof consists of a list of calls to *tactics* which build up the program step-by-step.

Consider a simple example. Suppose we want to prove that $n + 0$ is equal to n for all natural numbers n. Here is a full *Coq* proof of this property (this proof comes from the *Coq* standard library):

[1] `http://www.cs.tcd.ie/Edsko.de.Vries`

[*] The material in this chapter was published as *Uniqueness Typing Simplified—Technical Appendix*, Department of Computer Science, Trinity College Dublin, Technical Report TCD-CS-2008-19 (de Vries, 2008a).

```
Lemma plus_n_O : forall n:nat, n = n + 0.
Proof.
  induction n; simpl in |- *; auto.
Qed.
```

Although it should be clear what `induction n` does, the purpose of the other tactics (such as `simpl` or `auto`) is less obvious, *even to an experienced Coq user.* Tactics interact with the current state of the proof assistant, which includes information such as which lemmas are available, the types of all variables, etc. Trying to interpret a *Coq* proof without *Coq* is akin to hearing one part of a telephone conversation: half the script is missing.

Phrased another way, tactics provide an imperative specification of a declarative program. Consequently, it is difficult to understand a proof without being able to see how the imperative program changes the state of the proof assistant. Recently, there has been some work on a declarative proof language for Coq (Corbineau, 2008), but support for this language is still experimental in Coq and we make no use of it. The actual proof constructed by these tactics is

$$\lambda(n : \texttt{nat}) \cdot \texttt{nat\_ind}\ (\lambda(m : \texttt{nat}) \cdot m = m + 0)\ (\texttt{refl\_equal}\ 0)$$
$$(\lambda(m : \texttt{nat})(IH_m : m = m + 0) \cdot \texttt{f\_equal}\ \texttt{S}\ IH_m)\ n$$

which makes use of various other lemmas, such as induction on natural numbers (`nat_ind`—essentially a fold operation), the fact that equality is reflexive (`refl_equal`) and a lemma that states that if $x = y$, then for all f, $f\ x = f\ y$ (`f_equal`). The details do not matter; the point is that this is hardly more readable than the original proof. In this chapter, we would simply describe this proof as "*By induction on n*".

7.2 Equivalence

Suppose we have a set C of objects together with an equivalence relation $\approx$ on C, and some characterization P of objects of C. We want P to have the property that if $P\ x$ and $x \approx y$, then $P\ y$. There are three different ways in which we can guarantee that P has this property.

- We can prove that P has the required property.
- We can define P over the quotient set $C/\!\approx$ instead. This will give us the desired property by definition.
- It may be possible to choose an alternative representation C' of the objects in C, such that every equivalence set in $C'/\!\approx$ is a singleton set. In other words, so that the equivalence relation *is* the identity relation. The desired property of P then holds trivially.

For example, take the set of lambda terms together with alpha-equivalence, and the property of being well-typed. Then,

- We can prove that well-typedness is equivariant: if $\lambda x \cdot x$ is well-typed, so is $\lambda y \cdot y$.
- We can define the well-typedness over the set of alpha-equivalent terms.
- We can represent lambda terms using De Bruijn notation, in which case $\lambda x \cdot x$ and $\lambda y \cdot y$ are both represented as $\lambda \cdot 0$.

Not all options are always practical, and each option has its advantages and disadvantages. For the specific example of alpha-equivalent terms, the first option may be possible, but cumbersome as we may have many properties over lambda-terms; we will have to prove equivariance for each one. The second approach is inconvenient when we need to refer to the name of the bound variable in an abstraction, for example in the typing rule for abstraction. The final approach does not have these shortcomings, but introduces new ones: many operations on lambda terms in De Bruijn notation must juggle with the indices, leading to additional complexity in proofs.

In informal proofs, we tend to gloss over this issue:

> In this situation the common practice of human (as opposed to computer) provers is to say one thing and do another. We say that we will quotient the collection of parse trees by a suitable equivalence relation of alpha-conversion, identifying trees up to renaming of bound variables; but then we try to make the use of alpha-equivalence classes as implicit as possible by dealing with them via suitably chosen representatives. How to make good choices of representatives is well understood, so much so that it has a name—the "Barendregt Variable Convention": choose a representative parse tree whose bound variables are fresh, i.e., mutually distinct and distinct from any (free) variables in the current context. This informal practice of confusing an alpha-equivalence class with a member of the class that has sufficiently fresh bound variables has to be accompanied by a certain amount of hygiene on the part of human provers: our constructions and proofs have to be independent of which particular fresh names we choose for bound variables. Nearly always, the verification of such independence properties is omitted, because it is tedious and detracts from more interesting business at hand. Of course this introduces a certain amount of informality into "pencil-and-paper" proofs that cannot be ignored if one is in the business of producing fully formalized, machine-checked proofs.
>
> —Pitts (2001), *Nominal logic, a first order theory of names and binding*

(See also Berghofer and Urban, 2008.) In the remainder of this section, we detail how we tackle this issue for the specific examples of terms under alpha-equivalence, typing environments under substructural rules and boolean expressions under Huntington's Postulates.

7.2.1 Lambda terms

We discussed the problem of dealing with terms under alpha-equivalence as an example in the introduction to this section. There are various proposed solutions in the literature; we will adopt the *locally nameless* approach suggested by (Aydemir et al., 2008) in *Engineering Formal Metatheory* (we refer the reader to the same paper for an overview of alternatives).

In the locally nameless approach, bound variables are represented by De Bruijn indices, but free variables are represented by ordinary names. This means that alpha-equivalent terms are represented by the same term (and so we do not have to reason explicitly about alpha-equivalence), but we do not have to perform any arithmetic operations on terms. We do however have to solve one problem.

Consider the typing rule for application. In the locally nameless style, the rule is

$$\frac{\Gamma, x : \tau \vdash e^x : \tau' \qquad \text{fresh } x}{\Gamma \vdash \lambda \cdot e : \tau \rightarrow \tau'}\textsc{Abs}$$

When we typecheck the body e, we "open it up" using a fresh variable x, and then record the type of the variable as normal. That is, we replace bound variable 0 (the variable that was bound by the lambda) by a fresh variable (for some definition of "fresh"). This is a consequence of the locally nameless approach: every time a previously bound variable becomes free, we have to invent a fresh name for it. Without the freshness condition, we would be able to derive

$$\frac{\dfrac{\vdots}{x : \tau, x : \tau' \vdash (x, x) : (\tau', \tau')}}{x : \tau \vdash \lambda \cdot (0, x) : \tau' \rightarrow (\tau', \tau')}$$

where the (original) free variable x has suddenly changed type (the typing environment acts as a binder, and the variable x has been "captured"). The minimal freshness condition is therefore that the variable that is used to open up a term, does not already occur free in the term:

$$\frac{\Gamma, x : \tau \vdash e^x : \tau' \qquad x \notin \text{fv}\, e}{\Gamma \vdash \lambda \cdot e : \tau \rightarrow \tau'}\text{M-A}\textsc{bs}$$

A weak premise ($x \notin$ fv e) is good when using rule ABS to prove the type of a term since we only have to show that $\Gamma, x : \tau \vdash e^x$ holds for one particular x. It is however not so good when doing induction on a typing relation. In that case, we know that the e^x has type τ' *for one particular* x. But that x may not be fresh enough for our purposes, at which point we need to rename the term to avoid name clashes. To circumvent this problem, Aydemir et al. (2008, Section 4) propose to use cofinite quantification[1]:

$$\frac{\forall x \notin L \cdot \Gamma, x : \tau \vdash e^x : \tau'}{\Gamma \vdash \lambda \cdot e : \tau \rightarrow \tau'}\text{C-A}\textsc{bs}$$

To use C-ABS, we have to show that the e^x has type τ' for all x not in some finite set L (that is, x is chosen from a cofinite set), but using this rule is no more difficult than using M-ABS: we simply pick an arbitrary variable not in L. The induction principle however is now much stronger: we now know that e^x has type τ' *for any* x not in some set L'. Then when we have to prove that $\lambda \cdot e$ has type $\tau \rightarrow \tau'$, knowing that e^x has type τ' for all x not in L', and we need x to be distinct from some other variable y, we can simply apply rule ABS choosing $L' \cup \{y\}$ for L. We still occasionally need renaming lemmas, but they too become much more straightforward to prove
p. 175 when using cofinite quantification (we prove a number of renaming lemmas in Section 7.5.2).

Arthur Charguéraud, one of the authors of the *Engineering Formal Metatheory* paper, has developed a *Coq* library (Charguéraud, 2007) which facilitates the use of the locally nameless representation of terms and the use of cofinite quantification. The proofs in this chapter will make extensive use of this library, which we will dub the *Formal Metatheory* library. As an example, here is a trivial lemma that we can always pick a variable that is distinct from all other variables in a typing environment:

```
Lemma fresh_from_env : forall E e T fvars,
  E |= e ~: T | fvars -> exists x, x \notin dom E.
intros.
  pick_fresh x.
  exists x ; auto.
Qed.
```

[1] A cofinite subset of a set X is a subset Y whose complement in X is a finite set.

The proof is essentially just a call to the `pick_fresh` from the *Formal Metatheory* library. This tactic collects all variables in the environment, and then chooses a variable that is distinct from all these variables. The proof that x satisfies the necessary freshness condition is also handled automatically. The use of the locally nameless approach, and in particular the use of the *Formal Metatheory* library, meant that little of our soundness proof needs to be concerned with alpha-equivalence or freshness.

7.2.2 Environments

Consider this definition of a simple linear lambda calculus:

$$\frac{}{x : \tau \vdash x : \tau}\textsc{Var} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x \cdot e : \tau \to \tau'}\textsc{Abs} \qquad \frac{\Gamma \vdash f : \tau \to \tau' \qquad \Delta \vdash e : \tau}{\Gamma, \Delta \vdash f\ e : \tau'}\textsc{App}$$

Suppose we want to prove an exchange lemma:

> **Lemma** (Exchange). If $\Gamma, \Delta \vdash e : \tau$, then $\Delta, \Gamma \vdash e : \tau$.

In informal practice, we might not even consider proving this lemma, because we might represent environments as (multi-)sets so that Γ, Δ and Δ, Γ are the same environment. In a formal (constructive) proof, however, we must choose a concrete representation. If we represent environments by lists, we must prove *Exchange*, since Γ, Δ and Δ, Γ are certainly not the same list. Unfortunately, *the definition of the typing relation above does not permit Exchange*: *Exchange* does not hold.

One solution is to choose a different concrete representation. For example, if we choose to represent environments by sorted lists of pairs of variables and types (for some arbitrary ordering relation) then Γ, Δ and Δ, Γ again denote the same environment. Although this approach may work well, our definition of an environment is actually taken from the *Formal Metatheory* library (discussed in Section 7.2.1), which we preferred to use unmodified. p. 161

We must therefore explicitly allow for exchange in the type system. The traditional way is to include the exchange lemma as an axiom[1]:

$$\frac{\Gamma, \Delta, \Theta \vdash e : \tau}{\Gamma, \Theta, \Delta \vdash e : \tau}\textsc{Exch}$$

The downside of this approach is that the inversion lemmas for the typing relation become more difficult to state. For example, in the original type system we could prove

> **Lemma** (Inversion lemma for application). If $\Gamma \vdash f\ e : \tau$, then there exists Δ, Θ such that $\Gamma = \Delta, \Theta$, and there exists τ' such that $\Delta \vdash f : \tau' \to \tau$ and $\Theta \vdash e : \tau'$.

In the modified type system, however, this lemma no longer holds. Instead, we would have to allow for an application of the exchange rule, which makes the inversion lemma harder to state. This problem is amplified by the presence of other substructural rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau}\textsc{Weak} \qquad \frac{\Gamma, y : \tau', z : \tau' \vdash e : \tau}{\Gamma, x : \tau' \vdash e[y := x, z := x] : \tau}\textsc{Contr}$$

[1] It is often presented as

$$\frac{\Gamma, \Delta \vdash e : \tau}{\Delta, \Gamma \vdash e : \tau}\textsc{Exch}'$$

but that rule is not strong enough. In particular, we cannot show Exch from Exch′.

With these two rules, the inversion lemma for application becomes very difficult to state indeed. Fortunately, for an affine (as opposed to linear) substructural type system such as ours, weakening is unrestricted so that rule WEAK can easily be integrated into the typing rule for variables. We do however need to control contraction (only unique variables can be used more than once), and it is not so obvious how to integrate CONTR into the other rules.

The solution we adopt is the one described in (Walker, 2005), where it is attributed to (Cervesato and Pfenning, 2002). We define a generic context splitting operation as follows:

$$\frac{}{\emptyset = \emptyset \circ \emptyset}\textsc{Split-Empty} \qquad \frac{E = E_1 \circ E_2}{E, x : \tau = E_1, x : \tau \circ E_2}\textsc{Split-Left}$$

$$\frac{E = E_1 \circ E_2 \qquad \text{non-unique } \tau}{E, x : \tau = E_1, x : \tau \circ E_2, x : \tau}\textsc{Split-Both} \qquad \frac{E = E_1 \circ E_2}{E, x : \tau = E_1 \circ E_2, x : \tau}\textsc{Split-Right}$$

We can use the context splitting operation in the rule for application as follows:

$$\frac{\Gamma \vdash f : \tau \rightarrow \tau' \qquad \Delta \vdash e : \tau}{\Gamma \circ \Delta \vdash f\ e : \tau'}\textsc{App}'$$

With this rule, lemma *Exchange* becomes admissible because we can prove an auxiliary result that if $E = E_1 \circ E_2$ then $E = E_2 \circ E_1$. This approach is attractive for two reasons. First, the inversion lemma is straightforward to state and prove. Second, we can reason about context
p. 183 splitting as a separate notion, and we will do so extensively (Section 7.5.10). This means that in those proofs where we need to reason about reordering of the environment (in particular lemmas
p. 196 *preservation_commute* and *preservation_assoc*, Section 7.7), this reasoning is explicit and usually done in separate lemmas.

7.2.3 Boolean expressions

In our type system, we allow for arbitrary boolean expressions as uniqueness attributes: $t^\bullet$, $t^\times$, t^u, $t^{u \vee v}$, $t^{u \wedge v}$ and $t^{\neg u}$ are all valid types. Moreover, we want to identify "equivalent" boolean expressions: $t^{u \vee v}$ and $t^{v \vee u}$ are the same type. In other words, we want to identify uniqueness attributes (boolean expressions) that are equivalent under the usual set of axioms (Huntington's
p. 225 Postulates; see Appendix A.1).

Perhaps the most obvious solution is to quotient boolean expressions by Huntington's Postulates, and formally regard uniqueness attributes as equivalence classes of boolean expressions rather than boolean expressions. Since the equivalence class $[u \vee v]$ and $[v \vee u]$ are the same class (since both expressions are equivalent), the types $t^{[u \vee v]}$ and $t^{[v \vee u]}$ are then also identified.

Unfortunately, this solution is difficult to adopt for two reasons. First, since the equivalence class of a boolean expression is infinite, we would need to use co-induction to define the classes—not difficult conceptually, but technically awkward nevertheless. The other complication is that in our type system, and hence in the formalization, we do not distinguish between types and
p. 143 attributes (this is a key contribution of the Chapter 6). An attributed type t^u is syntactic sugar for the application of a special type constant `Attr` to two arguments (`Attr` t u); a kind system weeds out ill-formed types. This approach does not combine well with treating uniqueness attributes as equivalence classes.

Instead, we explicitly allow to replace a type by an equivalent type as a non-syntax directed rule:

$$\frac{\Gamma \vdash e : \tau|_{\mathit{fv}} \qquad \tau \approx \tau'}{\Gamma \vdash: e : \tau'|_{\mathit{fv}}}\textsc{Equiv}$$

As it turns out, adding this lemma does not make the inversion lemmas more difficult to state (we prove the inversion lemmas in Section 7.6.6; see also Section 7.2.2). Moreover, adding this rule is p. 195, p. 163 (almost) sufficient to be able to replace a type anywhere in a typing derivation[1]; in particular, it is sufficient to be able to replace a type in an environment (lemma *typ_equiv_env*, Section 7.6.5). p. 194 We will discuss the type equivalence relation proper in Section 7.3.2. p. 166

7.3 Inversion

As we saw in the previous section, adding additional typing rules makes forward reasoning easier, but backward reasoning more difficult. For example, if we add a contraction rule to the type system, it becomes trivial to prove $\Gamma, x : \tau' \vdash e[y := x, z := x] : \tau$ from $\Gamma, y : \tau', z : \tau' \vdash e : \tau$ (forward reasoning), but the inversion lemma for application becomes more difficult to state (backward reasoning). Generally, we want to make the definition of the type system permissive enough to facilitate forward reasoning, but not too permissive to avoid complicating backward reasoning. We already saw one example of this: rather than adding a separate contraction rule, it is better to integrate contraction into the other rules (by introduction a generic context splitting operation; see Section 7.2.2). In this section, we will see a number of other examples of this p. 163 tension between forward and backward reasoning.

7.3.1 Domain subtraction

In the definition of the type system we make use of a domain subtraction operation, denoted $⩥_x \mathit{fv}$, which removes x from the domain of *fv*. In this section we discuss how we should define this operation. In particular: if x occurs more than once in the domain of *fv*, should domain subtraction remove all of them, or only the first? Using an example, we will see that we will need to choose the latter option to be able to use backwards reasoning.

We will need a few definitions first. An environment is well-formed if it is *ok* and well-kinded: that is, if every variable occurs at most once in its domain and all the types in the codomain of the environment have the same kind. Two environments are equivalent, denoted $\Gamma \cong_k \Gamma'$, if they are both well-formed and map the same variables to the same types (k denotes the kind of the types in the codomain of the environments; these definitions are given formally in Section 7.4). p. 169

An important lemma is that if $\Gamma \vdash e : \tau|_{\mathit{fv}}$, $\Gamma \cong_* \Gamma'$ and $\mathit{fv} \cong_{\mathcal{U}} \mathit{fv}'$, then $\Gamma' \vdash e : \tau|_{\mathit{fv}'}$ (Lemma *env_equiv_typing*, Section 7.6.5). This lemma is important because it allows to change the order p. 194 of the assumptions in the environment (Lemma *exchange*) or replace a type by an equivalent type in an environment (Lemma *typ_equiv_env*). The proof of the lemma is by induction on the typing relation.

Consider the case for the rule for abstraction. We know that $\Gamma \cong_* \Gamma'$ and $\mathit{fv}' \cong_{\mathcal{U}} \mathit{fv}''$. The induction hypothesis gives us[2]

[1] In the typing rule for variables, we must also be careful to allow for a different (but equivalent) attribute in E and *fv*.

[2] This is a minor simplification of the proof; in the actual proof, we need to distinguish between the case where the bound variable of the abstraction is used in the body (the case which is shown here), and the case where it is not used. We do not discuss the second (easier) case.

$$(\Gamma, x : \tau \cong_* \Gamma', x : \tau) \rightarrow (\mathit{fv}', x : \nu \cong_{\mathcal{U}} \mathit{fv}) \rightarrow (\Gamma', x : \tau \vdash e^x : \tau'|_{\mathit{fv}})$$

and we have to show that

$$\Gamma' \vdash \lambda \cdot e : \tau \xrightarrow{\bigvee \mathit{fv}'} \tau'|_{\mathit{fv}''}$$

Replacing the attribute on the arrow by an equivalent one gives $\Gamma' \vdash \lambda \cdot e : \tau \xrightarrow{\bigvee \mathit{fv}''} \tau'|_{\mathit{fv}''}$, at which point we can apply the typing rule for abstraction. Remains to show that

$$\Gamma', x : \tau \vdash e^x : \tau'|_{\mathit{fv}}$$

where we know that $\mathit{fv}'' = \mathrel{|\!\!\rhd}_x \mathit{fv}$ and $x \notin \Gamma \cup \mathit{fv}''$. We can use the induction hypothesis to complete the proof, but only if we can prove its two premises. The first one is straightforward, but the second is more tricky:

$$\mathit{fv}', x : \nu \cong_{\mathcal{U}} \mathit{fv}$$

To be able to show this equivalence, we need to be able to show that fv is well-formed; in particular, we need to be able to show that it is *ok* (every variable occurs at most once in its domain). Since $\mathrel{|\!\!\rhd}_x \mathit{fv} = \mathit{fv}''$, we know that $\mathrel{|\!\!\rhd}_x \mathit{fv}$ is *ok* because $\mathit{fv}'' \cong_{\mathcal{U}} \mathit{fv}'$, and we know that $x \notin \mathrel{|\!\!\rhd}_x \mathit{fv}$ because $x \notin \mathit{fv}''$. However, it now depends on the definition of domain subtraction ($\mathrel{|\!\!\rhd}$) whether we can show that fv is *ok*.

If $\mathrel{|\!\!\rhd}_x \mathit{fv}$ removes *all* occurrences of x from fv, then we will be unable to complete the proof: even if $\mathrel{|\!\!\rhd}_x \mathit{fv}$ is ok, that does not allow us to conclude anything about the well-formedness of fv. On the other hand, if $\mathrel{|\!\!\rhd}_x \mathit{fv}$ only removes the *first* occurrence of x, then fv can contain at most one more assumption about x than $\mathrel{|\!\!\rhd}_x \mathit{fv}$; if additionally we know that $x \notin \mathrel{|\!\!\rhd}_x \mathit{fv}$, then we can conclude that fv must be ok.

Hence, we conclude that domain subtraction must remove the first occurrence of a variable only. This makes forward reasoning slightly more difficult, since where before we could prove a lemma that $x \notin \mathrel{|\!\!\rhd}_x \mathit{fv}$, now that only holds if fv is *ok*. Fortunately, we always require environments to be well-formed, so this is no problem in practice. On the other hand, backwards reasoning (proving that fv is *ok* given that $\mathrel{|\!\!\rhd}_x \mathit{fv}$ is *ok* and $x \notin \mathrel{|\!\!\rhd}_x \mathit{fv}$) is impossible if domain subtraction removes all variables from the domain of an environment.

7.3.2 Type equivalence

Huntington's Postulates give us an equivalence relation $\approx_{\mathrm{B}}$ on types. For example, we have that $u \vee v \approx_{\mathrm{B}} v \vee u$ (commutativity of disjunction) or $u \wedge \bullet \approx_{\mathrm{B}} u$ (identity element for conjunction). We want to extend this equivalence relation to a more general equivalence relation ($\approx_{\mathrm{T}}$), which is effectively ($\approx_{\mathrm{B}}$) extended with a closure rule for type application:

$$\frac{\tau \approx_{\mathrm{B}} \tau'}{\tau \approx_{\mathrm{T}} \tau'} \qquad \frac{\tau_1 \approx_{\mathrm{T}} \tau_1' \qquad \tau_2 \approx_{\mathrm{T}} \tau_2'}{\tau_1\ \tau_2 \approx_{\mathrm{T}} \tau_1'\ \tau_2'}$$

This allows us to derive that $\tau^{u \vee v} \approx_{\mathrm{T}} \tau^{v \vee u}$ or that if $a \approx_{\mathrm{T}} a'$, then $a \xrightarrow{u} b \approx_{\mathrm{T}} a' \xrightarrow{u} b$ (recall that $a \xrightarrow{u} b$ is syntactic sugar for `Attr` (`Arr` $a\ b$) u). However, we also occasionally need to reason backwards on the typing equivalence relation: if we know that $\tau^{\nu} \approx_{\mathrm{T}} \tau^{\nu'}$, we would like to be able prove that $\nu \approx_{\mathrm{T}} \nu'$.

It would seem that the easiest way to prove that would be to prove the following inversion lemma: if $\tau_1\ \tau_2 \approx_{\mathrm{T}} \tau_1'\ \tau_2'$, then $\tau_1 \approx_{\mathrm{T}} \tau_1'$ and $\tau_2 \approx_{\mathrm{T}} \tau_2'$. Unfortunately, that lemma does not hold. Recall that we do not distinguish between types and attributes in our type system. That is, the "attribute" $\nu \vee \nu'$ is a type (which happens to have kind $\mathcal{U}$). Moreover, $\nu \vee \nu'$ is really syntactic sugar for the application of a special type constant `Or` of kind $\mathcal{U} \to \mathcal{U} \to \mathcal{U}$ to two arguments (`Or` $\nu\ \nu'$). By Huntington's Postulates we have that $\nu \vee \nu' \approx_{\mathrm{T}} \nu' \vee \nu$, or desugared: `Or` $\nu\ \nu' \approx_{\mathrm{T}}$ `Or` $\nu'\ \nu$ for all ν and ν'. If the inversion lemma were true, we would thus be able to conclude that $\nu \approx_{\mathrm{T}} \nu'$, for *any* ν and ν'.

So, to make backwards reasoning possible, we need to redefine $\approx_{\mathrm{T}}$ slightly:

$$\frac{\nu \approx_{\mathrm{B}} \nu' \qquad \nu : \mathcal{U}, \nu' : \mathcal{U}}{\nu \approx_{\mathrm{T}} \nu'} \qquad \frac{\tau_1 \approx_{\mathrm{T}} \tau_1' \qquad \tau_2 \approx_{\mathrm{T}} \tau_2' \qquad (\tau_1\tau_2) \not: \mathcal{U},}{\tau_1\ \tau_2 \approx_{\mathrm{T}} \tau_1'\ \tau_2'}$$

(In addition, we need to introduce reflexivity, commutativity and transitivity rules; they were previously implied by ($\approx_{\mathrm{B}}$)). We can now prove the following inversion lemma: if $\tau_1\ \tau_2 \approx_{\mathrm{T}} \tau_1'\ \tau_2'$, and $(\tau_1\ \tau_2)$ does not have kind $\mathcal{U}$, then $\tau_1 \approx_{\mathrm{T}} \tau_1'$ and $\tau_2 \approx_{\mathrm{T}} \tau_2'$. Restricting the closure rule to types of kind other than $\mathcal{U}$ is not strictly necessary to prove this inversion lemma, but makes proving other lemmas easier (for example, Lemma *typ_equiv_BA_equiv*, Section 7.5.11) without p. 188
reducing the relation: closure for types of kind $\mathcal{U}$ is already implied by Huntington's Postulates.

This modification to the type equivalence relation has an additional benefit. Recall the following rule for context splitting:

$$\frac{E = E_1 \circ E_2 \qquad \text{non-unique } \tau}{E, x : \tau = E_1, x : \tau \circ E_2, x : \tau}\ \textsc{Split-Both}$$

Since the context splitting operation is applied both to typing environments (Γ) and the lists of free variables (*fv*), we give the following two axioms to prove "non-unique":

$$\frac{\nu \approx_{\mathrm{T}} \times}{\text{non-unique}(\tau^{\nu})}\ \mathrm{NU}_* \qquad \frac{\nu \approx_{\mathrm{T}} \times}{\text{non-unique}(\nu)}\ \mathrm{NU}_{\mathcal{U}}$$

Now consider proving the following lemma: if $\tau \xrightarrow{\nu} \tau'$ is non-unique, then $\nu \approx_{\mathrm{T}} \times$. The proof proceeds by inversion on non-unique$(\tau \xrightarrow{\nu} \tau')$. The case for rule NU_* is trivial, but how can we dismiss the case for rule $\mathrm{NU}_{\mathcal{U}}$? Without the kind requirements added to the type equivalence relation, we would have to show that it is impossible that $\tau \xrightarrow{\nu} \tau'$ is equivalent to $\times$ by Huntington's Postulates; not an easy proof![1]

7.3.3 Evaluation contexts

The operational semantics we use is the call-by-need semantics by Maraist *et al.* (Maraist et al., 1998). In this semantics, the definition of evaluation depends on the notion of an *evaluation context*, which is essentially a term with a hole in it (the difference between an evaluation context and the more general notion of a "context" (Barendregt, 1984) is that in an evaluation context we restrict where the hole can appear in the term). There are various ways in which we can formalize an evaluation context in Coq.

[1] If the proof seems trivial, perhaps the reader would like to attempt an even easier proof: prove that it is impossible to construct a proof using Huntington's postulates that "true" is equivalent to "false" without using an interpretation function, as it is not clear how to define an interpretation function for the broader class of types (rather than just the types of kind $\mathcal{U}$).

In simple cases, we can follow informal practice and define a context E inductively, followed by a definition of plugging a term M into the hole in the context $E[M]$. This is the approach taken in (Biernacka and Biernacki, 2007), for instance, but it does not apply here because we need the definition of $E[M]$ *when defining* $E[]$.

Another approach (Dubois, 2000) is to define a context as an ordinary function on terms, and then (inductively) define which functions on terms can be regarded as evaluation contexts. This is an attractive and elegant approach, but does not work so well in the locally-nameless approach: since some evaluation contexts place a term within the scope of a binder but others do not, we must distinguish between *binding* contexts which have the property that if t^x is a term for some fresh x, then $E[x]$ is also a term, and *regular* contexts (which do not have this property).

For example, consider the proof that reduction is regular:

Lemma (Regularity of reduction). If $e \longmapsto e'$, then both e and e' are locally closed[1].

The proof is by induction on $e \longmapsto e'$. In the case for the closure rule, we know that $E[e]$ and $E[e']$ are locally closed, and we have to show that e and e' are locally closed. However, we may or may not be able to show this (depending on whether E is a regular or a binding context). Thus, we need to distinguish the "closing" evaluation contexts from the others, at which point the elegance of the approach starts disappearing. We now need two closure rules (one for closing and one for regular contexts) and we have introduced a new characterization of evaluation contexts that we will need to reason about.

To avoid having to reason about closing contexts and regular contexts, we instead inline the definition of the evaluation contexts into the definition of the reduction relation. This gives only one more rule than when giving a closure rule for regular contexts and a closure rule for closing contexts, and moreover, the resulting closure rules correspond to intuitive notions about the semantics.

We still need to define the notion of an evaluation context, because the reduction relation depends on it in the other rules too. As mentioned before, we cannot define the notion of a context separately from plugging a term into the hole. The solution we adopt is to define E as a binary relation between a term and a free variable, where $E\ t\ x$ should be read as t evaluates x (there is an evaluation context E such that $t = E[x]$). This gives good inversion principles (suitable for backwards reasoning) and combines well with the locally nameless approach.

[1] We mentioned before that in the locally nameless approach to formal metatheory we distinguish between bound variables, represented by De Bruijn indices, and free variables, represented by ordinary names. A term is locally closed if it does not contain any "unbound bound variables"; that is, if it does not contain any De Bruijn indices without a corresponding binder.

7.4 Definitions

7.4.1 Types

A type is either a type constant or the application of one type to another.

```
Inductive typ : Set :=
    (** Type application *)
  | typ_app : typ → typ → typ
    (** Type constants *)
  | ARR : typ
  | ATTR : typ
  | UN : typ
  | NU : typ
  | OR : typ
  | AND : typ
  | NOT : typ.
```

For convenience, we define a number of functions to denote commonly used types, and some custom notation for attributed types.

```
Definition bi_app (f a b : typ) : typ := typ_app (typ_app f a) b.

Definition arr (a b : typ) : typ := bi_app ARR a b.
Definition attr (t u : typ) : typ := bi_app ATTR t u.
Definition or (u v : typ) : typ := bi_app OR u v.
Definition and (u v : typ) : typ := bi_app AND u v.
Definition not (u : typ) : typ := typ_app NOT u.

Notation "t ' u" := (attr t u) (at level 60).
Notation "a ⟨ u ⟩ b" := ((arr a b) ' u) (at level 68).
```

(A subset of the) language of types forms a boolean algebra.

```
Module TypeAsBooleanAlgebra <: BooleanAlgebraTerm.

Definition trm := typ.
Definition true := UN.
Definition false := NU.
Definition or := or.
Definition and := and.
Definition not := not.

End TypeAsBooleanAlgebra.

Module BA := BooleanAlgebra TypeAsBooleanAlgebra.
```

7.4.2 Kinding relation

The definition of kinds.

```
Inductive kind : Set :=
  | kind_T : kind
  | kind_U : kind
  | kind_star : kind
  | kind_arr : kind → kind → kind.
```

Kinding relation.

```
Inductive kinding : typ → kind → Prop :=
  | kinding_app : ∀ t1 t2 k1 k2,
      kinding t1 (kind_arr k1 k2) →
      kinding t2 k1 →
      kinding (typ_app t1 t2) k2
  | kinding_ARR : kinding ARR (kind_arr kind_star (kind_arr kind_star kind_T))
  | kinding_ATTR : kinding ATTR (kind_arr kind_T (kind_arr kind_U kind_star))
  | kinding_UN : kinding UN kind_U
  | kinding_NU : kinding NU kind_U
  | kinding_OR : kinding OR (kind_arr kind_U (kind_arr kind_U kind_U))
  | kinding_AND : kinding AND (kind_arr kind_U (kind_arr kind_U kind_U))
  | kinding_NOT : kinding NOT (kind_arr kind_U kind_U).
```

```
Hint Constructors kinding.
```

Equivalence between types

```
Inductive typ_equiv : typ → typ → Prop :=
  (** The type equivalence includes the boolean equivalence relation *)
  | typ_equiv_attr : ∀ u v,
      kinding u kind_U →
      kinding v kind_U →
      BA.equiv u v →
      typ_equiv u v
  (** Closure (does not apply to types of kind U) *)
  | typ_equiv_app : ∀ s t s' t',
      ¬ kinding (typ_app s t) kind_U →
      typ_equiv s s' →
      typ_equiv t t' →
      typ_equiv (typ_app s t) (typ_app s' t')
  (** Structural rules *)
  | typ_equiv_refl : ∀ t,
      typ_equiv t t
  | typ_equiv_sym : ∀ t s,
      typ_equiv t s → typ_equiv s t
  | typ_equiv_trans : ∀ t s r,
      typ_equiv t s → typ_equiv s r → typ_equiv t r.
```

```
Hint Constructors typ_equiv.
```

7.4.3 Environment

The definition of an environment comes from the Formal Metatheory library; we just need to instantiate it with our definition of a type.

```
Definition env : Set := Env.env typ.
```

An environment is well-formed if it is *ok* and well-kinded.

```
Definition env_kind (k : kind) : env → Prop :=
  env_prop (fun t ⇒ kinding t k).
```

```
Definition env_wf (E : env) (k : kind) : Prop :=
  ok E ∧ env_kind k E.
```

Two environments are considered equivalent if they both bind the same variables to equivalent types, and both are well-formed. For clarity, we introduce a special syntax to denote equivalence.

```
Definition env_equiv (E1 E2 : env) (k : kind) : Prop :=
  env_wf E1 k ∧ env_wf E2 k ∧
  (∀ x t, binds x t E1 → ∃ t', binds x t' E2 ∧ typ_equiv t t') ∧
  (∀ x t, binds x t E2 → ∃ t', binds x t' E1 ∧ typ_equiv t t').
```

```
Notation "E1 ≅ E2" := (env_equiv E1 E2) (at level 70).
```

The definition of the context split operation was explained in the introduction. The context split is used both to split the typing environment E and the list of free variables and their uniqueness attributes *fvars*. For this reason, we introduce a separate "non_unique" property of types, which applies to types of kind $*$ when they have a non-unique attribute, and to attributes (types of kind $\mathcal{U}$) when they are non-unique themselves.

```
Reserved Notation "'split_context' E 'as' ( E1 ; E2 )".
```

```
Inductive non_unique : typ → Prop :=
  | NU_star : ∀ t u,
      typ_equiv u NU → non_unique (t ' u)
  | NU_U : ∀ u,
      typ_equiv u NU → non_unique u.
```

```
Inductive context_split : env → env → env → Prop :=
  | split_empty :
      split_context empty as (empty ; empty)
  | split_both : ∀ E E1 E2 x t, split_context E as (E1 ; E2) → non_unique t →
      split_context (E & x ¬ t) as (E1 & x ¬ t; E2 & x ¬ t)
  | split_left : ∀ E E1 E2 x t, split_context E as (E1 ; E2) →
      split_context (E & x ¬ t) as (E1 & x ¬ t ; E2)
  | split_right : ∀ E E1 E2 x t, split_context E as (E1 ; E2) →
      split_context (E & x ¬ t) as (E1 ; E2 & x ¬ t)
  where
    "'split_context' E 'as' ( E1 ; E2 )" := (context_split E E1 E2).
```

```
Hint Constructors non_unique context_split.
```

7.4.4 Operations on the typing context

Disjunction of all types on the range of the environment

```
Fixpoint rng (E : env) : typ :=
  match E with
  | nil ⇒ NU
  | (x, u) :: tail ⇒ or u (rng tail)
  end.
```

Remove the first occurrence of *x* in *E*

```
Fixpoint dsub (x : var) (E : env) {struct E} : env :=
  match E with
  | nil ⇒ nil
  | (y, t) :: tail ⇒ if x == y then tail else (y, t) :: dsub x tail
  end.
```

Call *dsub* for every *x* in *xs*.

```
Fixpoint dsub_list (xs : list var) (E : env) : env :=
  match xs with
  | nil ⇒ E
  | x :: xs' ⇒ dsub_list xs' (dsub x E)
  end.
```

Variation on *dsub_list* working on sets *xs* rather than lists.

```
Definition dsub_vars (xs : vars) (E : env) : env := dsub_list (S.elements xs) E.
```

7.4.5 Typing relation

The rule for variables *typing_var* is subtle in two ways: since it only requires that *binds x (t ' u) E*, and therefore allows for other assumptions in *E*, it implicitly allows weakening on *E*. However, it is much more strict on *fvars* (the only assumption in *fvars* must be the assumption *x* : *u*; hence, no weakening is allowed on *fvars*). This is important, because while additional assumptions in *E* cannot affect the type of a term, additional assumptions in *fvars* can (by unnecessarily forcing an abstraction to be unique). The typing rule for abstraction uses the cofinite quantification discussed in the introduction.

```
Reserved Notation "E ⊢ t : T | fvars" (at level 69).
```

```
Inductive typing : env → trm → typ → env → Prop :=
  | typing_var : ∀ E x t u v,
      env_wf E kind_star →
      binds x (t ' u) E →
      typ_equiv u v →
      E ⊢ (trm_fvar x) : t ' u | x ¬ v
  | typing_abs : ∀ L E a b e fvars',
      (∀ x fvars, x \notin L → fvars' = dsub x fvars →
        (E & x ¬ a) ⊢ e ^ x : b | fvars) →
      E ⊢ (trm_abs e) : a ⟨ rng fvars' ⟩ b | fvars'
  | typing_app : ∀ E E1 E2 fvars fvars1 fvars2 e1 e2 a b u,
      E1 ⊢ e1 : a ⟨ u ⟩ b | fvars1 →
      E2 ⊢ e2 : a | fvars2 →
      split_context E as (E1 ; E2) → env_wf E kind_star →
      split_context fvars as (fvars1 ; fvars2) → env_wf fvars kind_U →
      E ⊢ (trm_app e1 e2) : b | fvars
  | typing_equiv : ∀ E e a b fvars,
      E ⊢ e : a | fvars →
      typ_equiv a b →
      E ⊢ e : b | fvars
  where "E ⊢ t : T | fvars" := (typing E t T fvars).
```

```
Hint Constructors typing.
```

7.4.6 Semantics

We treat "let $x = y$ in z" as syntactic sugar for $(\lambda x \cdot z)\, y$.

```
Notation "'lt' x 'in' y" := (trm_app (trm_abs y) x) (at level 70).
```

Definition of *answer*, *eval* and *red* as in Maraist et al. (1998); again, we're using cofinite quantification.

```
Inductive answer : trm → Prop :=
  | answer_abs : ∀ M, term (trm_abs M) →
      answer (trm_abs M)
  | answer_let : ∀ L M A, term (lt M in A) →
      (∀ x, x \notin L → answer (A ^ x)) →
      answer (lt M in A).
```

Definition of an evaluation context

```
Inductive evals : trm → var → Prop :=
  | evals_hole : ∀ x,
      evals (trm_fvar x) x
  | evals_app : ∀ x E M, evals E x →
      evals (trm_app E M) x
  | evals_let : ∀ L x E M,
      (∀ y, y \notin L → evals (E ^ y) x) →
      evals (lt M in E) x
  | evals_dem : ∀ L x E M, evals E x →
      (∀ y, y \notin L → evals (M ^ y) y) →
      evals (lt E in M) x.
```

```
Hint Constructors evals.
```

As mentioned before, the reduction relation we use is the standard reduction from Maraist et al. (1998) except that *red_value* is defined as in (Maraist et al., 1998, Section "*On types and logic*", p. 38), adapted for standard reduction. None of these rules adjust any of the bound variables (which are after all De Bruijn variables); this is justified by lemma *red_regular*, given in Section 7.5.7, which states that the reduction relation is defined for *locally closed* terms only (that is, they may contain free variables, but no unbound De Bruijn indices).

```
Inductive red : trm → trm → Prop :=
   (** Standard reduction rules *)
  | red_value : ∀ L M N, term (lt (trm_abs M) in N) →
       (∀ x, x \notin L → evals (N ^ x) x) →
       red (lt (trm_abs M) in N) (N ^^ trm_abs M)
  | red_commute : ∀ L M A N, term (trm_app (lt M in A) N) →
       (∀ x, x \notin L → answer (A ^ x)) →
       red (trm_app (lt M in A) N) (lt M in trm_app A N)
  | red_assoc : ∀ L M A N, term (lt (lt M in A) in N) →
       (∀ x, x \notin L → answer (A ^ x)) →
       (∀ x, x \notin L → evals (N ^ x) x) →
       red (lt (lt M in A) in N) (lt M in lt A in N)
   (** Compatible closure *)
  | red_closure_app : ∀ E E' M, term (trm_app E M) →
       red E E' →
       red (trm_app E M) (trm_app E' M)
  | red_closure_let : ∀ L E E' M, term (lt M in E) →
       (∀ x, x \notin L → red (E ^ x) (E' ^ x)) →
       red (lt M in E) (lt M in E')
  | red_closure_dem : ∀ L E0 E0' E1, term (lt E0 in E1) →
       red E0 E0' →
       (∀ x, x \notin L → evals (E1 ^ x) x) →
       red (lt E0 in E1) (lt E0' in E1).

Hint Constructors answer red.
```

7.5 Preliminaries

7.5.1 Some additional lemmas about *ok* and *binds*

Every variable occurs at most once.

```
Lemma ok_mid : ∀ (E2 E1 : env) x t,
  ok (E1 & x ¬ t & E2) → x # E1 ∧ x # E2.
```

By induction on *E2*.

If two environments are both *ok* and their domains are disjoint, then their concatenation is also *ok*.

```
Lemma ok_concat : ∀ (E2 E1 : env),
  ok E1 → ok E2 →
  (∀ x, x \in dom E1 → x \notin dom E2) →
  (∀ x, x \in dom E2 → x \notin dom E1) →
  ok (E1 & E2).
```

By induction on *E2*.

If the concatenation of two environments is *ok*, then their domains must be disjoint.

```
Lemma ok_concat_inv_2 : ∀ (E2 E1 : env),
  ok (E1 & E2) →
```

(∀ *x*, *x* \in *dom E1* → *x* \notin *dom E2*) ∧
(∀ *x*, *x* \in *dom E2* → *x* \notin *dom E1*).
By induction on *E2*.

We can change the order of the assumptions in an environment without affecting *ok*.
Lemma *ok_exch* : ∀ (*E1 E2* : *env*),
ok (*E1* & *E2*) → *ok* (*E2* & *E1*).
By induction on *E1*.

Generalization of *ok_exch*.
Lemma *ok_exch_3* : ∀ (*E1 E2 E3* : *env*),
ok (*E1* & *E2* & *E3*) → *ok* (*E1* & *E3* & *E2*).
Follows from *ok_concat_inv_2* and *ok_exch*.

If an environment binds a variable *x*, then *x* must be in the domain of the environment.
Lemma *binds_in_dom* : ∀ (*A* : Set) *x* (*T* : *A*) *E*,
binds x T E → *x* \in *dom E*.
By induction on *E*.

Inverse of *binds_in_dom*: if a variable *x* is in the domain of an environment, then the environment must bind *x*.
Lemma *in_dom_binds* : ∀ (*E* : *env*) *x*,
x \in *dom E* → ∃ *t*, *binds x t E*.
By induction on *E*.

Binds is unaffected by the order of the assumptions in an environment.
Lemma *binds_exch* : ∀ (*E1 E2* : *env*) *x t*, *ok* (*E1* & *E2*) →
binds x t (*E1* & *E2*) →
binds x t (*E2* & *E1*).
Follows from *ok_concat_inv_2*.

Generalization of *binds_exch*.
Lemma *binds_exch_3* : ∀ (*E1 E2 E3* : *env*) *x t*, *ok* (*E1* & *E2* & *E3*) →
binds x t (*E1* & *E2* & *E3*) →
binds x t (*E1* & *E3* & *E2*).
Trivial.

A variable can only be bound to one type.
Lemma *binds_head_inv* : ∀ (*E* : *env*) *x a b*,
binds x a (*E* & *x* ¬ *b*) → *a* = *b*.
Trivial.

7.5.2 Renaming Lemmas

All these renaming lemmas are proven in the same way. We first prove a substitution lemma which states that the names of the free variables do not matter, and then we prove the renaming lemma using the substitution lemma and the fact that *t* ^^ *u* = [*x* ⇝ *u*] *t* ^ *x*, as long as *x* \notin *fv t*.

If *e* is an answer, then it will still be an answer when we rename any of its free variables.
Lemma *subst_answer* : ∀ *e x y*,

answer e ⟶ *answer* ([*x* ⇝ *trm_fvar y*] *e*).
By induction on *answer e*.

If *t* ^ *x* is an answer, then *t* ^ *y* will also be an answer for any *y*.
Lemma *answer_rename* : ∀ *x y t*,
x \notin *fv t* ⟶
answer (*t* ^ *x*) ⟶ *answer* (*t* ^ *y*).
Follows from *subst_answer*.

If *M* evaluates *x* (by the evaluation context relation defined previously) then if we rename *y* to *z* in *M*, *M* will still evaluate *x* if $x \neq y$, or *M* will evaluate *z* otherwise.
Lemma *subst_evals* : ∀ *M x y z*,
evals M x ⟶ *evals* ([*y* ⇝ *trm_fvar z*] *M*) (if *x* == *y* then *z* else *x*).
By induction on *evals M x*.

If *M* ^ *x* evaluates *x*, then *M* ^ *y* will evaluate *y* for any *y*.
Lemma *evals_rename* : ∀ *M x y*,
x \notin *fv M* ⟶
evals (*M* ^ *x*) *x* ⟶ *evals* (*M* ^ *y*) *y*.
Follows from *subst_evals*.

Specialization of *subst_evals*, excluding the case that $x = y$.
Lemma *subst_evals_2* : ∀ *M x y z*, $x \neq y$ ⟶
evals M x ⟶ *evals* ([*y* ⇝ *trm_fvar z*] *M*) *x*.
Follows from *subst_evals*.

Generalization of *evals_rename*.
Lemma *evals_rename_2* : ∀ *M x y z*,
x \notin *fv M* ⟶ $z \neq x$ ⟶
evals (*M* ^ *x*) *z* ⟶ *evals* (*M* ^ *y*) *z*.
Follows from *subst_evals_2*.

If *e* reduces to *e'*, then if we rename a free variable by another in both terms the reduction relation will still hold.
Lemma *subst_red* : ∀ *e e' x y* ,
red e e' ⟶ *red* ([*x* ⇝ *trm_fvar y*] *e*) ([*x* ⇝ *trm_fvar y*] *e'*).
By induction on *red e e'*; uses *subst_evals_2*.

If *M* ^ *x* reduces to *N* ^ *x*, then *M* ^ *y* will reduce to *N* ^ *y* for any *y*.
Lemma *red_rename* : ∀ *x y M N*,
x \notin *fv M* ⟶ *x* \notin *fv N* ⟶
red (*M* ^ *x*) (*N* ^ *x*) ⟶ *red* (*M* ^ *y*) (*N* ^ *y*).
Follows trivially from *subst_read*.

7.5.3 Term opening

Auxiliary lemma used to prove *in_open*, below.
Lemma *in_open_aux* : ∀ *M x y k l*, $x \neq y$ ⟶
x \in *fv* ({*k* ⇝ *trm_fvar y*} *M*) ⟶ *x* \in *fv* ({*l* ⇝ *trm_fvar y*} *M*).

By induction on *M*.

If *x* is free in *M* ^ *y* and $y \neq x$, then *x* is free in *M*.
Lemma *in_open* : ∀ *M x y*,
x \in *fv* (*M* ^ *y*) → *y* ≠ *x* → *x* \in *fv M*.
By induction on *M*; uses *in_open_aux*.

If *x* is free in *e*, then *x* will still be free when we substitute any bound variable in *e*.
Lemma *in_open_2* : ∀ *e e' k x*,
x \in *fv e* → *x* \in *fv* ({*k* ⇝ *e'*} *e*).
By induction on *e*.

If *x* is not free in *t*, then if we replace a bound variable *k* by *y* (where $x \neq y$) in *t*, *x* will still not be free in *t*.
Lemma *open_rec_fv* : ∀ *t x y k*,
x \notin *fv t* → *x* ≠ *y* → *x* \notin *fv* ({*k* ⇝ *trm_fvar y*} *t*).
By induction on *t*.

If *t* ^ *x* is locally-closed, then substituting for any bound variables larger than 0 in *t* has no effect.
Lemma *open_rec_term_open* : ∀ *t x*,
term (*t* ^ *x*) → ∀ *k t'*, $k \geq 1$ → *t* = {*k* ⇝ *t'*} *t*.
Trivial.

7.5.4 Domain subtraction

Subtracting an element *x* from the domain of an environment *fvars* has no effect when *x* was not in the domain of *fvars* to start with.
Lemma *dsub_not_in_dom* : ∀ (*fvars* : *env*) *x*, *x* # *fvars* →
fvars = *dsub x fvars*.
By induction on *fvars*.

$⩥_x$ removes *x* from a domain
Lemma *not_in_dom_dsub* : ∀ *fvars x*, *ok fvars* →
x # *dsub x fvars*.
By induction on *fvars*.

Removing *x* from *E* & *x* ¬ *t* gives *E*.
Lemma *dsub_head* : ∀ *E x t*, *dsub x* (*E* & *x* ¬ *t*) = *E*.
Trivial.

(⩥) distributes over (++).
Lemma *dsub_app* : ∀ *E1 E2 x*, *ok* (*E1* ++ *E2*) →
dsub x (*E1* ++ *E2*) = *dsub x E1* ++ *dsub x E2*.
By induction on *E1*.

(⩥) distributes over (&).
Corollary dsub_concat : ∀ *fvars1 fvars2 x*, *ok* (*fvars1* & *fvars2*) →
dsub x (*fvars1* & *fvars2*) = *dsub x fvars1* & *dsub x fvars2*.
Follows trivially from *dsub_app*.

If removing *x* from *fvars* is the empty environment, then *y* cannot be in the domain of *fvars*.

Lemma *not_in_dom_empty* : $\forall\ fvars\ x\ y$,

$dsub\ x\ fvars = empty \rightarrow x \neq y \rightarrow y$ \in $dom\ fvars \rightarrow False$.

By case analysis on *fvars*.

If E binds x and $x \neq y$, then $(\triangleright_y E)$ binds x.

Lemma *binds_dsub* : $\forall\ E\ x\ y\ T$,

$binds\ x\ T\ E \rightarrow x \neq y \rightarrow binds\ x\ T\ (dsub\ y\ E)$.

By induction on E.

Inverse property of *binds_dsub_inv*.

Lemma *binds_dsub_inv* : $\forall\ E\ x\ y\ T$,

$binds\ x\ T\ (dsub\ y\ E) \rightarrow x \neq y \rightarrow binds\ x\ T\ E$.

By induction on E.

If x is in the domain of E and $x \neq y$, then x is in the domain of $dsub\ y\ E$.

Lemma *in_dom_dsub* : $\forall\ E\ x\ y$,

x \in $dom\ E \rightarrow x \neq y \rightarrow x$ \in $dom\ (dsub\ y\ E)$.

By induction on E.

Inverse property of *in_dom_dsub*.

Lemma *in_dom_dsub_inv* : $\forall\ E\ x\ y$,

x \in $dom\ (dsub\ y\ E) \rightarrow x$ \in $dom\ E$.

By induction on E.

If x is in the domain of E and $\triangleright_y E$ is the empty environment, then x must be y.

Lemma *in_dom_dsub_empty* : $\forall\ E\ x\ y$,

x \in $dom\ E \rightarrow dsub\ y\ E = empty \rightarrow x = y$.

By induction on E.

If an environment is *ok*, it will still be *ok* if we remove a variable from its domain.

Lemma *ok_dsub* : $\forall\ E\ x$,

$ok\ E \rightarrow ok\ (dsub\ x\ E)$.

By induction on *ok E*.

If an environment is *ok*, it will still be *ok* if we add a single assumption about x to the environment, provided that x was not already in the domain of E.

Lemma *ok_dsub_inv* : $\forall\ E\ x$,

$ok\ (dsub\ x\ E) \rightarrow x\ \#\ dsub\ x\ E \rightarrow ok\ E$.

By induction on E.

If removing x from an environment yields the empty environment, then either the environment was empty to start with, or it is the singleton environment binding x.

Lemma *dsub_empty* : $\forall\ E\ x$,

$dsub\ x\ E = empty \rightarrow E = empty \lor \exists\ t, E = x \neg t$.

By induction on E.

7.5.5 Kinding properties

An attributed type consists of a base type and an attribute.

Lemma *kinding_star_inv* : $\forall\ t\ u,\ kinding\ (t\ '\ u)\ kind\_star \rightarrow$

kinding t kind_T ∧ *kinding u kind_U.*
By inversion on *kinding* (*t* ' *u*) *kind_star.*

The domain and codomain of functions must have kind ∗, and the attribute on the arrow must have kind *U*.
Lemma *kinding_fun_inv* : ∀ *a u b*, *kinding* (*a* ⟨ *u* ⟩ *b*) *kind_star* ⟶
kinding a kind_star ∧ *kinding u kind_U* ∧ *kinding b kind_star.*
By inversion on *kinding* (*a* ⟨ *u* ⟩ *b*) *kind_star.*

Every type has at most one kind.
Lemma *kind_unique* : ∀ *t k1*, *kinding t k1* ⟶
∀ *k2*, *kinding t k2* ⟶ *k1* = *k2.*
By induction on *kinding t k1.*

Equivalent types must have the same kind.
Lemma *typ_equiv_same_kind* : ∀ *t s*, *typ_equiv t s* ⟶
∀ *k*, *kinding t k* ⟷ *kinding s k.*
By induction on *typ_equiv t s*; uses *kind_unique.*

or a b has kind *U* if *a* and *b* have kind *u*.
Lemma *kinding_or* : ∀ *a b*, *kinding a kind_U* ⟶ *kinding b kind_U* ⟶
kinding (*or a b*) *kind_U.*
Trivial.

7.5.6 Well-formedness of environments

If an environment is well-formed, it must be *ok*.
Lemma *env_wf_ok* : ∀ *E k*, *env_wf E k* ⟶ *ok E.*
Trivial.

The empty environment is well-formed.
Lemma *env_wf_empty* : ∀ *k*, *env_wf empty k.*
Trivial.

The singleton environment is well-formed.
Lemma *env_wf_singleton* : ∀ *x t k*, *kinding t k* ⟶
env_wf (*x* ¬ *t*) *k.*
Trivial.

An environment can be extended with (*x* ¬ *t*) if *x* is not already in *E* and *t* has the right kind.
Lemma *env_wf_extend* : ∀ *E k x t*, *x* # *E* ⟶ *kinding t k* ⟶
env_wf E k ⟶ *env_wf* (*E* & *x* ¬ *t*) *k.*
Trivial.

The tail of a well-formed environment is also well-formed.
Lemma *env_wf_tail* : ∀ *E x t k*,
env_wf (*E* & *x* ¬ *t*) *k* ⟶ *env_wf E k.*
Trivial.

Well-formedness of an environment is unaffected if we remove a variable.
Lemma *env_wf_dsub* : ∀ *E k x*,

env_wf E k ⟶ *env_wf (dsub x E) k*.
Follows from *binds_dsub_inv*.

Well-formedness of an environment is unaffected when we add a fresh variable of the right kind.
Lemma *env_wf_dsub_inv* : ∀ *E k x*,
env_wf (dsub x E) k ⟶
(∀ *t*, *binds x t E* ⟶ *kinding t k*) ⟶ *x # dsub x E* ⟶
env_wf E k.
Follows from *ok_dsub_inv* and *binds_dsub*.

Well-formed is unaffected if we replace a type by an equivalent one.
Lemma *env_wf_typ_equiv* : ∀ *E k x t s*, *typ_equiv t s* ⟶
env_wf (E & x ¬ *t) k* ⟶ *env_wf (E & x* ¬ *s) k*.
Follows from *typ_equiv_same_kind*.

Well-formedness of an environment is independent of the order of the assumptions.
Lemma *env_wf_exch* : ∀ *E1 E2 k*,
env_wf (E1 & E2) k ⟶ *env_wf (E2 & E1) k*.
Trivial (uses *binds_exch*).

Generalization of *env_wf_exch*.
Lemma *env_wf_exch_3* : ∀ *E1 E2 E3 k*,
env_wf (E1 & E2 & E3) k ⟶ *env_wf (E1 & E3 & E2) k*.
Trivial (uses *binds_exch_3*).

Every type in a well-formed environment has the same kind.
Lemma *env_wf_binds_kind* : ∀ *E x t k*, *env_wf E k* ⟶
binds x t E ⟶ *kinding t k*.
Trivial.

Every part of a well-formed environment must be well-formed.
Lemma *env_wf_concat_inv* : ∀ *E1 E2 k*, *env_wf (E1 & E2) k* ⟶
env_wf E1 k ∧ *env_wf E2 k*.
Trivial.

7.5.7 Regularity

A typing relation only holds when the environment is well-formed and the term is locally closed.
Lemma *typing_regular* : ∀ *E e T fvars*,
typing E e T fvars ⟶
env_wf E kind_star ∧ *env_wf fvars kind_U* ∧ *term e*.
By induction on *typing E e T fvars*.

The answer predicate only holds for locally closed terms.
Lemma *answer_regular* : ∀ *e*,
answer e ⟶ *term e*.
Trivial induction on *answer e*.

The reduction relation only holds for pairs of locally closed terms.
Lemma *body_app* : ∀ *e e'*, *term e'* ⟶

body e ⟶ *body* (*trm_app e e'*).
Trivial.

The reduction relation only applies to locally closed terms.
Lemma *red_regular* : ∀ *e e'*,
red e e' ⟶ *term e* ∧ *term e'*.
By induction on *red e e'*; uses *open_rec_term_open*.

7.5.8 Well-founded induction on subterms

Subterm relation on locally-closed terms.

```
Inductive subterm : trm ⟶ trm ⟶ Prop :=
  | sub_abs : ∀ x t, subterm (t ^ x) (trm_abs t)
  | sub_abs_trans : ∀ x t t', subterm t' (t ^ x) ⟶ subterm t' (trm_abs t)
  | sub_app1 : ∀ t1 t2, subterm t1 (trm_app t1 t2)
  | sub_app2 : ∀ t1 t2, subterm t2 (trm_app t1 t2)
  | sub_app1_trans : ∀ t' t1 t2, subterm t' t1 ⟶ subterm t' (trm_app t1 t2)
  | sub_app2_trans : ∀ t' t1 t2, subterm t' t2 ⟶ subterm t' (trm_app t1 t2).
```

Size is defined to be the number of constructors used to build up a term.

```
Fixpoint size (t:trm) : nat :=
  match t with
  | trm_fvar x ⇒ 1
  | trm_bvar i ⇒ 1
  | trm_abs t1 ⇒ 1 + size t1
  | trm_app t1 t2 ⇒ 1 + size t1 + size t2
  end.
```

Size is unaffected by substituting free variables for bound variables.
Lemma *size_subst_free* : ∀ *t i x*,
size t = *size* ({*i* ⇝ *trm_fvar x*} *t*).
By induction on *t*.

Special case of *size_subst_free*.
Lemma *size_open* : ∀ *t x*,
size t = *size* (*t* ^ *x*).
Follows directly from *size_subst_free*.

The subterm relation is well-founded[1].
Lemma *subterm_well_founded* : *well_founded subterm*.
We prove the more general property ∀ (*n:nat*) (*t:trm*), *size t* < *n* ⟶ *Acc subterm t* by induction on *n*.

7.5.9 Iterated domain subtraction

Removing a list of variables from the empty environment yields the empty environment.
Lemma *dsub_list_nil* : ∀ *xs*, *dsub_list xs nil* = *nil*.

[1] Proof suggested by Arthur Charguéraud.

Trivial.

Like *dsub_list_nil* but using *dsub_vars* instead of *dsub_list*.
Lemma *dsub_vars_nil* : ∀ *xs*, *dsub_vars xs nil* = *nil*.
Follows directly from *dsub_list_nil*.

Auxiliary lemma used to prove *dsub_list_inv*, below.
Lemma *dsub_list_inv_aux1* : ∀ *xs E v t*, *ok* ((*v*, *t*) :: *E*) ⟶
  In v xs ⟶ *dsub_list xs* ((*v*, *t*) :: *E*) = *dsub_list xs E*.
By induction on *xs*; uses *in_dom_dsub_inv*.

Auxiliary lemma used to prove *dsub_list_inv*, below.
Lemma *dsub_list_inv_aux2* : ∀ *xs E v t*,
  ¬ *In v xs* ⟶ *dsub_list xs* ((*v*, *t*) :: *E*) = (*v*, *t*) :: *dsub_list xs E*.
By induction on *xs*.

The following lemma is useful in proofs involving *dsub_list*. When we apply *dsub_list xs* to an environment with head (*v*, *t*), then either *v* is in the list *xs* and the head of the list will be removed, or *v* is not in the list *xs* and the head of the list will be left alone.
Lemma *dsub_list_inv* : ∀ *xs E v t*, *ok* ((*v*, *t*) :: *E*) ⟶
  (*In v xs* ∧ *dsub_list xs* ((*v*, *t*) :: *E*) = *dsub_list xs E*) ∨
  (˜ *In v xs* ∧ *dsub_list xs* ((*v*, *t*) :: *E*) = (*v*, *t*) :: *dsub_list xs E*).
Follows from *dsub_list_inv_aux_1* and *dsub_list_inv_aux_2*.

Like *dsub_list_inv* but using *dsub_vars* instead of *dsub_list*.
Lemma *dsub_vars_inv* : ∀ *xs E v t*, *ok* ((*v*, *t*) :: *E*) ⟶
  (*v* \in *xs* ∧ *dsub_vars xs* ((*v*, *t*) :: *E*) = *dsub_vars xs E*) ∨
  (*v* *\notin xs* ∧ *dsub_vars xs* ((*v*, *t*) :: *E*) = (*v*, *t*) :: *dsub_vars xs E*).
Follows from *dsub_list_inv*.

The order in which we remove variables from the domain of an environment is irrelevant.
Lemma *dsust_list_permut* : ∀ *E xs ys*, *ok E* ⟶
  (∀ *x*, *In x xs* ⟶ *In x ys*) ⟶
  (∀ *y*, *In y ys* ⟶ *In y xs*) ⟶
  dsub_list xs E = *dsub_list ys E*.
By induction on *E*; uses *dsub_list_inv* twice in the induction step (once for *xs* and once for *ys*).

Like *dsust_list_permut*, but using *dsub_vars* instead of *dsub_list*.
Lemma *dsust_vars_permut* : ∀ *E xs ys*, *ok E* ⟶
  (∀ *x*, *x* \in *xs* ⟶ *x* \in *ys*) ⟶
  (∀ *y*, *y* \in *ys* ⟶ *y* \in *xs*) ⟶
  dsub_vars xs E = *dsub_vars ys E*.
Proof analogous to *dsust_list_permut* but using *dsub_vars_inv* instead.

Special case of *dsub_vars_inv*.
Lemma *dsub_vars_concat_assoc* : ∀ *E xs x t*, *ok* (*E* & *x* ¬ *t*) ⟶
  x *\notin xs* ⟶ *dsub_vars xs* (*E* & *x* ¬ *t*) = (*dsub_vars xs E*) & *x* ¬ *t*.
Follows from *dsub_vars_inv*.

Special case of *dsub_vars_inv*.
Lemma *dsub_vars_cons* : ∀ *E xs x t*, *ok* (*E* & *x* ¬ *t*) ⟶ *x* \in *xs* ⟶

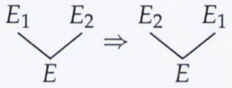
dsub_vars xs (E & x ¬ t) = dsub_vars xs E.
Follows from *dsub_vars_inv*.

To remove ({{*x*}} \u *xs*) from the domain of an environment, we first remove *x* and then *xs*.
Lemma *dsub_vars_to_dsub* : ∀ *E x xs*, *ok E* ⟶
dsub_vars ({{*x*}} \u *xs*) *E* = *dsub_vars xs* (*dsub x E*).
Follows from *dsust_list_permut*.

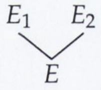
If *x* is in the domain of (*E* with *xs* removed), then *x* must be in the set (domain of *E*) with *xs* removed.
Lemma *in_dom_dsub_vars* : ∀ *E x xs*, *ok E* ⟶
x \in *dom* ((*dsub_vars xs*) *E*) ⟶ *x* \in (*S.diff* (*dom E*) *xs*).
By induction on *E*; uses *dsub_vars_inv* in the induction step.

If *x* is not in the domain of *E* to start with, then it certainly will not be in the domain of *E* after we have removed some variables from the domain of *E*.
Lemma *notin_dom_dsub_vars* : ∀ *E x xs*, *ok E* ⟶
x # *E* ⟶ *x* # (*dsub_vars xs E*).
Trivial.

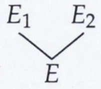
7.5.10 Context split

We can swap the two branches of a context split: 

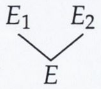
Lemma *split_exch* : ∀ *E E1 E2*,
split_context E as (*E1* ; *E2*) ⟶ *split_context E* as (*E2* ; *E1*).
Trivial induction on *split_context E* as (*E1* ; *E2*).

If 

and *x* is in the domain of E_1, then *x* must be in the domain of *E*.
Lemma *in_dom_split_1* : ∀ *E E1 E2 x*,
split_context E as (*E1* ; *E2*) ⟶ *x* \in *dom E1* ⟶ *x* \in *dom E*.
By induction on *split_context E* as (*E1* ; *E2*).

If 

and *x* is in the domain of E_2, then *x* must be in the domain of *E*.
Lemma *in_dom_split_2* : ∀ *E E1 E2 x*,
split_context E as (*E1* ; *E2*) ⟶ *x* \in *dom E2* ⟶ *x* \in *dom E*.
Follows from *in_dom_split_1* and *split_exch*.

If 

and *x* is in the domain of *E*, then *x* must either be in the domain of E_1 or in the domain of E_2 (or both).
Lemma *in_dom_split_inv* : ∀ *E E1 E2 x*,
split_context E as (*E1* ; *E2*) ⟶ *x* \in *dom E* ⟶ *x* \in *dom E1* ∨ *x* \in *dom E2*.
By induction on *split_context E* as (*E1* ; *E2*).

If $$\begin{tikzcd}[column sep=tiny, row sep=small] E_1 \arrow[dr, dash] & & E_2 \arrow[dl, dash] \\ & E & \end{tikzcd}$$ and E_1 binds x, then E must bind x. Note that unlike *in_dom_split_1*, we require E to be *ok*.

Lemma *binds_split_1* : ∀ *E E1 E2 x t*, *ok E* ⟶
  split_context E as (*E1* ; *E2*) ⟶ *binds x t E1* ⟶ *binds x t E*.

By induction on *split_context E* as (*E1* ; *E2*).

If $$\begin{tikzcd}[column sep=tiny, row sep=small] E_1 \arrow[dr, dash] & & E_2 \arrow[dl, dash] \\ & E & \end{tikzcd}$$ and E_2 binds x, then E must bind x. Note that unlike *in_dom_split_1*, we require E to be *ok*.

Lemma *binds_split_2* : ∀ *E E1 E2 x t*, *ok E* ⟶
  split_context E as (*E1* ; *E2*) ⟶ *binds x t E2* ⟶ *binds x t E*.

Follows from *binds_split_1* and *split_exch*.

If $$\begin{tikzcd}[column sep=tiny, row sep=small] E_1 \arrow[dr, dash] & & E_2 \arrow[dl, dash] \\ & E & \end{tikzcd}$$ and E binds x, then either E_1 or E_2 (or both) must bind x. Note that unlike *in_dom_split_inv*, we require E to be *ok*.

Lemma *binds_split_inv* : ∀ *E E1 E2 x t*,
  split_context E as (*E1* ; *E2*) ⟶ *binds x t E* ⟶ *binds x t E1* ∨ *binds x t E2*.

By induction on *split_context E* as (*E1* ; *E2*).

If $$\begin{tikzcd}[column sep=tiny, row sep=small] E_1 \arrow[dr, dash] & & E_2 \arrow[dl, dash] \\ & E & \end{tikzcd}$$ then $$\begin{tikzcd}[column sep=tiny, row sep=small] \rhd_x E_1 \arrow[dr, dash] & & \rhd_x E_2 \arrow[dl, dash] \\ & \rhd_x E & \end{tikzcd}$$.

Lemma *split_dsub* : ∀ *E E1 E2 x*,
  split_context E as (*E1* ; *E2*) ⟶ *ok E* ⟶
  split_context (*dsub x E*) as (*dsub x E1* ; *dsub x E2*).

By induction on split_context E as (E1 ; E2)].

We can always split an environment E as $$\begin{tikzcd}[column sep=tiny, row sep=small] E \arrow[dr, dash] & & \emptyset \arrow[dl, dash] \\ & E & \end{tikzcd}$$.

Lemma *split_empty* : ∀ *E*,
  split_context E as (*E* ; *empty*).

Trivial induction on *E*.

If $$\begin{tikzcd}[column sep=tiny, row sep=small] E' \arrow[dr, dash] & & \emptyset \arrow[dl, dash] \\ & E & \end{tikzcd}$$ then E must be *E'*.

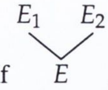
Lemma *split_empty_inv* : ∀ *E E'*,
  split_context E as (*E'* ; *empty*) ⟶ *E* = *E'*.

We prove ∀ *E E' E"*, *split_context E* as (*E'* ; *E"*) ⟶ *E"* = *empty* ⟶ *E* = *E'* by induction on *split_context E* as (*E'* ; *E"*).

We can always split $E, x : t$ as $$\begin{tikzcd}[column sep=tiny, row sep=small] E \arrow[dr, dash] & & x : t \arrow[dl, dash] \\ & E, x : t & \end{tikzcd}$$.

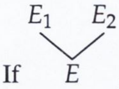
Lemma *split_tail* : ∀ *E x t*,
  split_context (*E* & *x* ¬ *t*) as (*E* ; *x* ¬ *t*).

Follows from *split_empty*.



If , *E* binds *x* and *x* is in the domain of E_1, then E_1 must bind *x*.

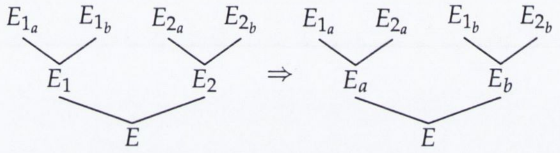
Lemma *split_binds_in_dom_1* : ∀ *E E1 E2*,
  split_context E as (*E1* ; *E2*) → *ok E* →
  ∀ *x t*, *binds x t E* → *x* \in *dom E1* → *binds x t E1*.
By induction on *split_context E* as (*E1* ; *E2*).



If , *E* binds *x* and *x* is in the domain of E_2, then E_2 must bind *x*.

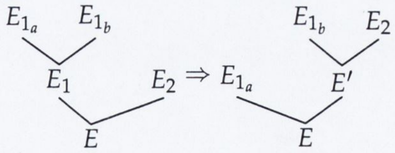
Lemma *split_binds_in_dom_2* : ∀ *E E1 E2*,
  split_context E as (*E1* ; *E2*) → *ok E* →
  ∀ *x t*, *binds x t E* → *x* \in *dom E2* → *binds x t E2*.
Follows from *split_binds_in_dom_1* and *split_exch*.

We prove a series of four reordering lemmas, with the first the most general and the basis for the other three.



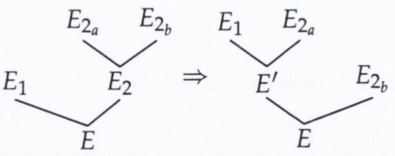
Lemma *reorder_ab'cd_ac'bd* : ∀ *E E1 E2*,
  split_context E as (*E1* ; *E2*) → ∀ *E1a E1b E2a E2b*,
  split_context E1 as (*E1a* ; *E1b*) →
  split_context E2 as (*E2a* ; *E2b*) →
  ∃ *Ea*, ∃ *Eb*,
    split_context E as (*Ea* ; *Eb*) ∧
    split_context Ea as (*E1a* ; *E2a*) ∧
    split_context Eb as (*E1b* ; *E2b*).
By induction on *split_context E* as (*E1* ; *E2*) followed by inversion on *split_context E1* as (*E1a* ; *E1b*) and *split_context E2* as (*E2a* ; *E2b*). There are 34 cases to consider but they are all trivial.

Restructure a three-way split (E_a, E_b, E_c).



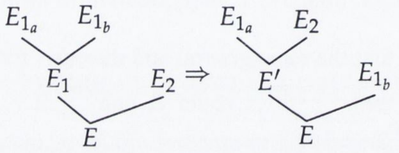
Corollary reorder_ab'c_a'bc : ∀ *E E1 E2 E1a E1b*,
  split_context E as (*E1* ; *E2*) →
  split_context E1 as (*E1a* ; *E1b*) →
  ∃ *E'*,
    split_context E as (*E1a* ; *E'*) ∧
    split_context E' as (*E1b* ; *E2*).
Follows from *reorder_ab'cd_ac'bd*.

Inverse of *reorder_ab'c_a'bc*:

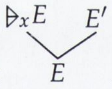


Corollary reorder_a'bc_ab'c : ∀ *E E1 E2 E2a E2b*,
split_context E as (*E1* ; *E2*) ⟶
split_context E2 as (*E2a* ; *E2b*) ⟶
∃ *E'*,
split_context E as (*E'* ; *E2b*) ∧
split_context E' as (*E1* ; *E2a*).
Follows from *reorder_ab'cd_ac'bd*.

The final reordering lemma is its own inverse:

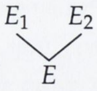


Corollary reorder_ab'c_ac'b : ∀ *E E1 E2 E1a E1b*,
split_context E as (*E1* ; *E2*) ⟶
split_context E1 as (*E1a* ; *E1b*) ⟶
∃ *E'*,
split_context E as (*E'* ; *E1b*) ∧
split_context E' as (*E1a* ; *E2*).
Follows from *reorder_ab'cd_ac'bd*.

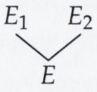
Remove the assumption about *x* from *E*:  . We use this lemma in *split_dom_inv*, below.

Lemma *split_dom* : ∀ *E x*,
∃ *E'*, *split_context E* as (*dsub x E* ; *E'*) ∧ *dsub x E'* = *empty*.
By induction on *E*.

Inverse property of *split_dom*.
Lemma *split_dom_inv* : ∀ *E E' x*,
E' = *dsub x E* ⟶
∃ *E_x*, *split_context E* as (*E'* ; *E_x*) ∧ *dsub x E_x* = *empty*.
By induction on *E*.

If  and *E* is *ok*, then both *E1* and *E2* must be *ok*.

Lemma *split_context_ok* : ∀ *E E1 E2*,
split_context E as (*E1* ; *E2*) ⟶ *ok E* ⟶ *ok E1* ∧ *ok E2*.
By induction on *split_context E* as (*E1* ; *E2*).

If  and *E* is well-formed, then both *E1* and *E2* must be well-formed.

Lemma *split_context_wf* : ∀ *E E1 E2 k*,
split_context E as (*E1* ; *E2*) ⟶ *env_wf E k* ⟶ *env_wf E1 k* ∧ *env_wf E2 k*.

Follows from *split_context_ok*, *binds_split_1* and *binds_split_2*.

We can always split the concatenation of two environments into its two constituents:

$$\begin{tikzcd} E_1 \arrow[dr, no head] & & E_2 \arrow[dl, no head] \\ & E_1\&E_2 & \end{tikzcd}$$

.

Lemma *split_concat* : ∀ *E2 E1*,

split_context (*E1* & *E2*) as (*E1* ; *E2*).

By induction on *E2*.

Split a domain *E* into two domains *E1* and *E2* so that all assumptions about variables in *xs* go into *E1* and the rest goes into *E2*:

$$\begin{tikzcd} \rhd_{\mathrm{dom}\ E\setminus xs} E \arrow[dr, no head] & & \rhd_{xs} E \arrow[dl, no head] \\ & E & \end{tikzcd}$$

.

Lemma *split_dom_set* : ∀ *E xs*, *ok E* ⟶

split_context E as (*dsub_vars* (*S.diff* (*dom E*) *xs*) *E*; *dsub_vars xs E*).

By induction on *E*. The proof is slightly tricky, and relies on *dsust_vars_permut*, *dsub_vars_cons*, *dsub_vars_concat_assoc* and *dsub_vars_to_dsub*.

7.5.11 Type equivalence

If *t1 t2* is equivalent to *s*, then *s* must be of the form *s1 s2* where *t1* and *s1*, and *t2* and *s2*, are equivalent. This however only holds for types of kind other than *U* (counterexample: *typ_equiv* (*or a* (*not a*)) *a*).

Lemma *typ_equiv_app_inv_ex* : ∀ *t s*, ¬ *kinding t kind_U* ⟶

typ_equiv t s ⟶

(∀ *t1 t2* , *t* = *typ_app t1 t2* ⟶ ∃ *s1*, ∃ *s2*,

s = *typ_app s1 s2* ∧ *typ_equiv t1 s1* ∧ *typ_equiv t2 s2*) ∧

(∀ *s1 s2* , *s* = *typ_app s1 s2* ⟶ ∃ *t1*, ∃ *t2*,

t = *typ_app t1 t2* ∧ *typ_equiv t1 s1* ∧ *typ_equiv t2 s2*).

By induction on *typ_equiv t s*. This is a slightly tricky proof, and we do need to prove it in both directions (as stated in the lemma). If we try to prove it in one direction only, we get stuck in the case for *typ_equiv_sym*.

If *t1 s1* is equivalent to *t2 s2*, then the components must be equivalent.

Lemma *typ_equiv_app_inv* : ∀ *t1 t2 s1 s2*,

¬ *kinding* (*typ_app t1 s1*) *kind_U* ⟶

typ_equiv (*typ_app t1 s1*) (*typ_app t2 s2*) ⟶

typ_equiv t1 t2 ∧ *typ_equiv s1 s2*.

Follows from *typ_equiv_app_inv*.

If *t* is equivalent to *ATTR*, it must be *ATTR*.

Lemma *typ_equiv_ATTR_inv* : ∀ *t s*, *typ_equiv t s* ⟶

(*t* = *ATTR* ⟶ *s* = *ATTR*) ∧ (*s* = *ATTR* ⟶ *t* = *ATTR*).

By induction on *typ_equiv t s*.

Special case of *typ_equiv_app_inv_ex* for attributed types.

Lemma *typ_equiv_attr_inv_ex* : ∀ *t u s*,

typ_equiv s (*t* ' *u*) ⟶ ∃ *t'*, ∃ *u'*,

s = *t'* ' *u'* ∧ *typ_equiv t t'* ∧ *typ_equiv u u'*.

Follows from *typ_equiv_app_inv_ex*.

Special case of *typ_equiv_app_inv* for attributed types.
Lemma *typ_equiv_attr_inv* : ∀ *t u s v*,
  typ_equiv (*t* ' *u*) (*s* ' *v*) → *typ_equiv t s* ∧ *typ_equiv u v*.
Follows from *typ_equiv_app_inv*.

Special case of *typ_equiv_app_inv* for function types.
Lemma *typ_equiv_fun_inv* : ∀ *a u b a' u' b'*,
  typ_equiv (*a* ⟨ *u* ⟩ *b*) (*a'* ⟨ *u'* ⟩ *b'*) →
  typ_equiv a a' ∧
  typ_equiv u u' ∧
  typ_equiv b b'.
Follows from *typ_equiv_attr_inv*.

Replace an attribute on an attributed type.
Lemma *typ_equiv_new_attr* : ∀ *t u v*, *typ_equiv u v* →
  typ_equiv (*t* ' *u*) (*t* ' *v*).
Trivial.

Replace the domain of an arrow
Lemma *typ_equiv_fun_new_dom* : ∀ *a u b a'*, *typ_equiv a a'* →
  typ_equiv (*a* ⟨ *u* ⟩ *b*) (*a'* ⟨ *u* ⟩ *b*).
Trivial.

Replace the codomain of an arrow
Lemma *typ_equiv_fun_new_cod* : ∀ *a u b b'*, *typ_equiv b b'* →
  typ_equiv (*a* ⟨ *u* ⟩ *b*) (*a* ⟨ *u* ⟩ *b'*).
Trivial.

If *t* and *s* are equivalent and have kind *U*, then they must also be equivalent by the boolean equivalence relation.
Lemma *typ_equiv_BA_equiv* : ∀ *t s*,
  typ_equiv t s → *kinding t kind_U* → *BA.equiv t s*.
By induction on *typ_equiv t s*.

Commutativity of *or*.
Lemma *typ_equiv_comm_or* : ∀ *a b*, *kinding* (*or a b*) *kind_U* →
  typ_equiv (*or a b*) (*or b a*).
Trivial.

7.5.12 Non-unique types

If *t* and *s* are equivalent and *t* is *non_unique*, *s* must be *non_unique*.
Lemma *non_unique_equiv* : ∀ *t s*, *typ_equiv t s* →
  non_unique t → *non_unique s*.
By inversion on *non_unique t*.

If t^u is non-unique, then *u* must be equivalent to false.
Lemma *non_unique_star* : ∀ *t u*,

non_unique (*t* ' *u*) ⟶ *typ_equiv u NU*.

By inversion on *non_unique* (*t* ' *u*). There are two possibilities (see the definition of *non_unique*). For the first case, (*t* ' *u*) of kind ∗, the lemma follows immediately. For the second, we show that *kinding* (*t* ' *u*) *kind_U* leads to contradiction.

If *u* is *non_unique* and has kind *U*, it must be equivalent to false.

Lemma *non_unique_U* : ∀ *u*,

non_unique u ⟶ *kinding u kind_U* ⟶ *typ_equiv u NU*.

By inversion on *non_unique u*. Proof analogous to *non_unique_star*.

If $\begin{array}{c} E_1 \quad E_2 \\ \diagdown\!\diagup \\ E \end{array}$, *E* binds *x*, and both *E1* and *E2* bind *x*, then *x* must have a non-unique type. That is, only variables of non-unique type can be duplicated.

Lemma *split_both_inv* : ∀ *E E1 E2 x t*, *ok E* ⟶

split_context E as (*E1* ; *E2*) ⟶

binds x t E ⟶ *x* \in *dom E1* ⟶ *x* \in *dom E2* ⟶

non_unique t.

By induction on *split_context E* as (*E1* ; *E2*).

If every type in *E* is non-unique, then $\begin{array}{c} E \quad E \\ \diagdown\!\diagup \\ E \end{array}$.

Lemma *split_non_unique* : ∀ *E*, *ok E* ⟶

(∀ *x t*, *binds x t E* ⟶ *non_unique t*) ⟶

split_context E as (*E* ; *E*).

By induction on *E*.

7.5.13 Equivalence of environments.

We start with a number of trivial consequences of ≅. These lemmas enable us to work directly with the notion of an equivalence, rather than having to unfold the definition of ≅ every time we need one of its constituents.

Equivalence only holds between well-formed environments.

Lemma *env_equiv_regular* : ∀ *E1 E2 k*,

(*E1* ≅ *E2*) *k* ⟶ *env_wf E1 k* ∧ *env_wf E2 k*.

Trivial.

If *E1* ≅ *E2* and *E1* binds *x*, then *E2* must bind *x*.

Lemma *env_equiv_binds_1* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* ⟶

∀ *x t*, *binds x t E1* ⟶ ∃ *t'*, *binds x t' E2* ∧ *typ_equiv t t'*.

Trivial.

If *E1* ≅ *E2* and *E2* binds *x*, then *E1* must bind *x*.

Lemma *env_equiv_binds_2* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* ⟶

∀ *x t*, *binds x t E2* ⟶ ∃ *t'*, *binds x t' E1* ∧ *typ_equiv t t'*.

Trivial.

If *E1* ≅ *E2* and *x* is in the domain of *E1*, *x* must be in the domain of *E2*.

Lemma *env_equiv_in_dom_1* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* ⟶

∀ *x*, *x* \in *dom E1* → *x* \in *dom E2*.
Follows directly from *binds_in_dom* and *env_equiv_binds_1*.

If *E1* ≅ *E2* and *x* is in the domain of *E2*, *x* must be in the domain of *E1*.
Lemma *env_equiv_in_dom_2* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* →
∀ *x*, *x* \in *dom E2* → *x* \in *dom E1*.
Follows directly from *binds_in_dom* and *env_equiv_binds_2*.

The equivalence relation is reflexive.
Lemma *env_equiv_refl* : ∀ *E k*, *env_wf E k* → (*E* ≅ *E*) *k*.
Trivial.

The equivalence relation is commutative.
Lemma *env_equiv_comm* : ∀ *E1 E2 k*, (*E1* ≅ *E2*) *k* → (*E2* ≅ *E1*) *k*.
Trivial.

The equivalence relation is transitive.
Lemma *env_equiv_trans* : ∀ *E1 E2 E3 k*,
(*E1* ≅ *E2*) *k* → (*E2* ≅ *E3*) *k* → (*E1* ≅ *E3*) *k*.
Trivial.

If *E* is equivalent to the empty environment, it must be the empty environment.
Lemma *env_equiv_empty* : ∀ *E k*, (*E* ≅ *empty*) *k* → *E* = *empty*.
By case analysis on *E*.

If *E* is equivalent to a singleton environment, it must be that singleton environment.
Lemma *env_equiv_singleton* : ∀ *E k y s*, (*E* ≅ (*y* ¬ *s*)) *k* →
∃ *s'*, *E* = *y* ¬ *s'* ∧ *typ_equiv s s'*.
By case analysis on *E*; distinguishing between the empty environment, the singleton environment, and the environment with more than one element. We show contradiction for all cases except the singleton case.

Equivalence between environments is unaffected if we remove a variable from both sides.
Lemma *env_equiv_dsub* : ∀ *E1 E2 k x*,
(*E1* ≅ *E2*) *k* → (*dsub x E1* ≅ *dsub x E2*) *k*.
Follows from *binds_in_dom*, *binds_dsub* and *binds_dsub_inv*.

Equivalence between environments is unaffected if we add a variable on both sides, provided that the variable was not already in the domain of the environments to start with and has the right kind.
Lemma *env_equiv_extend* : ∀ *E E' k x t s*, *x* # *E* →
kinding t k → *typ_equiv t s* →
(*E* ≅ *E'*) *k* → (*E* & *x* ¬ *t* ≅ *E'* & *x* ¬ *s*) *k*.
Trivial.

Special case of *env_equiv_extend*.
Lemma *env_equiv_typ_equiv* : ∀ *E k x t s*, *env_wf* (*E* & *x* ¬ *t*) *k* →
typ_equiv t s →
(*E* & *x* ¬ *t* ≅ *E* & *x* ¬ *s*) *k*.
Follows from *env_equiv_extend* and *env_wf_binds_kind*.

Special case of *env_equiv_dsub*.

Lemma *env_equiv_cons* : ∀ *E E' k x t*,
  (*E* & *x* ¬ *t* ≅ *E'*) *k* → (*E* ≅ *dsub x E'*) *k*.
Follows from *env_equiv_dsub* and *dsub_not_in_dom*.

Inverse property of *env_equiv_cons*.
Lemma *env_equiv_cons_inv* : ∀ *E E' k x t*,
  (*dsub x E* ≅ *E'*) *k* →
  binds x t E → *env_wf E k* →
  (*E* ≅ *E'* & *x* ¬ *t*) *k*.
Follows from *binds_dsub* and *binds_dsub_inv*.

We can take an environment *E*, remove its assumption about *x*, and then re-insert that assumption at the start of the environment; the result will be equivalent to the original environment.
Lemma *env_equiv_reorder* : ∀ *E k x t*,
  env_wf E k → *binds x t E* → (*E* ≅ *dsub x E* & *x* ¬ *t*) *k*.
Follows from *binds_dsub* and *binds_dsub_inv*.

If $\begin{tikzcd}[column sep=tiny, row sep=small] E_1 \arrow[dr, dash] & & E_2 \arrow[dl, dash] \\ & E & \end{tikzcd}$ and *E'* is equivalent to *E*, then there exist two environments E_1' and E_2' such that $\begin{tikzcd}[column sep=tiny, row sep=small] E_1' \arrow[dr, dash] & & E_2' \arrow[dl, dash] \\ & E' & \end{tikzcd}$ and E_1 and E_2 are equivalent to E_1' and E_2'.
Lemma *env_equiv_split* : ∀ *E' E E1 E2 k*,
  split_context E as (*E1* ; *E2*) → (*E* ≅ *E'*) *k* →
    ∃ *E1'*, ∃ *E2'*,
      split_context E' as (*E1'* ; *E2'*) ∧ (*E1* ≅ *E1'*) *k* ∧ (*E2* ≅ *E2'*) *k*.
By induction on E'. For the case (*v*, *t*) :: *E'*, we recurse on *dsub v E*, then add (*v*, *t*) to the partially constructed *E1'* or *E2'* depending on whether *v* \in *dom E1* or *v* \in *dom E2*.

Equivalence is unaffected by order.
Lemma *env_equiv_exch* : ∀ *E1 E2 k*, *env_wf* (*E1* & *E2*) *k* →
  (*E1* & *E2* ≅ *E2* & *E1*) *k*.
Follows trivially from *env_wf_exch* and *binds_exch*.

Generalization of *env_equiv_exch*.
Lemma *env_equiv_exch_3* : ∀ *E1 E2 E3 k*, *env_wf* (*E1* & *E2* & *E3*) *k* →
  (*E1* & *E2* & *E3* ≅ *E1* & *E3* & *E2*) *k*.
Follows trivially from *env_wf_exch_3* and *binds_exch_3*.

7.5.14 Range

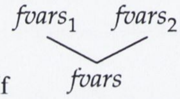
The range of an environment containing only types of kind *U* is *U*.
Lemma *rng_kind_U* : ∀ *E*, *env_wf E kind_U* → *kinding* (*rng E*) *kind_U*.
By induction on *E*.

Auxiliary lemma used to prove *rng_non_unique*.
Lemma *rng_non_unique_BA* : ∀ *fvars*,
  BA.equiv (*rng fvars*) *NU* →
  (∀ *x u*, *binds x u fvars* → *BA.equiv u NU*).

By induction on *fvars*, using lemma *or_false_both* from the Boolean Algebra formalization.

If the range of an environment is equivalent to false, then every attribute in that environment must be equivalent to false.
Lemma *rng_non_unique* : ∀ *fvars*, *env_wf fvars kind_U* ⟶
  typ_equiv (*rng fvars*) *NU* ⟶
  (∀ *x u*, *binds x u fvars* ⟶ *typ_equiv u NU*).
Follows from *rng_non_unique_BA*, *env_wf_binds_kind* and *typ_equiv_BA_equiv*.

Auxiliary lemma used to prove *split_rng*.
Lemma *split_rng_BA* : ∀ *fvars fvars1 fvars2*,
  split_context fvars as (*fvars1* ; *fvars2*) ⟶
  BA.equiv (*rng fvars*) (*or* (*rng fvars1*) (*rng fvars2*)).
By induction on *split_context fvars* as (*fvars1* ; *fvars2*), using properties of the boolean equivalence relation and *rng_concat*.

If  then the range of *fvars* is equivalent to the range of the concatenation of *fvars1* and *fvars2*. This holds because if there is an assumption about *x* in both *fvars1* and *fvars2*, then that must be the same assumption, and we know that *t* is equivalent to *or t t* for any *t*.

Lemma *split_rng* : ∀ *fvars fvars1 fvars2*, *env_wf fvars kind_U* ⟶
  split_context fvars as (*fvars1* ; *fvars2*) ⟶
  typ_equiv (*rng fvars*) (*or* (*rng fvars1*) (*rng fvars2*)).
Follows from *split_rng_BA*.

Auxiliary lemma needed to prove *env_equiv_rng*.
Lemma *rng_reorder* : ∀ (*E* : *env*) *x t*, *binds x t E* ⟶
  BA.equiv (*rng E*) (*or* (*rng* (*dsub x E*)) *t*).
By induction on *E*.

If two environments are equivalent, then their ranges must be equivalent.
Lemma *env_equiv_rng* : ∀ *E E'*,
  (*E* ≅ *E'*) *kind_U* ⟶ *typ_equiv* (*rng E*) (*rng E'*).
By induction on *E*, using properties of the boolean equivalence relation, *rng_reorder*, *rng_concat* and *typ_equiv_BA_equiv*.

7.6 Properties of the typing relation

7.6.1 Kinding properties

Every assumption in *E* must have kind $*$.
Lemma *kinding_env* : ∀ *E e t fvars*,
  E ⊢ *e* : *t* | *fvars* ⟶ ∀ *x s*,
  binds x s E ⟶ *kinding s kind_star*.
Follows trivially from regularity and *env_wf_binds_kind*.

Every assumption in *fvars* must have kind $\mathcal{U}$.

Lemma *kinding_fvars* : $\forall\ E\ e\ t\ \mathit{fvars}$,

$E \vdash e : t \mid \mathit{fvars} \rightarrow \forall\ x\ u$,

$\mathit{binds}\ x\ u\ \mathit{fvars} \rightarrow \mathit{kinding}\ u\ \mathit{kind\_U}$.

Follows trivially from regularity and *env_wf_binds_kind*.

If e has type t, then t must have kind $*$.

Lemma *typing_kind_star* : $\forall\ E\ e\ t\ \mathit{fvars}$,

$E \vdash e : t \mid \mathit{fvars} \rightarrow \mathit{kinding}\ t\ \mathit{kind\_star}$.

By induction on $E \vdash e : t \mid \mathit{fvars}$.

7.6.2 Free variables

If $E \vdash e : T \mid \mathit{fvars}$, then if x is free in e it must be in the domain of E and in the domain of *fvars*.

Lemma *typing_fv* : $\forall\ E\ e\ T\ \mathit{fvars}$,

$E \vdash e : T \mid \mathit{fvars} \rightarrow \forall\ x, x \text{ \\in } \mathit{fv}\ e \rightarrow x \text{ \\in } \mathit{dom}\ E \wedge x \text{ \\in } \mathit{dom}\ \mathit{fvars}$.

By induction on $E \vdash e : T \mid \mathit{fvars}$.

If there is an evaluation context E such that $t = E[x]$, then x must be free in t.

Lemma *eval_fv* : $\forall\ t\ x$,

$\mathit{evals}\ t\ x \rightarrow x \text{ \\in } \mathit{fv}\ t$.

By induction on *evals t x*.

7.6.3 Consistency of *E* and *fvars*

Every assumption in *fvars* must have a corresponding assumption in E.

Lemma *fvars_and_env_consistent* : $\forall\ E\ e\ S\ \mathit{fvars}\ x\ u$,

$E \vdash e : S \mid \mathit{fvars} \rightarrow \mathit{binds}\ x\ u\ \mathit{fvars} \rightarrow$

$\exists\ t, \exists\ v, \mathit{binds}\ x\ (t\ '\ v)\ E \wedge \mathit{typ\_equiv}\ u\ v$.

By induction on $E \vdash e : S \mid \mathit{fvars}$.

Every assumption in E must have a corresponding assumption in *fvars*.

Lemma *env_and_fvars_consistent* : $\forall\ E\ e\ S\ \mathit{fvars}\ x\ t\ u$,

$E \vdash e : S \mid \mathit{fvars} \rightarrow \mathit{binds}\ x\ (t\ '\ u)\ E \rightarrow x \text{ \\in } \mathit{fv}\ e \rightarrow$

$\exists\ v, \mathit{binds}\ x\ v\ \mathit{fvars} \wedge \mathit{typ\_equiv}\ u\ v$.

By induction on $E \vdash e :^{\sim} S \mid \mathit{fvars}$.

7.6.4 Weakening

Auxiliary lemma used to prove *unused_assumptions*.

Lemma *unused_assumption_env* : $\forall\ E\ e\ T\ \mathit{fvars}\ x$,

$E \vdash e : T \mid \mathit{fvars} \rightarrow x \text{ \\notin } \mathit{fv}\ e \rightarrow \mathit{dsub}\ x\ E \vdash e : T \mid \mathit{fvars}$.

By induction on $E \vdash e : T \mid \mathit{fvars}$.

Auxiliary lemma used to prove *unused_assumptions*.

Lemma *unused_assumptions_list* : $\forall\ \mathit{xs}\ E\ e\ T\ \mathit{fvars}$,

$E \vdash e : T \mid \mathit{fvars} \rightarrow (\forall\ x, \mathit{In}\ x\ \mathit{xs} \rightarrow x \text{ \\notin } \mathit{fv}\ e) \rightarrow$

$\mathit{dsub\_list}\ \mathit{xs}\ E \vdash e : T \mid \mathit{fvars}$.

By induction on *xs*, using *unused_assumption_env*.

We can remove all assumptions in *E* about variables that are not free in *e*.

Lemma *unused_assumptions* : ∀ *xs E e T fvars*,
  E ⊢ *e* : *T* | *fvars* → (∀ *x*, *x* \in *xs* → *x* \notin *fv e*) →
  dsub_vars xs E ⊢ *e* : *T* | *fvars*.
Follows trivially from *unused_assumptions_list*.

We can append unused assumptions to the typing environment.

Lemma *weakening_1* : ∀ *E1 e T fvars*,
  E1 ⊢ *e* : *T* | *fvars* → ∀ *E E2*, *env_wf E kind_star* →
  split_context E as (*E1* ; *E2*) → *E* ⊢ *e* : *T* | *fvars*.
By induction on *E1* ⊢ *e* : *T* | *fvars*.

We can prepend unused assumptions to the typing environment.

Lemma *weakening_2* : ∀ *E2 e T fvars*,
  E2 ⊢ *e* : *T* | *fvars* → ∀ *E E1*, *env_wf E kind_star* →
  split_context E as (*E1* ; *E2*) → *E* ⊢ *e* : *T* | *fvars*.
Follows trivially from *weakening_1* and *split_exch*.

Every assumption in *fvars* must be used.

Lemma *no_fvars_weakening* : ∀ *E e T fvars*,
  E ⊢ *e* : *T* | *fvars* → ∀ *x*, *x* \notin *fv e* → *x* # *fvars*.
By induction on *E* ⊢ *e* : *T* | *fvars*.

Since every assumption in *fvars* must be used, if *x* is not free in *e* then removing *x* from *fvars* has no effect (since it wasn't in *fvars* to start with).

Lemma *unused_assumption_fvars* : ∀ *E e T fvars x*,
  E ⊢ *e* : *T* | *fvars* → *x* \notin *fv e* → *E* ⊢ *e* : *T* | *dsub x fvars*.
Follows trivially from *no_fvars_weakening* and *dsub_not_in_dom*.

Combination of *unused_assumption_env* and *unused_assumption_fvars*.

Lemma *unused_assumption* : ∀ *E e T fvars x*,
  E ⊢ *e* : *T* | *fvars* → *x* \notin *fv e* → *dsub x E* ⊢ *e* : *T* | *dsub x fvars*.
Follows directly from *unused_assumption_fvars* and *unused_assumption_env*.

If *e* can be typed in environment *E*, we can split *E* into two environments *E1* and *E2* such that every assumption about variables in *e* will be in *E1*; then *e* can also be typed in environment *E1*.

Lemma *split_env* : ∀ *E e t u fvars*,
  E ⊢ *e* : *t* ' *u* | *fvars* →
  (∃ *E1*, ∃ *E2*,
    split_context E as (*E1* ; *E2*) ∧
    E1 ⊢ *e* : *t* ' *u* | *fvars* ∧
    (∀ *x*, *x* \in *dom E1* → *x* \in *fv e*)).
Follows from *split_dom_set*.

7.6.5 Exchange

We can replace both *E* and *fvars* by equivalent environments. This is a powerful lemma, because the definition of equivalence for environment is very general (in particular, it allows to replace a type by an equivalent type).

Lemma *env_equiv_typing* : ∀ *E e T fvars*,
  E ⊢ *e* : *T* | *fvars* ⟶ ∀ *E' fvars'*,
  (*E* ≅ *E'*) *kind_star* ⟶ (*fvars* ≅ *fvars'*) *kind_U* ⟶
  E' ⊢ *e* : *T* | *fvars'*.

By induction on *E* ⊢ *e* : *T* | *fvars*. This proof is slightly tricky. The case of variables relies on *env_equiv_singleton*. In the case for abstraction, we need *env_equiv_rng*, *env_equiv_extend* and *env_equiv_cons_inv*, and in the case for application we need *env_equiv_split*.

Change the order of the assumptions in the environment.
Lemma *exchange* : ∀ *E1 E2 E3 e T fvars*,
  E1 & *E2* & *E3* ⊢ *e* : *T* | *fvars* ⟶
  E1 & *E3* & *E2* ⊢ *e* : *T* | *fvars*.
Follows trivially from *env_equiv_typing* and *env_equiv_exch_3*.

Replace an assumption in the environment by an equivalent one.
Lemma *typ_equiv_env* : ∀ *E x s s' e t fvars*,
  E & *x* ¬ *s* ⊢ *e* : *t* | *fvars* ⟶ *typ_equiv s s'* ⟶
  E & *x* ¬ *s'* ⊢ *e* : *t* | *fvars*.
Follows trivially from *env_equiv_typing* and *env_equiv_typ_equiv*.

7.6.6 Inversion lemmas

Inversion lemma for variables.
Lemma *typing_var_inv* : ∀ *E x s fvars*,
  E ⊢ *trm_fvar x* : *s* | *fvars* ⟶
    ∃ *t*, ∃ *u*, ∃ *v*,
      typ_equiv s (*t* ' *u*) ∧
      fvars = *x* ¬ *v* ∧
      env_wf E kind_star ∧
      binds x (*t* ' *u*) *E* ∧
      typ_equiv u v.

We prove the more general lemma ∀ *E e s fvars*, *E* ⊢ *e* : *s* | *fvars* ⟶ ∀ *x*, *e* = *trm_fvar x* ⟶ ∃ *t*, ∃ *u*, ∃ *v*, *typ_equiv s* (*t* ' *u*) ∧ *fvars* = *x* ¬ *v* ∧ *env_wf E kind_star* ∧ *binds x* (*t* ' *u*) *E* ∧ *typ_equiv u v*) by induction on *E* ⊢ *e* : *s* | *fvars*. The case for variables is trivial, the cases for application and abstraction can be dismissed, and the case for *typing_equiv* is a straightforward application of the induction hypothesis.

Inversion lemma for application.
Lemma *typing_app_inv* : ∀ *E e1 e2 s fvars*,
  E ⊢ *trm_app e1 e2* : *s* | *fvars* ⟶
    ∃ *E1*, ∃ *E2*, ∃ *fvars1*, ∃ *fvars2*,
    ∃ *a*, ∃ *b*, ∃ *u*,
      typ_equiv s b ∧
      E1 ⊢ *e1* : *a* ⟨ *u* ⟩ *b* | *fvars1* ∧
      E2 ⊢ *e2* : *a* | *fvars2* ∧
      split_context E as (*E1* ; *E2*) ∧ *env_wf E kind_star* ∧
      split_context fvars as (*fvars1* ; *fvars2*) ∧ *env_wf fvars kind_U*.

Analogous to the proof of the inversion lemma for variables.

Inversion lemma for abstraction.

```
Lemma typing_abs_inv : ∀ E e s fvars',
  E ⊢ trm_abs e : s | fvars' →
    ∃ L, ∃ a, ∃ b,
      typ_equiv s (a ⟨ rng fvars' ⟩ b) ∧
      (∀ x fvars, x \notin L → fvars' = dsub x fvars →
        (E & x ¬ a) ⊢ e ^ x : b | fvars).
```

Analogous to the proof of the inversion lemma for variables.

Tactic *typing_inversion* can be used instead of a call to the standard Coq tactic *inversion* to do inversion on the typing relation using the inversion lemmas we just proved.

```
Ltac typing_inversion H :=
  match type of H with
  | ?E ⊢ trm_fvar ?x : ?T | ?fvars ⇒
      let t := fresh "t" in
      let u := fresh "u" in
      let v := fresh "v" in
      elim3 (typing_var_inv H) t u v (?, (?, (?, (?, ?))))
  | ?E ⊢ trm_app ?e1 ?e2 : ?T | ?fvars ⇒
      let E1 := fresh "E1" in
      let E2 := fresh "E2" in
      let fvars1 := fresh "fvars1" in
      let fvars2 := fresh "fvars2" in
      let a := fresh "a" in
      let b := fresh "b" in
      let u := fresh "u" in
      elim7 (typing_app_inv H) E1 E2 fvars1 fvars2 a b u (?, (?, (?, (?, (?, (?, ?))))))
  | ?E ⊢ trm_abs ?e : ?T | ?fvars ⇒
      let L := fresh "L" in
      let a := fresh "a" in
      let b := fresh "b" in
      elim3 (typing_abs_inv H) L a b (?, ?)
  end.
```

7.7 Soundness

7.7.1 Progress

If *e* is locally-closed, then either it is an answer, it reduces to some other term *e'*, or there exists an evaluation context *E* such that *e* = *E*[*x*] for some free variable *x* in *e*.

```
Lemma weak_progress : ∀ e, term e →
  answer e ∨
  (∃ e':trm, red e e') ∨
```

$(\exists\, x, x \backslash\text{in}\ \mathit{fv}\ e \wedge \mathit{evals}\ e\ x)$.
By complete structural induction on *term e* (using *subterm_well_founded*).

If *e* can be typed in the empty environment, then either *e* is an answer or it reduces to some other term *e'*.
Theorem *progress* : $\forall\, e\ T\ \mathit{fvars}$,
$\quad \mathit{empty} \vdash e : T \mid \mathit{fvars} \longrightarrow \mathit{answer}\ e \vee \exists\, e', \mathit{red}\ e\ e'$.
Follows from *weak_progress* and *typing_fv*.

7.7.2 Preservation

When a function is non-unique, then all of the elements in its closure must be non-unique. In other words, all assumptions about the free variables of the function must be non-unique. That means that we can type the function in an environment *E'* (which is *E* stripped from all unnecessary assumptions) so that we can duplicate *E'* (split it into *E'* twice). We will need this lemma in the substitution lemma, when we have to substitute a function for a free variable in both terms of an application (i.e., when we have to duplicate the function, or in other words, apply it twice).
Lemma *shared_function* : $\forall\, E\ e\ a\ b\ \mathit{u\_f}\ \mathit{fvars}$,
$\quad E \vdash \mathit{trm\_abs}\ e : a\ \langle\ \mathit{u\_f}\ \rangle\ b \mid \mathit{fvars} \longrightarrow$
$\quad \mathit{typ\_equiv}\ (\mathit{rng}\ \mathit{fvars})\ \mathit{NU} \longrightarrow$
$\quad \exists\, E', \exists\, E''$,
$\qquad E' \vdash \mathit{trm\_abs}\ e : a\ \langle\ \mathit{u\_f}\ \rangle\ b \mid \mathit{fvars} \wedge$
$\qquad \mathit{split\_context}\ E\ \text{as}\ (E'\ ;\ E'') \wedge$
$\qquad \mathit{split\_context}\ E'\ \text{as}\ (E'\ ;\ E') \wedge$
$\qquad \mathit{split\_context}\ \mathit{fvars}\ \text{as}\ (\mathit{fvars}\ ;\ \mathit{fvars})$.
Follows from *split_env*, *rng_non_unique* and *fvars_and_env_consistent*.

The substitution lemma is probably the most difficult lemma in the subject reduction proof. This is not surprising, because when we substitute a term *e2* for *x* in *e1*, *e2* may be duplicated (when there is more than one use for *x* in *e1*). That is not necessarily a problem, because when there is more than one use of *x* in *e1*, then *x* must have a non-unique type and therefore it should be okay to duplicate *e2*. However, for the result of the substitution to be well-typed, if *e2* is duplicated, we must also duplicate all the assumptions that are needed to type *e2*, and that is not possible in the general case (we may need a unique assumption even when the result is non-unique). However, in the specific case that *e2* is an abstraction, we know that if *e2* is non-unique, that all of the elements in its closure must be non-unique, and so we can actually duplicate all assumptions required to type *e2* (this is what we proved in the previous lemma).
Lemma *substitution* : $\forall\, \mathit{e1}, \mathit{term}\ \mathit{e1} \longrightarrow$
$\quad \forall\, E\ \mathit{E1}\ \mathit{E2}\ \mathit{fvars}\ \mathit{fvars1}\ \mathit{fvars2}\ x\ a\ b\ \mathit{e2}\ T$,
$\qquad \mathit{split\_context}\ E\ \text{as}\ (\mathit{E1}\ ;\ \mathit{E2}) \longrightarrow \mathit{env\_wf}\ E\ \mathit{kind\_star} \longrightarrow$
$\qquad \mathit{split\_context}\ \mathit{fvars}\ \text{as}\ (\mathit{fvars1}\ ;\ \mathit{fvars2}) \longrightarrow \mathit{env\_wf}\ \mathit{fvars}\ \mathit{kind\_U} \longrightarrow$
$\qquad \mathit{E1}\ \&\ x \neg\ (a\ \langle\ \mathit{rng}\ \mathit{fvars2}\ \rangle\ b) \vdash \mathit{e1} : T \mid \mathit{fvars1}\ \&\ x \neg\ \mathit{rng}\ \mathit{fvars2} \longrightarrow$
$\qquad \mathit{E2} \vdash \mathit{trm\_abs}\ \mathit{e2} : a\ \langle\ \mathit{rng}\ \mathit{fvars2}\ \rangle\ b \mid \mathit{fvars2} \longrightarrow$
$\qquad x\ \backslash\mathit{notin}\ (\mathit{dom}\ \mathit{E1}\ \backslash u\ \mathit{dom}\ \mathit{E2}\ \backslash u\ \mathit{dom}\ \mathit{fvars1}) \longrightarrow$
$\qquad x\ \backslash\text{in}\ \mathit{fv}\ \mathit{e1} \longrightarrow$
$\qquad E \vdash$ [x ~> trm_abs e2] $\mathit{e1} : T \mid \mathit{fvars}$.

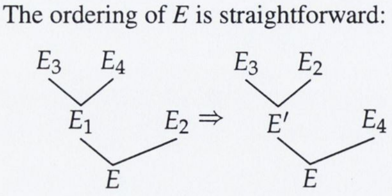
By induction on *term e1*. For the case of variables, we know that *e1* must be *x* (it cannot be a different variable because of the requirement that *x* must be free in *e1*), and the lemma follows from *weakening_2*. In the case for an application *e1 e1'*, we do case analysis on *x* \in *fv e1* and *x* \in *fv e2* (again, it cannot be in neither because of the same requirement). If it is *e1* but not in *e1'*, or in *e1'* but not in *e1*, then it is a matter of reordering the environment so that the assumptions about *e2* are passed to the appropriate branch of the application. If it is in both, then we know that *e2* must be non-unique, and we can use *shared_function* to distribute the assumptions to type *e2* to both branches. Finally, the case for abstraction uses *split_dom_inv*, *exchange* and *simplify_rng* (and we make sure to include the assumption about the bound variable of the abstraction when using the induction hypothesis).

Preservation for evaluation rule *red_value*.
Lemma *preservation_value* : ∀ *L M N*,
  term (*lt trm_abs M* in *N*) ⟶
  (∀ *x* : *S.elt*, *x* \notin *L* ⟶ *evals* (*N* ^ *x*) *x*) ⟶
  ∀ *E T fvars*,
    (*E* ⊢ *lt trm_abs M* in *N* : *T* | *fvars*) ⟶
    (*E* ⊢ *N* ^^ *trm_abs M* : *T* | *fvars*).
Follows from *substitution* and *eval_fv*.

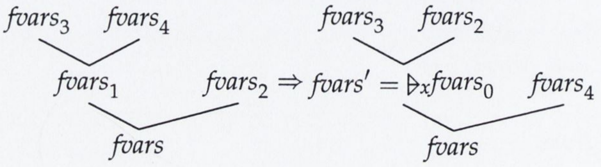
Preservation for evaluation rule *red_commute*.
Lemma *preservation_commute* : ∀ *L M A N*,
  term (*trm_app* (*lt M* in *A*) *N*) ⟶
  (∀ *x* : *S.elt*, *x* \notin *L* ⟶ *answer* (*A* ^ *x*)) ⟶
  ∀ *E T fvars*,
    (*E* ⊢ *trm_app* (*lt M* in *A*) *N* : *T* | *fvars*) ⟶
    (*E* ⊢ *lt M* in *trm_app A N* : *T* | *fvars*).
This and the next lemma are mainly a matter of re-ordering the assumptions in the environments *E* and *fvars* in a useful way. Graphically, what we want is

$$\underbrace{\underbrace{\Big(\overbrace{(\lambda\cdot A)}^{E_3\vdash -:a_0\xrightarrow{u_0}(a\xrightarrow{u}T)|_{fvars_3}} \quad \overbrace{M}^{E_4\vdash -:a_0|_{fvars_4}} \Big)}_{E_1\vdash -:a\xrightarrow{u}T|_{fvars_1}} \quad \overbrace{N}^{E_2\vdash -:a|_{fvars_2}}}_{E\vdash -:T|_{fvars}} \mapsto \underbrace{\underbrace{\Big(\lambda\cdot \overbrace{A}^{E_3,x:a_0\vdash -^{x}:a\xrightarrow{u}T|_{fvars''}} \quad \overbrace{N}^{E_2\vdash -:a|_{fvars_2}} \Big)}_{E'\vdash -:a_0\xrightarrow{\vee\triangleright_x fvars_0}T|_{\triangleright_x fvars_0}} \quad \overbrace{M}^{E_4\vdash -:a_0|_{fvars_4}}}_{E\vdash -:T|_{fvars}}$$

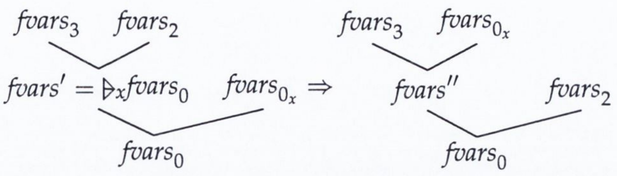
The ordering of *E* is straightforward:



but the reordering of *fvars* is slightly more involved. We have



Here, the equality on *fvars'* comes from the premise of the abstraction rule. In addition, we can use *split_dom_inv* to get



Together with *split_empty_inv*, that is sufficient to prove the lemma.

Preservation for evaluation rule *red_assoc*.

Lemma *preservation_assoc* : ∀ *L M A N*,

term (*lt lt M* in *A* in *N*) ⟶

(∀ *x* : *S.elt*, *x* \notin *L* ⟶ *answer* (*A* ^ *x*)) ⟶

(∀ *x* : *S.elt*, *x* \notin *L* ⟶ *evals* (*N* ^ *x*) *x*) ⟶

∀ *E T fvars*,

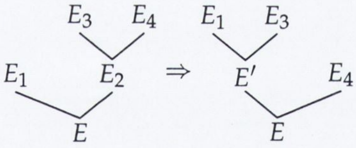
(*E* ⊢ *lt lt M* in *A* in *N* : *T* | *fvars*) ⟶

(*E* ⊢ *lt M* in (*lt A* in *N*) : *T* | *fvars*).

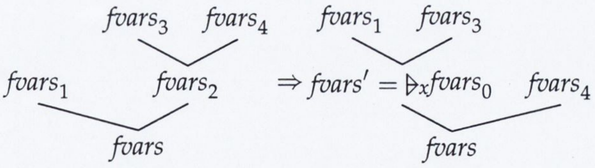
Like in the previous lemma, proving this lemma is mainly a matter of reordering the environments. The following diagram shows roughly what we're trying to achieve:

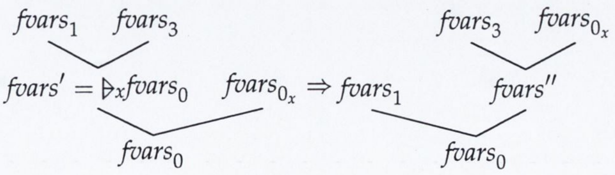
$$\underbrace{\overbrace{(\lambda\cdot N)}^{E_1\vdash -:a\xrightarrow{u}T|_{fvars_1}} \; \underbrace{(\;\overbrace{(\lambda\cdot A)}^{E_3\vdash -:a_0\xrightarrow{u_0}a|_{fvars_3}}\;\;\overbrace{M}^{E_4\vdash -:a_0|_{fvars_4}}\;)}_{E_2\vdash -:a|_{fvars_2}}}_{E\vdash -:T|_{fvars}} \mapsto \underbrace{\underbrace{(\lambda\cdot\;\overbrace{(\lambda\cdot N)}^{E_1\vdash -:a\xrightarrow{u}T|_{fvars_1}}\;\;\overbrace{A}^{E_3,x:a_0\vdash A^x:a|_{fvars''}}\;)}_{E'\vdash -:a_0\xrightarrow{\vee\triangleright_x fvars_0}T|_{\triangleright_x fvars_0}}\;\;\overbrace{M}^{E_4\vdash -:a_0|_{fvars_4}}}_{E\vdash -:T|_{fvars}}$$

Also, as for the last lemma, the reordering on *E* is straightforward,



but the ordering on *fvars* is again slightly more involved:





Preservation for evaluation rule *red_closure_app*.

Lemma *preservation_closure_app* : ∀ *E E' M*,

term (*trm_app E M*) ⟶

red E E' ⟶

```
  (∀ (E0 : env) (T : typ) (fvars : env),
        E0 ⊢ E : T | fvars → E0 ⊢ E' : T | fvars) →
  ∀ E0 T fvars,
    (E0 ⊢ trm_app E M : T | fvars) →
    (E0 ⊢ trm_app E' M : T | fvars).
```

Trivial.

Preservation for evaluation rule *red_closure_let*.

```
Lemma preservation_closure_let : ∀ L E E' M,
  term (lt M in E) →
  (∀ x : S.elt, x \notin L → red (E ^ x) (E' ^ x)) →
  (∀ x : S.elt,
      x \notin L →
      ∀ (E0 : env) (T : typ) (fvars : env),
      E0 ⊢ E ^ x : T | fvars → E0 ⊢ E' ^ x : T | fvars) →
  ∀ E0 T fvars,
    (E0 ⊢ lt M in E : T | fvars) →
    (E0 ⊢ lt M in E' : T | fvars).
```

Trivial.

Preservation for evaluation rule *red_closure_dem*.

```
Lemma preservation_closure_dem : ∀ L E0 E0' E1,
  term (lt E0 in E1) →
  red E0 E0' →
  (∀ (E : env) (T : typ) (fvars : env),
        E ⊢ E0 : T | fvars → E ⊢ E0' : T | fvars) →
  (∀ x : S.elt, x \notin L → evals (E1 ^ x) x) →
  ∀ E T fvars,
    (E ⊢ lt E0 in E1 : T | fvars) →
    (E ⊢ lt E0' in E1 : T | fvars).
```

Trivial.

If *e* has type *T* and *e* reduces to *e'*, then *e'* will also have type *T*.

```
Theorem preservation : ∀ e e', red e e' →
  ∀ E T fvars, E ⊢ e : T | fvars → E ⊢ e' : T | fvars.
```

Follows trivially by induction on $E \vdash e : T$ from the preceding preservation lemmas.

Conclusions and Future Work

Through the development of a series of successively simpler type systems we have shown how to define a uniqueness type system for the lambda calculus that is only a minor deviation from more conventional Hindley/Milner based type systems, thus demonstrating how to make uniqueness typing less unique. The advantages of such an endeavour should be self-evident: it makes uniqueness typing more accessible to the functional programming community, facilitates retrofitting uniqueness to existing compilers, and enables incorporating existing techniques for modern type system extensions such as higher rank types or generalized algebraic data types into a uniqueness type system.

In this final chapter we will review some of our design decisions in Section 8.1. Section 8.2 discusses future work, and we conclude the thesis in Section 8.3.

p. 208

p. 223

8.1 An exploration of the design space

We have presented three different uniqueness type systems in this thesis; add to that the original Clean type system, the uniqueness logic proposed by Harrington and the uniqueness type/sharing analysis system proposed by Hage et al. and it is clear that the design space for uniqueness type systems is large. In this section, we will discuss some of the dimensions of this space, the possible choices within those dimensions and why we prefer some options over others.

8.1.1 Boolean attributes versus inequality constraints

The choice between using boolean attributes as uniqueness attributes and using a separate constraint language is partly aesthetic. For example, when Hage et al. (2007, Section 3.3) argue that uniqueness polymorphism and constraints are simultaneously useful, they give the following example:

$$\texttt{apply ::}\ (t^u \xrightarrow{u_f} s^v) \xrightarrow{\times} t^w \xrightarrow{u'_f} s^v, [w \leq u, u'_f \leq u_f]$$

```
apply f x = f x
```

(Although Clean too supports both polymorphism and constraints, this is not a valid type in Clean due to the non-uniformity of the subtyping relation; instantiating t and s with `Int` would however yield a valid type.) Using boolean attributes we can recode this type as

$$\texttt{apply ::}\ (t^u \xrightarrow{u_f} s^v) \xrightarrow{\times} t^{u \vee w} \xrightarrow{u_f \vee u'_f} s^v$$

Which type is to be favoured is a matter of taste, and in cases such as the one above where one uniqueness attribute (u) constrains another (w) and both attributes are necessary there is no clear advantage to either approach.

p. 132, p. 147 On the other hand, we have repeatedly argued in this thesis (Section 5.1.5, Section 6.3) that we should allow terms to be as polymorphic in their uniqueness as possible, and never assign a unique type to a term when not strictly necessary. We can take advantage of that approach and (without loss of generality) simplify the type of `apply` to

$$\texttt{apply ::}\ (t^u \xrightarrow{u_f} s^v) \xrightarrow{\times} t^u \xrightarrow{u_f} s^v$$

or even

$$\texttt{apply ::}\ (a \xrightarrow{u_f} b) \xrightarrow{\times} a \xrightarrow{u_f} b$$

by taking advantage of type variables of kind $\star$. Of course, we can adopt the same strategy (and keep terms polymorphic in their uniqueness) in a system based on constraints, and consequently simplify the type of `apply` the same way. However, when attributes are constrained by more than one other attribute, this simplification is no longer possible in a system based on constraints. For example, the constraint-based type

$$\texttt{swap ::}\ (a^u, b^v)^{u_p} \xrightarrow{\times} (b^v, a^u)^{u'_p}, [u_p \leq u, u_p \leq v]$$

```
swap (x, y) = (y, x)
```

can be simplified to

$$\texttt{swap ::}\ (a^u, b^v)^{u \vee v} \xrightarrow{\times} (b^v, a^u)^w$$

which we feel is simpler and preferable to the type with constraints. Note again that the simplification takes advantage of the fact that for any derivation $e : (a^u, b^v)^\bullet$ there is also a derivation of $e : (a^u, b^v)^\times$, so that this is in fact a correct translation of the type with constraints (as discussed p. 132 in Section 5.1.5). This simplification relies essentially on using a boolean expression and is frequently useful. For example, functions with more than two arguments typically have types such as

$$\texttt{f ::}\ t^u \xrightarrow{\times} s^v \xrightarrow{u_f} r^w \xrightarrow{u'_f} \ldots, [u_f \leq u, u'_f \leq u, u'_f \leq v]$$

```
f x y z = ..
```

Using boolean attributes we can simplify this to

$$\texttt{f ::}\ t^u \xrightarrow{\times} s^v \xrightarrow{u} r^w \xrightarrow{u \vee v} \ldots$$

where we have eliminated u_f and u'_f (using constraints, we can eliminate u_f but not u'_f).

Moreover, boolean attributes offer a clear advantage over a constraint based approach in the formalization and implementation of the type system. Although some of the technical difficulties p. 115 we mentioned in Chapter 4 can be resolved by recasting the type system using the qualified types framework[1], various difficulties remain—and of course, an implementation and formalization would have to deal with qualified types; this is an especially valid concern in formal proofs.

Simplification of constraint sets (taking transitive closures and removing "redundant" constraints, p. 117 exemplified in Section 4.1.4) are *ad hoc* techniques and must therefore be studied with care. For example, Jones (1995a, Section 6.3) notes that the restriction to unambiguous type schemes may cause difficulties in the context of a system with subtyping. Whether or not this is a problem in a uniqueness type system (with its very shallow subtyping relation) is an open question.

[1] Qualified types are well-known for causing a loss of sharing (Jones, 1995a, Section 6.1.2, *Unnecessary polymorphism*; see also a discussion of Haskell's monomorphism restriction, for example in Hudak et al., 2007). A uniqueness type system based on qualified types framework, which uses evidence parameters for subtyping constraints—such as the one proposed by Hage et al. (2007)—must be careful that this does not cause unique objects to be shared during optimization.

If it is, then a weakened ambiguity restriction needs to be used (Jones, 1995a, Section 5.8.3) in which uniqueness variables are allowed to appear in constraints that do not appear in the type proper. However, this will complicate the implementation of subsumption (Section 2.5.2). For p. 45 example, consider checking whether

$$\forall a\ u\ v\ w \cdot a^{u} \xrightarrow{\times} a^{w}, [u \leq v, v \leq w] \qquad \preceq \qquad \forall a\ u\ w \cdot a^{u} \xrightarrow{\times} a^{w}, [u \leq w]$$

(Example adopted from Fuh and Mishra, 1989.) As part of the subsumption check we will need to check whether $[\mathfrak{u} \leq \mathfrak{w}]$ entails $[\mathfrak{u} \leq v, v \leq \mathfrak{w}]$, where $\mathfrak{u}$ and $\mathfrak{w}$ are the skolemized versions of the uniqueness attributes of the right-hand type scheme—for *some* variable v. It is non-obvious how to define the necessary combination of the entailment check with unification.

By contrast, the boolean approach requires only the addition of two well-understood algorithms, which can be implemented and tested or proven correct independently from the rest of the compiler: boolean unification and boolean simplification. For example, the subsumption check between two type schemes does not need to do *anything* special regarding the uniqueness attributes when using boolean attributes (Section 5.2.4). p. 138

Finally, although the approach based on constraints is relatively easy to use in compilers that already support qualified types, retrofitting uniqueness typing to compilers that do not support qualified types is much easier with the boolean attribute approach.

8.1.2 Subtyping

A uniqueness type system distinguishes between unique types and non-unique types. Duplication of terms of a unique type is disallowed, so that the type system guarantees that there is only a single reference to terms of a unique type. Although it seems safe to assume that a unique term is non-unique—after all, we are simply forgetting about the uniqueness guarantee—we have seen that we need to be careful with partial applications. There are various solutions to this problem:

1. In Clean (Section 3.2.4), the partial application of a function must be unique when any of p. 82 the supplied arguments is unique. Moreover, unique functions are *necessarily* unique: the subtyping relation is modified so we cannot regard unique functions as non-unique. This non-uniformity of the subtyping relation has various disadvantages:

    - Lack of principal types (Barendsen and Smetsers, 1996)
    - Since functions cannot lose their uniqueness, type variables cannot either (since they could be instantiated to function types). For example, the function dup must be assigned the type

      ```
      dup :: $t^{\times} \xrightarrow{\times} (t^{\times}, t^{\times})^{u}$
      dup x = (x,x)
      ```

    - Type inference becomes more difficult: when inferring a type for a variable marked as shared ($x^{\otimes}$), if we do not yet know the type of the variable (the type is still a meta-variable), we cannot execute the uniqueness correction until later when the meta-variable is either instantiated by a type or is universally quantified in a generalization step.

2. The essence of the problem is that the type of a function is not specific enough. Generally, when we extract something from a container (such as a pair of elements), if we extract a unique element from the container the container must be unique itself. Executing a function may involve extracting unique elements from the closure of the function (a container)—but the type of a function does not tell us the types of the elements in the closure of the function. p. 115 We therefore proposed a form of closure typing in Chapter 4, and added a second uniqueness attribute to the function arrow indicating whether the closure of the function contained any unique elements. With this approach, it is sound to duplicate functions; the typing rules will enforce that functions with unique elements in their closure are unique *when applied.* For example, `dup` now has the type

   $$\texttt{dup ::}\ t^u \xrightarrow[\times]{\times} (t^\times,t^\times)^v$$

   This restores the uniformity of the subtyping relation at the cost of introducing an additional attribute into the type language. This approach was adopted by Hage et al. (2007; 2008).

   p. 123 Although this approach seems to introduce rather complicated types (Section 4.3), we p. 129 showed in Chapter 5 that this complexity can be reduced by using boolean attributes.

3. p. 143 We argued in Chapter 6 that we do not need uniqueness coercion at all if we are careful in the types we assign to primitive functions. For example, the function that clears all elements of an array should get the type

   $$\texttt{resetArray ::}\ \texttt{Array}^\bullet \xrightarrow{\times} \texttt{Array}^u$$

   rather than

   $$\texttt{resetArray ::}\ \texttt{Array}^\bullet \xrightarrow{\times} \texttt{Array}^\bullet$$

   If `resetArray` is assigned the second type, we will not be able to pass the result of `resetArray` to a function that expects a non-unique argument (or indeed, duplicate the array) without a uniqueness coercion. However, if we use the first type instead then we will be able to treat the result of `resetArray` as unique or non-unique, depending on what we need to do. We are taking advantage of polymorphism to encode subtyping.

   A downside of this approach is that we need higher rank types (and thus perhaps a type p. 216 annotation, but see Section 8.2.5) if we are using one term at two different uniqueness p. 147 levels. The example we considered in Section 6.3 was

   $$\texttt{f ::}\ \forall v.(\forall u.\texttt{Array}^u) \xrightarrow{\times} \texttt{Array}^v$$
   $$\texttt{f arr = }\textbf{if}\ \texttt{isEmpty arr}^\otimes\ \textbf{then}\ \texttt{shrink arr}^\odot\ \textbf{else}\ \texttt{grow arr}^\odot$$

   Since we are using `arr` both as a unique array (in both branches) and as a non-unique array (in the condition), `arr` needs to be polymorphic in its uniqueness in the absence of an explicit coercion relation. However, we feel that this is a small price to pay.

4. We can avoid the partial application problem completely by disallowing unique elements p. 108 inside function closures. This is the approach taken in Mercury (Section 3.6.3) and in some p. 89 of Wadler's linear type systems (Section 3.3.2). In practical terms, it means that functions can only have a single unique argument, which must be the last. Functions that require more than one unique argument must be uncurried. For example,

   $$\texttt{closeFile ::}\ (\texttt{File}^\bullet\texttt{,World}^\bullet)^\bullet \longrightarrow \texttt{World}^\bullet$$

   Similarly, the function that returns the first of two arguments gets the type

   $$\begin{array}{l}\texttt{const ::}\ \forall(t:\mathcal{T})(a:\star)\cdot t^{\times} \to a \to t^{\times}\\ \texttt{const x y = x}\end{array}$$

   Of course, the type of `curry` is also affected:

   $$\begin{array}{l}\texttt{curry ::}\ \forall(t:\mathcal{T})(u:\mathcal{U})(a\,b:\star)\cdot((t^{\times},a)^{u} \to b) \to t^{\times} \to a \to b\\ \texttt{curry f =}\ \lambda\texttt{x.}\ \lambda\texttt{y. f (x, y)}\end{array}$$

   Although this approach makes it possible to introduce a general coercion relation from unique to non-unique terms, it is also of interest in the absence of such a coercion relation as it allows to assume that *every* function is non-unique; hence we can interpret $(\to)$ as $(\xrightarrow{\times})$ and we no longer need to add uniqueness attributes to arrows.

5. For completeness, we also mention the approach suggested by Harrington (2001), where the result of a non-unique function must be non-unique. However, as we argued in Section 3.4, we consider this approach inadequate for maintaining purity. p. 91

8.1.3 Attributes as types

We have a choice between treating uniqueness attributes and "base types" as two different syntactic categories, or treating both as types and distinguish based on a kind system. We argued in Section 6.1 that treating uniqueness attributes as types (of kind $\mathcal{U}$) has various advantages: we gain free additional expressive power in the definition of algebraic data types and type synonyms, and the presentation of the type system as well as the presentation of types is simplified. For example, the types we considered in the previous section p. 143

$$\begin{array}{l}\texttt{const ::}\ t^{\times} \to a \to t^{\times}\\ \texttt{curry ::}\ ((t^{\times},a)^{u} \to b) \to t^{\times} \to a \to b\end{array}$$

make essential use of type variables over kind $\mathcal{T}$, $\mathcal{U}$ and kind $\star$ to be able to present this types in a natural way. Note that kind annotations are not necessary: the kind inferencer will be able to infer the kinds of type variables without user assistance.

Distinguishing base types and uniqueness attributes using a kind system also cleans up various other aspects of the type system. For example, in type systems such as Clean there are three different kinds of variables: type variables, uniqueness variables, and variables that denote arguments to algebraic data types and correspond to a type with an attribute. Using a kind system, we only need one notion of a type variable (of kind $\mathcal{T}$, $\mathcal{U}$ or $\star$, respectively).

Moreover, the definition of a type scheme is clarified. For example, the usual kinding rule for type schemes is (e.g., van Bakel et al., 1997)

$$\frac{\Gamma, a:\kappa \vdash \sigma : * \qquad \kappa : \Box}{\Gamma \vdash \forall(a:\kappa)\cdot\sigma : *}\ \textsc{Univ}$$

The universally quantified variable can range over types of an arbitrary kind κ (recall from Section 2.3 that $\Box$ is the universe of kinds), but the type that we are quantifying over is restricted to be of kind $\star$ (value types). We use exactly the same rule in our system (Chapter 6). For example, the type of the identity function is p. 29 p. 143

$$\texttt{id ::}\ \forall(a:\star), \left(a \to a\right)^{\times}$$

(We are avoiding the shorthand notation $a \xrightarrow{\times} a$ to facilitate the comparison, below.) In the approach proposed by Hage et al. (2007) universal quantification effectively ranges over types of kind $\mathcal{T}$:

$$\frac{\Gamma, t : \mathcal{T} \vdash \sigma : \mathcal{T}}{\Gamma \vdash \forall (t : \mathcal{T}) \cdot \sigma : \mathcal{T}}\ \textsc{Univ}\mathcal{T} \qquad \frac{\Gamma, u : \mathcal{U} \vdash \sigma : \mathcal{T}}{\Gamma \vdash \forall (u : \mathcal{U}) \cdot \sigma : \mathcal{T}}\ \textsc{Univ}\mathcal{U}$$

In this approach, the universal quantifier ranges over non-value types (types of kind other than $\star$). p. 86 This is slightly unconventional and has some disadvantages (Section 3.2.8): the top-level attribute of a type cannot be universally quantified, and types become more difficult to interpret (we will p. 210 discuss a benefit of their approach in Section 8.2.3). For example, the type of `id` is now

$$\texttt{id ::}\ \left(\forall (t : \mathcal{T})(u : \mathcal{U}), t^u \to t^u\right)^{\times}$$

Hage et al. do not give an explicit kinding rule, but enforce the structure of universal quantification syntactically. We feel that even if $\textsc{Univ}\mathcal{T}$ is preferred over rule $\textsc{Univ}$ (Section 8.2.3), using a kind system makes it possible to be more explicit about this decision and consider its consequences. One minor disadvantage of using a kind system is that the metatheory needs to include a kinding relation. However, for any serious type system this needs to be the case anyway.

Finally, we note that it is possible to reduce the number of kinds by interpreting $\mathcal{T}$ as $\mathcal{U} \to \star$ p. 143 or $\mathcal{U}$ as $\mathcal{T} \to \star$ (Section 6.1). This makes little difference from a theoretical point of view; whether or not it is more elegant we leave up to the reader.

8.1.4 Uniqueness propagation in constructors or destructors

One design decision that we have not emphasized in this thesis but deserves a discussion at this point is whether constructors or destructors (or both) should propagate uniqueness. We discussed p. 81 uniqueness propagation in Section 3.2.3, where we mentioned the following Clean type for the first projection function for pairs:

$$\texttt{fst ::}\ (t^u, s^v)^w \to t^u, [w \leq u]$$

The constraint $[w \leq u]$ indicates that if we want to extract a unique element from the pair, the pair itself must be unique. Hence, destructors (such as `fst`) in Clean enforce uniqueness propagation.

Constructors in Clean *also* enforce uniqueness propagation; the type of the pair constructor is

$$\texttt{(,) ::}\ t^u \to s^v \to (t^u, s^v)^w, [w \leq u, w \leq v]$$

It is important that destructors enforce propagation because even though the constructor may enforce propagation too, a unique pair may later become non-unique; hence, we must check "again" when destructing the pair. Nevertheless, since Clean demands that containers containing necessarily unique objects are necessarily unique themselves, it is also important that the constructor enforces uniqueness propagation: if a necessarily unique argument is given to the pair constructor, then the pair itself becomes necessarily unique.

In a sense, Clean uses the constructor to guarantee uniqueness propagation for partial applications (which become necessarily unique when any of the supplied arguments are unique, and are therefore guaranteed to remain unique when created), and the destructor for everything else. Propagation for partial applications *cannot* be enforced during destruction (that is, when the function is applied) since the function type does not tell us if there are any unique objects in the p. 115 function closure, unless we use a form of closure typing such as the one suggested in Chapters 4 p. 129, p. 203 and 5 (see also point 2 in Section 8.1.2, above).

If we use closure typing, it is possible to enforce uniqueness propagation in destructors *only*. In fact, this may even be possible in Clean (where there is no closure typing) if we allow necessarily unique objects to be wrapped inside containers that are "unique now but may become non-unique later", as long as the destructor for that container will make it impossible to extract the necessarily unique object from a non-unique container.

If however we always interpret "unique" as "necessarily unique" and no longer support a notion of "unique now, but may become non-unique later" (as we suggest in Chapter 6), it also p. 143
becomes possible to enforce propagation in *constructors* only. For example, if the type of the pair constructor is

$$\texttt{(,)} \;\; \texttt{::} \;\; t^u \xrightarrow{\times} s^v \xrightarrow{u} (t^u, s^v)^{u \vee v}$$

then when we construct a pair with a unique component, the pair itself must be unique and will therefore remain unique. That means that we no longer need to enforce uniqueness propagation in the destructor, so that we can give the following type to the first projection function:

$$\texttt{fst} \;\; \texttt{::} \;\; (t^u, s^v)^w \xrightarrow{\times} t^u$$

In theory this function can be used to extract a unique object from a non-unique pair but that is okay since there *are* no non-unique pairs with unique components (due to the constructor).

The obvious question is whether it is better to enforce propagation in destructors or in constructors. Enforcing propagation in destructors admits more programs: if we have a non-unique pair with one unique component and one non-unique component, we will still be able to extract the non-unique component from the pair. If the constructor enforces propagation, this will not be possible since a pair with a unique component cannot become non-unique in that approach.

On the other hand, it is not clear how often it is useful to extract a non-unique component from a non-unique container with some unique components. Enforcing propagation in the constructor is more uniform since some attributes *must* be propagated in the constructor. In particular, the attribute on arrows must be propagated in the function space constructor (rule for abstraction) from the attributes in the function closure. (The observer attribute we discuss in Section 8.2.9 p. 221
must also be propagated in constructors rather than destructors.)

Finally, if GADTs are supported (Section 5.3) the user can decide the type of the constructors p. 140
of a data type so that it may be possible to support both: if the user specifies that the constructor propagates uniqueness it is no longer necessary to do so in the destructor and vice versa.

In summary, it is always possible to enforce propagation in destructors only; even necessarily unique objects may be wrapped in non-unique containers as it will be impossible to extract them. When unique *always* means "necessarily unique", it is also possible to enforce propagation only in the constructors. Enforcing propagation in both constructors and destructors is not necessary.

8.1.5 Number of aspects considered

In uniqueness typing, we consider only one usage aspect of a term: is there one or more than one reference to a term? Various authors have considered systems that consider more than one aspect; for example, the single-threaded polymorphic lambda calculus (Section 3.5) uses three: read-only, p. 95
free, and single-threaded. When multiple aspects are subsequently combined into one "abstract use", the definition of the type system becomes rather opaque. For example, the definition of the various operators in the STPLC (Figure 3.7) is far from obvious, and it is hard to tell how the p. 98
STPLC differs from similar type systems such as the system proposed by Odersky (Section 3.5.5). p. 102

Another puzzling feature of the STPLC is that it allows for arbitrary disjunctions and conjunctions between these aspects, resulting in a lattice of 20 different properties whose inclusions are
p. 96 not self-evident (Figure 3.6). It is not clear whether such generality gives any additional expressive power over the simple product of the three aspects (i.e., 8 different properties: $\overline{RSF}$, $\overline{RS}F$, $\overline{R}S\overline{F}$, $\overline{R}SF$, $R\overline{SF}$, $R\overline{S}F$, $RS\overline{F}$ and RSF), especially when aspect polymorphism is also supported.

Although considering more aspects can make the system more precise, it is important that the type system does not become too complex (for example, consider the `folda` example in
p. 101 Section 3.5.4). It therefore seems important that the individual aspects can be considered and understood independently. Moreover, twenty years of experience with uniqueness typing seems to suggest that considering only uniqueness (the multi-threaded aspect in the STPLC) is sufficient
p. 221 (but see Section 8.2.9).

8.2 Future work

The design of a type system is a major effort and we have only focused on certain aspects in this thesis. There is therefore plenty of scope for future work; we will identify the most important in this section.

8.2.1 Simplifying the type language

Although the typing rules only make use of disjunction between boolean attributes, unification may introduce conjunctions and negations. In this section we discuss when this arises and whether it can be avoided. Consider

$$\begin{array}{ll} \texttt{use2} & \texttt{::}\ t^u \xrightarrow{\times} s^v \xrightarrow{u} a \xrightarrow{u \vee v} \texttt{Int}^\times \\ \texttt{req\_notu\_fn} & \texttt{::}\ (a \xrightarrow{\times} b) \xrightarrow{\times} \texttt{Int}^\times \end{array}$$

The definitions of these functions is not important; `use2` is a function of more than two arguments so that `use2` x y must be unique if either x or y is, and `req_notu_fn` is a higher order function that requires a non-unique function as argument (perhaps because that function is applied more than once). Since $u \vee v \overset{?}{\approx} \times$ only if $u \approx \times$ and $v \approx \times$, it follows that if we pass `use2` x y as argument to `req_notu_fn`, both x and y must be non-unique:

$$\begin{array}{l} \texttt{bunif1 ::}\ t^\times \xrightarrow{\times} s^\times \xrightarrow{\times} \texttt{Int}^\times \\ \texttt{bunif1 x y = req\_notu\_fn (use2 x y)} \end{array}$$

However, suppose we have a higher order function that demands to be passed a *unique* function:

$$\texttt{req\_uniq\_fn ::}\ (a \xrightarrow{\bullet} b) \xrightarrow{\times} \texttt{Int}^\times$$

The rule for abstraction takes advantage of the fact that the typing rules *never* require a unique
p. 132 function (Section 5.1.5), and indeed there is never a need to demand a unique function. However, a function such as `req_uniq_fn` can easily be defined by providing a type annotation. We mentioned before that `use2` x y is unique if either x or y is; if we *require* this partial application to be unique, then at least one of x or y must be unique:

$$\begin{array}{l} \texttt{bunif2 ::}\ t^{\neg u \vee v} \xrightarrow{\times} s^u \xrightarrow{\neg u \vee v} \texttt{Int}^\times \\ \texttt{bunif2 x y = req\_uniq\_fn (use2 x y)} \end{array}$$

$$\begin{array}{l} \texttt{bunif3 ::}\ t^u \xrightarrow{\times} s^{\neg u \vee v} \xrightarrow{u} \texttt{Int}^\times \\ \texttt{bunif3 x y = req\_uniq\_fn (use2 y x)} \end{array}$$

The strange attribute $\neg u \vee v$ comes from the most general unifier

$$\begin{bmatrix} u & \mapsto u \\ v & \mapsto \neg u \vee v \end{bmatrix}$$

of the unification equation $u \vee v \overset{?}{\approx} \bullet$, and encodes exactly the requirement that at least one of x and y must be unique. This shows that the boolean attribute approach can express much more complex relations between uniqueness attributes than the constraints approach (using only implications between attributes).

The important question is whether this additional expressive power is useful. The example above is contrived because it unnecessarily requires a function to be unique. Nevertheless, we have seen in Section 6.4.3 that it is sometimes useful to express implications using a conjunction p. 153 rather than a disjunction. An interesting open question is under which conditions (if at all) we can simplify the type language to support disjunction only (disallowing conjunction and negation).

As well as simplifying the type language, this has an important computational advantage: simplification of general boolean expressions is expensive (exponential complexity). Simplification of boolean expressions that use only disjunctions, on the other hand, is simple operation that can be performed in linear time.

8.2.2 Simplifying types

As well as simplifying the type language, it may be possible to further simplify types; unfortunately, boolean simplification of individual attributes does not appear to be sufficient. For example, suppose that strings have a polymorphic uniqueness[1]:

$$\texttt{"a" :: String}^{u}$$

If we pass a string to the function `use2` from the previous section, we get

$$\begin{array}{l} \texttt{need\_simpl :: } a \xrightarrow{u \vee v} \texttt{Int}^{\times} \\ \texttt{need\_simpl = use2 "a" "b"} \end{array}$$

Although the expression $u \vee v$ cannot be further simplified, the type of this function can be simplified to

$$\texttt{need\_simpl :: } a \xrightarrow{u} \texttt{Int}^{\times}$$

Indeed, when a user provides this type annotation the type checker will accept this type as valid since the subsumption check

$$\forall a\, u\, v \cdot a \xrightarrow{u \vee v} \texttt{Int}^{\times} \quad \preceq \quad \forall a\, u \cdot a \xrightarrow{u} \texttt{Int}^{\times}$$

succeeds. In fact, the subsumption holds in both directions:

$$\forall a\, u \cdot a \xrightarrow{u} \texttt{Int}^{\times} \quad \preceq \quad \forall a\, u\, v \cdot a \xrightarrow{u \vee v} \texttt{Int}^{\times}$$

It is clear therefore that these two types are isomorphic. However, the individual uniqueness attributes cannot be further simplified when considered independently but only when considered as part of the larger type, and it is not obvious how to automate this simplification.

[1] Assuming that every occurrence of a string denotes a *different* instances of that string, of course.

This problem looks fairly innocuous in the above examples, but it can get sufficiently bad to become a real problem. For example, we discussed the function `swap` in Section 6.4.3:

$$\texttt{swap ::}\ (t^u, s^v)^{u \vee v \vee w} \xrightarrow{\times} (s^v, t^u)^{w'}$$
$$\texttt{swap p = (p}^{\odot}\texttt{.field2, p}^{\odot}\texttt{.field1)}$$

Unfortunately, the *inferred* type for `swap` is

$$\texttt{swap ::}\ (s^{(\neg v \wedge u) \vee (\neg v \wedge w) \vee (u1 \wedge u) \vee (u1 \wedge w)}, t^u)^{u \vee w} \xrightarrow{\times} (t^u, s^{(\neg v \wedge u) \vee (\neg v \wedge w) \vee (u1 \wedge u) \vee (u1 \wedge w)})^{v1}$$

As it turns out these two types are isomorphic, but this is non-obvious to say the least (even the type checker requires a long time to verify it) and it is clear that further simplification of types is a necessity—even though the attribute on s has been minimized.

8.2.3 Improving Impredicativity

At first sight, it may seem that the two types

$$\forall (a : \star) \cdot a \xrightarrow{\times} a$$

and

$$\forall (t : \mathcal{T})(u : \mathcal{U}) \cdot t^u \xrightarrow{\times} t^u$$

are interchangeable in all contexts. Unfortunately, that is not the case. A type variable a of kind $\star$ can be instantiated with a polymorphic type (impredicative instantiation); impredicative instantiation is not possible if we replace a by t^u for two type variables of kind $\mathcal{T}$ and $\mathcal{U}$.

p. 51 Consider the following example using HMF (Section 2.5.4), Morrow's original type system. Suppose we define a function that wraps an object in a record:

$$\texttt{wrap ::}\ \forall (a : \star) \cdot a \rightarrow \{\!(l : a)\!\}$$
$$\texttt{wrap x = \{l = x\}}$$

p. 38, p. 51 The application (`wrap id`) has an ambiguous type (Section 2.4.4, Section 2.5.4); by default, Morrow will never choose impredicative instantiation and assign the type

$$\texttt{wrap id ::}\ \forall a \cdot \{\!(l : a \rightarrow a)\!\}$$

To override this behaviour, we can use a rigid type annotation (Leijen, 2008a) to indicate that we require impredicative instantiation:

$$\texttt{wrap (id ::}\ \forall a \cdot a \rightarrow a\texttt{) ::}\ \{\!(l : \forall a \cdot a \rightarrow a)\!\}$$

Alas, this does not transfer easily to our version of Morrow with support for uniqueness. In our system, `wrap` has the type

$$\texttt{wrap ::}\ \forall t\, u\, v \cdot t^u \xrightarrow{\times} \{\!(l : t^u)\!\}^v$$

Like in HMF, the type of (`wrap id`) is

$$\texttt{wrap id ::}\ \forall t\, u\, v \cdot \{\!(l : t^u \xrightarrow{\times} t^u)\!\}^v$$

Unlike in HMF however we cannot use the rigid type annotation to force impredicative instantiation. If we try

$$\texttt{wrap (id ::}\ \forall t\, u \cdot t^u \xrightarrow{\times} t^u\texttt{)}$$

Morrow will give the following error message:

```
Types do not match
  context        : wrap (id :: $\forall t\, u \cdot t^u \xrightarrow{\times} t^u$)
  term           : (id :: $\forall t\, u \cdot t^u \xrightarrow{\times} t^u$)
  inferred type  : $\forall s\, v . s^v \xrightarrow{\times} s^v$
  does not match: $t^u$
```

which is exactly the problem we alluded to above: a type t^u for two type variables of kind $\mathcal{T}$ and $\mathcal{U}$ cannot be instantiated with a polymorphic type.

If this example seems somewhat arcane, we briefly mentioned in Section 6.4.2 that the fields of records are limited to have monomorphic types. This is due to the same problem. Consider the type of record field selection:

p. 151

```
(_.l) :: $\forall r\, t\, u\, v \cdot r \setminus l \Rightarrow \{\llparenthesis l : t^u \mid r \rrparenthesis\}^{u \vee v} \xrightarrow{\times} t^u$
```

Notice that the type of the field must have type t^u; hence, if we create a record with a polymorphic field[1]

```
rec :: $\forall u . \{\llparenthesis l : \forall t\, v . t^v \xrightarrow{\times} t^v \rrparenthesis\}^u$
rec = {l = id :: $\forall t\, v . t^v \xrightarrow{\times} t^v$}
```

and try to select that field

```
rec.l
```

we get a similar error message:[2]

```
Types do not match
  context        : rec.l
  term           : rec
  inferred type  : $\{\llparenthesis l : \forall s\, u1 . s^{u1} \xrightarrow{\times} s^{u1} \rrparenthesis\}^w$
  does not match: $\{\llparenthesis l : t^u \mid r \rrparenthesis\}^{u \vee v}$
```

This is unfortunate, because restricting records to monomorphic fields rather limits their usefulness.

There is no trivial solution to this problem. Although the type of `wrap` could conceivably be changed to

```
wrap :: $\forall a\, v \cdot a \xrightarrow{\times} \{\llparenthesis l : a \rrparenthesis\}^v$
```

this solution it not possible for field selection, because we need to know what the attribute on the type is to be able to express uniqueness propagation (the requirement that the record must be unique if we are extracting a unique field).

One solution is to adopt the approach suggested by Hage, Holdermans, and Middelkoop (2007) and change the universal quantifier to quantify over types of kind $\mathcal{T}$ instead. We discussed some of the disadvantages of this approach in Section 8.1.3, but an advantage of their approach is that the problems discussed in this section disappear since a type t^u now *can* be instantiated with a polymorphic type, which will have the shape $(\forall \ldots)^u$.

p. 205

One potential alternative is to promote the ceiling $\lceil\rceil$ operator from Section 4.2.4 from the meta-level to the object level as a type constructor of kind

p. 122

$$\lceil\rceil : \star \rightarrow \mathcal{U}$$

[1] The reason we can even create this record is that the internal type of record creation quantifies over a type of kind $\star$, like the modified type of `wrap` shown later in the section.

[2] Note that Morrow's type system is invariant, so that we cannot instantiate type schemes under type constructors. In a covariant type system we could type `rec.l` but this circumvents the problem by avoiding impredicative instantiation—it does not solve the more general problem.

For example, the function `const` might get the type

$$\begin{aligned}&\texttt{const} :: \forall a\,b \cdot a \xrightarrow{\times} b \xrightarrow{\lceil a \rceil} a\\ &\texttt{const x y = x}\end{aligned}$$

instead of

$$\begin{aligned}&\texttt{const} :: \forall t\,u\,b \cdot t^{u} \xrightarrow{\times} b \xrightarrow{u} t^{u}\\ &\texttt{const x y = x}\end{aligned}$$

Similarly, the type of field selection would become

$$\texttt{(\_.l)} :: \forall r\,a\,v \cdot r \setminus l \Rightarrow \{(\!| l : a \mid r |\!)\}^{\lceil a \rceil \vee v} \xrightarrow{\times} a$$

As mentioned in Section 4.2.4, we have to be careful defining this operator. If we allow for arbitrary boolean expressions as attributes, then we might define it as

$$\lceil \forall \vec{a} \cdot \rho^{\nu} \rceil = \begin{cases} \nu & \text{if } fv(\nu) \cup \vec{a} = \emptyset \\ \bullet & \text{otherwise} \end{cases}$$

The introduction of this operator might streamline other aspects of the type system as we must
p. 137 frequently be careful with terms with a polymorphic uniqueness (for example, see Section 5.2.3): introducing an object-level ceiling operator might mean that we have to worry about this problem in fewer places (of course, this is exactly the problem that the approach by Hage et al. avoids, by not allowing universal quantification over the top-most attribute of a type).

However, it seems that if we pursue this approach to its logical conclusion we will also need various other object-level operators. For example, we might need the "dual" operator

$$\lfloor \cdot \rfloor : \star \rightarrow \mathcal{T}$$

so that we can write

$$\texttt{dup} :: \forall a\,u \cdot a \xrightarrow{\times} (\lfloor a \rfloor^{\times}, \lfloor a \rfloor^{\times})^{u}$$

Depending on the approach we take to subtyping, we must be careful with the definition of this operator. In particular, if we do not use subtyping at all we should be able to instantiate this type of `dup` to

$$\texttt{dup} :: t^{\times} \xrightarrow{\times} (t^{\times}, t^{\times})^{u}$$

or

$$\texttt{dup} :: (\forall u \cdot t^{u}) \xrightarrow{\times} (t^{\times}, t^{\times})^{u}$$

but not to

$$\texttt{dup} :: t^{\bullet} \xrightarrow{\times} (t^{\times}, t^{\times})^{u}$$

That means that $\lfloor t^{\bullet} \rfloor$ must be undefined (just like the uniqueness correction $\llcorner\lrcorner$ is undefined for some types).

We may need yet another operator to be able to define the `to` operator discussed in Sec-
p. 151 tion 6.4.2, which was defined so that

$$\texttt{to}\ u\ a\ (\!| x : t^{v}, y : s^{w} \mid r |\!)$$

evaluates to

$$\llparenthesis x : t^{v \wedge u} \longrightarrow a, y : s^{w \wedge u} \longrightarrow a \mid r \rrparenthesis$$

If we wanted to support polymorphic fields, this operator must be modified to be defined over general (possibly polymorphic) types

$$\texttt{to}\ u\ a\ \llparenthesis x : \sigma, y : \sigma' \mid r \rrparenthesis$$

The example we considered was

$$\begin{array}{l} \texttt{select ::}\ \ \langle \llparenthesis l : t^{u \wedge v} \rrparenthesis \rangle^{v} \xrightarrow{\times} t^{u \wedge v} \\ \texttt{select v = }\textbf{case}\texttt{ v \{l = }\lambda\texttt{x} \rightarrow \texttt{x\}} \end{array}$$

However we modify the type of `select` to support polymorphic fields, we should be able to instantiate the result type to

$$\texttt{select ::}\ \ \langle \llparenthesis l : \forall u \cdot t^{u} \rrparenthesis \rangle^{\bullet} \xrightarrow{\times} t^{\bullet}$$

but (importantly) not to

$$\texttt{select ::}\ \ \langle \llparenthesis l : \forall u \cdot t^{u} \rrparenthesis \rangle^{\times} \xrightarrow{\times} t^{\bullet}$$

The combination of $\lfloor\rfloor$ and $\lceil\rceil$ does not appear to be sufficient to achieve this. On the other hand, the `to` operator is rather unusual: we have only found one use for using conjunctive attributes on the fields of a record (rather than a disjunctive attribute on the record itself). Giving up first-class pattern matching to remove the need for `to` may be a small price to pay for a simpler type system. When pattern matching becomes a language construct, we can give `select` the same type as for field selection, above:

$$\texttt{select ::}\ \ \{\llparenthesis l : a \rrparenthesis\}^{\lceil a \rceil \vee v} \xrightarrow{\times} a$$

or even

$$\texttt{select ::}\ \ \{\llparenthesis l : a \rrparenthesis\}^{\lceil a \rceil} \xrightarrow{\times} a$$

An in-depth examination of which of these operators are essential and of their properties is future work; in particular, their interaction with the existence of principal types and the decidability of type inference must be carefully studied.

8.2.4 Syntactic sugar

When we compared uniqueness typing to the monadic approach to side effects in Section 2.8.7, we p. 66 mentioned that the most important objection to uniqueness typing is that programs (in particular, types) become more cluttered. We feel that some of the techniques we propose in this thesis reduce this problem to some extent:

- The use of type variables of kind $\star$ allows us to write

  $$\texttt{id ::}\ \ a \xrightarrow{\times} a$$

  instead of

  $$\texttt{id ::}\ \ t^{u} \xrightarrow{\times} t^{u}$$

  Unfortunately, the fact that we must often know the attribute on a type limits the applicability of this technique. For example, we must write `const ::` $t^u \xrightarrow{\times} b \xrightarrow{u} t^u$

p. 210 Unless, that is, we adopt the approach outlined in 8.2.3 so that we can write

`const ::` $a \xrightarrow{\times} b \xrightarrow{\lceil a \rceil} a$

- Our suggestion of keeping terms polymorphic in their uniqueness as long as possible (rather than making them unique) allows us to write

  `apply ::` $(a \xrightarrow{u_f} b) \xrightarrow{\times} a \xrightarrow{u_f} b$

  instead of

  `apply ::` $(t^u \xrightarrow{u_f} s^v) \xrightarrow{\times} t^w \xrightarrow{u'_f} s^v, [w \leq u, u'_f \leq u_f]$

- As mentioned above, we feel that the approach using boolean attributes yields more readable types than the approach using uniqueness constraints; we believe that

  `swap ::` $(t^u, s^v)^{u \vee v} \xrightarrow{\times} (s^v, t^u)^w$

  is more readable than

  `swap ::` $(t^u, s^v)^{u_p} \xrightarrow{\times} (s^v, t^u)^{u'_p}, [u_p \leq u, u_p \leq v]$

- Support for synonyms for types of kind $\star$ can also be used to lighten the burden. For example, we can define a `Unit` type as a non-unique empty record:

  type `Unit =` $\{(\!|\,|\!)\}^{\times}$

  Likewise, we might define synonyms for non-unique versions of often used data structures such as lists:

  type `List t =` $[t^{\times}]^{\times}$

  Such synonyms can then be used to simplify function signatures. Note that this `List` synonym has kind $\mathcal{T} \rightarrow \star$. Clean only supports synonyms of kind $\star \rightarrow \ldots \rightarrow \star \rightarrow \mathcal{T}$; the additional expressive power we offer follows very naturally from the "attributes are types" approach (Section 6.1). p. 143

Even with these improvements the uniqueness type system still permeates the language, affecting nearly every type: even the identity function has a different type than in the standard Hindley/Milner style type systems. Syntactic conventions can be used to alleviate the problem, but we must be careful that syntactic sugar does not become syntactic heroin (Bates, 2005). Although judicious use of syntactic sugar may hide the uniqueness properties of a function, one may wonder whether that is desirable: after all, the uniqueness properties are important. Worse, types with *some* of their uniqueness properties hidden (for example, when some inequality constraints are "implied" and therefore not shown) may confuse the non-expert user.

With that proviso, however, some syntactic sugar is undoubtedly useful. Clean for example allows type annotations without inequality constraints, which are inferred. Unfortunately, in a qualified types framework this is difficult to formalize since there are no "constraint variables" (we discussed a possible type system with constraints that did use constraint variables in Chapter 4). p. 115

The situation improves with a type system based on boolean expressions. The key ingredient is the use of partial type annotations by means of the "some" binder (Leijen, 2008a; Rémy, 2005), denoted by $\exists$. For example, we can write the type of the identity function as

$$\texttt{id ::}\ \exists(u : \mathcal{U}) \cdot \forall(a : \star) \cdot a \xrightarrow{u} a$$

These binders are read as "for some u, and for all a". Note that the $\exists$ binder is not an existential binder: we are not trying to hide the attribute on the arrow, but merely want to express that there is "some" attribute without having to be specific about what it is. The syntactic sugar we can now add is that we allow users to omit attributes from types; missing attributes will be interpreted as fresh "some"-bound variables. Thus, the annotation

```
id :: a → a
```

is equivalent to the previous annotation with the explicit $\exists$ binder. If we additionally introduce an object-level ceiling operator, this syntactic sugar will allow us to write many types without any uniqueness annotations. For instance, we could write

$$\texttt{const ::}\ a \rightarrow b \rightarrow a$$

which would be syntactic sugar for the partial type annotation

$$\texttt{const ::}\ \exists u v \cdot a \xrightarrow{u} b \xrightarrow{v} a$$

and can be completed by the type inferencer to

$$\texttt{const ::}\ a \xrightarrow{\times} b \xrightarrow{\lceil a \rceil} a$$

As an aside, note that the annotation

```
const :: a → b → a
```

is allowed in Clean, too (and frequently used), but denotes a very different type: Clean does not have a special syntax to denote "non-unique"; instead, this is denoted by the absence of an attribute. Hence, this type of `const` denotes the type

$$\texttt{const ::}\ a^{\times} \xrightarrow{\times} b^{\times} \xrightarrow{\times} a^{\times}$$

Other useful syntactic sugar we might introduce is to allow the use of a type σ of kind $\star$ where a type of kind $\mathcal{U}$ is expected and treat that use as an implicit application of the ceiling operator, $\lceil \sigma \rceil$; this allows us to write

$$\texttt{const ::}\ a \xrightarrow{\times} b \xrightarrow{a} a$$

to denote the same type as the one above. Similarly, we might allow the use of a type σ of kind $\star$ where a type of kind $\mathcal{T}$ is expected and treat that use as an implicit application of the floor operator, $\lfloor \sigma \rfloor$; this allows us to write

$$\texttt{dup ::}\ a \xrightarrow{\times} (a^{\times}, a^{\times})^{u}$$

Together with the convention that missing attributes are interpreted as fresh type variables bound by an implicit "some" quantifier, this allows us to write down those aspects of types that are interesting and leave out the rest:

$$\begin{aligned} &\texttt{const ::}\ a \rightarrow b \xrightarrow{a} a \\ &\texttt{dup}\ \ \ \ \texttt{::}\ a \rightarrow (a^{\times}, a^{\times}) \end{aligned}$$

We must remind the reader that the definition of these two operators is not without difficulty and should be studied before more pragmatic matters are dealt with. Moreover, these conventions will complicate kind inference: for instance, it is not clear whether

$$\texttt{dup ::}\ a \rightarrow (a^{\times}, a^{\times})$$

means

$$\texttt{dup :: } \exists v w \cdot \forall (a : \star) \cdot a \xrightarrow{v} (\lfloor a \rfloor^{\times}, \lfloor a \rfloor^{\times})^{w}$$

or

$$\texttt{dup :: } \exists u v w \cdot \forall (a : \mathcal{T}) \cdot a^{u} \xrightarrow{v} (a^{\times}, a^{\times})^{w}$$

p. 210 although in light of Section 8.2.3 it should be clear that the first interpretation is to be preferred.

8.2.5 Integration in Clean

We have proposed various changes to the Clean type system in this thesis. Some of these changes will not cause many problems with backwards compatibility, whereas others are more difficult to integrate into Clean without sacrificing the large "legacy" code base. We will consider each change in turn:

Boolean attributes instead of inequality constraints When no type annotation is provided for an expression, this will make no difference. When a type annotation with inequality constraints is given, however, we can easily interpret the inequality constraints as syntactic sugar for boolean expressions (Section 5.1.5). p. 132

No subtyping Clean distinguishes between objects that are non-unique, unique "now" but that may become non-unique later, and "necessarily" unique objects that are unique now and must remain unique (Section 3.2, especially 3.2.4). p. 78, p. 82 We have proposed to distinguish only between terms that are non-unique and terms that are necessarily unique, but to leave terms polymorphic in their uniqueness whenever possible. This is a more radical departure from the original type system, and consequently more difficult to integrate in a backwards compatible manner. Some of the difficulties can be resolved by interpreting type annotations differently. For example, one might treat a unique (but not necessarily unique!) type in the codomain of a function as it if were polymorphic, so that we interpret

$$\texttt{resetArray :: Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{\bullet}$$

as

$$\texttt{resetArray :: Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{u}$$

Not all problems can be resolved in this fashion since there are types that are valid in Clean but not valid in our system, such as

$$\texttt{dupArray :: \{Char\}}^{\bullet} \rightarrow \texttt{(\{Char\}}^{\times}\texttt{, \{Char\}}^{\times}\texttt{)}$$
```
dupArray arr = (arr, arr)
```

Fortunately such types are rare, especially since the subtyping relation is non-uniform so that type variables cannot loose their uniqueness. For instance, the following type is invalid in both Clean and in our proposal:

```
// Illegal example
```
$$\texttt{dupAny :: t}^{\bullet} \rightarrow \texttt{(t}^{\times}\texttt{, t}^{\times}\texttt{)}$$
```
dupAny x = (x, x)
```

Attributes as types We have proposed to treat attributes as types and use a kind system to distinguish between the two. Although this will of course require re-engineering (indeed, perhaps simplifying) parts of the Clean type checker, we do not foresee any problems with backwards compatibility due to this modification. (Of course, programs that take advantage of the additional expressive power provided by treating attributes as types of kind $\mathcal{U}$ will not be compatible with the original type system.)

Higher rank types *Support* for higher rank types will obviously not cause any difficulties with backwards compatibility. However, we sometimes *require* higher rank types if a term is used at two different uniqueness levels (Section 6.3). For example: p. 147

$$\texttt{f} \;::\; \forall v. (\forall u. \texttt{Array}^u) \xrightarrow{\times} \texttt{Array}^v$$
$$\texttt{f arr} = \textbf{if}\ \texttt{isEmpty arr}^{\otimes}\ \textbf{then}\ \texttt{shrink arr}^{\odot}\ \textbf{else}\ \texttt{grow arr}^{\odot}$$

If no type annotation was given in the original code, backwards compatibility depends on the higher rank type system that is adopted. If this is PTI (Section 2.5.3) or HMF p. 46 (Section 2.5.4) then a type annotation *must* be given for this definition, as higher rank types p. 51 will never be inferred. However, since type inference for rank-2 types is decidable (Kfoury and Wells, 1994; Lushman, 2007) (albeit at the cost of losing principal types), it may be possible to modify the type system so that the type of f, above, can be inferred, in which case backwards compatibility can be retained.

However, it may be that the user has provided a type annotation such as

$$\texttt{f} \;::\; \texttt{Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^v$$

or probably even

$$\texttt{f} \;::\; \texttt{Array}^{\bullet} \xrightarrow{\times} \texttt{Array}^{\bullet}$$

This is a valid annotation in Clean, since $\texttt{Array}^{\bullet}$ is a subtype of $\texttt{Array}^{\times}$. It is *not* a valid annotation in our system. In this case, it may be necessary to ignore a given type annotation and infer the correct type instead. This is not an elegant solution, but it may help to integrate a new type system without sacrificing legacy code.

Apart from the changes to the core type system, there are also some areas of the type system that we have ignored completely in this thesis. The two most important ones are type classes and polytypic programming. We will briefly consider these in turn.

Type classes The difficulty with type classes is that uniqueness propagation must somehow be guessed from the type of the class members (rather than from their definition). The example that is given in (Plasmeijer and van Eekelen, 2002, Section 9.5, *Combining uniqueness typing and overloading*) is the class

$$\textbf{class}\ \texttt{Functor f}\ \textbf{where}$$
$$\quad \texttt{fmap} \;::\; (t^u \to s^v) \to f\ t^u \to f\ s^v$$

(We have modified the notation somewhat to be in line with the notation used elsewhere in this thesis.) The type that is specified for `fmap` is underspecified (even ill-kinded): the attribute on f is missing. The problem is that this attribute depends on the particular f. For example, the `Functor` instance for lists has type

$\texttt{fmap}_{[]}$ :: $(t^u \to s^v) \to [t^u]^w \to [s^v]^{w'}, [w \leq u, w' \leq v]$

But if we define a data type T as follows:

```
:: T a = MkT .(Int → a)
```

then the corresponding `Functor` instance has type

$\texttt{fmap}_{\texttt{T}}$ :: $(t^u \xrightarrow{u_f} s^v) \to T^w\, t^u \to T^{w'}\, s^v, [w' \leq u_f, w' \leq w]$

In the first case, the arguments t^u and s^v that are supplied to f describe the type of the elements of the list and thus the uniqueness of f is constrained by u and v (recall p. 206 from Section 8.1.4 that Clean enforces uniqueness propagation in both constructors and destructors). In the second example t^u and s^v do not describe the type of the elements in the container f but rather give the codomain of the function that is stored in a `T` object. In this case, the type of f is constrained by the uniqueness of that function[1] but is not at all related to t^u or s^v.

This is a complicated part of uniqueness typing, and a full description is even "far beyond the scope of the reference manual" (Plasmeijer and van Eekelen, 2002, Section 9.6)—though one might wonder about the right scope of that description, if not the reference manual! The bad news is that most of this complexity will remain even after the simplifications we propose. The good news is that *some* of the complexity will disappear: since we suggest to abandon subtyping, covariance/contravariance behaviour of constructors no longer needs to be taken into account.

Polytypic programming The current implementation of polytypic programming in Clean (Alimarine, 2005) has rudimentary support for uniqueness typing. However, since polytypic programming relies essentially on higher rank types (Hinze, 2000, especially Section 3.1.3, *Specializing generic values*, `encodeFMapForm` example on page 62) but Clean does not currently support higher rank types, full support for uniqueness has so far been impossible. Indeed, this was the original motivation for our work. However, even with a uniqueness system that supports higher rank types support for polytypic programming is not trivial. For example, the type of a polytypic function in n arguments is usually defined to be a type-level function of kind $\underbrace{\star \to \star \to \star}_{n} \to \star$ (Verbruggen et al., 2008). It is not obvious how this kind should be modified in a uniqueness type system, but at least the kind system we propose in this thesis will make it easier to discuss the possibilities. Other difficulties include attribution during type specialization and the uniqueness properties of the bimaps. An in-depth study of this problem is future work.

8.2.6 Embedding affine logic

In both affine logic and uniqueness typing, terms of a unique (affine) type cannot be duplicated. That is, in both systems we want to reject the function

`dup_unique ::` $t^\bullet \xrightarrow{\times} (t^\bullet, t^\bullet)^u$
`dup_unique x = (x, x)`

[1] The uniqueness of the function stored inside the `T` object must be the same as the uniqueness of the `T` object itself (as indicated by the dot; Section 5.3). The function in the new `T` object is the composition of the original function (with uniqueness w) and the argument to `fmap` (with uniqueness u_f). If either of these two functions is unique, the new function and therefore the new `T` object must be unique. Hence the constraint $[w' \leq u_f, w' \leq w]$.

However, uniqueness typing and affine logic are "dual" in their subtyping relation. In uniqueness typing, the following term is type-correct:

```
coerce_unique :: $t^\bullet \xrightarrow{\times} t^\times$
coerce_unique x = x
```

(provided we deal with partial application); in affine logic, `coerce_unique` is ill-typed but

```
coerce_affine :: $t^\times \xrightarrow{\times} t^\bullet$
coerce_affine x = x
```

is correct. We argued in Section 8.1.2 that we do not need to include a specific subtyping relation p. 203 in the type system as long as we are careful in the types we assign to primitive functions. It should be possible to use a similar encoding for affine logic. For instance, suppose we can buy a pizza with one bank note. This is encoded in affine logic as

```
buy_pizza :: Note$^\bullet \xrightarrow{\times}$ Pizza$^\bullet$
```

We should also be able to buy a pizza given an infinite supply of bank notes:

```
bank :: Note$^\times$
```

To be able to typecheck the application (`buy_pizza bank`), we need the coercion from a non-affine type to a affine type. In the absence of such a coercion we can still type the application if we give `buy_pizza` a polymorphic type:

```
buy_pizza :: Note$^u \xrightarrow{\times}$ Pizza$^\bullet$
```

The duality between affine logic and uniqueness is again evident from this example: in uniqueness typing, we needed make to the *codomain* of functions (such as `resetArray`) more polymorphic while in affine logic we needed to make the *domain* of functions more polymorphic.

This has an intuitive semantic explanation too: when a function returns a unique object, it returns an object with a guarantee that that object is not shared. The function does not care whether or not this guarantee is useful, so we can modify the function to return a term with a polymorphic uniqueness instead. When a function requires a unique object as input, however, it relies on that guarantee and so it really requires a unique object. In affine logic the situation is reversed. When a function demands a bounded resource (a single bank note) it does not matter if it passed an unbounded resource instead (the whole bank), and so we can make the domain of the function polymorphic in its affinity. However, when it returns a bounded resource, then that resource cannot be interpreted as an unbounded resource and therefore it really must return a affine result.

It thus seems that we can unify affine logic and uniqueness typing as a single language, simply by taking advantage of polymorphism. A more formal investigation into an embedding from both uniqueness typing with explicit coercions and affine logic with explicit coercions into a single language without coercions could therefore be very interesting.

8.2.7 Formalization

Although we have given a formal specification of the core type system and the associated formal soundness proof in Chapter 7, a full formalization of the type system should be based on an p. 159 extension of the core logic to a higher rank type system such as PTI or HMF, and should include proofs of principal types, correctness of type inference (decidability, soundness, completeness), conservativity (with respect to the underlying type system without support for uniqueness), etc.

Fully formal (where we use "formal" strictly in the sense of "machine verified") proofs of these properties are very labour intensive; proof engineering is an area of computer science that only recently started to mature. The work by Aydemir et al. (2008) has been extremely helpful in the development of our proofs, and it is only through the development of proof techniques and proof libraries that full formal proofs of non-trivial type systems become feasible.

A fully formal proof does give strong guarantees: one has only to trust the definition of the type system and the statement of the theorems one is interested in and the type checker will verify that the proof is correct. Moreover, a proof in a constructive logic such as Coq has the additional benefit that a formal definition of a type inference algorithm (say) can be extracted automatically to compilable code (for example, in Haskell) so that there is no gap between the formalization and the implementation.[1] Moreover, although constructive logic did force us to be very precise about some aspects of the proof (for example, we had to be very precise about the nature of a typing environment, rather than simply treating it as a set) we certainly did not p. 17 feel "completely paralysed" with Boolos (Section 2.1.1). In fact, we felt very comfortable with constructive logic—it matches a computer scientist's way of thinking!

Although we could potentially have given informal proofs of these other properties, we are not convinced that an informal ("pencil-and-paper") proof provides much more insight into a type system than an implementation. In both cases one must consider all corner cases, and in both cases one is likely to make assumptions that will later turn out to be incorrect—during the development of the formal proof of soundness it happened on numerous occasions that we temporarily assumed an "obvious" lemma so that we could complete another theorem, only to realize later that the assumed was in fact not true. In a pencil-and-paper proof those lemmas would probably not be considered at all. Moreover, the implementation has the important advantage over the pencil-and-paper proof that it is easier to experiment with; one is therefore more likely to discover errors in an implementation than to discover errors in an informal proof. In addition, when the type system evolves (as it inevitably will), bugs in an implementation are likely to be discovered, but bugs in an informal proof are likely to be missed unless one is extremely careful when rechecking each and every lemma. In a fully formal proof on the other hand the proof checker will automatically verify that our lemmas still hold when we make a modification, and will tell us exactly which lemmas need to be updated.

Finally, although it is not at all obvious that soundness holds for a uniqueness type system (after all, it is very sensitive to loss of sharing), other properties such as principal types and decidability of type inference are mostly inherited from the base system. This is true in particular for the type system with boolean attributes instead of inequality constraints: since boolean unification p. 25 is unitary (we proved this in Section 2.2.4), unification with boolean attributes is guaranteed to return most general unifiers, a key ingredient when inferring principal types.

8.2.8 Purity

We discussed the definition of "purity" and related properties such as referential transparency p. 62 and definiteness in Section 2.8.4, and mentioned that a more formal treatment of purity and substructural logic (in particular, uniqueness typing) is necessary. In this section we briefly discuss two ways in which we might approach this problem.

[1] One may object that a formal definition of type inference is unlikely to be efficient, and that may be true. On the other hand, one could argue that a more efficient inference algorithm should *also* be proven correct formally.

One option is to adopt the method described by Guzmán and Hudak (1990), who start with a pure lambda calculus with an associated non-deterministic (graph-rewriting) operational semantics and prove that the order of evaluation is irrelevant in this language (they prove that the language is confluent). They then show that confluence is lost when side effects are added: once side effects are added to the language, we can write programs for which one order of evaluation gives a different result than another. Finally, they show that when their type system (the single-threaded polymorphic lambda calculus) is imposed on top of this language, the non-confluent programs are considered ill-typed so confluence is recovered for well-typed programs.

As we have seen in Section 2.8.3, Sabry (1998) argues that confluence is not a strong enough p. 62 property to guarantee purity. An alternative approach might therefore be to adopt his definition of purity (essentially, equivalence of the call-by-name, call-by-value and call-by-need semantics for the language). To do so, we will need to introduce a notion of "typed" rewriting: we can only hope to prove equivalence of these semantics if we require that evaluation maintains type correctness at every step (we discussed this in more detail in Section 2.8.3).

8.2.9 Observer types

We mentioned the problem of allowing multiple reads before a write in Section 3.2.6. Adam p. 83 Megacz has suggested (Megacz, 2007) that the boolean attribute approach may be naturally extended to support Odersky's observer types (Section 3.3.3) by adding a second attribute to terms p. 90 to indicate whether the term is "observed".

This corresponds to the "read-only" attribute in Guzmán's single-threaded polymorphic lambda calculus (Section 3.5), although Guzmán (like Odersky) then tries to combine the read- p. 95 only attribute with the "multi-threaded" (uniqueness) attribute. If we want to use the boolean attribute approach, it is better to leave the two attributes separate; for example, an "observed" non-unique array might have type

`arr ::` $\mathit{Array}^{\times,\bullet}$

This additional attribute is "ignored" (left free) by most of the typing rules. For example, the identity function might get the type

`id ::` $t^{u,v} \xrightarrow{\times,w} t^{u,v}$

Only in the strict-let expression would the observer attribute of a term be set to true (•); we can then easily check whether any terms "escape" the strict-let expression by attempting to unify the observer attribute of the types of variables bound in the strict-let with false (×). (We must make sure that the observer property is propagated outwards by constructors for algebraic data types.)

This provides a more principled and more general (more permissive) solution than the one currently adopted in Clean. For example, we might have a function that copies an array:

`copyArray ::` $\texttt{Array}^{u,u'} \xrightarrow{\times,w} \texttt{Array}^{v,v'}$

With the new approach, we can now (correctly!) define

`dupArray ::` $\texttt{Array}^{\bullet,u} \xrightarrow{\times,w} (\texttt{Array}^{\bullet,v}, \texttt{Array}^{\bullet,v'})^{u_p}$

```
dupArray arr #! copy = copyArray arr
             = (arr, copy)
```

while

```
// invalid definition
dupArray' :: Array$^{\bullet,u}$ $\xrightarrow{\times,w}$ (Array$^{\bullet,v}$,Array$^{\times,v'}$)$^{u_p}$
dupArray' arr #! copy = id arr
                  = (arr, copy)
```

would still be rejected: we can detect that `arr` escapes since the type of `arr` within the strict-let is `Array`$^{\times,\bullet}$.

A potential alternative way to distinguish between these two examples was suggested by John van Groningen (personal communication): we might use the information that `copy` (but not `id`) can return a unique array even when given a non-unique array to accept the first example but reject the second. This analysis might become more difficult to execute however when the bodies of the definitions in the strict-let are more complex.

Although the additional attribute on types complicates the type system and the types, the second attribute on types could always be hidden from the user. Even when a type error arises because of a unification failure on the observer attribute, an error message such as "observed term escaping in strict-let on line *xxx*" would probably suffice. The advantage of this approach is that it is simple to define and fits in well with the rest of the type system.

8.2.10 Improving error messages

An often heard complaint about Clean is that its type checker generates error messages that are hard to understand. Since we have made uniqueness typing less unique, research on improved error messages for Hindley/Milner type systems (such as Heeren, 2005) can probably be incorporated into a uniqueness type checker. On the other hand, the use of boolean attributes may benefit from
p. 138 a special treatment. For example, if we reconsider the example from Section 5.2.4,

$$f :: (\forall u\, v . t^u \xrightarrow[u_c]{u_f} s^v) \to \ldots$$

$$g :: t^{u \vee v} \xrightarrow[u_c]{u_f} s^v$$

if we attempt to apply f to g, our type checker will complain

```
Types do not match
  context       : f g
  term          : g
  inferred type : $\forall u\, v \cdot t^{u \vee v} \xrightarrow{\times} t^v$
  does not match: $\forall u\, v \cdot t^u \xrightarrow{\times} t^v$
```

Perhaps it would be good if the type checker would explain *why* these types do not match. For example, it could tell us that the first type cannot be instantiated to $(t^\times \xrightarrow{\times} t^\bullet)$, whereas the second one can.

In addition, it might be helpful if the type checker gives hints about *why* a term is inferred to be non-unique (for instance, when a variable is used twice in its scope) or unique (for instance, when a function is partially applied and the supplied argument is unique).

Other scope for improvement will become clearer once a serious implementation of the new type system is attempted, and it is used for non-toy examples.

8.2.11 Dependent types

There has been surprisingly little research into substructural dependent type systems or logics. In the type systems community, there are two notable exceptions:

- LF (Harper et al., 1993; Harper and Licata, 2007) is a dependently typed programming language (equivalent to system λP in the Barendregt Cube, Figure 2.6, or first-order p. 30 predicate logic). Although LF itself is a logic, it was designed as a (meta) logic for formalizing other (object) logics; it is however less suitable to formalizing substructural logics. Both Linear LF (Cervesato and Pfenning, 2002) and Relevant LF[1] (Ishtiaq and Pym, 1998) are attempts to adapt LF so that it uses a linear version of λP instead.

- The logic of bunched implications (Section 3.6.4) has a predicate version (Pym, 1999; p. 112 O'Hearn and Pym, 1999).

Although there is some work on substructural predicate logic (Grišin, 1982; Došen, 1992; Bellin and Ketonen, 1992; Brünnler et al., 2008), there is much less than one might expect (a discussion of these papers, and of LLF/RLF and bunched predicate logic, is beyond the scope of this thesis).

It seems that the development of a substructural dependent type system might yield substantial benefits. A dependent uniqueness type system can be used as a basis for a pure functional programming language that uses uniqueness typing to support side effects without losing referential transparency, but also supports the proofs-as-programs methodology supported by systems such as Coq (Section 2.3.5). Obviously, this would be a major undertaking and might form the basis p. 34 for a Ph.D. or postdoctoral research project.

8.3 Coda

Uniqueness typing is not as well-known in the functional programming community as it deserves to be; we feel that this is partly due to the origin in and emphasis on graph rewriting in the original research papers on the topic. In this thesis, we have shown how to make uniqueness typing less unique: by allowing boolean attributes as uniqueness attributes we can give a formalization of uniqueness typing that is so similar to the conventional Hindley/Milner style type systems that modern extensions such as higher rank types or GADTs can be incorporated without difficulty. Treating uniqueness attributes as types of a special kind facilitates retrofitting uniqueness typing to existing compilers, allows for a more elegant formalization of the type system and gives additional expressive power virtually for free. Removing subtyping but making sure that terms are never more unique than they need to be means that we do not to give partial application any special treatment, and makes it possible to simplify many types. Finally, we have shown that the type system is sound with respect to the standard call-by-need semantics, which is the most accurate semantics for more conventional programming languages such as Haskell too. We have surveyed related work, discussed some of the advantages and disadvantages of our design decisions, and identified future work.

[1] The name "Relevant LF" is confusing: the authors use "relevant" to describe a logic where both contraction and weakening are restricted (that is, a linear logic) rather than a logic where weakening is restricted but contraction is permitted (the usual meaning of "relevant", Section 2.6).

Boolean algebra

A.1 Boolean algebra

This formalization is based on the second chapter ("The self-dual system of axioms") in Goodstein's book "Boolean Algebra" (Goodstein, 2007).

A.1.1 Abstraction over the structure of terms

```
Module Type BooleanAlgebraTerm.

Parameter trm : Set.
Parameter true : trm.
Parameter false : trm.
Parameter or : trm → trm → trm.
Parameter and : trm → trm → trm.
Parameter not : trm → trm.

End BooleanAlgebraTerm.
```

A.1.2 Huntington's postulates

```
Module BooleanAlgebra (Term : BooleanAlgebraTerm).
Import Term.

Inductive equiv : trm → trm → Prop :=
   (** Commutativity *)
  | comm_or : ∀ (a b:trm), equiv (or a b) (or b a)
  | comm_and : ∀ (a b:trm), equiv (and a b) (and b a)
   (** Distributivity *)
  | distr_or : ∀ (a b c:trm), equiv (or a (and b c)) (and (or a b) (or a c))
  | distr_and : ∀ (a b c:trm), equiv (and a (or b c)) (or (and a b) (and a c))
   (** Identities *)
  | id_or : ∀ (a:trm), equiv (or a false) a
  | id_and : ∀ (a:trm), equiv (and a true) a
   (** Complements *)
  | compl_or : ∀ (a:trm), equiv (or a (not a)) true
  | compl_and : ∀ (a:trm), equiv (and a (not a)) false
  (** Closure *)
| clos_not : ∀ (a b:trm), equiv a b → equiv (not a) (not b)
| clos_or : ∀ (a b c:trm), equiv a b → equiv (or a c) (or b c)
| clos_and : ∀ (a b c:trm), equiv a b → equiv (and a c) (and b c)
  (** Structural rules *)
| refl : ∀ (a:trm), equiv a a
| sym : ∀ (a b:trm), equiv a b → equiv b a
| trans : ∀ (a b c:trm), equiv a b → equiv b c → equiv a c.
```

A.1.3 Setup for Coq setoids

Thanks to Adam Megacz.

```
Add Relation trm equiv
  reflexivity proved by refl
  symmetry proved by sym
  transitivity proved by trans
  as equiv_relation.

Add Morphism or
  with signature equiv ==> equiv ==> equiv
  as or_morphism.

Add Morphism and
  with signature equiv ==> equiv ==> equiv
  as and_morphism.

Add Morphism not
  with signature equiv ==> equiv
  as not_morphism.
```

A.1.4 Derived Properties

```
Lemma false_unique : ∀ (x:trm), (∀ (a:trm), equiv (or a x) a) → equiv false x.

Lemma true_unique : ∀ (y:trm), (∀ (a:trm), equiv (and a y) a) → equiv true y.

Lemma complement_unique : ∀ (a a' a":trm),
  (** if a' has the property of the complement *)
  equiv (or a a') true → equiv (and a a') false →
  (** and so does a" *)
  equiv (or a a") true → equiv (and a a") false →
  (** then a' and a" must be equivalent *)
  equiv a' a".

Lemma involution : ∀ (a:trm), equiv (not (not a)) a.

Lemma true_compl_false : equiv false (not true).

Lemma false_compl_true : equiv (not false) true.
```

Lemma *zero_or* : ∀ (*a:trm*), *equiv* (*or a true*) *true*.

Lemma *zero_and* : ∀ (*a:trm*), *equiv* (*and a false*) *false*.

Lemma *idem_or* : ∀ (*a:trm*), *equiv a* (*or a a*).

Lemma *idem_and* : ∀ (*a:trm*), *equiv a* (*and a a*).

Lemma *abs_or* : ∀ (*a b:trm*), *equiv* (*or a* (*and a b*)) *a*.

Lemma *abs_and* : ∀ (*a b:trm*), *equiv* (*and a* (*or a b*)) *a*.

Lemma *equiv_or_and3* : ∀ (*a b c:trm*),
equiv (*or a b*) (*or a c*) → *equiv* (*and a b*) (*and a c*) → *equiv b c*.

Lemma *equiv_or_not* : ∀ (*a b c:trm*),
equiv (*or a b*) (*or a c*) → *equiv* (*or* (*not a*) *b*) (*or* (*not a*) *c*) → *equiv b c*.

Lemma *equiv_and_not* : ∀ (*a b c:trm*),
equiv (*and a b*) (*and a c*) → *equiv* (*and* (*not a*) *b*) (*and* (*not a*) *c*) → *equiv b c*.

Lemma *assoc_or* : ∀ (*a b c:trm*), *equiv* (*or a* (*or b c*)) (*or* (*or a b*) *c*).

Lemma *assoc_and* : ∀ (*a b c:trm*), *equiv* (*and a* (*and b c*)) (*and* (*and a b*) *c*).

Lemma *equiv_or_and2* : ∀ (*a b:trm*), *equiv* (*or a b*) (*and a b*) → *equiv a b*.

Lemma *DeMorgan_or* : ∀ (*a b:trm*), *equiv* (*not* (*or a b*)) (*and* (*not a*) (*not b*)).

Lemma *DeMorgan_and* : ∀ (*a b:trm*), *equiv* (*not* (*and a b*)) (*or* (*not a*) (*not b*)).

A.1.5 "Non-standard" properties (not proven in Goodstein)

Lemma *abs_or_or* : ∀ (*a b:trm*), *equiv* (*or* (*or a b*) *a*) (*or a b*).

Lemma *abs_and_and* : ∀ (*a b:trm*), *equiv* (*and* (*and a b*) *a*) (*and a b*).

Lemma *distr_or_or* : ∀ *a b c*, *equiv* (*or a* (*or b c*)) (*or* (*or a b*) (*or a c*)).

Lemma *distr_and_and* : ∀ *a b c*, *equiv* (*and a* (*and b c*)) (*and* (*and a b*) (*and a c*)).

Lemma *or_false_left* : ∀ (*a b:trm*), *equiv* (*or a b*) *false* → *equiv a false*.

Lemma *or_false_right* : ∀ (*a b:trm*), *equiv* (*or a b*) *false* → *equiv b false*.

Lemma *or_false_both* : ∀ (*a b:trm*),
equiv (*or a b*) *false* → *equiv a false* ∧ *equiv b false*.

Lemma *and_true_left* : ∀ (*a b:trm*), *equiv* (*and a b*) *true* → *equiv a true*.

Lemma *and_true_right* : ∀ (*a b:trm*), *equiv* (*and a b*) *true* → *equiv b true*.

Lemma *and_true_both* : ∀ (*a b:trm*),
equiv (*and a b*) *true* → *equiv a true* ∧ *equiv b true*.

A.1.6 Conditional

Definition *ifbool* (*b P Q*:*trm*) : *trm* := *or* (*and b P*) (*and* (*not b*) *Q*).

Lemma *if_ident_branch* : ∀ (*b P*:*trm*),
  equiv (*ifbool b P P*) *P*.

Lemma *distr_or_if* : ∀ (*b P Q R*:*trm*),
  equiv (*or* (*ifbool b P Q*) *R*) (*ifbool b* (*or P R*) (*or Q R*)).

Lemma *distr_or_if2* : ∀ (*b P Q*:*trm*),
  equiv (*ifbool b P Q*) (*or* (*ifbool b P Q*) (*and P Q*)).

Lemma *distr_and_if* : ∀ (*b P Q R*:*trm*),
  equiv (*and* (*ifbool b P Q*) *R*) (*ifbool b* (*and P R*) (*and Q R*)).

Lemma *distr_not_if* : ∀ (*b P Q*:*trm*),
  equiv (*not* (*ifbool b P Q*)) (*ifbool b* (*not P*) (*not Q*)).

End *BooleanAlgebra*.

Bibliography

Journal and book series titles are listed in full, and *Lecture Notes in Computer Science* volume numbers are always quoted (where applicable). We feel that publishers' addresses are no longer useful today and do not list them. Conference papers do not include the full details of the conference proceedings, but simply mention the name of the conference; details for the conference proceedings are listed separately (p. 242). Page numbers are listed for journal articles and conference proceedings, but not for electronic or informal proceedings or for articles that are yet to appear. Every entry concludes with a list of pages where the entry is cited in parentheses, so that the bibliography can be used for "reverse look up".

Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1-2):3–57, 1993. (p. 89)

P. M. Achten and M. J. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995. (pp. 13 and 69)

Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In ICFP05, pages 78–91. (p. 77)

Artem Alimarine. *Generic Functional Programming: Conceptual Design, Implementation and Applications*. PhD thesis, Radboud Universiteit Nijmegen, 2005. (pp. 15 and 218)

Thorsten Altenkirch and Nicolas Oury. ΠΣ: A core language for dependently typed programming. Unpublished manuscript, 2008. (p. 32)

Pablo Armelín. *Programming with Bunched Implications*. PhD thesis, Queen Mary, University of London, 2002. (p. 112)

David Aspinall and Martin Hofmann. Another type system for in-place update. In ESOP02, pages 36–52. (p. 105)

David Aspinall, Martin Hofmann, and Michal Konečný. A type system with usage aspects. *Journal of Functional Programming*, 18(2):141–178, 2008. (p. 105)

Steve Awodey. *Category Theory*. Oxford University Press, 2006. (p. 73)

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In POPL08, pages 3–15. (pp. 161, 162, and 220)

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. (pp. 22, 26, and 27)

Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In ICFP02, pages 157–166. (p. 60)

H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984. (p. 167)

Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991. (p. 29)

Henk Barendregt and Silvia Ghilezan. Lambda terms for natural deduction, sequent calculus and cut elimination. *Journal of Functional Programming*, 10(1):121–134, 2000. (p. 22)

Erik Barendsen and Sjaak Smetsers. A derivation system for uniqueness typing. In SEGRAGRA95, pages 151–158. (p. 78)

—. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996. (pp. 7, 78, 79, 82, 85, 126, 127, 128, 147, 148, 150, 154, 155, and 203)

—. Conventional and uniqueness typing in graph rewrite systems. Technical Report CSI-R9328, University of Nijmegen, December 1993a. (p. 126)

—. Conventional and uniqueness typing in graph rewrite systems. In FSTTCS93, pages 41–51. (p. 78)

—. Uniqueness type inference. In PHILPS95, pages 189–206. *Also known as "Uniqueness typing in Theory and Practice"*. (p. 78)

Michael Barr and Charles Wells. *Category Theory for Computing Science*. Les Publications CRM, third edition, 1999. (p. 73)

Gilles Barthe and Morten Heine Sørensen. Domain-free pure type systems. *Journal of Functional Programming*, 10(5):417–452, 2000. (p. 39)

Rodney Bates. Syntactic heroin. *Queue*, 3(5):64, 62–63, 2005. (p. 214)

Gianluigi Bellin and Jussi Ketonen. A decision procedure revisited: notes on direct logic, linear logic and its implementation. *Theoretical Computer Science*, 95(1):115–142, March 1992. (p. 223)

Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In LICS96, page 420. (p. 77)

Josh Berdine and Peter W. O'Hearn. Strong update, disposal, and encapsulation in bunched typing. In MFPS06, pages 81–98. Electronic Notes in Theoretical Computer Science, volume 158. (p. 112)

Stefan Berghofer and Christian Urban. Nominal inversion principles. In TPHOLs08. To appear. (p. 161)

Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development (Coq'Art: The Calculus of Inductive Constructions)*. Springer-Verlag, 2004. (p. 34)

G. M. Bierman. Program equivalence in a linear functional language. *Journal of Functional Programming*, 10(2):167–190, 2000. (p. 77)

G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In HOOTS00, pages 70–88. (p. 77)

Malgorzata Biernacka and Dariusz Biernacki. Formalizing constructions of abstract machines for functional languages in Coq. In WRS07pre. (p. 168)

Richard Bird, Geraint Jones, and Oege De Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5):541–547, 1997. (p. 56)

Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!* College Publications, 2006. Freely available online. (p. 108)

George Boole. *An Investigation of the Laws of Thought on which are founded the mathematical theories of logic and probabilities*. Macmillan, 1854. Republished by Dover, 1958. (pp. 25 and 27)

George Boolos. The hardest logic puzzle ever. *The Harvard Review of Philosophy*, 6:62–65, 1996. (p. 19)

Didier Le Botlan and Didier Rémy. ML^{F}: raising ML to the power of System F. In ICFP03, pages 27–38. (pp. 35, 41, 51, and 149)

John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, May 2001. (p. 77)

L. E. J. Brouwer. *Over de Grondslagen der Wiskunde*. PhD thesis, Universiteit van Amsterdam, 1907. (p. 19)

Frank Markham Brown. *Boolean Reasoning, The Logic of Boolean Equations*. Dover Publications, Inc., 2003. (p. 27)

Kai Brünnler, Dieter Probst, and Thomas Studer. On contraction and the modal fragment. *Mathematical Logic Quarterly*, 54(4):345–349, 2008. (p. 223)

Stanley N Burris. The laws of Boole's thought. Presented to the American Mathematical Society, January 2001. (p. 25)

Andrew Butterfield and Glenn Strong. Proving correctness of programs with IO—a paradigm comparison. In IFL01, pages 72–87. (p. 66)

Venanzio Capretta. General recursion via coinductive types. *Logical methods in computer science*, 1(2):1–28, July 2005. (p. 71)

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002. (pp. 154, 164, and 223)

Jorge Cham. Phd comics: Thesis topic, May 1998. `http://www.phdcomics.com/comics/archive.php?comicid=82`. (p. vii)

Arthur Charguéraud. Formal PL metatheory: Locally nameless developments (Coq development), 2007. `http://www.chargueraud.org/arthur/research/2007/binders`. (p. 162)

Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2): 346–366, April 1932. (p. 11)

Koen Claessen and David Sands. Observable sharing for functional circuit description. In ASIAN99, pages 62–73. (p. 63)

Stephen Cooper. *On linear types and imperative update*. PhD thesis, Syracuse University, 1997. (p. 90)

Coq Development Team. The Coq proof assistant—reference manual (version 8.1), 2006. (p. 35)

T. Coquand and H. Herbelin. A-translation and looping combinators in pure type systems. *Journal of Functional Programming*, 4:77–88, 1994. (p. 32)

Pierre Corbineau. A declarative language for the Coq proof assistant. In TYPES07, pages 69–84. (p. 160)

Olivier Coudert. Two-level logic minimization: an overview. *Integration, the VLSI Journal*, 17(2): 97–140, October 1994. (p. 150)

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In POPL82, pages 207–212. (pp. 41, 44, 45, and 134)

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In POPL06, pages 206–217. (p. 61)

Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers—Sparkle: A functional theorem prover. In IFL01, pages 55–71. (p. 61)

Edsko de Vries. Serious bug in the type system, December 2007. Message on the Clean mailing list. (p. 84)

—. Uniqueness typing simplified––technical appendix. Technical Report TCD-CS-2008-19, Department of Computer Science, Trinity College Dublin, 2008a. (p. 159)

—. Status of unique modes in Mercury?, July 2008b. Message on the Mercury users mailing list. (p. 110)

Edsko de Vries, Rinus Plasmeijer, and David M Abrahamson. Equality-based uniqueness typing. In TFP07pre. Technical report TR-SHU-CS-2007-04-1. (p. 129)

—. Uniqueness typing redefined. In IFL06, pages 181–198. (pp. 16, 87, and 115)

—. Uniqueness typing simplified. In IFL07, pages 181–198. (pp. 16 and 143)

Péter Diviánszky. Unique identifiers in pure functional languages. In TFP06pre. (p. 63)

Kosta Došen. Nonmodal classical linear predicate logic is a fragment of intuitionistic linear logic. *Theoretical Computer Science*, 102(1):207–214, 1992. (p. 223)

Malcolm Dowse, Glenn Strong, and Andrew Butterfield. Proving Make correct: I/O proofs in Haskell and Clean. In IFL02, pages 68–83. (p. 66)

Catherine Dubois. Proving ML type soundness within Coq. In THOLs00, pages 126–144. Published version is incorrect; corrected version available from the author's website. (p. 168)

Catherine Dubois and Valérie Ménissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3):319–346, November 1999. (p. 44)

Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In PLDI02, pages 13–24. (p. 77)

Matthew Fluet. *Monadic and Substructure Type Systems for Region-Based Memory Management*. PhD thesis, Cornell University, January 2007. (pp. 114 and 143)

You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In TAPSOFT89, pages 167–183. Volume 2 of the conference proceedings. (p. 203)

Jacques Garrigue. Relaxing the value restriction. In FLOPS04, pages 196–213. (pp. 114 and 117)

Benedict R. Gaster. *Records, variants and qualified types*. PhD thesis, University of Nottingham, July 1998. Technical report NOTTCS-TR-98-3. (p. 151)

Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, November 1996. (pp. vi and 151)

Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986. (pp. 36, 39, and 54)

—. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. (p. 87)

—. On the unity of logic. Technical Report 1467, Institut National de Recherche en Informatique et en Automatique (INRIA), June 1991. (p. 87)

Dina Goldin and Peter Wegner. The Church-Turing thesis: Breaking the myth. In CiE05, pages 152–168. (p. 13)

Dina Q. Goldin. Persistent turing machines as a model of interactive computation. In FoIKS00, pages 116–135. (p. 13)

R. L. Goodstein. *Boolean Algebra*. Dover Publications, Inc, 2007. Unabridged republicaton of the 1966 printing of the work originally published by Pergamon Press, London, in 1963. (p. 225)

Clemens Grelck and Sven-Bodo Scholz. Classes and objects as basis for I/O in SAC. In IFL95, pages 30–44. (p. 106)

V. N. Grišin. Predicate and set-theoretic calculi based on logic without contractions. *Mathematics of the USSR-Izvestiya*, 18(1):41–59, 1982. (p. 223)

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In PLDI02, pages 282–293. (p. 113)

J. C. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In LICS90, pages 333–343. (pp. 95 and 221)

Juan Carlos Guzmán. *On Expressing the Mutation of State in a Functional Programming Language*. PhD thesis, Yale University, 1993. (pp. 7, 95, 96, and 102)

Jurriaan Hage and Stefan Holdermans. Heap recycling for lazy languages. In PEPM08, pages 189–197. (pp. 87, 106, and 204)

Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In ICFP07, pages 235–246. (pp. 16, 85, 86, 87, 106, 201, 202, 204, 206, 211, and 212)

Theodore Hailperin. Boole's algebra isn't boolean algebra. *Mathematics Magazine*, 54(4): 173–184, September 1981. (p. 25)

Chris Hankin. *Lambda calculi: a guide for computer scientists.* Oxford Clarendon, 1994. (p. 12)

Robert Harper. A note on "a simplified account of polymorphic references". *Information Processing Letters*, 57(1):15–16, 1996. (p. 58)

—. A simplified account of polymorphic references. *Information Processing Letters*, 51(4): 201–206, 1994. (p. 58)

Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007. (p. 223)

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. (p. 223)

Dana Harrington. Uniqueness logic. *Theoretical Computer Science*, 354(1):24–41, 2006. (pp. 91, 92, 93, and 201)

Dana G. Harrington. A type system for destructive updates in declarative programming languages. Master's thesis, University of Calgary, 2001. (pp. 7, 88, 91, 92, and 205)

Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In ICFP07, pages 251–264. (p. 86)

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Science, July 2002. (p. 128)

Bastiaan Johannes Heeren. *Top Quality Type Error Messages.* PhD thesis, Universiteit Utrecht, 2005. (pp. 44 and 222)

Fergus Henderson. Strong modes can change the world!, November 1992. Honours Report, Department of Computer Science, University of Melbourne. (p. 109)

Fritz Henglein, Henning Makholm, and Henning Niss. Effect types and region-based memory management. In Pierce (2005). (pp. 113 and 114)

A. Heyting. *Intuitionism: An Introduction.* North-Holland Publishing Company, 1966. (p. 18)

R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969. (p. 41)

Ralf Hinze. Generic programs and proofs, October 2000. Habilitationsschrift, Universität Bonn. (pp. 15 and 218)

Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors. *Engineering theories of software construction.* IOS Press, 2001. (p. 238)

Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. (p. 105)

Douglas R. Hofstadter. *Gödel Escher Bach: An Eternal Golden Braid.* Basic Books, Inc., 1999. (p. 18)

Paul Hudak and Raman S. Sundaresh. On the expressiveness of purely functional I/O systems. Technical Report YALEU/DCS/RR-665, Department of Computer Science, Yale University, 1988. (p. 13)

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In HOPL07, pages 12–1–12–55. (pp. 13, 37, 68, and 202)

J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989. (p. 56)

E. V. Huntington. Sets of independent postulates for the algebra of logic. *Transactions of the American Mathematical Society*, 5:288–309, 1904. (pp. 25 and 159)

Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004. (p. 125)

S. S. Ishtiaq and D. J. Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998. (p. 223)

Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In TFP06, pages 73–90. (p. 12)

Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In FPCA93, pages 52–61. (pp. 32, 37, and 143)

—. *Qualified types: theory and practice.* Cambridge Distinguished Dissertations In Computer Science. Cambridge University Press, 1995a. (pp. 72, 202, and 203)

—. Simplifying and improving qualified types. In FPCA95, pages 160–169. (p. 72)

—. A theory of qualified types. *Science of Computer Programming*, 22(3):231–256, 1994. (p. 72)

Simon B. Jones. Experiences with Clean I/O. In FPWS95. (p. 69)

Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory—From its Origins until Today.* Kluwer Academic Publishers, 2004. (p. 18)

A. J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation*, 98(2):228–257, 1992. (p. 38)

A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In LFP94, pages 196–207. (pp. 149 and 217)

A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41 (2):368–398, 1994. (p. 150)

Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung chieh Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). In FLOPS08, pages 64–80. (p. 109)

Pieter Koopman and Rinus Plasmeijer. Generic generation of elements of types. In TFP05, pages 167–179. (p. 15)

Johannes John Carel Kuiper. *Ideas and Explorations: Brouwer's Road to Intuitionism.* PhD thesis, Universiteit Utrecht, 2004. (p. 19)

John Launchbury. A natural semantics for lazy evaluation. In POPL93, pages 144–154. (pp. 58 and 155)

Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4): 707–723, 1998. (p. 44)

Daan Leijen. HMF: Simple type inference for first-class polymorphism. In ICFP08. To appear. (pp. vi, 16, 35, 51, 149, 210, and 214)

—. Flexible types: robust type inference for first-class polymorphism. Technical Report MSR-TR-2008-55, Microsoft Research, March 2008b. (p. 41)

—. A type directed translation of MLF to System F. In ICFP07, pages 111–122. (p. 41)

—. Extensible records with scoped labels. In TFP05, pages 179–194. (pp. 127 and 151)

—. wxHaskell: a portable and concise GUI library for Haskell. In HW04, pages 57–68. (p. 127)

Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001. (p. 56)

Xavier Leroy. Coinductive big-step operational semantics. In ESOP06, pages 54–68. (pp. 58 and 59)

Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In POPL91, pages 291–302. (pp. 113 and 117)

Ben Lippmeier. The Disciplined Disciple Compiler (DDC), July 2008. `http://www.haskell.org/haskellwiki/DDC`. (p. 113)

Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently-typed lambda calculus. Unpublished manuscript, 2008. (p. 40)

Bradley M. Lushman. *Direct and Expressive Type Inference for the Rank 2 Fragment of System F.* PhD thesis, University of Waterloo, 2007. (pp. 41, 149, and 217)

Ian Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994. (p. 89)

Jan-Willem Maessen. Eager haskell: resource-bounded execution yields efficient iteration. In HW02, pages 38–50. (p. 56)

Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In POPL90, pages 382–401. (p. 150)

John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998. (pp. 57, 58, 143, 155, 167, and 173)

Simon Marlow. Update avoidance analysis by abstract interpretation. In FPW93. (p. 86)

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982. (pp. 24 and 25)

Adam Megacz. Relationship between uniqueness types and single-threaded lambda calculus?, June 2007. Message on the Clean mailing list. (p. 221)

Bertrand Meyer. *Object-Oriented Software Construction (Second Edition)*. Prentice Hall PTR, 1997. (p. 14)

Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992. (p. 50)

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. (p. 41)

Naftaly Minsky. Towards alias-free pointers. In ECOOP96, pages 189–209. (p. 77)

Rasmus Ejlers Mogelberg and Alex Simpson. Relational parametricity for computational effects. In LICS07, pages 346–355. (p. 77)

James H. Morris. *Lambda calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1969. Also published as technical report MIT-LCS-TR-057. (p. 70)

Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23(3):299–318, November 1999. (p. 44)

Sara Negri and Jan Von Plato. *Structural Proof Theory*. Cambridge University Press, 2001. (p. 20)

Martin Odersky. Observers for linear types. In ESOP92, pages 390–407. (p. 90)

—. How to make destructive updates less destructive. In POPL91, pages 25–36. (pp. 7, 102, and 104)

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In POPL96, pages 54–67. (pp. 46 and 50)

Peter O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003. (p. 112)

Peter W. O'Hearn. Linear logic and interference control. In CTCS01, pages 74–93. (p. 112)

Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *The Bulletin of Symbolic Logic*, 5(2):215–244, June 1999. (pp. 87, 112, and 223)

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. (p. 56)

David Overton. *Precise and expressive mode systems for typed logic programming languages*. PhD thesis, The University of Melbourne, December 2003. (pp. 25, 109, and 110)

Christine Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In TLCA93, pages 328–360. (p. 34)

R. Peña, C. Segura, and M. Montenegro. A sharing analysis for SAFE. In TFP06, pages 109–128. (p. 113)

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions and foreign-language calls in haskell. In Hoare et al. (2001), pages 47–96. (pp. 13, 66, 67, and 68)

—. Haskell 98 language and libraries—the revised report, December 2002. (p. 15)

—. Wearing the hair shirt: a retrospective on Haskell. Invited talk at POPL 2003, 2003. (p. 70)

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In ICFP06, pages 50–61. (pp. vi, 140, and 142)

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, January 2007. (pp. vi, 16, 17, 43, 45, 46, 48, 49, 119, 121, 122, 125, 127, 135, 136, 137, 140, and 149)

Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In POPL93, pages 71–84. (p. 66)

Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In MFPS89, pages 209–228. (p. 34)

Benjamin Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005. (pp. 234 and 241)

Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002. (pp. 17 and 39)

Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991. (p. 73)

Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In TACS01, pages 219–242. (p. 161)

Rinus Plasmeijer and Peter Achten. The implementation of iData—a case study in generic programming. In IFL05, pages 106–123. (p. 15)

Rinus Plasmeijer and Marko van Eekelen. *Clean Language Report (version 2.1)*, November 2002. (pp. 15, 84, 116, 217, and 218)

—. Keep it Clean: a unique approach to functional programming. *ACM SIGPLAN Notices*, 34(6): 23–31, 1999. (p. 69)

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In ICFP07, pages 141–152. (p. 66)

G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3): 223–255, December 1977. (p. 12)

David J. Pym. On bunched predicate logic. In LICS99, pages 183–192. (p. 223)

Willard Van Orman Quine. *From a logical point of view*. Harvard University Press, 1953. (p. 59)

Brian Rabern and Landon Rabern. A simple solution to the hardest logic puzzle ever. *Analysis*, 68(2):105–111, April 2008. (p. 20)

Didier Rémy. Simple, partial type-inference for system F based on type-containment. In ICFP05, pages 130–143. (p. 214)

Greg Restall. *An Introduction to Substructural Logics*. Routledge, 2000. (pp. 53 and 54)

J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In LICS02, pages 55–74. (p. 112)

John C. Reynolds. Towards a theory of type structure. In PROGSYM74, pages 408–423. (p. 36)

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. (p. 24)

Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1): 1–22, 1998. (pp. 62 and 221)

Sven-Bodo Scholz. Single assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003. (p. 106)

Ronny Wichers Schreur and Rinus Plasmeijer. Dynamic construction of generic functions. In IFL04, pages 160–176. (p. 15)

Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In LOPSTR01, pages 1–24. (p. 77)

Tim Sheard. Putting Curry-Howard to work. In HW05, pages 74–85. (p. 143)

Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In GRAPH94, pages 358–379. (p. 78)

Sjaak Smetsers, Arjen van Weelden, and Rinus Plasmeijer. Efficient and type-safe generic data storage. In WGT08. Electronic Notes in Theoretical Computer Science. (p. 15)

Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–517, 1990. (p. 61)

M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006. (pp. 12 and 17)

Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000/1967. Reprint of lecture notes originally presented in 1967. (p. 61)

N. I. Styazhkin. *History of Mathematical Logic from Leibniz to Peano*. M.I.T. Press, 1969. (p. 25)

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In TLDI07, pages 53–66. (pp. 35, 37, and 143)

Hugh Tredennick. *Aristotle—Metaphysics, books I–IX*. Harvard University Press, 1933. (p. 17)

A. S. Troelstra. *Lectures on Linear Logic*. Center for the Study of Language and Information, 1992. (p. 54)

A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996. (p. 19)

A. M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937. (pp. 11 and 13)

David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1-2):231–248, 1999. (p. 77)

David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In FPCA95, pages 1–11. (pp. 86, 90, and 91)

C. Umans, T. Villa, and A.L. Sangiovanni-Vincentelli. Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1230–1246, July 2006. (p. 150)

S. van Bakel, L. Liquori, R. Ronchi della Rocca, , and P. Urzyczyn. Comparing cubes of typed and type assignment systems. *Annals of Pure and Applied Logic*, 86(3):267–303, 1997. (pp. 39 and 205)

Steffen van Bakel, Sjaak Smetsers, and Simon Brock. Partial type assignment in left linear applicative term rewriting systems. In CAAP92, pages 300–321. (p. 78)

Jean van Heijenoort, editor. *From Frege to Gödel—A source book in mathematical logic, 1879–1931*. Harvard University Press, 1967. (p. 18)

Arjen van Weelden, Sjaak Smetsers, and Rinus Plasmeijer. Polytypic syntax tree operations. In IFL05, pages 106–123. (p. 15)

Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Polytypic programming in Coq. In WGP08. To appear. (p. 218)

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In ICFP06, pages 251–262. (pp. 16, 41, 68, and 149)

—. FPH: First-class polymorphism for Haskell. In ICFP08. To appear. (p. 41)

Philip Wadler. Linear types can change the world! In IFIP90, pages 561–581. (pp. 7, 89, and 90)

—. How to declare an imperative. *ACM Computing Surveys (CSUR)*, 29(3):240–263, 1997. (p. 70)

—. Monads for functional programming. In AFP95, pages 24–52. (p. 65)

—. There's no substitute for linear logic. In MFPS92pre. (p. 77)

—. A syntax for linear logic. In MFPS94, pages 513–529. (p. 77)

—. A taste of linear logic. In MFCS93, pages 185–210. (pp. 17, 54, and 126)

—. Is there a use for linear logic? In PEPM91, pages 255–273. (pp. 7, 88, 89, and 90)

David Wakeling. *Linearity and Laziness*. PhD thesis, University of York, 1990. (p. 67)

David Walker. Substructural type systems. In Pierce (2005). (pp. 17, 126, 154, and 164)

Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In POPL99, pages 15–28. (pp. 86 and 87)

Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, 1997. (p. 13)

J. B. Wells. The essence of principal typings. In ICALP02, pages 913–925. (p. 43)

—. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999. (pp. 40 and 119)

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. (p. 58)

Jialong Zhang and Fenrong Liu. Some thoughts on Mohist logic. In LORI07, pages 85–102. (p. 18)

Conferences

AFP95. Johan Jeuring and Erik Meijer, editors. *Proceedings of the 1st international spring school on Advanced Functional Programming (AFP)*, volume 925 of *Lecture Notes in Computer Science*, 1995. Springer-Verlag. (p. 240)

ASIAN99. P. S. Thiagarajan and R. Yap, editors. *Proceedings of the 5th Asian computing science conference (ASIAN)*, volume 1742 of *Lecture Notes in Computer Science*, 1999. Springer-Verlag. (p. 232)

CAAP92. Jean-Claude Raoult, editor. *Proceedings of the 17th Colloquium on trees in Algebra And Programming (CAAP)*, volume 581 of *Lecture Notes in Computer Science*, 1992. Springer-Verlag. (p. 240)

CiE05. S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors. *Proceedings of the 1st conference on Computability in Europe—New Computational Paradigms (CiE)*, volume 3526 of *Lecture Notes in Computer Science*, June 2005. Springer Berlin. (p. 233)

CTCS01. David H. Pitt, Pierre-Louis Curien, Samson Abramsky, Andrew M. Pitts, Axel Poigné, and David E. Rydeheard, editors. *Proceedings of the 4th international conference on Category Theory and Computer Science (CTCS)*, volume 530 of *Lecture Notes in Computer Science*, 1991. Springer-Verlag. (p. 237)

ECOOP96. Pierre Cointe, editor. *Proceedings of the 10th European Conference on Object Oriented Programming (ECOOP)*, volume 1098 of *Lecture Notes in Computer Science*, July 1996. Springer-Verlag. (p. 237)

ESOP02. Daniel Le Métayer, editor. *Proceedings of the 11th European Symposium on Programming languages and systems (ESOP)*, volume 2305 of *Lecture Notes in Computer Science*, 2002. Springer-Verlag. (p. 229)

ESOP06. Peter Sestoft, editor. *Proceedings of the 15th European Symposium on Programming languages and systems (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, 2006. Springer-Verlag. (p. 236)

ESOP92. Bernd Krieg-Brückner, editor. *Proceedings of the 4th European Symposium on Programming (ESOP)*, volume 582 of *Lecture Notes in Computer Science*, 1992. Springer-Verlag. (p. 237)

FLOPS04. Yukiyoshi Kameyama and Peter J. Stuckey, editors. *Proceedings of the 7th international symposium on Functional and Logic Programming (FLOPS)*, volume 2998 of *Lecture Notes in Computer Science*, 2004. Springer-Verlag. (p. 233)

FLOPS08. Jacques Garrigue and Manuel Hermenegildo, editors. *Proceedings of the 9th international symposium on Functional and Logic Programming (FLOPS)*, volume 4989 of *Lecture Notes in Computer Science*, 2008. Springer-Verlag. (p. 236)

FoIKS00. Klaus-Dieter Schewe and Bernhard Thalheim, editors. *Proceedings of the 1st international symposium on Foundations of Information and Knowledge Systems (FoIKS)*, volume 1762 of *Lecture Notes in Computer Science*, 2000. Springer-Verlag. (p. 233)

FPCA93. *Proceedings of the 6th international conference on Functional Programming languages and Computer Architecture (FPCA)*, 1993. ACM. (p. 235)

FPCA95. *Proceedings of the 7th international conference on Functional Programming languages and Computer Architecture (FPCA)*, 1995. ACM. (pp. 235 and 240)

FPW93. Kevin Hammond and David N. Turner, editors. *Proceedings of the 6th Glasgow Workshop on Functional Programming*, Electronic Workshops in Computing, 1993. Springer-Verlag. (p. 237)

FPWS95. David N. Turner, editor. *Proceedings of the 8th Glasgow Workshop on Functional Programming*, Electronic Workshops in Computing, 1995. Springer-Verlag. (p. 235)

FSTTCS93. R. K. Shyamasundar, editor. *Proceedings of the 13th conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, December 1993. Springer-Verlag. (p. 230)

GRAPH94. Hans Jürgen Schneider and Hartmut Ehrig, editors. *Proceedings of the international workshop on graph transformations in computer science*, volume 776 of *Lecture Notes in Computer Science*, 1994. Springer-Verlag. (p. 239)

HOOTS00. *Proceedings of the 4th international workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 41 of *Electronic Notes in Theoretical Computer Science*, September 2000. Elsevier. (p. 231)

HOPL07. *Proceedings of the 3rd ACM SIGPLAN conference on History Of Programming Languages (HOPL)*, 2007. ACM. (p. 235)

HW02. *Haskell '02: Proceedings of the 6th ACM SIGPLAN Workshop on Haskell*, 2002. ACM. (p. 236)

HW04. *Haskell '04: Proceedings of the 8th ACM SIGPLAN Workshop on Haskell*, September 2004. ACM. (p. 236)

HW05. *Haskell '05: Proceedings of the 9th ACM SIGPLAN Workshop on Haskell*, 2005. ACM. (p. 239)

ICALP02. Peter Widmayer, Francisco Triguero, Rafael Morales, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors. *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2380 of *Lecture Notes in Computer Science*, 2002. Springer-Verlag. (p. 241)

ICFP02. *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002. ACM. (p. 230)

ICFP03. *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2003. ACM. (p. 231)

ICFP05. *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2005. ACM. (pp. 229 and 239)

ICFP06. *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2006. ACM. (pp. 238 and 240)

ICFP07. *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2007. ACM. (pp. 234, 236, and 238)

ICFP08. *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008. ACM. (pp. 236 and 240)

IFIP90. M. Broy and C. B. Jones, editors. *Proceedings of the IFIP TC2 WG 2.2/2.3 Working Conference on Programming Concepts and Methods*, April 1990. North-Holland. (p. 240)

IFL01. Thomas Arts and Markus Mohnen, editors. *Proceedings of the 13th international workshop on Implementation of Functional Languages (IFL)*, volume 2312 of *Lecture Notes in Computer Science*, 2002. Springer-Verlag. (pp. 231 and 232)

IFL02. Ricardo Peña and Thomas Arts, editors. *Proceedings of the 14th international workshop on Implementation of Functional Languages (IFL)*, volume 2670 of *Lecture Notes in Computer Science*, 2003. Springer-Verlag. (p. 232)

IFL04. C. Grelck, F. Huch, G. J. Michaelson, and Ph. Trinder, editors. *Proceedings of the 16th international workshop on Implementation and Application of Functional Languages (IFL)*, volume 3474 of *Lecture Notes in Computer Science*, 2004. (p. 239)

IFL05. Andrew Butterfield, editor. *Proceedings of the 17th international workshop on Implementation and Application of Functional language (IFL)*, volume 4015 of *Lecture Notes in Computer Science*, December 2005. (pp. vii, 238, and 240)

IFL06. Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors. *Proceedings of the 18th international symposium on Implementation and Application of Functional Languages (IFL)*, volume 4449 of *Lecture Notes in Computer Science*, 2007. Springer-Verlag. (p. 232)

IFL07. Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors. *Proceedings of the 19th international symposium on Implementation and Application of Functional Languages (IFL)*, volume 5083 of *Lecture Notes in Computer Science*, 2008. Springer-Verlag. (p. 232)

IFL95. Thomas Johnsson, editor. *Proceedings of the 7th international workshop on Implementation of Functional Languages (IFL)*, 1995. Chalmers University. (p. 233)

LFP94. *Proceedings of the 8th ACM conference on LISP and Functional Programming (LFP)*, 1994. ACM. (p. 235)

LICS02. *Proceedings of the 17th annual IEEE symposium on Logic In Computer Science (LICS)*, 2002. IEEE Computer Society. (p. 239)

LICS07. *Proceedings of the 22nd annual IEEE symposium on Logic In Computer Science (LICS)*, 2007. IEEE Computer Society. (p. 237)

LICS90. *Proceedings of the 5th annual IEEE symposium on Logic In Computer Science (LICS)*, June 1990. IEEE. (p. 233)

LICS96. *Proceedings of the 11th annual IEEE symposium on Logic In Computer Science (LICS)*, 1996. IEEE Computer Society. (p. 230)

LICS99. *Proceedings of the 14th annual IEEE symposium on Logic In Computer Science (LICS)*, 1999. IEEE Computer Society. (p. 239)

LOPSTR01. Alberto Pettorossi, editor. *Proceedings of the 11th international workshop on Logic based Program Synthesis and Transformation (LOPSTR)*, volume 2372 of *Lecture Notes in Computer Science*, 2001. Springer-Verlag. (p. 239)

LORI07. Johan van Benthem, Shier Ju, and Frank Veltman, editors. *A Meeting of the Minds—Proceedings of the workshop on Logic, Rationality and Interaction (LORI)*, 2007. College Publications. (p. 241)

MFCS93. *Proceedings of the 18th international symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 711 of *Lecture Notes in Computer Science*, 1993. Springer-Verlag. (p. 241)

MFPS06. *Proceedings of the 22nd annual conference on Mathematical Foundations of Programming Semantics (MFPS)*, May 2006. Electronic Notes in Theoretical Computer Science, volume 158. (p. 230)

MFPS89. M. Main, A. Melton, M. Mislove, and D. Schmidt, editors. *Proceedings of the 5th international conference on Mathematical Foundations of Programming Semantics (MFPS)*, volume 442 of *Lecture Notes in Computer Science*, 1990. Springer-Verlag. (p. 238)

MFPS92pre. *Informal Proceedings of the 8th international workshop on the Mathematical Foundations of Programming Semantics (MFPS)*, April 1992. (p. 241)

MFPS94. *Proceedings of the 9th international conference on Mathematical Foundations of Programming Semantics (MFPS)*, volume 802 of *Lecture Notes in Computer Science*, 1994. Springer-Verlag. (p. 241)

PEPM08. *Proceedings of the 18th ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM)*, 2008. ACM. (p. 234)

PEPM91. *Proceedings of the 2nd ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM)*, 1991. ACM. (p. 241)

PHILPS95. Manuel Hermenegildo and S. Doaitse Swierstra, editors. *Proceedings of the 7th international symposium on Programming Languages: Implementations, Logics and Programs (PLILPS)*, volume 982 of *Lecture Notes in Computer Science*, 1995. Springer-Verlag. (p. 230)

PLDI02. *Proceedings of the 23rd ACM SIGPLAN 2002 conference on Programming Language Design and Implementation (PLDI)*, June 2002. ACM. (p. 233)

POPL06. *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2006. ACM. (p. 232)

POPL08. *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2008. ACM. (p. 229)

POPL82. *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1982. ACM. (p. 232)

POPL90. *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1990. ACM. (p. 237)

POPL91. *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1991. ACM. (pp. 236 and 237)

POPL93. *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1993. ACM. (pp. 236 and 238)

POPL96. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1996. ACM. (p. 237)

POPL99. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1999. ACM. (p. 241)

PROGSYM74. B. Robinet, editor. *Proceedings of the 1st Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, 1974. Springer-Verlag. (p. 239)

SEGRAGRA95. Corradini and Montanari, editors. *Proceedings of the joint COMPUGRAPH/SEMAGRAPH workshop on graph rewriting and computation (SEGRAGRA)*, 1995. Elsevier Electronic Notes in Theoretical Computer Science. (p. 230)

TACS01. Naoki Kobayashi and Benjamin C. Pierce, editors. *Proceedings of the 4th international symposium on Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *Lecture Notes in Computer Science*, 2001. Springer-Verlag. (p. 238)

TAPSOFT89. Josep Díaz and Fernando Orejas, editors. *Proceedings of the 3rd international joint conference on Theory And Practice of Software development (TAPSOFT)*, volume 352 of *Lecture Notes in Computer Science*, 1989. Springer-Verlag. Volume 2 of the conference proceedings. (p. 233)

TFP05. Marko van Eekelen, editor. *Proceedings of the 6th symposium on Trends in Functional Programming (TFP)*, volume 6 of *Trends in Functional Programming*, 2007. Intellect. (p. 236)

TFP06. Henrik Nilsson, editor. *Proceedings of the 7th symposium on Trends in Functional Programming (TFP)*, volume 7 of *Trends in Functional Programming*, 2007. Intellect. (pp. 235 and 238)

TFP06pre. *Informal proceedings of the 7th symposium on Trends in Functional Programming (TFP)*, 2006. (p. 232)

TFP07pre. Marco T. Morazán and Henrik Nilsson, editors. *Informal proceedings of the 8th symposium on Trends in Functional Programming (TFP)*, 2007. Seton Hall University. Technical report TR-SHU-CS-2007-04-1. (p. 232)

THOLs00. Mark Aagaard and John Harrison, editors. *Proceedings of the 13th international conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1869 of *Lecture Notes in Computer Science*, 2000. Springer-Verlag. (p. 232)

TLCA93. M. Bezem and J. F. Groote, editors. *Proceedings of the 1st international conference on Typed Lambda Calculi and Applications (TLCA)*, volume 664 of *Lecture Notes in Computer Science*, 1993. (p. 238)

TLDI07. *Proceedings of the 3rd ACM SIGPLAN international workshop on Types in Language Design and Implementation (TLDI)*, 2007. ACM. (p. 240)

TPHOLs08. Otmane Ait Mohamed, César Mu noz, and Sofiène Tahar, editors. *Proceedings of 21st international conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *Lecture Notes in Computer Science*, 2008. Springer-Verlag. (p. 230)

TYPES07. Marino Miculan, Ivan Scagnetto, and Furio Honsell, editors. *Proceedings of the 6th international conference on Types for proofs and programs (TYPES)*, volume 4941 of *Lecture Notes in Computer Science*, 2008. Springer-Verlag. (p. 232)

WGP08. *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming (WGP)*, 2008. ACM. (p. 240)

WGT08. *Proceedings of the Workshop on Generative Technologies (WGT)*, April 2008. Elsevier. Electronic Notes in Theoretical Computer Science. (p. 239)

WRS07pre. Jürgen Giesl, editor. *Informal proceedings of the 7th international Workshop on Reduction Strategies in rewriting and programming (WRS)*, June 2007. (p. 231)

Index