On the foundations of functional programming: a programmer's point of view

Rob R. Hoogerwoord

13 may 1994

Contents

Chapter 0

Introduction

Functional programming is a style of programming where, possibly recursive definitions of functions are derived to satisfy a priori given specifications. Examples of this programming style can be found in [4, 7, 9]. In this study we provide a mathematical justification for the rules of this game. So, this study is not about (functional) programming but about its mathematical foundations.

Recursive function definitions can be considered as equations of which the functions thus defined are solutions. This raises two questions: does such an equation always have a solution, and if so, which of the possibly very many solutions is intended? One of the reasons for undertaking this study is the observation that these two questions are entirely different and that each of them can be answered in isolation.

A second, more important, reason is that, nowadays, it is taken very much for granted that to answer these questions one should study domains that are based on complete lattices or complete partial orders [12]. Although this approach is viable and although it permits the formulation of stronger theorems, it certainly is not the simplest possible. Moreover, some of the arguments to motivate this approach simply are wrong. To give an example, we quote from [12, p.73]:

> An even more fundamental difficulty, however, is highlighted by the fact that our definition [...] involves the application of x to itself. The possibility of self-application can lead to paradoxes. For example, suppose we define
>
> $$u \;=\; \lambda y.\ \mathsf{if}\ y(y) = a\ \mathsf{then}\ b\ \mathsf{else}\ a \quad .$$
>
> Then an attempt to evaluate $u(u)$ gives
>
> $$u(u) \;=\; \mathsf{if}\ u(u) = a\ \mathsf{then}\ b\ \mathsf{else}\ a$$
>
> which is a contradiction.

The alleged contradiction in this example, however, is not due to the occurrence of self-application, as the author suggests, but by his failing to be explicit about the rules of his game. As is often the case with paradoxes, the contradiction disappears when these rules are made explicit. In this example, the symbol $=$ in the expression $y(y) = a$ is the culprit: the contradiction follows from the author's tacid assumption that $=$ denotes equality; as we shall show in Chapter 8, in the λ-calculus $=$ does not exist.

To show that the above mentioned equations have solutions we use the (type free) λ-calculus: in Chapter 5 we prove that in the λ-calculus every (so-called) "admissible equation" has a solution. The use of the λ-calculus seems to pose a few additional problems, though. First, in the λ-calculus equations such as $x\colon x = \neg x$ and $x\colon x = 1+x$ have solutions as well, and this seems paradoxical. Second, as folklore has it, in the λ-calculus functions can be applied to themselves, which seems to make it difficult to attribute meaningful types to such functions. Fortunately, these aspects can be discussed in isolation, that is, without reference to the λ-calculus at all. In this respect, this study also is an exercise in disentanglement.

* * *

This study is about the foundations of functional programming, not about the foundations of mathematics in general. Therefore, we may (and shall) use without justification whatever well-understood mathematical concepts we need. Their justification is a legitimate concern, but it is a different one and it is not ours, for the simple reason that it is not specific for functional programming. Moreover, whatever mathematical game one wishes to play, one must take some of the rules for granted, lest there be no game.

I make this point with so much emphasis because the λ-calculus plays a possibly confusing role here. The λ-calculus was designed to be used for the formalisation of mathematics and logic, which is a much more ambitious purpose than ours: we only use the λ-calculus as a prototype functional-programming language. Therefore, this is just an ordinary mathematical study, not a meta-mathematical one.

Via the notion of *reduction* the λ-calculus also provides a prototype implementation of a functional-programming language: the λ-calculus is to functional programming what the Turing machine is to sequential programming. Hence, contrary to common belief, we need not construct *models*[0] for the λ-calculus in order to understand functional programming: rather we would say that the λ-calculus itself *is* the (computational) model underlying many functional languages.

* * *

As may be clear from the above, the nature of this study is more technical than mathematical. Most if not all of its mathematical contents are well-known and can,

[0] Ever heard of models for a Turing machine?

for example, be found in Barendregt's textbook [2]. Therefore, we shall not elaborate in all details those parts of the story that are well-understood but technically tedious and boring, such as dummy renaming and substitution.

The main purpose of this study is to show how *simple* the rules of the functional programming game are —which does not imply that playing that game is equally simple— . Strangely enough, this is not so trivial a matter as might be expected: authors of works like [11, 12, 13] succeed very well in making their lives more difficult than necessary. In this respect I must stress that the reader of this study who finds himself left with a feeling of dissatisfaction because this study seems to offer nothing new, has missed my point: this study is a strong plea *against* undue complexity.

Wherever the following presentation is (too) tutorial, this should not be interpreted as pedantry but as the result of a conscious attempt to be as explicit as possible and to make this study self-contained.

Chapter 1

Mathematical preliminaries

We use well-known mathematical concepts such as sets, relations, functions, predicate calculus —in the sense of [6]— , and some elementary algebra.

1.0 Notational issues

For function application we use the binary infix operator $\cdot$ ("dot"): we write $f \cdot x$ instead of $f(x)$. The dot binds stronger than all other operators and it is left-binding: $f \cdot x \cdot y$ must be read as $(f \cdot x) \cdot y$. Function composition is denoted by $\circ$, with:

$$(\forall x :: (f \circ g) \cdot x \;=\; f \cdot (g \cdot x))$$

For any binary operator $\odot$ we use $(\odot), (x \odot)$, and $(\odot y)$ as denotations of functions, defined as follows:

$$(\forall x, y :: (\odot) \cdot x \cdot y \;=\; x \odot y)$$
$$(\forall y :: (x \odot) \cdot y \;=\; x \odot y)$$
$$(\forall x :: (\odot y) \cdot x \;=\; x \odot y)$$

examples: $(+1)$ is the function that adds 1 to its argument, $(+)$ is the (curried) addition function, and $(=b)$ is the point predicate that is true when its argument equals b and false otherwise. Also, $(\cdot b)$ is the (higher-order) function that applies its (function) argument to b: $(\cdot b) \cdot f \;=\; f \cdot b$. The latter example shows that it really helps to have an explicit operator for function application.

□

1.1 Functions, equality, and congruences

In the set-theoretical interpretation a function of type $X \to V$ is a subset of the cartesian product $X \times V$ with certain properties. As a result, a function of type $X \to V$ is also a function of type $X \to W$, for all W with $V \subseteq W$. This is quite practical: every integer is a real as well, so it stands to reason that every integer-valued function is also a real-valued function. On the other hand, the set-theoretical interpretation does not allow the equally convenient conclusion that every function of type $X \to V$ is a function of type $Y \to V$ as well, whenever $Y \subseteq X$. In the set-theoretical interpretation we can only formulate this by stating that $f{\uparrow}Y$ has type $Y \to V$, where $f{\uparrow}Y$ denotes "f restricted to Y". For practical purposes, though, it is more convenient to omit ${\uparrow}Y$; this is harmless as long as we are explicit about the domains on which the function is used.

This is reflected in the following definition of *function type*, which is based on the observation that the only thing we can do with a function is apply it to an argument. For sets X and V we say that function f *has type* $X \to V$ whenever:

$$(\forall x : x \in X : f \cdot x \in V)$$

We write $f \in (X \to V)$ to denote "f has type $X \to V$". Thus defined, the operator $\to$ has the following (anti)monotonicity properties:

$$\begin{array}{lll} Y \subseteq X & \Rightarrow & f \in (X \to V) \;\Rightarrow\; f \in (Y \to V) \\ V \subseteq W & \Rightarrow & f \in (X \to V) \;\Rightarrow\; f \in (X \to W) \end{array}$$

examples: Every function that has type $X \to \mathsf{Nat}$ also has type $X \to \mathsf{Int}$ and every function that has type $\mathsf{Int} \to V$ also has type $\mathsf{Nat} \to V$.
□

For function composition we have, of course:

$$f \in (X \to Y) \;\wedge\; g \in (Y \to Z) \quad \Rightarrow \quad g \circ f \in (X \to Z)$$

A fundamental property, connecting functions with equality, is Leibniz's rule of *substitution of equals for equals*. In its simplest form the rule is:

$$(\forall x, y :: \; x = y \;\Rightarrow\; (\forall f :: f \cdot x = f \cdot y)\,)$$

Moreover, it is convenient to consider function application as a function itself, to which Leibniz's rule can be applied, so we also have:

$$(\forall f, g :: \; f = g \;\Rightarrow\; (\forall x :: f \cdot x = g \cdot x)\,)$$

By contraposition these rules can be transformed into the following ones, which are useful for proving *differences*:

$$(\forall x,y :: \ x \neq y \ \Leftarrow \ (\exists f :: f\cdot x \neq f\cdot y)\,)$$
$$(\forall f,g :: \ f \neq g \ \Leftarrow \ (\exists x :: f\cdot x \neq g\cdot x)\,)$$

We shall document applications of any of these rules with the hint "Leibniz".

As a matter of fact, there is not much more to equality than that it satisfies Leibniz's rule and that it is an equivalence relation. Every equivalence relation $\simeq$ that, for a given collection of functions, satisfies:

$$(\forall x,y :: \ x \simeq y \ \Rightarrow \ (\forall f :: f\cdot x = f\cdot y)\,) \quad ,$$

may be considered as an equality relation with respect to that collection of functions. In such a case $\simeq$ is called *congruent* with these functions; conversely, we call the functions *compatible with* $\simeq$. (Notice the difference in emphasis between the two notions.) To all intents and purposes, an equivalence relation may be treated as equality, provided that every function in our universe of discourse is compatible with that relation.

For example, within a collection of functions on the same domain we may define $\simeq$ by:

$$(\forall f,g :: \ f \simeq g \ \equiv \ (\forall x :: f\cdot x = g\cdot x)\,)$$

Then $\simeq$ is an equivalence relation and function application is, by definition, compatible with $\simeq$. Because function application is the only (primitive) operation applicable to functions, $\simeq$ may be treated as function equality: functions related by $\simeq$ can not be distinguished. So, we may safely write $f = g$ instead of $f \simeq g$. As a matter of fact, we now have justified what is known as the axiom of *extensionality*:

$$(\forall f,g :: \ f = g \ \Leftarrow \ (\forall x :: f\cdot x = g\cdot x)\,)$$

In this formula the range of dummy x is *the* (i.e.: largest possible) domain of f and g. Very often, though, we are only interested in properties of functions on a smaller domain than the largest possible; whether or not we consider two functions as equal then depends on the domain of our interest: functions can be equal on some domain but different on a larger one.

example: We consider the following equation (with unknown f):

$$f : \quad f\cdot 0 = 3 \ \wedge \ (\forall i : i \in \mathsf{Nat} : \ f\cdot(i{+}1) = f\cdot i + 2\,)$$

In $\mathsf{Int}\rightarrow\mathsf{Int}$ this equation has infinitely many solutions; in $\mathsf{Nat}\rightarrow\mathsf{Int}$, however, all these solutions are equal.

□

Finally, I wish to point out that, strictly speaking, there is no such thing as a *recursive function*: recursiveness is a property of a function's definition, not of the function itself[0].

example: Consider the following definitions of functions f and g:

$$f\cdot 0 = 3 \ \wedge \ (\forall i : i{\in}\mathsf{Nat} : \ f\cdot(i{+}1) = f\cdot i + 2\,) \quad \text{, and:}$$
$$(\forall i : i{\in}\mathsf{Nat} : \ g\cdot i = 2{*}i + 3\,)$$

The definition of f is recursive whereas g's definition is not, and within $\mathsf{Nat}\rightarrow\mathsf{Int}$ we have $f = g$.

□

1.2 On representations

In this study we use the terms *represent* and *representation* in a strictly technical meaning, which we define here. For sets U and V and a *surjective* function F of type $U\rightarrow V$, we say that the elements of U represent the elements of V; we also say that set U represents set V. Function F, called the *abstraction function*, maps representations to values represented. Because F is surjective every element of V is represented by at least one element in U. Conversely, it is not precluded that different elements of U represent the same value in V: F need not be injective.

The situation becomes slightly more complicated when we also wish to represent functions from or to V by similar functions from or to U. For example, for a fixed set X, function g, $g\in(X\rightarrow V)$, is represented by function f, $f\in(X\rightarrow U)$, provided that f and g satisfy:

$$(\forall x : x{\in}X : g\cdot x \ = \ F\cdot(f\cdot x)\,) \quad (\text{, i.e. on } X : \quad g = F\circ f)$$

In this way, every function in $X\rightarrow U$ represents a function in $X\rightarrow V$.

Not every function in $U\rightarrow X$ represents a function in $V\rightarrow X$, though. In this case f only represents g provided that:

$$(\forall u :: g\cdot(F\cdot u) \ = \ f\cdot u) \quad (\text{, i.e.: } \ g\circ F = f)$$

From this it follows immediately that f must satisfy a kind of consistency condition, stating that f is invariant under *changes of representation*:

$$(1.0) \quad (\forall u, u' :: \ F\cdot u = F\cdot u' \ \Rightarrow \ f\cdot u = f\cdot u'\,)$$

Conversely, every f that satisfies (1.0) represents a unique function in $V\rightarrow X$ (namely $f\circ \overleftarrow{F}$).

[0] In common parlance, a function is called recursive if it admits a recursive definition.

Every function F on U induces an equivalence relation on U, namely the relation $\simeq$ defined by:

(1.1) $(\forall u, u' :: \; u \simeq u' \;\equiv\; F \cdot u = F \cdot u')$

In terms of $\simeq$ consistency condition (1.0) can now be rephrased as

$$(\forall u, u' :: \; u \simeq u' \;\Rightarrow\; f \cdot u = f \cdot u') \quad ,$$

which is the proposition that f is compatible with $\simeq$. Apparently, the functions on U that represent functions on V are exactly those functions that are compatible with $\simeq$. This is not so surprising: the relation $\simeq$ (on U) represents $=$ (on V).

This is well-known and so is the fact that for every equivalence relation $\simeq$ on a set U, a set V and an abstraction function F in $U \to V$ exist such that $\simeq$ is the equivalence relation defined by (1.1). (Just take for V the set $U/\simeq$ of the equivalence classes of $\simeq$.) Thus, abstraction functions —also called *homomorphisms*— and congruence relations are two faces of the same coin.

1.3 On semantics

A formal language is a collection of objects called "sentences" or "terms". Such a collection is not amorphous but exhibits a certain algebraic structure, because terms in it can be composed, by means of "constructors", to form larger terms. Usually, a formal language and its algebraic structure are defined by means of a (context-free or context-sensitive) grammar, but for discussions on semantics the algebraic point of view is more adequate. The semantics of a formal language can be defined in two ways, which are mathematically equivalent but technically different.

In the denotational approach to semantics, a domain of "values" is defined; the terms of the language are supposed to represent values in this domain and the term constructors are supposed to represent operators on this domain. This is formalised by defining an abstraction function mapping terms to their values. This function is required to be such that the value of a composite term depends on the values of its constituent terms only. This is called *compositionality*; algebraically speaking, this is the requirement that the abstraction function be a homomorphism. An example of a successful application of the denotational approach is the predicate-transformer semantics for sequential programs [6], where the "value" of a statement is a pair of predicate transformers.

Alternatively, the semantics of the formal language can be defined by means of an equivalence relation on the set of terms; the only requirement now is that this relation is congruent with the term constructors. The technical advantage of this approach is that both the domain of values and the abstraction function can remain anonymous. This approach is particularly useful when the language will be used for *equational reasoning*, as is the case in functional programming.

The requirements of compositionality and of congruence serve the same purpose, namely to validate the rule of Leibniz: a subterm of a composite term may then be replaced by any equivalent term without affecting the value of the composite term.

summary: The discussion in the last two sections shows that the following phrases all refer to the same mathematical concept:

Leibniz's rule of equals for equals
substitutivity
congruence relations
homomorphisms
compositionality
referential transparency

□

Chapter 2

Self application

One of the alleged problems with the λ-calculus pertains to expressions of the form $f \cdot f$: if f is a function then what is its type? The obvious answer is that if f has type $X \to Y$ then $f \cdot f$ is meaningful if f is also an element of X. This is certainly so when X and Y satisfy

$$X \to Y \ \subseteq \ X$$

For nontrivial sets X, Y this is impossible, because then $X \to Y$ has greater cardinality than X. Hence, the obvious answer will not do.

The only other simple possibility —apart from rejecting self application altogether— is to require that X and Y satisfy

$$X \ \subseteq \ X \to Y$$

This does not make $f \cdot f$ meaningful for every f in $X \to Y$, but it does so for all f in X. As we shall show this is more than sufficient and, by the discrepancy between the cardinalities of $X \to Y$ and X, this is about the best we may expect.

Nevertheless, X is a strange set. Its elements are also elements of $X \to Y$ and so they are functions, but they are also the arguments to which these functions can be applied. Do such sets exist? (In the set-theoretical model the answer is: no! [3].) Instead of answering this question, we avoid it by taking the inclusion in $X \subseteq X \to Y$ with a grain of salt, that is, modulo a change of representation. We devote the remainder of this chapter to elaborating this.

2.0 Representation of functions

We consider a set Ω and a binary operator $\odot$ of type $\Omega \times \Omega \to \Omega$. In what follows all variables range over Ω. With every f in Ω we associate a function $(f\odot)$, of type $\Omega \to \Omega$, which satisfies —according to the conventions from Section 1.0—:

$$(\forall x :: (f\odot)\cdot x \ = f \odot x \,)$$

In this way, f, which is not a function, *represents* the function $(f\odot)$ in $\Omega\to\Omega$. The corresponding abstraction function, mapping Ω onto a subset of $\Omega\to\Omega$, is $(\odot)$, with:

$$(\forall f :: (\odot)\cdot f \;=\; (f\odot))$$

In this arrangement $\odot$ represents function application: we have $f\odot x = (f\odot)\cdot x$, so $f\odot x$ is the value of the function (represented by) f applied to x, for all f and x in Ω. In particular, $f\odot f$ and $x\odot x$ can be considered as function applications as well. The crux is that the left operand of $\odot$ is interpreted differently from the right operand: self application does not apply a function to itself but to its representation, because we have $f\odot f = (f\odot)\cdot f$.

When it is the operator $\odot$'s sole purpose to represent function application we may safely use $\cdot$ instead of $\odot$, thus (deliberately) ignoring the distinction between functions and their representations. Such an abuse of notation is quite common, it is harmless, and it simplifies the formulae —$f\cdot x$ is simpler than $(f\odot)\cdot x$—; yet, the distinction is crucial to a proper understanding of self application.

The reader who is tempted to consider the above as cheating should bear in mind two things. First, all that we, as human symbol manipulators, or computers, as mechanical symbol manipulators, can do is manipulate representations of the objects of our interest. The expressions in a (functional) programming language are not functions but representations of functions, that is, they are function definitions. We can, for example, say that a certain definition is recursive, or we can discuss the efficiency of a definition, but recursiveness and efficiency are not properties of the function thus represented; one and the same function usually admits several definitions, with different properties such as efficiency. Second, the above representation trick is also used in the domain-theoretic models, so in this respect these models offer no help. For example, in the graph model $\mathsf{P}\omega$ by Plotkin and Scott the corresponding abstraction function is called *fun* [2, 12].

The above game can be played with any set Ω and any binary operator $\odot$. As we have seen, the mapping $(\odot)$, which embeds Ω into $\Omega\to\Omega$, is not surjective. Hence, not every function in $\Omega\to\Omega$ is representable in Ω. This raises the question whether Ω and $\odot$ can be chosen in such a way that Ω represents a "sufficiently interesting" subset of $\Omega\to\Omega$.

examples: Let Ω be Int and let $\odot$ be $+$. Then we have $(m+)\cdot n = m+n$, so m represents the function adding m to its argument; in particular, 0 represents the identity function on Int, we have $((m+n)+) = (m+)\circ(n+)$, and $((-m)+) = (m+)^{-1}$, so the subset of representable functions forms a group (, namely the group of *translations*). In this setup function application and composition even coincide: both are represented by $+$.

A less trivial example is the matrix calculus, where matrices represent linear mappings on vector spaces of finite dimension. Vectors themselves can

be represented by matrices, and if we do so both function application and function composition are represented by matrix multiplication. (By the way, matrix multiplication itself is a linear mapping.)

□

2.1 Concluding remarks

Every formal language is a set of finite-length sequences over an at most countably infinite alphabet. Hence, every formal language is at most countably infinite, so in it at most countably many objects —functions included— can be represented. Similarly, the state space of a computer is at most countably infinite, so in a computer at most countably many values —programs included— can be represented. Therefore, we need not be worried at all about the fact that, when we choose Ω to be countable, only a countable subset of $\Omega \to \Omega$ is representable in Ω.

In this chapter the λ-calculus plays no role whatsoever. The λ-calculus only enters the picture when we wish to make the representable subset of $\Omega \to \Omega$ "sufficiently interesting"; that is the subject of the next chapters.

The game we have played here yields a free bonus. For "function" f and "argument" x, $f \odot x$ is an element of Ω that itself represents a function in $\Omega \to \Omega$. That is, in an expression like $(f \odot x) \odot y$ we may consider f as a *two-argument function* and as a *higher-order function* at the same time. Thus, Ω not only represents a subset of $\Omega \to \Omega$, but also of $\Omega \to (\Omega \to \Omega)$, and so on. Hence, by playing the game in this way we obtain multi-argument and higher-order functions for free.

Chapter 3

The λ-calculus

3.0 Introduction

In functional programming, functions are the main components from which programs are constructed. Essential ingredients of a functional-programming language are notations for function application and for function definitions. The λ-calculus provides exactly these two ingredients, and nothing more. The λ-calculus is *a* calculus of functions, not *the* calculus of *all* functions. As we explained in Chapter 2, this should not worry us. On the contrary, that the λ-calculus is a restricted calculus even has advantages; for example, in the λ-calculus all functions have fixed-points.

The λ-calculus has very simple syntax and manipulation rules, but proofs of theorems about the calculus can be quite tedious and laborious, even when they are not difficult. In this respect the current presentation is a compromise between two conflicting desires: on the one hand, I wish to make this study self-contained; on the other hand, I wish to avoid redoing work done by others many times before.

3.1 Syntax

Syntactically, the ingredients of the λ-calculus are a set Name of *names*, a set Exp of *terms*, a unary constructor $^-$ ("name"), and two binary constructors $\dashv$ ("lambda") and $\odot$ ("dotl"), with the following properties:

$^-$ has type $\mathsf{Name} \to \mathsf{Exp}$
$\dashv$ has type $\mathsf{Name} \times \mathsf{Exp} \to \mathsf{Exp}$
$\odot$ has type $\mathsf{Exp} \times \mathsf{Exp} \to \mathsf{Exp}$

Set Name is (countably) infinite and Exp is the *smallest* set that is closed under $^-$, $\dashv$, and $\odot$; hence, properties of Exp may be proved by mathematical induction on the size of the terms, and by structural induction in particular.

Terms of the form $\bar{x}$ are just called "names", terms of the form $x \dashv E$ are called "abstractions", and terms of the form $E \odot F$ are called "applications". In order to save parentheses we adopt the following binding conventions:

$\odot$ binds stronger than $\dashv$, $\odot$ is left-binding, and $\dashv$ is right-binding

example: $x \dashv y \dashv \bar{x} \odot \bar{y} \odot \bar{z} \;=\; x \dashv (y \dashv ((\bar{x} \odot \bar{y}) \odot \bar{z}))$
□

notes: Traditionally, $\bar{x}$ is written as x, $x \dashv E$ as $(\lambda x.E)$, and $E \odot F$ is written as (EF). We have deliberately adopted a more explicit notation, to emphasize the algebraic view that the term constructors are *operators*. Nevertheless, the usual syntactical connotations are implied as well: $\bar{x} \neq \bar{y}$ if $x \neq y$, the classes of names, abstractions, and applications are disjoint, et cetera.

It is (still) amazing that a calculus with so simple a syntax turns out to be so effective: Church did a great job indeed.
□

Throughout this chapter, x, y, z denote names and E, F, G denote terms. Terms are finite, as they are formed by a finite number of applications of the constructors. Hence, every term contains a finite number of names only. Because Name is infinite we have for every term an unbounded supply of names not occurring in the term. We call these names *fresh* with respect to that term.

In an abstraction $x \dashv E$ the operator $\dashv$ is said to *bind* all occurrences of name x in E. Bound names are also called *dummies*, whereas we call names that are not bound *free variables*. The same name can occur as a dummy and as a free variable in the same term, such as x in $(x \dashv \bar{x}) \odot \bar{x}$. Formally, we denote the free variables of term E by the predicate $fv \cdot E$; fv is defined as follows:

$$\begin{array}{lcl} fv \cdot \bar{x} \cdot y & \equiv & x = y \\ fv \cdot (x \dashv E) \cdot y & \equiv & x \neq y \;\wedge\; fv \cdot E \cdot y \\ fv \cdot (E \odot F) \cdot y & \equiv & fv \cdot E \cdot y \;\vee\; fv \cdot F \cdot y \end{array}$$

For term E and names y, z the term $E(y \lhd z)$ is obtained from E by *renaming y to z*, that is, by systematically replacing all free occurrences of y in E by z. Its formal definition is:

$$\begin{array}{lcll} & \bar{y}(y \lhd z) & = & \bar{z} \\ & \bar{x}(y \lhd z) & = & \bar{x} \;\text{, if } x \neq y \\ & (y \dashv E)(y \lhd z) & = & y \dashv E \\ (*) & (x \dashv E)(y \lhd z) & = & x \dashv E(y \lhd z) \;\text{, if } x \neq y \wedge x \neq z \\ & (E \odot F)(y \lhd z) & = & E(y \lhd z) \odot F(y \lhd z) \end{array}$$

notes: The condition $x \neq z$ in rule $(*)$ is necessary to avoid a so-called *name clash*: if $x = z$ free variable y would be replaced by bound variable x, which is generally not what we want.

Renaming does not change the (tree) structure of terms. So, proofs by induction on the structure of terms are insensitive to occasional renamings.
□

Terms that can be transformed into each other by renaming of dummies are equal: we consider the actual choice of dummies as an aspect of the textual representation of terms, not of the terms themselves. (See [2, app. C] for a formal justification.) So we have:

$$y \text{ is fresh} \quad \Rightarrow \quad x \dashv E \;=\; y \dashv E(x \lhd y)$$

Because E is finite and Name is infinite, dummy renamings are always possible.

example: $x \dashv \overline{x} \;=\; y \dashv \overline{y}$, although $\overline{x} \neq \overline{y}$.
□

3.2 Algebraic semantics

We define the semantics of the λ-calculus by defining an equivalence relation $\simeq$ on Exp, with the interpretation that $E \simeq F$ if and only if E and F have "the same value". As we explained in Section 1.3 this is a legitimate approach, provided that we see to it that $\simeq$ is congruent with $\dashv$ and $\odot$.

We proceed in a number of steps and define $\simeq$ by a number of *semantical rules*. We require $\simeq$ to be the *strongest relation* satisfying these rules, and this is essential: we do not want every term to be equivalent to every term, because that would make the calculus useless. We return to this in Chapter 4.

The first semantical rule defines how abstractions can be interpreted as functions, by connecting $\dashv$ and $\odot$. For its formulation we need *substitution*, which is a generalisation of renaming, and which we shall define later:

(β) $\quad (x \dashv E) \odot F \;\simeq\; E(x \lhd F)$, for all x, E, F

This rule states that $x \dashv E$ represents —see Chapter 2— the function mapping F to $E(x \lhd F)$, for all F.

The second rule turns $\simeq$ into a congruence relation; it consists of 3 parts:

$(\gamma 0)$ $\quad E \simeq E' \;\Rightarrow\; x \dashv E \;\simeq\; x \dashv E'$, for all x, E, E'
$(\gamma 1)$ $\quad E \simeq E' \;\Rightarrow\; E \odot F \;\simeq\; E' \odot F$, for all E, E', F
$(\gamma 2)$ $\quad F \simeq F' \;\Rightarrow\; E \odot F \;\simeq\; E \odot F'$, for all E, F, F'

The third rule states that $\simeq$ is an equivalence relation and, finally, the fourth rule completes the job:

(δ) $\quad \simeq$ is reflexive, symmetric, and transitive
(ε) $\quad \simeq$ is the strongest relation satisfying (β) through (δ)

note: The rules given here are slightly redundant: rule $(\gamma 1)$ can be derived from the other rules and the properties of substitution.
□

Rules (γ) and (δ) imply that $\simeq$ may be used as equality in calculations involving $\dashv$ and $\odot$. Hence, we may use equational reasoning to derive properties. For example, for E and E' with $E \simeq E'$ we derive:

$$\begin{array}{ll} & E(x \lhd F) \\ \simeq & \quad \{\ (\beta)\ \} \\ & (x \dashv E) \odot F \\ \simeq & \quad \{\ E \simeq E' \colon (\gamma 0) \text{ and } (\gamma 1)\ \} \\ & (x \dashv E') \odot F \\ \simeq & \quad \{\ (\beta)\ \} \\ & E'(x \lhd F)\ . \end{array}$$

By a similar calculation we can derive $F \simeq F' \ \Rightarrow\ E(x \lhd F) \simeq E(x \lhd F')$. So, without even knowing how substitution is defined, we have proved:

lemma 0: Substitution, as a function of type $\mathsf{Exp} \times \mathsf{Name} \times \mathsf{Exp} \to \mathsf{Exp}$, is compatible with $\simeq$.
□

Substitution is a generalisation of renaming; it is defined as follows:

$$\begin{array}{lll} & \overline{y}(y \lhd G) & = G \\ & \overline{x}(y \lhd G) & = \overline{x} \quad \text{, if } x \neq y \\ (*) & (x \dashv E)(y \lhd G) & = z \dashv E(x \lhd z)(y \lhd G) \quad \text{, for fresh } z \\ & (E \odot F)(y \lhd G) & = E(y \lhd G) \odot F(y \lhd G) \end{array}$$

The dummy renaming in rule $(*)$ is necessary to avoid name clashes (, as was the case with renaming, in the previous section). Substitution has a number of well-known properties which we shall not prove here:

properties of $\lhd$: For x, y, $x \neq y$, we have:

$$\begin{array}{lll} E(x \lhd y) & = & E(x \lhd \overline{y}) \quad \text{(renaming is a special case of substitution)} \\ E(x \lhd y)(y \lhd F) & = & E(x \lhd F) \quad \text{, if } \neg fv \cdot E \cdot y \\ E(x \lhd F) & = & E \quad \text{, if } \neg fv \cdot E \cdot x \\ E(x \lhd \overline{x}) & = & E \\ E(x \lhd F)(x \lhd G) & = & E(x \lhd F(x \lhd G)) \\ E(x \lhd F)(y \lhd G) & = & E(y \lhd G)(x \lhd F) \quad \text{, if } \neg fv \cdot F \cdot y \wedge \neg fv \cdot G \cdot x \end{array}$$

□

example: We derive rule $(\gamma 1)$, by assuming $E \simeq E'$ and:

$$
\begin{array}{ll}
 & E \odot F \\
= & \quad \{ \text{ let } x \text{ be fresh, so } \neg fv \cdot F \cdot x : \text{ substitution properties (twice) } \} \\
 & \bar{x}(x \triangleleft E) \odot F(x \triangleleft E) \\
= & \quad \{ \text{ substitution } \} \\
 & (\bar{x} \odot F)(x \triangleleft E) \\
\simeq & \quad \{ (\beta) \} \\
 & (x \dashv \bar{x} \odot F) \odot E \\
\simeq & \quad \{ E \simeq E' : (\gamma 2) \} \\
 & (x \dashv \bar{x} \odot F) \odot E' \\
\simeq & \quad \{ \text{ the above, in reverse order } \} \\
 & E' \odot F \quad .
\end{array}
$$

□

In the previous section we have identified terms that differ in the choice of dummies only. Therefore, we must prove that the relation $\simeq$ is insensitive to dummy renaming. Because rules (γ), (δ), (ε) define $\simeq$ as a *closure* of the relation defined by rule (β), it suffices to verify that rule (β) is insensitive to dummy renaming:

$$
\begin{array}{ll}
 & (y \dashv E(x \triangleleft y)) \odot F \\
\simeq & \quad \{ (\beta) \} \\
 & E(x \triangleleft y)(y \triangleleft F) \\
= & \quad \{ y \text{ is fresh: property of substitution } \} \\
 & E(x \triangleleft F) \\
\simeq & \quad \{ (\beta) \} \\
 & (x \dashv E) \odot F \quad .
\end{array}
$$

3.3 Denotational semantics

The relation $\simeq$ is an equivalence relation. We now define the value of a term as the equivalence class it belongs to; that is, denoting the value of E by $[\![E]\!]$ we have:

$$[\![E]\!] \;=\; (\mathsf{set}\, F :\; E \simeq F \;:\; F)$$

Because $\simeq$ is congruent with $\bar{}$, $\dashv$, $\odot$, and $\triangleleft$, well-defined operators $\bar{\bar{}}$, $\dashv\!\!\dashv$, $\bullet$, and $\triangleleft\!\!\triangleleft$ exist satisfying:

$$\begin{array}{lcl}
\overline{x} & = & [\![\, \bar{x} \,]\!] \\
x \dashv [\![\, E \,]\!] & = & [\![\, x \dashv E \,]\!] \\
[\![\, E \,]\!] \bullet [\![\, F \,]\!] & = & [\![\, E \odot F \,]\!] \\
[\![\, E \,]\!] \, (x \lhd [\![\, F \,]\!]) & = & [\![\, E (x \lhd F) \,]\!]
\end{array}$$

That is, the abstraction function $[\![\ \cdot\ ]\!]$ is a homomorphism and all properties about $\simeq$ can be reformulated in the value domain $\mathsf{Exp}/\simeq$; for example:

$$\begin{array}{lcl}
[\![\, E \,]\!] = [\![\, F \,]\!] & \equiv & E \simeq F \\
(x \dashv e) \bullet f & = & e (x \lhd f) \\
\overline{x} (x \lhd f) & = & f
\end{array}$$

and so on.

The value domain defined here is known as the *term model*; yet, it hardly deserves to be called a model because it is defined in terms of the calculus itself and because, as a model, it is trivial.

* * *

In conventional denotational semantics for expression languages, each term is interpreted within a so-called *environment* that provides values for the free variables of that term. That is, in this approach functions of type $\mathsf{Name} \to \Omega$ are used to represent environments. The value of a term E is then denoted by $M \cdot E \cdot p$, with $p \in \mathsf{Name} \to \Omega$. Function M, which maps terms onto their meanings, has type:

$$\mathsf{Exp} \to (\mathsf{Name} \to \Omega) \to \Omega$$

We deliberately distinguish *values* and *meanings* here: the (denotational) meaning of E is $M \cdot E$, of type $(\mathsf{Name} \to \Omega) \to \Omega$, whereas $M \cdot E \cdot p$ is the value of E, of type Ω, in environment p. The latter is the one usually referred to in calculations, but the former is the true "meaning" of the term.

Two terms E and F are equivalent if and only if $M \cdot E = M \cdot F$, which amounts to

$$(\forall p :: M \cdot E \cdot p = M \cdot F \cdot p)$$

In the case of the λ-calculus, function M must be compatible with $\simeq$, because $\simeq$ and particularly rule (β) will be used in calculations. That is, M must satisfy:

$$(\forall\, E, F ::\ E \simeq F \ \Rightarrow\ M \cdot E = M \cdot F)$$

We even can define Ω and M in such a way that this implication is an equivalence.

To start with we need an operator to modify environments. (This is standard in denotational semantics.) For p in $\mathsf{Name} \to \Omega$ and x in Name and e in Ω, we define $p(x{:}e)$ in $\mathsf{Name} \to \Omega$ by:

$$\begin{array}{lcl} p(x{:}e){\cdot}x & = & e \\ p(x{:}e){\cdot}y & = & p{\cdot}y \quad \text{, if } x \neq y \end{array}$$

Obviously, if $x \neq y$ then $p(x{:}e)(y{:}f) = p(y{:}f)(x{:}e)$. We now define Ω and M as follows:

$$\begin{array}{lcl} \Omega & = & \mathsf{Exp}/\simeq \\ M{\cdot}\overline{x}{\cdot}p & = & p{\cdot}x \\ M{\cdot}(x \dashv E){\cdot}p & = & z \dashv M{\cdot}E{\cdot}\,p(x{:}\overline{z}) \quad \text{(note)} \\ M{\cdot}(E \odot F){\cdot}p & = & M{\cdot}E{\cdot}p \bullet M{\cdot}F{\cdot}p \end{array}$$

note: z must satisfy:
$(\forall y, e : y \neq x \wedge fv{\cdot}E{\cdot}y : p{\cdot}y(z \triangleleft e) = p{\cdot}y)$
□

Thus defined, function M has the following properties. Because we mention these properties, and the following theorem, for the sake of illustration only, we omit their (somewhat laborious) proofs here:

(3.0) $(\forall x : fv{\cdot}E{\cdot}x : p{\cdot}x = q{\cdot}x) \;\Rightarrow\; M{\cdot}E{\cdot}p = M{\cdot}E{\cdot}q$

(3.1) $M{\cdot}((x \dashv E) \odot F) \;=\; M{\cdot}(E(x \triangleleft F))$

(3.2) $(\forall x :: p{\cdot}x = \overline{x}) \;\Rightarrow\; M{\cdot}E{\cdot}p = [\![\, E \,]\!]$

With these properties we can now prove:

Theorem: $(\forall\, E, F ::\; E \simeq F \;\equiv\; M{\cdot}E = M{\cdot}F\,)$

proof:

"⇒" : Because its definition is compositional, function M is a homomorphism with respect to the term constructors. So, the equivalence relation induced by M is congruent with the term constructors; that is, this equivalence relation satisfies rule (γ). Of course, it also satisfies rule (δ). On account of the above property (3.1), this equivalence relation satisfies rule (β). From rule (ε) we now conclude:

$$(\forall\, E, F ::\; E \simeq F \;\Rightarrow\; M{\cdot}E = M{\cdot}F)$$

"⇐" : Define p such that $(\forall x :: p{\cdot}x = \overline{x})$, then we have:

$$\begin{array}{ll} & M{\cdot}E = M{\cdot}F \\ \Rightarrow & \quad \{ \text{ Leibniz } \} \\ & M{\cdot}E{\cdot}p = M{\cdot}F{\cdot}p \\ \equiv & \quad \{ \text{ property (3.2) } \} \end{array}$$

$$
\begin{array}{ll}
 & [\![E]\!] = [\![F]\!] \\
\equiv & \quad \{ \text{ definition of } [\![\cdot]\!] \ \} \\
 & E \simeq F \quad .
\end{array}
$$

□

* * *

The above comparison of the algebraic and denotational approaches shows that both can yield the same mathematical result. Indeed, an immediate corollary of the theorem is that $\mathsf{Exp}/\simeq$ and $(\mathsf{set}\, E :: M \cdot E)$ are isomorphic.

The above comparison also shows that the two approaches are different from a technical point of view: the algebraic approach is simpler, for two reasons. First, if the formalism will be used for equational reasoning then the rules (β) through (ε) are the ones that matter. In the denotational approach these rules must be proved as theorems —such as property (3.1), which justifies rule (β)—; this is a roundabout way compared to the algebraic approach where the rules simply are definitions. Second, the introduction of environments to represent mappings of free variables to values is a technical complication that, after all, is unnecessary.

The reason that we can do without environments is that substitution is so well-behaved: as we have seen, it is compatible with $\simeq$ and we have:

$$[\![E(x \lhd F)]\!] \;=\; [\![E]\!] \, (x \lhd [\![F]\!])$$

As a result, substitution can be considered as a semantical instead of a purely syntactical operation. Therefore, terms with free variables have well-defined values as well, such as $\bar{x}$ in:

$$[\![x \dashv \bar{x}]\!] \;=\; x \dashv [\![\bar{x}]\!]$$

Chapter 4

Consistency of the λ-calculus

4.0 Introduction

The λ-calculus is an algebra with an equality relation. In no way could such a calculus be considered as "inconsistent": in this setting the term just does not make sense. Yet, the (type free) λ-calculus has been called inconsistent for quite some time, wrongly so, but this was caused by how it was used: the λ-calculus was invented to provide a basis for the formalisation of logic, and the (naive) formalisation of propositional logic in the λ-calculus turns out to be inconsistent. This inconsistency, however, is not an inherent defect of the calculus itself: it only shows that this particular use of the calculus is incorrect.

aside: This is known under the name Curry's paradox [5, 8]: Curry's paradox shows that the properties of logical implication are incompatible with the fact that every term has fixed points.
□

The equality relation $\simeq$ might be such that $(\forall E, F :: E \simeq F)$; that is, all terms might have the same value. Although, logically, there would be nothing wrong about this, it would certainly make the calculus useless.

In order to be useful as a programming language the λ-calculus must "contain" at least the natural numbers. Hence, the λ-calculus must contain infinitely many values —that is: equivalence classes—. Fortunately, this is true; to prove this we need the so-called Church-Rosser theorem. To start with, however, we investigate in more general terms how sets can be proved to be infinite.

4.1 Intermezzo on infinite sets

A set is infinite if every finite subset of it is not the whole set. A more specific, and therefore sometimes easier, definition is: a set is infinite if it contains an infinite sequence all whose elements are different.

One way to construct such a sequence is as follows. With Ω for the set and with $b \in \Omega$ and $f \in \Omega \to \Omega$ we define a sequence $x_i\,(0 \leq i)$ by:

$$\begin{array}{lcl} x_0 & = & b \\ x_{i+1} & = & f \cdot x_i \quad , 0 \leq i \end{array}$$

We now derive what properties b and f should have in order that all elements of x be different; we do so by proving the latter by mathematical induction on i:

$$\begin{array}{ll} & (\forall j : 0 < j : x_0 \neq x_j) \\ \equiv & \quad \{ \text{ dummy transformation } j := j{+}1 \text{ ; definition of } x \text{ } \} \\ & (\forall j : 0 \leq j : b \neq f \cdot x_j) \\ \Leftarrow & \quad \{ \text{ assumption } (4.0) \text{ , see below } \} \\ & \mathsf{true} \quad , \end{array}$$

and:

$$\begin{array}{ll} & (\forall j : i{+}1 < j : x_{i+1} \neq x_j) \\ \equiv & \quad \{ \text{ dummy transformation } j := j{+}1 \text{ ; definition of } x \text{ } \} \\ & (\forall j : i < j : f \cdot x_i \neq f \cdot x_j) \\ \Leftarrow & \quad \{ \text{ assumption } (4.1) \text{ , see below } \} \\ & (\forall j : i < j : x_i \neq x_j) \quad . \end{array}$$

Here we have assumed that b and f satisfy:

(4.0) $\quad (\forall y :: b \neq f \cdot y)$

(4.1) $\quad (\forall y, z :: \; f \cdot y \neq f \cdot z \;\Leftarrow\; y \neq z \,)$

In words, the requirements are that f is *injective* but not *surjective*. Thus, we have derived the following lemma.

infinity lemma: for $b \in \Omega$ and $f \in \Omega \to \Omega$ we have:

$$(4.0) \;\wedge\; (4.1) \quad \Rightarrow \quad \text{“}\Omega \text{ is infinite”}$$

□

4.2 The λ-calculus is infinite

In order to be able to apply the infinity lemma we must choose a value b and a function f that satisfy (4.0) and (4.1). With:

$$K \;=\; x\!\dashv\! y\!\dashv\!\overline{x} \quad \text{, we have}$$

$$(4.2)\quad (\forall g,h :: \; K\bullet g\bullet h \;=\; g \;)$$

We now show that $b, f \;:=\; K\,,\; (K\bullet)$ does the job; first, we derive:

$$\begin{array}{ll}
 & K \;\neq\; K\bullet g \\
\Leftarrow & \quad \{ \text{ Leibniz, to prepare for application of } (4.2) \; \} \\
 & (\exists h :: \; K\bullet h \;\neq\; K\bullet g\bullet h \;) \\
\equiv & \quad \{ \;\; (4.2) \;\; \} \\
 & (\exists h :: \; K\bullet h \;\neq\; g \;) \\
\Leftarrow & \quad \{ \text{ Leibniz, for the same reason } \} \\
 & (\exists e,h :: \; K\bullet h\bullet e \;\neq\; g\bullet e \;) \\
\equiv & \quad \{ \;\; (4.2) \;\; \} \\
 & (\exists e,h :: \; h \;\neq\; g\bullet e \;) \\
\Leftarrow & \quad \{ \text{ instantiation } e := g \; \} \\
 & (\exists h :: h \;\neq\; g\bullet g \;) \\
\Leftarrow & \quad \{ \text{ generalisation } \} \\
 & (\forall e :: \; (\exists h :: h \neq e) \;) \quad .
\end{array}$$

The formula $(\forall e :: \; (\exists h :: h \neq e) \;)$ expresses that Ω is not a singleton set; this follows from the existence of at least two different values in Ω. With, for example,

$$I \;=\; x\!\dashv\!\overline{x} \quad \text{, we have} \quad I \;\neq\; K \quad ,$$

but to prove this we need the Church-Rosser theorem; this is the subject of the next section.

Using $I \neq K$, we have proved condition (4.0) of the infinity lemma, where we have chosen $b, f \;:=\; K\,,\; (K\bullet)$. Now we prove (4.1):

$$\begin{array}{ll}
 & K\bullet g \;\neq\; K\bullet h \\
\Leftarrow & \quad \{ \text{ Leibniz, to prepare for application of } (4.2) \; \} \\
 & (\exists e :: \; K\bullet g\bullet e \;\neq\; K\bullet h\bullet e \;) \\
\equiv & \quad \{ \;\; (4.2) \;\; \}
\end{array}$$

$$
\begin{array}{ll}
 & (\exists e :: \ g \neq h \ ) \\
\Leftarrow & \quad \{ \ \Omega \neq \phi \ \} \\
 & g \neq h \ \ .
\end{array}
$$

By the lemma, this establishes the infinity of the λ-calculus provided that

(4.3) $\quad x \dashv \overline{x} \ \neq \ x \dashv y \dashv \overline{x}$

4.3 The Church-Rosser theorem

In the previous section we have reduced the proof that the λ-calculus is infinite to the obligation to prove (4.3), which (in more syntactical terms) amounts to:

(4.4) $\quad \neg (\, x \dashv \overline{x} \ \simeq \ x \dashv y \dashv \overline{x} \,)$

(In this section the distinction between syntax and semantics is crucial.)

On Exp we define the relation $\rightsquigarrow$ ("reduces to") by means of the following rules, which are very similar to the rules for $\simeq$; the only difference lies in rules (δ) and (δ'): $\simeq$ is symmetric whereas $\rightsquigarrow$ is not.

(β') $\quad (x \dashv E) \odot F \ \rightsquigarrow \ E(x \triangleleft F)$, for all x, E, F
$(\gamma' 0)$ $\quad E \rightsquigarrow E' \ \Rightarrow \ x \dashv E \ \rightsquigarrow \ x \dashv E'$, for all x, E, E'
$(\gamma' 1)$ $\quad E \rightsquigarrow E' \ \Rightarrow \ E \odot F \rightsquigarrow \ E' \odot F$, for all E, E', F
$(\gamma' 2)$ $\quad F \rightsquigarrow F' \ \Rightarrow \ E \odot F \rightsquigarrow \ E \odot F'$, for all E, F, F'
(δ') $\quad \rightsquigarrow$ is reflexive and transitive
(ε') $\quad \rightsquigarrow$ is the strongest relation satisfying (β') through (δ')

The following theorem is known as the Church-Rosser theorem; for proofs we refer the reader to [2]. Although we have formulated the theorem as an equivalence, its "mathematical contents" is in the implication from left to right.

Theorem: $\quad (\forall E, F :: \ E \simeq F \ \equiv \ (\exists G :: E \rightsquigarrow G \wedge F \rightsquigarrow G) \,)$
□

Terms containing no subterms of the form $(x \dashv E) \odot F$ are called *normal forms*. Normal forms are characterised by the predicate nf defined recursively by:

$$
\begin{array}{lcl}
nf \cdot \overline{x} & \equiv & \mathsf{true} \\
nf \cdot (x \dashv E) & \equiv & nf \cdot E \\
nf \cdot (\overline{x} \odot G) & \equiv & nf \cdot G \\
nf \cdot ((x \dashv E) \odot G) & \equiv & \mathsf{false} \\
nf \cdot ((E \odot F) \odot G) & \equiv & nf \cdot (E \odot F) \wedge nf \cdot G
\end{array}
$$

Normal forms are *irreducible*; that is, we have:

(4.5) $(\forall E :: \ nf\cdot E \ \Rightarrow \ (\forall F :: E \rightsquigarrow F \ \equiv \ E = F) \ )$

exercise: prove (4.5)
□
remark: The converse to (4.5) is not true; that is, we do not have $nf\cdot E \ \Leftarrow \ (\forall F :: E \rightsquigarrow F \ \equiv \ E = F)$. As a counterexample, take $E := W \odot W$ where $W \ = \ x \dashv \bar{x} \odot \bar{x}$.
□

We are now ready for the following lemma, which is an immediate corollary of the Church-Rosser theorem and of (4.5).

lemma: For normal forms E, F: $\quad E \simeq F \ \equiv \ E = F \quad .$

proof:

$$
\begin{array}{ll}
 & E \simeq F \\
\equiv & \quad \{ \text{ Church-Rosser theorem } \} \\
 & (\exists G :: E \rightsquigarrow G \wedge F \rightsquigarrow G) \\
\equiv & \quad \{ \ E \text{ and } F \text{ are normal forms: (4.5) } \} \\
 & (\exists G :: E = G \wedge F = G) \\
\equiv & \quad \{ \text{ predicate calculus } \} \\
 & E = F \quad .
\end{array}
$$

□

In words the lemma states that equivalent normal forms are (syntactically) identical; so, (syntactically) different normal forms have different values. Now, the terms $x \dashv \bar{x}$ and $x \dashv y \dashv \bar{x}$ are normal forms and they are different; hence, by the lemma they have different values, which establishes (4.4) at the beginning of this section.

4.4 Evaluation and normal-order reduction

A different way to phrase the above lemma is: every equivalence class of $\simeq$ contains *at most* 1 normal form . We can not do better than this, as the following exercise shows.

exercise: show that $\neg (\exists E :: \ W \odot W \simeq E \wedge nf\cdot E)$, where $W \ = \ x \dashv \bar{x} \odot \bar{x}$.
□

The unique normal form in an equivalence class can be used as a canonical representation of (the value of) the terms in that class. An *evaluator* for the λ-calculus is a mechanism —machine, program, ... — that takes a term as input

and that produces the equivalent normal form as output, provided that such normal form exists. Formally, an evaluator can be modelled as a function *eval* of type $\mathsf{Exp} \to \mathsf{Exp}$, as follows:

specification: $(\forall E :: \ NF{\cdot}E \ \Rightarrow \ E \simeq eval{\cdot}E \ \wedge \ nf{\cdot}(eval{\cdot}E)\ )$,

where predicate *NF* is defined by:

$$(\forall E :: \ NF{\cdot}E \ \equiv \ (\exists F :: \ E \simeq F \ \wedge \ nf{\cdot}F)\ )$$

□

That some of the equivalence classes contain no normal form and, hence, that some terms have values that cannot be computed, is no reason for worry: the set of terms that do have computable values is rich enough to yield a useful programming language. Nevertheless, the terms without equivalent normal forms cannot simply be discarded: they are needed as subterms in terms that do have normal forms.

That functions *eval* satisfying the above specification exist follows from the following example; the definition given here is about the simplest possible, but its proof of correctness is far from trivial [2] and, considered as a mechanism, it is very inefficient. The algorithm defined here is known as Normal-Order-Reduction.

definition:

$$\begin{array}{lcl} eval{\cdot}E & = & \mathsf{if} \quad nf{\cdot}E \ \to \ E \\ & & [] \ \neg nf{\cdot}E \ \to \ eval{\cdot}(red{\cdot}E) \\ & & \mathsf{fi} \quad , \end{array}$$

where function *red* is defined by:

$$\begin{array}{lcl} red{\cdot}(x \dashv E) & = & x \dashv red{\cdot}E \\ red{\cdot}((x \dashv E) \odot F) & = & E(x \triangleleft F) \\ red{\cdot}(E \odot F) & = & \mathsf{if} \quad nf{\cdot}E \ \to \ E \odot red{\cdot}F \\ & & [] \ \neg nf{\cdot}E \ \to \ red{\cdot}E \odot F \\ & & \mathsf{fi} \quad , \text{ for } E \text{ not an abstraction} \end{array}$$

□

Chapter 5

Recursion

In the previous sections we have defined the λ-calculus and we have shown its consistency and implementability. From this section onwards we study its use as a programming language; hence, the distinction between syntax and semantics becomes less relevant: the values now are the objects of our interest and the terms are only textual representations of these values. We reflect this change of emphasis by adopting a simpler and more conventional notation for the λ-calculus:

instead of	$\bar{x}$	we use	x
instead of	$x \dashv E$	we use	$\lambda x : E$
instead of	$E \odot F$	we use	$E \cdot F$
instead of	$E(x \triangleleft F)$	we use	$E(x := F)$
instead of	$E \simeq F$	we use	$E = F$

remarks: The use of $=$ for semantical equality is common mathematical practice: usually $2{+}3 = 5$ does not mean equality of the symbol strings "2+3" and "5" but of their values. The use of $\cdot$ instead of $\odot$ reflects that the one and only purpose of $\odot$ is to represent function application, in the way discussed in Chapter 2.

□

5.0 Function definitions

In functional programming we use (function) definitions of the following shape:

(5.0) $\quad x \cdot y \cdot z \;=\; E \;,$

in which x, y, z are names and E is an expression. In E names x, y, z, as well as other names, may occur as free variables. How can we interpret (5.0) as a definition, that is, what if anything at all does it define?

The answer is that we consider formula (5.0) as an abbreviation of an equation in the unknown x, namely:

$$x: \quad (\forall y, z :: \ x \cdot y \cdot z \ = \ E \ )$$

This shows that x is the unknown and that y and z are only dummies; we call y and z the *parameters* of x. In this particular case x has 2 parameters; generally, the unknown may have any number of parameters. We call equations of this shape "admissible equations". This equation contains universal quantification over y, z; this may seem strange from a strictly formal point of view, because y, z are names in a λ-calculus expression. Nevertheless, this is harmless, as long as we consider this equation as an abbreviation of the following (formally correct) equation:

(5.1) $\quad x: \quad (\forall F, G : \neg fv \cdot F \cdot z \ : \ x \cdot F \cdot G \ = \ E(y := F)(z := G) \ )$

note: The restriction $\neg fv \cdot F \cdot z$ is necessary to prevent a name conflict with the substitution $z := G$; for any particular F this restriction can be met by a suitable renaming of z, which is a dummy.
□

The answer to the above question now is that (5.0) defines x as a solution of equation (5.1). This answer, however, raises two new questions: do equations like (5.1) always have solutions, and what if they have many —more than 1— solutions? To answer the first question we use the λ-calculus, whereas the second one can be answered in more general terms, without reference to the λ-calculus.

5.1 A solution to equation (5.1)

In the λ-calculus every admissible equation has at least one solution: we prove that equation (5.1) has a solution by constructing one. The pattern of reasoning is the same for any number of parameters. We proceed in a number of steps. First, assuming $\neg fv \cdot F \cdot z$, we derive:

$$\begin{array}{ll} & E(y := F)(z := G) \\ = & \quad \{ \ (\beta) \ \} \\ & (\lambda z : E(y := F)) \cdot G \\ = & \quad \{ \text{ substitution, using } \neg fv \cdot F \cdot z \ \} \\ & ((\lambda z : E)(y := F)) \cdot G \\ = & \quad \{ \ (\beta) \ \} \\ & (\lambda y : \lambda z : E) \cdot F \cdot G \quad , \end{array}$$

hence:

$$
\begin{array}{ll}
 & x\cdot F\cdot G \;=\; E(y:=F)(z:=G) \\
\equiv & \quad \{ \text{ as above } \} \\
 & x\cdot F\cdot G \;=\; (\lambda y:\lambda z:E)\cdot F\cdot G \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & x \;=\; \lambda y:\lambda z:E \quad .
\end{array}
$$

So, any solution of the following equation also solves (5.1):

$$x: \quad x \;=\; \lambda y:\lambda z:E$$

Because x may occur in E as a free variable, this equation is not entirely trivial. It is an instance of the more general equation:

$$x: \quad x \;=\; E \quad ,$$

for any term E in which x (and other names) may occur as free variables. We now show that this equation has solutions by calculating first as follows:

$$
\begin{array}{ll}
 & E \\
= & \quad \{ \text{ substitution } \} \\
 & E(x:=x) \\
= & \quad \{ \ (\beta) \ \} \\
 & (\lambda x:E)\cdot x \quad .
\end{array}
$$

The purpose of this calculation is to isolate the free occurrences of x: it shows that every term can be rewritten as $F\cdot x$, where F is a term without free occurrences of x. Therefore, without loss of generality, we can safely confine our attention to equations of the following shape:

(5.2) $\quad x: \quad x \;=\; F\cdot x \quad ,$ for term F with $\neg fv\cdot F\cdot x$

Solutions of (5.2) are usually called *fixed-points* of F; hence, the question whether our original equation (5.0) has solutions boils down to the question whether the expressions in the language, when interpreted as functions, have fixed-points. In the λ-calculus the answer is "yes", as the following calculation shows:

$$
\begin{array}{ll}
 & x \;=\; F\cdot x \\
\equiv & \quad \{ \text{ choose } x:=y\cdot y \text{ (to obtain more manipulative freedom) } \} \\
 & y\cdot y \;=\; F\cdot(y\cdot y) \\
\equiv & \quad \{ \text{ substitution } (z \text{ fresh}) \text{ and } (\beta) \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & y\cdot y \;=\; (\lambda z : F\cdot(z\cdot z))\cdot y \\
\Leftarrow & \quad \{\ \text{Leibniz}\ \} \\
 & y \;=\; \lambda z : F\cdot(z\cdot z) \quad .
\end{array}
$$

corollary: $(\lambda z : F\cdot(z\cdot z))\cdot(\lambda z : F\cdot(z\cdot z))$ is a solution of (5.2)
□
exercise: Solve $x, y :\ x = F\cdot x\cdot y \ \wedge\ y = G\cdot x\cdot y$
□

The λ-calculus offers even more: the operation of mapping a function to one of its fixed-points itself is a function that can be represented in the λ-calculus; calling that function Y, we have:

$$Y \;=\; \lambda f : (\lambda z : f\cdot(z\cdot z))\cdot(\lambda z : f\cdot(z\cdot z))$$

5.2 Which solution?

In the previous section we have shown that in the λ-calculus every "function" has at least one fixed-point; as a result, every admissible equation has at least one solution. In this section we study the question which solution is the intended one, whenever there are many. This question can be answered completely without reference to the λ-calculus.

The number of solutions of an equation depends, of course, on the universe in which the equation is considered.

example: We consider the following equation (with unknown f):

$$f :\quad f\cdot 0 = 3 \ \wedge\ (\forall i : i{\in}\mathsf{Nat} :\ f\cdot(i{+}1) = f\cdot i + 2\,)$$

In $\mathsf{Int}\to\mathsf{Int}$ this equation has infinitely many solutions. In $\mathsf{Nat}\to\mathsf{Int}$, however, the solution is unique.
□

The question "which solution?" admits of two obvious answers. The first, most commonly used, answer is to impose a suitable partial order onto the set of all solutions and to pick the *smallest* one. Here, "suitable" means that the resulting ordered set is a complete lattice or a complete partial order. The second answer is to decide not to care and to pick an *arbitrary* solution. These two approaches are not so different as they may seem at first. Both are based on a kind of pessimistic view, and although the former allows stronger propositions to be proved —because Knaster-Tarski's theorem can be used— the latter has the charm of being simpler. Moreover,

the latter one perfectly fits into a style of programming that uses *modularisation* or *refinement*.

The use of least solutions and, in the case of recursion, least fixed-points is well-known. Here, we investigate the "we-do-not-care"-interpretation in somewhat greater detail, by formulating proof rules for it.

5.3 A general recursion theorem

We study a generalised case first and then we specialise this for the case of recursive definitions.

Let Ω be a set and let R be a predicate on Ω. Suppose we wish to construct a *definition* —program, expression, mechanism, ... — of a value X in Ω satisfying:

$$R{\cdot}X$$

In such a case we call $R{\cdot}X$ the *specification* of X, to stress that it is a *desired* property of X. A specification is a coin with two faces. Whenever we use X the specification is all we (care to) know about X: from the viewpoint of its use a specification has the same status as a definition or an axiom. On the other hand, when we construct a definition for X the specification represents a proof obligation: from this point of view a specification has the same status as an equation to be solved or as a theorem to be proved. Thus, a specification forms the interface between the use of a mechanism and its implementation.

This shows that, whenever a given specification admits many solutions, the user of a particular solution shall never complain about its particularities: the specification is supposed to capture all the user's wishes (and nothing more). So, that a specification admits many solutions means that the user did not care in the first place, nor should he start caring afterwards as well.

Now suppose that we come up with a solution X that satisfies $Q{\cdot}X$, for some predicate Q on Ω. What, then, must we prove in order that X meets its specification $R{\cdot}X$? When we already know $Q{\cdot}X$ and when this is all we know about X, then the only thing we can do is prove:

$$(5.3) \quad (\forall x :: R{\cdot}x \Leftarrow Q{\cdot}x)$$

This rule can be considered as the basis of all proof obligations in (functional) programming: to prove that a proposed solution satisfies a specification, it suffices to show that its definition (viewed as a predicate) implies the specification. Of course, when Q is used to define X, we must also prove that such X exists, but that is a separate proof obligation, which can be rendered as follows:

$$(\exists x :: Q{\cdot}x)$$

If, however, $Q{\cdot}X$ is an admissible equation, as discussed in Section 5.1, the existence of solutions is guaranteed.

We now use rule (5.3) to derive a few more specific proof rules, namely for the case of recursive definitions; that is, we consider predicates Q of the following form, where F is some function of type $\Omega \to \Omega$:

$$Q{\cdot}x \quad \equiv \quad x = F{\cdot}x$$

Thus, rule (5.3) becomes:

(5.4) $\quad (\forall x :: R{\cdot}x \;\Leftarrow\; x = F{\cdot}x\,)$

This rule can be considered as the prototype proof rule for recursive definitions. A few very trivial special instances of (5.4) are:

$(\forall x :: R{\cdot}x\,)$
$(\forall x :: R{\cdot}(F{\cdot}x)\,)$
$(\forall x :: R{\cdot}(F{\cdot}(F{\cdot}x))\,)$
et cetera .

These trivial instances exhibit a pattern; indeed, by mathematical induction on the natural numbers we can derive the following strengthening of (5.4) :

(5.5) $\quad (\forall x :: (\exists k : 0 \le k : R{\cdot}(F^k{\cdot}x)\,)$

In functional programmer's jargon this rule states that, whatever the initial value of x , a finite number of applications of F suffices to obtain a value that satisfies the specification. In a way, the elements of the sequence x , $F{\cdot}x$, $F{\cdot}(F{\cdot}x)$, $\cdots$ can be considered as successive approximations of F's fixed-points. Rule (5.5) then requires that every sequence of approximations contains an element satisfying R .

Unfortunately, rule (5.5) is too strong to be of any practical use. Recursive definitions are mainly used to define functions, or similar objects like infinite lists; usually, the number of *unfoldings* required to obtain a value satisfying the specification depends on the argument of the function. Therefore, we need a more sophisticated rule that takes into account the properties of an additional domain.

example: We consider the following function F :

$$F{\cdot}f \;=\; \lambda i :\; \mathsf{if}\;\; i = 0 \;\to\; 3 \;\;[]\;\; i \ge 1 \;\to\; f{\cdot}(i-1)+2 \;\;\mathsf{fi}$$

Then we have:

$$f = F{\cdot}f \quad \Rightarrow \quad (\forall i : i \in \mathsf{Nat} :\; f{\cdot}i = 2{*}i + 3\,) \quad ,$$

but this cannot be proved by means of rule (5.5) ; yet, this proof is simple if only we use mathematical induction on i .

□

The following proof rule is only applicable to specifications of a certain shape, but it gives a weaker condition for F. We assume that C is a set on which $<$ is a binary relation that is *well-founded*, which means that properties about C may be proved by mathematical induction. Furthermore, we assume that Q is a predicate on $C \times \Omega$, such that:

(5.6) $(\forall x :: R{\cdot}x \equiv (\forall c : c{\in}C : Q{\cdot}c{\cdot}x))$

In what follows dummies b, c range over C. We now derive:

$$
\begin{array}{ll}
 & (\forall x : x = F{\cdot}x : R{\cdot}x) \\
\equiv & \quad \{\ (5.6)\ \} \\
 & (\forall x : x = F{\cdot}x : (\forall c :: Q{\cdot}c{\cdot}x)) \\
\Leftarrow & \quad \{\ (C, <) \text{ is well-founded: mathematical induction}\ \} \\
 & (\forall x : x = F{\cdot}x : (\forall c :: Q{\cdot}c{\cdot}x \Leftarrow (\forall b : b < c : Q{\cdot}b{\cdot}x))) \\
\equiv & \quad \{\ \text{Leibniz}\ \} \\
 & (\forall x : x = F{\cdot}x : (\forall c :: Q{\cdot}c{\cdot}(F{\cdot}x) \Leftarrow (\forall b : b < c : Q{\cdot}b{\cdot}x))) \\
\Leftarrow & \quad \{\ x = F{\cdot}x \text{ has played its role: generalisation; unnesting dummies}\ \} \\
 & (\forall x, c :: Q{\cdot}c{\cdot}(F{\cdot}x) \Leftarrow (\forall b : b < c : Q{\cdot}b{\cdot}x))\ .
\end{array}
$$

Thus, we have derived the following rule for specifications R satisfying (5.6).

recursion theorem: Every fixed-point X of F satisfies $(\forall c :: Q{\cdot}c{\cdot}X)$, for functions F satisfying:

$$(\forall x, c :: Q{\cdot}c{\cdot}(F{\cdot}x) \Leftarrow (\forall b : b < c : Q{\cdot}b{\cdot}x))$$

□

For the special case that C is the natural numbers, with the usual ordering, the premiss of this theorem can be reformulated as:

(5.7) $(\forall x, i :: Q{\cdot}0{\cdot}(F{\cdot}x) \ \wedge\ (Q{\cdot}(i{+}1){\cdot}(F{\cdot}x) \Leftarrow Q{\cdot}i{\cdot}x))$

These proof rules are not very deep, but they provide a useful separation of concerns: these rules give proof obligations with respect to F for drawing conclusions about the fixed-points of F. That these rules are only applicable to specifications of a particular shape is harmless, because specifications of functions almost always have that shape.

In practical situations it is not necessary to introduce a name like F when we wish to prove properties from a recursive definition. In such a case, the recursive definition itself can be used to formulate the proof obligations.

example: In practice the recursive definition from the previous example is cast in the following form; so, in this form f represents *some* fixed-point of function F from the previous example:

$$(5.8) \quad f\cdot 0 = 3 \;\wedge\; (\forall i : i{\in}\mathsf{Nat} : \; f\cdot(i{+}1) = f\cdot i + 2\,)$$

Then we have:

$$(\forall i : i{\in}\mathsf{Nat} : \; f\cdot i = 2{*}i + 3\,) \quad ,$$

and we prove this by mathematical induction on i, as follows:

$$\begin{array}{ll} & f\cdot 0 \\ = & \quad \{ \text{ definition } (5.8) \text{ of } f \text{ } \} \\ & 3 \\ = & \quad \{ \text{ algebra } \} \\ & 2{*}0 + 3 \quad , \end{array}$$

and:

$$\begin{array}{ll} & f\cdot(i{+}1) \\ = & \quad \{ \text{ definition of } f \text{ } \} \\ & f\cdot i + 2 \\ = & \quad \{ \text{ specification of } f\text{, by induction hypothesis } \} \\ & 2{*}i + 3 + 2 \\ = & \quad \{ \text{ algebra } \} \\ & 2{*}(i{+}1) + 3 \quad . \end{array}$$

Notice how the "base" and the "step" in this proof correspond to the two conjuncts of rule (5.7). This kind of reasoning is correct because all we have used about f is (5.8), and (5.8) is (exactly) what all solutions to the equation have in common.

□

Chapter 6

On types

6.0 Introduction

The notion of *type* is particularly important in connection with functions: we say that a function F has type $X \to Y$ if and only if for all x of type X the application $F{\cdot}x$ has type Y. Because Y does not depend on x, the proposition that $F{\cdot}x$ has type Y is not the whole story about the value $F{\cdot}x$: having type Y is a common property of *all* values of F on the domain X. Generally, a type is a common property of (the elements of) a collection of objects.

In order that types be useful in mathematical discussions, we need rules for deriving the types of the objects under discussion —type introduction— and rules for using information about the types of objects —type elimination—. Together, rules for type introduction and elimination are known as type inference rules.

In this chapter I wish to contrast two views on types, which I shall call the "syntactical view" and the "semantical view". In particular, I wish to shed some light on the limitations of the syntactical view and I wish to argue that, from the point of view of programming methodology, the semantical view is to be preferred. Thus, one may well wonder why the syntactical view is so popular nowadays. (One explanation might be found in the influence of the logicians on computing science, with their concern for syntax and decidability, and their fear of Russell's paradox.)

Two remarks are in order here. First, it should be clear that in the above I am mainly referring to the use of types in programming and programming languages. Second, it should be clear that I am *not* challenging (the usefulness of) the type concept itself; I only want to discuss the forms in which the concept can be cast.

6.1 The syntactical view

In the syntactical view types are properties of expressions, as syntactical entities, independent of the values these expressions may assume. The type inference rules are

syntactical rules; they are formulated in such a way that the type(s) of an expression can be derived in an entirely mechanical way from the (syntactical) structure of that expression and its context. (For example, it is mechanically decidable that $x{+}2$ has type Int if this expression occurs in a context where x has type Int.) This is known as *strong typing*.

Expressions for which no type can be derived by means of the inference rules are considered as "meaningless" and are considered not to belong to the language. As a consequence, a great many (but not all) "meaningless" expressions are excluded from the language by syntactical means. (A trivial example of such an expression is $\mathsf{true}{+}2$.)

The syntactical approach has two advantages. First, it provides some protection against programming errors, because the type inference mechanism can detect violations of the typing rules. One might argue that a careful and competent programmer, who is well aware of the role types play in his programs, will not make such errors, but many clerical errors, like writing errors, manifest themselves as violations of the typing rules. Thus, any (not too elaborate) mechanism that contributes to the robustness of programs is welcome, even if it provides a partial protection only. Second, by exploiting the type information compilers (possibly) can generate more efficient code than without this information.

These two advantages pertain to robustness and efficiency, which are *engineering concerns*; as such they are valuable, but they bear no relevance to the methodology of programming, where methods for designing correct and efficient programs are studied, not techniques for coping with clerical errors. Notice that I have not mentioned as an advantage that, as folklore has it, the syntactical notion of types is indispensible for a proper definition of the semantics of programming languages, for the simple reason that this is not true. And this is about the best I can say about the virtues of the syntactical view.

* * *

We now investigate some of the limitations of the syntactical view. It is well-known that for a general-purpose programming language, termination of computations is undecidable. In particular, it is generally undecidable whether or not the value of an expression is an integer. Consequently, the syntactical notion "having type Int" cannot be the same as the semantical notion "having an integer value".

For example, the value of $x \mathbin{\mathsf{div}} y$ is an integer only if $y \neq 0$, which is a semantical condition. As a more complicated example, for F of type $\mathsf{Int} \to \mathsf{Int}$ and for x a fixed-point of F we may not conclude that x is an integer. Take, for instance, $F{\cdot}y = 1{+}y$, then F has no integer fixed-points. Now take $G{\cdot}y = 2{-}y$, then G does have a unique integer fixed-point, but the non-integer fixed-points of F may be fixed-points of G as well; as a matter of fact, models exist in which this is true indeed.

Thus, to save the syntactical view we have to admit a little bit of cheating: syntactical conclusions about the type of an expression give information about the value of that expression only modulo *well-definedness*, whereas well-definedness is not syntactically decidable[0].

A more fundamental disadvantage of the syntactical view is that it induces a rather artificial dichotomy between syntactically decidable and semantical properties. For example, whereas "integer" is a type, "even integer" and "prime number" are not; whereas "list" is a type, "finite list" and "infinite list" cannot be distinguished syntactically; whereas "integer list" is a type, "increasing integer list" is not. As a result, that an expression is an infinite integer list requires proof, but once this proof has been given the verification that the expression is an integer list is superfluous.

Finally, a minor disadvantage is that the typing rules not only exclude meaningless expressions from the language, but also possibly meaningful ones. This is relevant when the language is also used for the implementation of data structuring constructs. For example, for boolean B and integer E the expression

$$\lambda i : \ \mathsf{if}\ i = 0 \to B \ [] \ i = 1 \to E \ \mathsf{fi} \quad ,$$

is a perfectly defensible implementation of the pair consisting of boolean B and integer E. In most type inference systems this expression is untypable, whereas in the semantical view it has type

$$(\{0\} \to \mathsf{Bool}) \quad \cup \quad (\{1\} \to \mathsf{Int}) \quad .$$

6.2 The semantical view

In the semantical view types themselves can, of course, not be used for the definition of the semantics of the language, so we must assume that the semantics has been defined first. In this section we use Ω for the universe (or domain) of values for the expressions of the language. (For example, in the (type free) λ-calculus Ω is the set of all equivalence classes of terms that are congruent modulo convertibility.) This universe will be rather *amorphous* and types can now be used to impose some structure onto it. In this section we elaborate upon this view.

Properties of elements of Ω can be specified by means of predicates on Ω. We now define a type to be a predicate on Ω. The idea is that a statement like "x has type T" is a (logical) proposition as good as any other; hence, by abstraction from x, "has type T" is a predicate as good as any other. Thus we obtain a nice

[0]Strictly speaking there is a way out: when the program would be accompanied by a completely formalised specification and correctness proof, then all of its relevant properties would be mechanically verifiable; the type checker would then become a proof verifier. In that case, the distinction between types and other properties would disappear (as in the semantical view), but it remains to be seen whether this ultimate form of robustness is worth the price of complete formalisation.

homogeneous view in which there is no formal difference between types and other predicates or specifications. As a result, the type inference rules now are just special instances of the common proof rules; we shall illustrate this later. Moreover, the inherent dichotomy of the syntactical view is absent here. For example, when we have proved that an expression's value is an increasing integer list, then it also has type "integer list", for the simple reason that the increasing integer lists form a subset of the integer lists. Because of the (standard) one-to-one correspondence between the predicates on Ω and the subsets of Ω, a type is also a subset of Ω, with the obvious interpretation: a type is the subset of all elements having that type.

An immediately obvious advantage of this approach is that *subtypes* pose no problems whatsoever. Type T is a subtype of type U whenever $[T \Rightarrow U]$, or, in set notation, $T \subseteq U$. For example, Nat is a subtype of Int in the ordinary sense that every natural number also is an integer. A single value always has many types. For every value x the *strongest* type containing x is, of course, the point predicate —singleton set— $(=x)$.

To illustrate the viability of the semantical view, we discuss a few common type constructors and we shall show that only a few modest additional assumptions about Ω are sufficient. Remember that types are predicates, so we denote "x has type T" by $T{\cdot}x$. In this interpretation Ω —that is, the predicate $(\in\Omega)$— represents the universal type of all values in our language: that a value has type Ω gives no information at all about that value. We exploit this when we discuss polymorphic data structures: a polymorphic list, for example, will be a list over Ω.

Let $\odot$ be some fixed, but as yet unspecified, binary operator in $\Omega \times \Omega \to \Omega$. For types X, Y we *define* the function-type $X \to Y$ as follows, for every f in Ω:

$$(6.0) \quad (X \to Y){\cdot}f \;\equiv\; (\forall x :: X{\cdot}x \;\Rightarrow\; Y{\cdot}(f \odot x))$$

Thus, the binary operator $\odot$ represents function application; the function involved is $(f\odot)$, although in common parlance we mostly refer to "function f". Whether or not this yields an interesting game depends, of course, on the properties of $\odot$, but for understanding this definition these properties are irrelevant. Notice that (6.0) does *not* define $X \to Y$ as the set of *all* functions from X to Y: the type $X \to Y$ now denotes the subset of those elements in Ω that can be interpreted as functions from X to Y when $\odot$ is interpreted as function application. (As I have argued in Chapter 2, this is the best possible for any programming language[1]. As a result, self applications like $x \odot x$ present no problems, because the types X and $X \to Y$ need not be disjoint. It is all a matter of interpretation now.)

For example, when $\odot$ is λ-calculus application we can derive, for terms E, F and dummy x:

[1] Ω is countable, whereas the set of all functions from X to Y is uncountable.

$$
\begin{array}{ll}
 & (X \to Y) \cdot (\lambda x : E) \\
\equiv & \quad \{ \text{ definition } (6.0) \text{ of } \to \} \\
 & (\forall F :: \ X \cdot F \ \Rightarrow \ Y \cdot ((\lambda x : E) \cdot F) \ ) \\
\equiv & \quad \{ \ \lambda\text{-calculus } \} \\
 & (\forall F :: \ X \cdot F \ \Rightarrow \ Y \cdot (E(x := F)) \ ) \\
\equiv & \quad \{ \text{ dummy transformation } x := F \ \} \\
 & (\forall x :: \ X \cdot x \ \Rightarrow \ Y \cdot E \ ) \ .
\end{array}
$$

Thus we have *derived*, from our general definition of $\to$, the following type inference rule for λ-calculus abstractions:

type inference for abstractions:

$$(X \to Y) \cdot (\lambda x : E) \quad \equiv \quad (\forall x :: \ X \cdot x \ \Rightarrow \ Y \cdot E \,)$$

□

Another elementary type constructor is the cartesian product operator $\times$; in our approach it can be defined as follows, for types X, Y and for all z in Ω:

$$(X \times Y) \cdot z \ \equiv \ (\exists x, y :: \ X \cdot x \wedge Y \cdot y \wedge z = \mathsf{pair} \cdot x \cdot y \ ) \ ,$$

where pair is a function in $\Omega \to \Omega \to \Omega$ for which functions left and right in $\Omega \to \Omega$ exist with:

$$(\forall x, y :: \ \mathsf{left} \cdot (\mathsf{pair} \cdot x \cdot y) \ = \ x \ \wedge \ \mathsf{right} \cdot (\mathsf{pair} \cdot x \cdot y) \ = \ y \ ) \ .$$

(In the λ-calculus terms pair, left, and right exist that satisfy $\mathsf{left} \cdot (\mathsf{pair} \cdot x \cdot y) = x$ and $\mathsf{right} \cdot (\mathsf{pair} \cdot x \cdot y) = y$. Thus, cartesian product can be implemented in the λ-calculus. See Chapter 7.)

From this definition of $\times$ (and the properties of pair, left, right) the following inference rules for product types can be derived:

type inference for products:

$$
\begin{array}{l}
(\forall x, y :: \ X \cdot x \wedge Y \cdot y \ \Rightarrow \ (X \times Y) \cdot (\mathsf{pair} \cdot x \cdot y) \ ) \\
(\forall z :: \ (X \times Y) \cdot z \ \Rightarrow \ X \cdot (\mathsf{left} \cdot z) \ ) \\
(\forall z :: \ (X \times Y) \cdot z \ \Rightarrow \ Y \cdot (\mathsf{right} \cdot z) \ )
\end{array}
$$

□

In this way the type $X \times Y$ contains precisely the values $\mathsf{pair} \cdot x \cdot y$, for all x and y of type X and Y, and *nothing else*. As a result we have

$$(\forall z :: \ (X \times Y) \cdot z \ \Rightarrow \ \mathsf{pair} \cdot (\mathsf{left} \cdot z) \cdot (\mathsf{right} \cdot z) \ = \ z \ )$$

Questions like whether or not $\mathsf{pair}\cdot\bot\cdot\bot = \bot$ are irrelevant here, for the simple reason that $\bot$ plays no role in this approach.

6.3 Polymorphism

Polymorphism is possible without any problems. For example, the prototype of a polymorphic function is the identity function I; from our definition of $\rightarrow$ it follows immediately that I has type $X \rightarrow X$, for all types X. Furthermore, in Chapter 7 we show how data structures with "elements", such as tuples, lists, trees, can be represented in a truly polymorphic way: when we wish to discuss their structure only, we consider them as structures over Ω, which amounts to ignoring all information about (the types of) their elements. For example, every type is a subtype of Ω, so the type of lists over any type is a subtype of the type of lists over Ω. Hence, every function that is well-defined on the lists over Ω, is also well-defined on the lists over any particular type. In this way, polymorphic properties of lists are properties of the lists over Ω; such properties are inherited by lists over any type. As a matter of fact, in this approach it is not even necessary to require all elements of a list to have the *same* type.

6.4 On the shape of inference rules

In Section 6.2 we have derived the following inference rule for abstractions.

type inference for abstractions:

$$(X \rightarrow Y)\cdot(\lambda x : E) \quad \equiv \quad (\forall x :: X\cdot x \;\Rightarrow\; Y\cdot E)$$

□

This rule establishes the equivalence of two predicates; by splitting the equivalence into two implications we can rewrite the left-to-right implication as follows:

$$\begin{array}{ll} & (X \rightarrow Y)\cdot(\lambda x : E) \quad \Rightarrow \quad (\forall x :: X\cdot x \;\Rightarrow\; Y\cdot E) \\ \equiv & \quad \{\ (P \Rightarrow)\ \text{distributes over}\ \forall\ \} \\ & (\forall x :: (X \rightarrow Y)\cdot(\lambda x : E) \quad \Rightarrow \quad (X\cdot x \;\Rightarrow\; Y\cdot E)\ ) \\ \equiv & \quad \{\ \text{trading}\ \} \\ & (\forall x :: (X \rightarrow Y)\cdot(\lambda x : E) \;\wedge\; X\cdot x \quad \Rightarrow \quad Y\cdot E\ ) \\ \equiv & \quad \{\ \text{dummy transformation}\ x := F\ \} \\ & (\forall F :: (X \rightarrow Y)\cdot(\lambda x : E) \;\wedge\; X\cdot F \quad \Rightarrow \quad Y\cdot(E(x := F))\ ) \quad . \end{array}$$

In this shape this implication is known as the *elimination rule* for $\rightarrow$, whereas the reverse implication of the above equivalence, that is

$$(X \rightarrow Y) \cdot (\lambda x : E) \quad \Leftarrow \quad (\forall x :: X \cdot x \;\Rightarrow\; Y \cdot E) \quad ,$$

is known as the *introduction rule* for $\rightarrow$. In the traditional natural-deduction style these rules take the following shape:

$$\frac{\Gamma, x : X \;\vdash\; E : Y}{\Gamma \;\vdash\; (\lambda x.E) : X \rightarrow Y} \qquad\qquad \frac{\Gamma \;\vdash\; (\lambda x.E) : X \rightarrow Y \quad , \quad \Gamma \;\vdash\; F : X}{\Gamma \;\vdash\; E(x := F) : Y}$$

Apart from the abundance of redundant —as I showed in [10]— Γ's, the mere shape of these two rules obfuscates that they are actually the two halves of a simple equivalence. This illustrates for the n-th time, $n \gg 1$, the clumsiness of natural deduction when it comes to effective reasoning.

Chapter 7

Towards a functional-programming language

In this chapter we show how a functional-programming language can be based upon the λ-calculus. For the interpretation of (recursive) definitions we adopt the views developed in Chapter 5, whereas for types we use the semantical view from Chapter 6. We shall discuss the representation of booleans, guarded expressions, product and sum types, naturals and integers, tuples, and finite and infinite lists.

To what extent syntactic sugar is added to the language to support these concepts is mainly a matter of pragmatic considerations and, perhaps, of personal taste. Because here we are only interested in feasibility, we shall introduce as little additional syntax as possible.

This chapter is mainly included for the sake of completeness, and to demonstrate how utterly straightforward the application of the ideas from the previous chapters is.

7.0 Function definitions

In Chapter 5 we have already shown how recursive definitions can be interpreted in the λ-calculus. Here we recall the main results from that chapter.

In functional programming we use (function) definitions of the following shape:

(7.0) $\quad x \cdot y \cdot z \;=\; E \;\;,$

in which x, y, z are names and E is an expression. In E names x, y, z, as well as other names, may occur as free variables. Such as a definition is an abbreviation of the equation

(7.1) $\quad x : \;\; (\forall y, z :: \; x \cdot y \cdot z \;=\; E \,) \;\;,$

and it defines x as *some* —otherwise unspecified— solution of this equation; we have shown that in the λ-calculus this equation has solutions and we adopt the convention that any solution is as good as any other. Hence, all we know and all we shall use about x is (7.1). Dummies y, z are called the *parameters* of x; generally, a function can have any number of parameters. (Functions with 0 parameters are usually called *constants*.)

7.1 The type Bool

For the implementation of the booleans and their operations it suffices to have available the following *boolean primitives*, in terms of which all boolean operations can be defined:

> constants true and false, and a 3-argument function if satisfying
> $(\forall x, y :: \mathsf{if}{\cdot}\mathsf{true}{\cdot}x{\cdot}y = x)$ and $(\forall x, y :: \mathsf{if}{\cdot}\mathsf{false}{\cdot}x{\cdot}y = y)$

By introducing these primitives we keep the interface between the datatype to be implemented and the λ-calculus as thin as possible. Notice that we did not even require true and false to be different, as this is implied by the specification:

$$
\begin{array}{ll}
 & \mathsf{true} \neq \mathsf{false} \\
\Leftarrow & \quad \{ \text{ Leibniz (what else?) } \} \\
 & (\exists x, y :: \mathsf{if}{\cdot}\mathsf{true}{\cdot}x{\cdot}y \neq \mathsf{if}{\cdot}\mathsf{false}{\cdot}x{\cdot}y) \\
\equiv & \quad \{ \text{ specification of } \mathsf{if} \} \\
 & (\exists x, y :: x \neq y) \\
\equiv & \quad \{ \Omega \text{ has at least two elements (Section 4.2) } \} \\
 & \mathsf{true} \quad .
\end{array}
$$

Solutions to the above specification are easily calculated; for example:

$$
\begin{array}{ll}
 & \mathsf{if}{\cdot}\mathsf{true}{\cdot}x{\cdot}y \\
= & \quad \{ \text{ specification of } \mathsf{if} \} \\
 & x \\
= & \quad \{ \lambda\text{-calculus } \} \\
 & (\lambda y : x){\cdot}y \\
= & \quad \{ \lambda\text{-calculus } \}
\end{array}
$$

$$(\lambda x : \lambda y : x) \cdot x \cdot y \quad .$$

The driving force behind this calculation is the desire to bring the expression in the right-hand side of if's specification into a form similar to its left-hand side: now we conclude that the first conjunct of if's specification is implied by:

$$\mathsf{if} \cdot \mathsf{true} \;=\; \lambda x : \lambda y : x$$

In a similar way the second conjunct of the specification can be strengthened to:

$$\mathsf{if} \cdot \mathsf{false} \;=\; \lambda x : \lambda y : y$$

Solutions to these two equations are easily obtained. We choose:

$$\mathsf{if} \;=\; \lambda z : z \quad ,$$

in which case we have no choice but to define

$$\begin{array}{l} \mathsf{true} \;=\; \lambda x : \lambda y : x \\ \mathsf{false} \;=\; \lambda x : \lambda y : y \end{array}$$

We now define the datatype Bool as the set $\{\mathsf{true}, \mathsf{false}\}$. In terms of the primitives all boolean operations can be defined; for example:

$$\begin{array}{lcl} \mathsf{not} \cdot x & = & \mathsf{if} \cdot x \cdot \mathsf{false} \cdot \mathsf{true} \\ \mathsf{and} \cdot x \cdot y & = & \mathsf{if} \cdot x \cdot y \cdot \mathsf{false} \\ \mathsf{or} \cdot x \cdot y & = & \mathsf{if} \cdot x \cdot \mathsf{true} \cdot y \\ \mathsf{eqv} \cdot x \cdot y & = & \mathsf{if} \cdot x \cdot y \cdot (\mathsf{not} \cdot y) \end{array}$$

and so on ...

These operations are functions on Bool; that is, not has type $\mathsf{Bool} \to \mathsf{Bool}$, whereas $\mathsf{and}, \mathsf{or}, \mathsf{eqv}, \ldots$ have type $\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$; moreover, all the usual properties of these functions are derivable from these definitions. As usual, we call an expression whose value is in Bool a *boolean expression*; what is unusual is that establishing such a fact requires proof going beyond a mere syntactical check.

Finally, we remark that the expressions for true and false have normal forms: they even are normal forms. As a consequence, evaluation of any expression of type Bool terminates —see Section 4.4—. This is a nice consequence of our decision to treat types semantically —see Section 6.2—. Thus, in functional programming we never need to give separate termination proofs: termination of an expression's evaluation is always implicit in the expression's type. This leaves us, of course, with the obligation to prove that the expression has the right type, but the latter has much less operational connotations than the notion of termination. In this way, functional programming becomes an entirely non-operational game[0].

In the following sections we shall use the conventional notation for the boolean operations, such as $\neg$ for not, $\wedge$ for and, et cetera.

[0] as it should be, but not always is.

7.2 Guarded expressions

In this section we introduce guarded expressions, mainly because we need them in the following sections. A guarded expression is an expression of the shape $\mathsf{if}\ B \to E \,[]\, C \to F\ \mathsf{fi}$, where B and C are boolean expressions called *guards*, and where E and F are expressions called *alternatives*. That B and C are boolean is a proof obligation that we leave implicit in the following discussions. For guarded expressions we use the following proof rule as their specification:

first proof rule for guarded expressions: For every predicate R:

$$R\cdot(\mathsf{if}\ B \to E \,[]\, C \to F\ \mathsf{fi})$$

follows from the conjunction of the following three conditions:

$$\begin{array}{lcl} B & \vee & C \\ B & \Rightarrow & R\cdot E \\ C & \Rightarrow & R\cdot F \end{array}$$

□

This rule does not require B and C to be disjoint and indeed expressions exist for which this rule does not give complete information about the values of these expressions. For example, the strongest conclusion that can be drawn about $\mathsf{if}\ \mathsf{true} \to 2 \,[]\, \mathsf{true} \to 3\ \mathsf{fi}$ is that its value is either 2 or 3. This does not mean, however, that the language is *nondeterministic*. On the contrary, it is only the proof rule, as a specification, that gives incomplete information about the expression. By choosing the rule this way we avoid *overspecification*: in this form the rule captures all that is relevant for using guarded expressions, and nothing else. For example, now the expressions $\mathsf{if}\ B \to E \,[]\, C \to F\ \mathsf{fi}$ and $\mathsf{if}\ C \to F \,[]\, B \to E\ \mathsf{fi}$ are equivalent in the sense that all we can prove about the one can be proved about the other: by means of the proof rule they cannot be distinguished; nevertheless, they may have different values.

To obtain complete information about the value of a guarded expression, one should prove in addition that all alternatives have the same value. This is represented by the following special case of the proof rule, obtained by substituting $(=X)$ for R:

second proof rule for guarded expressions: For all X:

$$\mathsf{if}\ B \to E \,[]\, C \to F\ \mathsf{fi}\ \ =\ X$$

follows from the conjunction of the following three conditions:

$$\begin{array}{lcl} B & \vee & C \\ B & \Rightarrow & E = X \\ C & \Rightarrow & F = X \end{array}$$

□

For the implementation in the λ-calculus we use the boolean primitives from the previous section. Actually, the above proof rule is a weakened version of the specification of the boolean primitive if; so, we simply define

$$\mathsf{if}\ B \to E \,[]\, C \to F\ \mathsf{fi} \quad = \quad \mathsf{if}{\cdot}B{\cdot}E{\cdot}(\mathsf{if}{\cdot}C{\cdot}F{\cdot}?) \ ,$$

where ? represents a value that is left completely unspecified —a so-called *don't care*— .

7.3 Product and sum types

For the implementation of cartesian product we need functions pair, left, and right with the following specification:

$$(\forall x, y :: \mathsf{left}{\cdot}(\mathsf{pair}{\cdot}x{\cdot}y) = x) \ \text{ and: } \ (\forall x, y :: \mathsf{right}{\cdot}(\mathsf{pair}{\cdot}x{\cdot}y) = y) \ .$$

A simple and straightforward implementation is

$$\begin{array}{lcl} \mathsf{pair}{\cdot}x{\cdot}y & = & \lambda b{:}\ \mathsf{if}\ b \to x \,[]\, \neg b \to y\ \mathsf{fi} \\ \mathsf{left}{\cdot}p & = & p{\cdot}\mathsf{true} \\ \mathsf{right}{\cdot}p & = & p{\cdot}\mathsf{false} \end{array}$$

As in Section 6.2 we denote the type of pairs with left element of type X and right element of type Y by $X \times Y$. As a more readable alternative for $\mathsf{pair}{\cdot}x{\cdot}y$ we also use $\langle x, y \rangle$. In subsection 7.6 we shall define the more general notion of *tuples* in a homogeneous way.

Disjoint sums can now be formed by pairing the elements with either true or false to encode the subtype they belong to. Thus, for types X and Y we define their disjoint sum $X + Y$ by

$$X + Y \quad = \quad \{\mathsf{true}\} \times X \ \cup \ \{\mathsf{false}\} \times Y$$

7.4 The types Nat and Int

The natural numbers can be defined in terms of the following *natural primitives*:

a constant zero and functions isz, succ, and pred satisfying,
for all x in Ω:

$$\begin{array}{lcl} \mathsf{isz}{\cdot}\mathsf{zero} & = & \mathsf{true} \\ \mathsf{isz}{\cdot}(\mathsf{succ}{\cdot}x) & = & \mathsf{false} \\ \mathsf{pred}{\cdot}(\mathsf{succ}{\cdot}x) & = & x \end{array}$$

In terms of these primitives the type Nat is then defined as the strongest predicate on Ω —i.e.: the smallest subset of Ω— satisfying:

$$\mathsf{Nat}\cdot\mathsf{zero} \;\text{ and }\; (\forall x :: \mathsf{Nat}\cdot x \;\Rightarrow\; \mathsf{Nat}\cdot(\mathsf{succ}\cdot x))$$

The first of these requirements states that zero has type Nat and the second one states that succ has type $\mathsf{Nat}\rightarrow\mathsf{Nat}$.

Thus defined, the type Nat is infinite, which follows from the infinity lemma from Section 4.1 with $b, f := \mathsf{zero}, \mathsf{succ}$, because we have:

$$\begin{array}{ll} & \mathsf{zero} \neq \mathsf{succ}\cdot x \\ \Leftarrow & \quad \{\text{ Leibniz }\} \\ & \mathsf{isz}\cdot\mathsf{zero} \neq \mathsf{isz}\cdot(\mathsf{succ}\cdot x) \\ \equiv & \quad \{\text{ specification of } \mathsf{isz}\ \} \\ & \mathsf{true} \neq \mathsf{false} \\ \equiv & \quad \{\ \} \\ & \mathsf{true}\ , \end{array}$$

and:

$$\begin{array}{ll} & \mathsf{succ}\cdot x \neq \mathsf{succ}\cdot y \\ \Leftarrow & \quad \{\text{ Leibniz }\} \\ & \mathsf{pred}\cdot(\mathsf{succ}\cdot x) \neq \mathsf{pred}\cdot(\mathsf{succ}\cdot y) \\ \equiv & \quad \{\text{ specification of } \mathsf{pred}\ \} \\ & x \neq y\ . \end{array}$$

The elements of type Nat now represent the natural numbers in the obvious way: $\mathsf{succ}^k\cdot\mathsf{zero}$ represents k, for every natural k. Moreover, all arithmetic operators can be defined in terms of the primitives. For example, a recursive definition for addition is:

$$\mathsf{plus}\cdot x\cdot y \;=\; \mathsf{if}\;\; \mathsf{isz}\cdot y \;\rightarrow\; x \;[]\; \neg\mathsf{isz}\cdot y \;\rightarrow\; \mathsf{succ}\cdot(\mathsf{plus}\cdot x\cdot(\mathsf{pred}\cdot y))\;\; \mathsf{fi}$$

Thus defined, plus satisfies:

$$\begin{array}{lcl} \mathsf{plus}\cdot x\cdot\mathsf{zero} & = & x \\ \mathsf{plus}\cdot x\cdot(\mathsf{succ}\cdot y) & = & \mathsf{succ}\cdot(\mathsf{plus}\cdot x\cdot y) \end{array}$$

The natural primitives can be defined easily in the programming notation:

$$\begin{array}{lcl} \mathsf{zero} & = & \langle \mathsf{true}, ? \rangle \\ \mathsf{succ}\cdot x & = & \langle \mathsf{false}, x \rangle \\ \mathsf{isz}\cdot x & = & \mathsf{left}\cdot x \\ \mathsf{pred}\cdot x & = & \mathsf{right}\cdot x \end{array}$$

Actually, in this way the type Nat is defined as the *recursive datatype* given by:

$$\mathsf{Nat} \;=\; \{?\} + \mathsf{Nat}$$

The integers can be defined directly in terms of the naturals as follows. An integer is (represented by) a pair of naturals, in such a way that the pair $\langle x, y\rangle$ represents the integer $x - y$. This yields a non-unique representation for the integers, but the representation can be made unique by restricting the set to those pairs $\langle x, y\rangle$ that satisfy $x = 0 \vee y = 0$. In order to assure that the naturals are a true subtype of the integers we must, of course, embed them into the integers by redefining them. Thus we obtain:

$$\begin{array}{lcl} \mathsf{Nat}' & = & (\mathsf{set}\, x : \mathsf{Nat} \cdot x : \langle x, 0\rangle\,) \\ \mathsf{Int} & = & (\mathsf{set}\, y : \mathsf{Nat} \cdot y : \langle 0, y\rangle\,) \;\cup\; \mathsf{Nat}' \end{array}$$

As with the booleans, we shall use conventional notation for naturals and integers throughout the remainder of this study.

7.5 Recursive datatypes

The example of the natural numbers shows how recursive datatypes in general can be discussed: instead of imposing a complete partial order upon the value domain, we use that the collection of all predicates on the value domain —or: the powerset of the value domain— forms a complete lattice. The structure of this lattice is much richer than the complete partial orders used in domain theory; moreover, because for reasoning about programs and datastructures we need the predicate calculus anyhow, this gives rise to a nice, homogeneous system. A definite advantage of this modus operandi is that the need for the introduction of awkward elements like $\bot$, with all their associated problems —e.g.: is or is not $\mathsf{succ} \cdot \bot = \bot$ true?—, is avoided.

The definition of the type Nat as the strongest predicate having a certain property, amounts to defining the type as the strongest fixed point of a monotonic predicate transformer [6]. In the case of Nat, the predicate transformer involved is $X \mapsto (= \mathsf{zero}) \vee \mathsf{succ} * X$, where $*$ ("map") denotes the operation of applying a function to all elements of a set. The monotonicity of this predicate transformer follows immediately from the fact that $(F*)$ is *universally disjunctive*, for every F.

Generally, every recursive datatype whose definition only involves the unit type, cartesian product, and disjoint sum can be dealt with in this way; these datatypes are also known as *polynomial types* [1].

7.6 Tuples

Having the integers at our disposal we are now able to define *tuples* as a generalisation of pairs. This yields a homogeneous treatment in that the same (projection) functions for element selection can be used for tuples of any size. Moreover, it is possible to introduce a unary operator $\#$ ("size") mapping tuples to their sizes. For instance, we define 3-tuples as follows:

$$\begin{array}{lcl} \langle x,y,z\rangle & = & \lambda i:\ \mathsf{if}\ \ i=-1\ \to\ 3 \\ & & \ []\ \ i=\ \ 0\ \to\ x \\ & & \ []\ \ i=\ \ 1\ \to\ y \\ & & \ []\ \ i=\ \ 2\ \to\ z \\ & & \mathsf{fi} \end{array}$$

With this definition we have that $tp\cdot -1$ equals the number of elements of triple tp, so we can define $\# = (\cdot -1)$. Similarly, for element selection we can use $(\cdot 0)$, $(\cdot 1)$, and so on.

In many applications the sizes of the tuples used can take over the role of the boolean tags needed for the implementation of disjoint union. For example, the natural primitives could also be defined by

$$\begin{array}{lcl} \mathsf{zero} & = & \langle\,\rangle \\ \mathsf{succ}\cdot x & = & \langle x\rangle \\ \mathsf{isz}\cdot x & = & \#\cdot x = 0 \\ \mathsf{pred}\cdot x & = & x\cdot 0 \end{array}$$

Similarly, the datatype of *labelled binary trees* can be defined by means of tuples as follows; the type Tree is the strongest predicate satisfying:

$$\mathsf{Tree}\cdot\langle\,\rangle \ \text{ and } \ (\forall x,s,t::\ \mathsf{Tree}\cdot s \wedge \mathsf{Tree}\cdot t\ \Rightarrow\ \mathsf{Tree}\cdot\langle x,s,t\rangle\ )$$

remark: This way of dealing with tuples resembles the concept of *extensible records* [14] that are used in object-oriented programming.
□

7.7 Finite and infinite lists

Lists can be defined by means of the following *list primitives*. These primitives have been chosen so as to enable us to view lists in two complementary ways. On the one hand, lists are structured objects that can be composed and taken apart. On the other hand, lists can be considered as functions on natural intervals of the shape $[0..n)$, for $0\le n\le\infty$; if $n<\infty$ the list is finite with length n and for $n=\infty$ the

list is infinite. We call the former the *structural view*, whereas we call the latter the *functional view* of lists.

The list primitives are $[\,]$, $;$, ise , and tail and their specification is, for all x, y and natural i :

$$\begin{array}{lcl} \mathsf{ise}\cdot[\,] & = & \mathsf{true} \\ \mathsf{ise}\cdot(x\,;y) & = & \mathsf{false} \\ (x\,;y)\cdot 0 & = & x \\ (x\,;y)\cdot(i{+}1) & = & y\cdot i \\ \mathsf{tail}\cdot(x\,;y) & = & y \end{array}$$

The (polymorphic) datatype $\mathcal{L}$ consists of two subtypes $\mathcal{L}_*$ —the finite lists— and $\mathcal{L}_\infty$ —the infinite lists— . These subtypes are disjoint, so we have:

$$\begin{array}{lcl} \mathcal{L}_* \,\cup\, \mathcal{L}_\infty & = & \mathcal{L} \quad , \text{ and} \\ \mathcal{L}_* \,\cap\, \mathcal{L}_\infty & = & \phi \end{array}$$

Moreover, $\mathcal{L}_*$ is the *strongest* and $\mathcal{L}$ is the *weakest* of all predicates R satisfying:

$$(7.2) \quad R\cdot[\,] \;\wedge\; (\forall x, y :: R\cdot y \;\Rightarrow\; R\cdot(x\,;y)\,)$$

This definition is all we need for programming with lists. For reasoning about (recursive definitions of) infinite lists we need a kind of *productivity theory*, which is obtained by application of the recursion theorem from Section 5.3 to the case of infinite lists. According to this theory (so-called) *list-productive* functions have unique infinite lists as their fixed points; moreover, the theory provides a means for proving properties about these fixed points. As a very simple example, the following definition is productive and it yields the increasing infinite list containing the natural numbers. (An extensive discussion of productivity and more interesting examples can be found in [9].) In this example $*$ denotes the "map" operator:

$$x \;=\; 0 \;\;;\;\; (1{+}) * x$$

The list types thus defined comprise the lists over Ω , which means that the type of their elements is left unspecified. In order to obtain lists over some element type T , the range of dummy x in equation (7.2) must be strengthened to $T\cdot x$.

A correct implementation of the list primitives is:

$$\begin{array}{lcl} x\,;y & = & \lambda i : \ \mathsf{if} \ \ i{=}{-}2 \ \to \ \mathsf{false} \\ & & \quad [\!] \ \ i{=}{-}1 \ \to \ y \\ & & \quad [\!] \ \ i{=}\ \ 0 \ \to \ x \\ & & \quad [\!] \ \ i{>}\ \ 0 \ \to \ y\cdot(i{-}1) \\ & & \quad \mathsf{fi} \\ [\,] & = & \lambda i : \ \mathsf{if} \ \ i{=}{-}2 \ \to \ \mathsf{true} \ \ \mathsf{fi} \\ \mathsf{ise}\cdot x & = & x\cdot{-}2 \\ \mathsf{tail}\cdot x & = & x\cdot{-}1 \end{array}$$

Chapter 8

What paradoxes?

In Chapter 7 we have defined the type Bool as the set $\{\mathsf{true}, \mathsf{false}\}$, where $\mathsf{true} \in \Omega$ and $\mathsf{false} \in \Omega$, and we have defined boolean operations like not. Recall that not has type $\Omega \to \Omega$, which means that not is total on Ω but that otherwise we know nothing about it, and that it has type $\mathsf{Bool} \to \mathsf{Bool}$, which means that (within Ω) it maps booleans to booleans. Because $\mathsf{not} \in \Omega$ and because every value in Ω has a fixed-point we have:

(8.0) $(\exists x : x \in \Omega : x = \mathsf{not} \cdot x)$,

which *seems* contradictory, because we also know that:

$\mathsf{true} \neq \mathsf{not} \cdot \mathsf{true} \ \wedge \ \mathsf{false} \neq \mathsf{not} \cdot \mathsf{false}$, hence:
$(\forall x : x \in \mathsf{Bool} : x \neq \mathsf{not} \cdot x)$

Nevertheless, there is no contradiction, for the range of x in (8.0) is Ω; the only conclusion we can draw is that, apparently, Ω contains more than just the booleans, which we already knew (because Ω is infinite).

Similarly, we have:

$(\exists x : x \in \Omega \ : x = 1 + x)$, but also:
$(\forall x : x \in \mathsf{Int} : x \neq 1 + x)$,

but the only conclusion to be drawn from this is:

$(\exists x : x \in \Omega : \neg(x \in \mathsf{Int}))$.

In words: in whatever way we represent the integers in Ω, we shall always have $\mathsf{Int} \neq \Omega$. Again, this is not surprising, because (in the λ-calculus) the integers have normal forms, whereas not all values have normal forms.

A more interesting paradox is the one I quoted [12] in Chapter 0. We repeat it here:

> An even more fundamental difficulty, however, is highlighted by the fact that our definition [...] involves the application of x to itself. The possibility of self-application can lead to paradoxes. For example, suppose we define
>
> $$u \;=\; \lambda y.\ \mathsf{if}\ y(y)=a\ \mathsf{then}\ b\ \mathsf{else}\ a \quad .$$
>
> Then an attempt to evaluate $u(u)$ gives
>
> $$u(u) \;=\; \mathsf{if}\ u(u)=a\ \mathsf{then}\ b\ \mathsf{else}\ a$$
>
> which is a contradiction.

The argument giving rise to the contradiction runs as follows:

The assumption $u(u)=a$ leads to the conclusion $u(u)=b$, which implies $a=b$; hence, when $a\neq b$ has been given[0], we conclude $u(u)\neq a$. From $u(u)\neq a$ (and the definition of u) we conclude $u(u)=a$, so we have $a\neq a$, which is false.

This paradox is not due to an inconsistency in the calculus: the above argument just is wrong. To expose the error we must be aware of the fact that u is a term in the λ-calculus, in which $\mathsf{if}\ldots\mathsf{then}\ldots\mathsf{else}$ and $=$ do not occur; so, these must be represented by terms in the λ-calculus. That is, the λ-calculus should contain terms true, false, if, and eq, with the following specifications:

$$(\forall x,y ::\ \mathsf{if}\cdot\mathsf{true}\cdot x\cdot y \;=\; x)$$
$$(\forall x,y ::\ \mathsf{if}\cdot\mathsf{false}\cdot x\cdot y \;=\; y)$$
$$(\forall x,y : x=y :\ \mathsf{eq}\cdot x\cdot y \;=\; \mathsf{true})$$
$$(\forall x,y : x\neq y :\ \mathsf{eq}\cdot x\cdot y \;=\; \mathsf{false})$$

In particular, this specification implies that eq has type $\Omega\to\Omega\to\mathsf{Bool}$. In Chapter 7 we have seen that if exists indeed. The contradiction in the above argument now arises from the assumption that eq exists as well, because in that argument we (tacidly) used that $u(u)=a$ had a boolean value and that that boolean value represented equality of $u(u)$ and a. (Notice that $u(u)=a$ should be encoded as $\mathsf{eq}\cdot(u\cdot u)\cdot a$.) Apparently, this assumption is false, and the only conclusion we can draw from this "paradox" is that (a general purpose) equality test cannot be implemented in the λ-calculus.

Again, this conclusion is not that surprising: it only shows that mathematical equality —any two values in Ω are either equal or different— and computational equality are different notions; in this light the use of the symbol $=$ for computational equality is misleading. Fortunately, for terms having normal forms mathematical equality is also computable, so for terms having normal forms the two notions coincide. That is, when restricted to normal forms, eq does exist[1], with $\mathsf{eq}\cdot x\cdot y \;\equiv\; x=y$.

[0] The author of [12] has left this implicit.

[1] Although I do not know whether eq exists in Ω, but that is not too relevant here.

Bibliography

[1] R.C. Backhouse c.s., *Polynomial relators.*
Computing Science Notes 91/10, Eindhoven University of Technology, 1991.

[2] H.P. Barendregt, *The lambda calculus, its syntax and semantics.*
Elsevier Science Publishers, Amsterdam, 1984.

[3] A. Bijlsma, R.R. Hoogerwoord, *No self-application in set theory.*
memorandum AB33a/rh176, Eindhoven, 1992.

[4] R.S. Bird, Ph. Wadler, *Introduction to functional programming.*
Prentice Hall International, Hemel Hempstead, 1988.

[5] H.B. Curry, R. Feys, *Combinatory logic* (vol. 1).
North-Holland Publishing Company, Amsterdam, 1958.

[6] E.W. Dijkstra, C.S. Scholten, *Predicate calculus and program semantics.*
Springer-Verlag, New York, 1990.

[7] M.C. Henson, *Elements of functional languages.*
Blackwell Scientific Publications, Oxford, 1987.

[8] J.R. Hindley, J.P. Seldin, *Introduction to combinators and λ-calculus.*
Cambridge University Press, Cambridge, 1986.

[9] R.R. Hoogerwoord, *The design of functional programs: a calculational approach.*
Ph.D. thesis, Eindhoven University of Technology, 1989.

[10] R.R. Hoogerwoord, *Whither inference rules?*
memorandum rh172, Eindhoven, 1992.

[11] G.E. Revesz, *Lambda-calculus, combinators, and functional programming.*
Cambridge University Press, Cambridge, 1988.

[12] J.E. Stoy, *Denotational semantics: the Scott-Strachey approach to programming language theory.*
The MIT Press, Cambridge Massachusetts, 1977.

[13] R. Turner, *Constructive foundations for functional languages.*
McGraw-Hill Book Company, Maidenhead, 1991.

[14] N. Wirth, Type extensions.
ACM Toplass, **10**(1988), 204-214.