Specificational Functions

J. M. MORRIS and A. BUNKENBURG
University of Glasgow

Mathematics supplies us with various operators for creating functions from relations, sets, known functions, and so on. Function inversion is a simple example. These operations are useful in specifying programs. However, many of them have strong constraints on their arguments to ensure that the result is indeed a function. For example, only functions that are bijective may be inverted. This is a serious impediment to their use in specifications, because at best it limits the specifier's expressive power, and at worst it imposes strong proof obligations on the programmer. We propose to loosen the definition of functions so that the constraints on operations such as inversion can be greatly relaxed. The *specificational functions* that emerge generalize traditional functions in that their application to some arguments may yield no good outcome, while for other arguments their application may yield any of several outcomes unpredictably. While these functions are not in general algorithmic, they can serve as specifications of traditional functions as embodied in programming languages. The idea of specificational functions is not new, but accommodating them in all their generality without falling foul of a myriad of anomalies has proved elusive. We investigate the technical problems that have hindered their use, and propose solutions. In particular, we develop a formal axiomatization for reasoning about specificational functions, and we prove its consistency by constructing a model.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs*; *formal methods*

General Terms: Design, Theory, Verification

Additional Key Words and Phrases: Expression, function, logic, nondeterminacy, refinement calculus

1. INTRODUCTION

The square function on the integers, defined by $sqr \mathrel{\hat=} (\lambda z{:}\mathbf{Z} \bullet z * z)$, is not traditionally regarded as having a well-defined inverse, because it is neither injective nor surjective. Suppose, however, we were to broaden our definition of functions so that the inverse of *sqr*, call it *sqrt*, is indeed a function. We might define *sqrt* thus:

$$sqrt \mathrel{\hat=} \lambda z{:}\mathbf{Z} \bullet (\Box x{:}\mathbf{Z} \mid x * x = z).$$

We have used above an instance of what we can call a *prescriptive expression.* This has the general form $(\Box x{:}T|P)$ and the intuitive meaning *some x of type T satisfying*

Supported by the Leverhulme Trust. Authors' address: Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, Scotland.

predicate P. If there is no such x, we regard $(\Box x{:}T \mid P)$ as being equivalent to the special term $\bot$, pronounced "bottom." If there are many such x, then we have no information about which outcome is actually produced. For example, *sqrt* 4 may yield 2 or -2, and we do not know or care which. Indeed, we cannot even determine its behavior by experiment, because if *sqrt* 4 yields 2 in the morning, it may well yield -2 in the afternoon. Both *sqrt* 7 and $\mathit{sqrt}(-4)$ yield $\bot$.

Here is another example. Suppose the type *PhoneBook* is comprised of all relations from type *Name* to type *PhoneNumber*. Then the following function looks up a person's phone number in a phone book:

$$\mathit{lookUp} \mathrel{\hat=} \lambda n{:}\mathit{Name} \bullet \lambda b{:}\mathit{PhoneBook} \bullet (\Box x{:}\mathit{PhoneNumber} \mid (n, x) \in b).$$

Again, *lookUp* is not necessarily a function in the traditional sense (because some people are not listed in phone books, and some have several entries), but we would like to treat it as such.

For a more elaborate example, consider the function

$$\mathit{leastWRT} : (T \to \mathbf{Z}) \to \mathbb{P}T \to T,$$

which takes as arguments a function f and a set s, and selects some element a of s such that $f\ a$ is minimized. For example, instantiating T with $\mathbf{Z}$,

$$\mathit{leastWRT}\ \mathit{sqr}\ \{-3, -1, 1, 2, 4\}$$

yields either 1 or -1. We can define *leastWRT* for any type T thus:

$$\mathit{leastWRT} \mathrel{\hat=} \lambda f{:}T \to \mathbf{Z} \bullet \lambda s{:}\mathbb{P}T \bullet (\Box x{:}T \mid x \in s \wedge (\forall y{:}s \bullet f\ x \le f\ y)).$$

To illustrate its use, we make a function to be used by two lovers each of whom travels from city to city as a computer consultant. The function should yield a city to which they should both travel if they want to be together as soon as possible. We assume a type *City* whose elements are all cities, and a function $\mathit{time}{:}\mathit{City} \times \mathit{City} \to \mathbf{N}$ which yields the least traveling time (in minutes, say) between any two cities. The function is

$$\lambda \mathit{his}, \mathit{hers}{:}\mathit{City} \bullet \mathit{leastWRT}\ (\lambda c{:}\mathit{City} \bullet \mathit{time}(\mathit{his}, c)\ \max\ \mathit{time}(\mathit{hers}, c))\ \mathit{City}.$$

In $(\Box x{:}T \mid P)$, the type T need not be a so-called "flat" type such as the integers, but can be a more structured type such as a function type. For example,

$$(\Box f : \mathbf{Z} \to \mathbf{Z} \mid f\ 0 = 1 \wedge (\forall n{:}\mathbf{N} \bullet f(n+1) = (n+1) * f\ n))$$

specifies the familiar factorial function. For a more elaborate example of the usefulness of choice over function types, let us specify a program for playing a one-player game such as Rubik's Cube. We assume a context which provides a set *Cube* of all the legitimate states of the game, a set *LegalMoves* which is a subset of $\mathit{Cube} \times \mathit{Cube}$ describing the set of legal moves, and a value $\mathit{goal} \in \mathit{Cube}$ which describes the goal of the game. A player will be modeled as a function $f{:}\mathit{Cube} \to \mathit{Cube}$ where $f\ b$ is the position to which the player moves when in position b:

$$\mathit{Players} \mathrel{\hat=} \{f{:}\mathit{Cube} \to \mathit{Cube} \mid (\forall b{:}\mathit{Cube} \bullet (b, f\ b) \in \mathit{LegalMoves})\}.$$

We eliminate players who get stuck:

$$\mathit{GoodPlayers} \mathrel{\hat=} \{f{:}\mathit{Players} \mid (\forall b{:}\mathit{Cube} \bullet \exists i{:}\mathbf{N} \bullet f^i b = \mathit{goal})\}.$$

Each player has a cost, which is the total number of moves taken to play all possible games:

$$cost \mathrel{\hat{=}} \lambda f{:}Cube \rightarrow Cube \bullet (\Sigma b{:}Cube \bullet (\min i{:}\mathbf{N} \mid f^i b = goal)).$$

The program we want is

$$GamePlayer \mathrel{\hat{=}} leastWRT\ cost\ GoodPlayers.$$

All of *sqrt*, *lookUp*, *leastWRT*, and *GamePlayer* are examples of functions more liberally defined; we call them *specificational functions.*

Although specificational functions are not in general algorithmic, they do play an important role in *specifying* algorithmic functions. For example, *sqrt* might be presented to a programmer as a specification of a program he should implement, by which is meant that he should produce an algorithmic function $SQRT$ whose behavior is consistent with that of *sqrt*. By "consistent" we mean that when $SQRT$ is applied to a perfect square x it yields a square root of x (either negative or non-negative), and otherwise it behaves in any way that the programmer fancies. For example, the programmer might well design $SQRT$ such that its graph is

$$\{\ldots -2 \mapsto 49, -1 \mapsto 49, 0 \mapsto 0, 1 \mapsto -1, 2 \mapsto 57, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto -15, \ldots\}.$$

We say that $SQRT$ is a *refinement* of *sqrt* and write $sqrt \sqsubseteq SQRT$. Roughly, $E \sqsubseteq F$ holds of expressions E and F if in all contexts the set of possible outcomes of E is a superset of the set of possible outcomes of F. The expression $\perp$ is interpreted as the set of all possible outcomes, including the outcome non-termination, so it is a superset of every set of outcomes. Readers not familiar with refinement calculi may feel, with good cause, that $\sqsupseteq$ would be a more appropriate symbol for refinement, but $\sqsubseteq$ is what is traditionally used (it may help to think of $\sqsubseteq$ as suggesting increasing information content). For example, in the context of applying $SQRT$ to 4, the outcome 2 is among the outcomes of applying *sqrt* to 4 (2 and -2), and in the context of applying $SQRT$ to -2, the outcome 49 is acceptable because *sqrt* applied to -2 yields $\perp$.

A *refinement calculus* is, in essence, a language with a transitive refinement relation $\sqsubseteq$, where some of the terms in the language are algorithmic, and the remainder are available for describing desired behavior. The process of making a program consists in producing a sequence of language terms, $t_0 \sqsubseteq t_1 \sqsubseteq \ldots \sqsubseteq t_n$, where t_0 is the customer's specification, t_n is algorithmic, and each term in the sequence (except the first) is constructed incrementally from its predecessor. We say that each term in the sequence is a *refinement* of its predecessors. Programming by stepwise refinement [Wirth 1973] is an example of such a process, albeit a somewhat informal one because the terms of the language are written partly in informal pseudocode. The motivation for studying specificational functions is a desire to use them as part of a refinement calculus that supports the development of functions, whether in functional or imperative programming.

Unlike stepwise refinement, we have in mind a fully formal mathematical system in which each refinement is formally provable by deduction from given axioms. This requires an axiomatization of specificational functions. Traditional functions are typically axiomatized by postulating properties such as those in Figure 1. $E[x \backslash F]$ denotes expression E with all free occurrences of x replaced by F (subject to the

α	$(\lambda x{:}T \bullet E) = (\lambda y{:}T \bullet E[x\backslash y])$, fresh y
β	$(\lambda x{:}T \bullet E)F = E[x\backslash F]$
ξ	$(\forall x{:}T \bullet E = F) \Leftrightarrow ((\lambda x{:}T \bullet E) = (\lambda x{:}T \bullet F))$
η	$E = (\lambda x{:}T \bullet E\ x)$
extensionality	$(E = F) \Leftrightarrow (\forall x{:}T \bullet E\ x = F\ x)$
Skolem	$(\forall x{:}T \bullet \exists y{:}U \bullet P) \Rightarrow \exists f{:}T \to U \bullet \forall x{:}T \bullet P[y\backslash f\ x]$

Fig. 1. Axiomatic properties of functions.

usual caveat of renaming to avoid variable capture), and similarly for predicates. In η and extensionality, E and F are of functional type $T \to U$, and x is fresh. (The listed properties are not independent of one another, so no one axiomatization would use all of them.)

If we employ such axioms on specificational functions, we fall foul of a myriad of anomalies, as we shall see shortly. The effect has been to inhibit their deployment seriously. For example, it is common to restrict choice to flat types only, which rules out, for example, the specification *GamePlayer* above. We investigate the technical problems and propose solutions. In particular, we develop a formal axiomatization for reasoning about them and show that it sees off the anomalies.

1.1 Outline of article

The next section will introduce the concepts and notations for equivalence, refinement, choice, and "proper" values. The third section explains in detail the anomalies that occur when functions and choice meet, and suggests ways of avoiding them. The fourth section axiomatizes these language constructs. The fifth section is the core of the article. It presents an axiomatization of specificational functions that avoids the discussed anomalies. The sixth section discusses logic, and argues for our preferred logic. The seventh section gives a denotational model of the calculus. Finally, we draw conclusions and review related literature.

1.2 Contributions

—Detailed exposition of six anomalies that occur when functions and choice are combined.

—Suggestions how these anomalies can be avoided.

—An axiomatization of specificational functions.

—A proof of consistency by constructing a denotational model.

2. MATHEMATICAL PRELIMINARIES

2.1 Equivalence, Choice, and Refinement

We presume the availability of a (strong) equality operator $\equiv$ on terms, reserving the symbol $=$ for the weak or algorithmic equality operator found in programming languages. (Weak equality is further explained below.) We will use "equivalence" as a synonym for strong equality. Equivalence is reflexive, symmetric, transitive, and a congruence: if $E \equiv F$, then E can replace F in any term without changing its meaning.

The choice inherent in $(\Box x{:}T \mid P)$ may be unbounded, as in $(\Box x{:}\mathbf{Z} \mid \mathit{true})$ which may yield any integer. On the other hand, there may be no x satisfying P, as in

$(\Box x{:}T \,|\, \mathit{false})$; we introduce the special term $\bot_T$ as an abbreviation for $(\Box x{:}T \,|\, \mathit{false})$. It may seem intuitively reasonable that $\bot_T \sqsubseteq F$ should hold for *no* term F other than $\bot_T$, but refinement calculi commonly depart from intuition by postulating $\bot_T \sqsubseteq F$ for *all* terms F. We adopt the latter approach (known as *excluding miracles*). This choice is not central to our work, and indeed the axiomatization of specificational functions given in this article does not depend on it. Note that $(\Box x{:}T \mid \mathit{true})$ differs from $\bot_T$ in that $\bot_T$ is refined even by a "nonterminating" expression such as an application of the recursive function f where $f \mathrel{\hat=} \lambda x{:}T \bullet f\ x$. There is a bottom for each type, indicated by subscripting, but we nearly always omit the type indicator, either because it is not significant in the context or because it can be easily inferred. In refinement calculi, partial operations such as 3/0 are commonly equated with $\bot$, and similarly for nonterminating expressions. It is also customary to use $\bot$ as a "don't care" term by which the customer indicates that he or she has no interest in the outcomes. Although it may be useful in other contexts to distinguish these various roles for $\bot$, in program derivation they are similar in that they represent error situations in which the outcome is unpredictable and unconstrained.

When there is precisely one x of type T satisfying predicate P—call it k—then $(\Box x{:}T|P)$ is equivalent to k. One consequence of this is that specificational functions include traditional total functions. Any finite choice can be expressed in terms of a binary choice $E \sqcap F$ which specifies a choice among terms E and F (we use the words "term" and "expression" interchangeably). For example, $2 \sqcap 3$ specifies 2 or 3, without preference; it can be written equivalently as $(\Box x{:}\mathbf{Z} \mid x{\equiv}2 \vee x{\equiv}3)$. It should be evident that $\sqcap$ is commutative, associative, and idempotent. It is standard in refinement calculi that refinement and equivalence are related by $(E \sqsubseteq F) \Leftrightarrow (E \sqcap F \equiv E)$. It follows that $\sqsubseteq$ is reflexive, transitive and antisymmetric (with respect to $\equiv$). It also follows that $\bot$ is a zero of $\sqcap$. In other words, we have a meet semilattice with a bottom.

Readers coming from a background in formal logic might be tempted to view $(\Box x{:}T \mid P)$ as some rough equivalent of Hilbert's ϵ operator [Leisenring 1969], but that would be a mistake. There is little connection between the two. Roughly, ϵ represents choices that have already been made for you and for everyone now and for all time, whereas $\Box$ represents choices that have yet to be made and which may be made differently on different occasions. A Hilbert choice from 2 and 3 is always 2 or always 3—we just do not know which one it is. On the other hand, $2 \sqcap 3$ is sometimes 2 and sometimes 3. In formal terms, the ϵ operator satisfies the axiom $(\exists x{:}T \bullet P) \Rightarrow P[x \backslash (\epsilon x{:}T \mid P)]$; in words this is "if P is true of some x of type T, then it is true of $(\epsilon x{:}T \mid P)$." This assertion does not hold when ϵ is replaced with $\Box$. For example, letting J abbreviate $(\epsilon x{:}\mathbf{Z} \mid x{\equiv}2 \vee x{\equiv}3)$, we have the truth of $J{\equiv}2 \vee J{\equiv}3$, but with K standing for $(\Box x{:}\mathbf{Z} \mid x{\equiv}2 \vee x{\equiv}3)$, i.e., $2 \sqcap 3$, $K{\equiv}2 \vee K{\equiv}3$ is false. The ϵ operator is not useful for program refinement, because it requires that any "looseness" in a specification be resolved by programmers in the same way at all times and in all places. For example, if a customer asked that some error message be displayed if a file was unavailable, without stating any preference as to the wording of the message, it would require every programmer to deliver exactly the same message. More amusingly, if all choice was Hilbert's choice, then the diners in a restaurant would each choose the same meal from the menu.

We postulate that the operators of the base types (like $+$ and $-$ on the integers) are strict (i.e., $\bot$ is a zero for them) and distribute over choice. This design decision properly reflects the fact that our brand of choice allows different resolutions on different occasions. For example, although $(2 \sqcap 3) - (2 \sqcap 3)$ would be equivalent to 0 according to Hilbert's brand of choice, we admit the possibility that the first occurrence of $(2 \sqcap 3)$ has outcome 2, while the second has outcome 3, and vice versa. Hence $(2 \sqcap 3) - (2 \sqcap 3) \equiv -1 \sqcap 0 \sqcap 1$. Note that $=$ behaves strictly and distributively, unlike $\equiv$ (for example, $\bot = 3$ is just $\bot$ whereas $\bot \equiv 3$ is false).

2.2 Propers

The instantiation rule for universal quantification asserts that from the truth of $\forall x{:}T \bullet P$, we may infer $P[x\backslash E]$ for any term E. But such instantiation rules can easily lead to inconsistencies in the presence of bottom or choice. For example, $\forall z{:}\mathbf{Z} \bullet z - z \equiv 0$ is a theorem of arithmetic, so we might be tempted to infer $(2 \sqcap 3) - (2 \sqcap 3) \equiv 0$ in contradiction of our earlier conclusion that $(2 \sqcap 3) - (2 \sqcap 3) \equiv -1 \sqcap 0 \sqcap 1$. A similar anomaly arises if we instantiate with $\bot$—we can infer $\bot \equiv 0$. Either we modify the standard laws of arithmetic to take into account bottom and choice, or we modify the rules of instantiation. The easiest fix is to modify the instantiation rules by forbidding instantiations with $\bot$ or terms involving (an unresolvable) choice; we call such terms *improper* and all other terms *proper*. For example, $\bot$ and $2 \sqcap 3$ are improper, whereas 3, $2+3$, $2 \sqcap 2$, and $2 \sqcap 3 < 6$ are proper (in the final example, we have assumed that $<$ distributes over $\sqcap$). Properness may depend on the environment: in some environments $1 \sqcap x$ is proper, in others not. Our intention is that every expression should be either proper or bottom or a choice among propers. In a calculus including miracles, an expression could be the choice among zero propers. The requirement that $\forall x{:}T \bullet P$ (and similarly, $\exists x{:}T \bullet P$) only be instantiated with proper terms is a condition on the underlying logic. In the terminology of Søndergaard and Sestoft [1992], we have opted for *strict singular* semantics; they list alternative approaches.

If E is proper, we state ΔE. If E is not equivalent to $\bot$, we state τE. After postulating that variables are always proper ($\forall x{:}T \bullet \Delta x$) and axiomatizing Δ and refinement, it can be proven that $\Delta E \Leftrightarrow (\exists x{:}T \bullet E \equiv x)$. Furthermore, after postulating that variables are always nonbottom ($\forall x{:}T \bullet \tau x$), $\Delta E \Rightarrow \tau E$ can also be proven. We postulate that the common base types $\mathbf{Z}$, *Char*, *Float*, ... are *flat*, i.e., $\forall x, y{:}T \bullet (x \sqsubseteq y) \Rightarrow (x \equiv y)$. From this we can deduce, that, for flat types, $E \sqcap F$ is proper if and only f E is proper and equivalent to F. We also postulate that the constants (such as, in the case of the integers, 0, 1, -1, 2, -2, ..., etc.) are proper. We can now formally deduce that $2 \sqcap 3$, say, is not proper because 2 is proper and differs from 3.

Nonflat types (such as function types) are considerably more complex than flat types, and determining just which expressions should be proper need not be at all obvious, as we shall see. Because most of us came to know functions through studying functions on the integers and reals, it is easy to be seduced into accepting properties of functions that may hold for functions on flat types, but which are not true in general. This is even more so in the case of specificational functions, and we advise the reader to think of nonflat types whenever he or she is looking for intuitive understanding. We have ourselves sometimes been misguided by thinking in terms of flat types only.

For the purpose of examples, we will denote by *Two* a type which has exactly two propers u and v such that $u \sqsubset v$, i.e., $u \sqcap v \equiv u$ and $u \not\equiv v$. Whether such a type exists is of course a consequence of our design decisions. However, usually functional types are nonflat, and if there is a type 1 with only one proper element, also written 1, then $(\lambda x \bullet \bot_1)$ and $(\lambda x \bullet 1)$ might be taken for u and v.

> *For the remainder of the article we use lowercase letters for proper expressions, and uppercase letters for arbitrary (possibly improper) expressions.*

3. THE ANOMALIES

In this section, we expose some of the problems that are encountered when a straightforward axiomatization of functions drawn from Figure 1 is used in the presence of bottom and choice.

3.1 Beta-Equivalence and Extensionality

For specificational functions, the combination of axioms β and extensionality leads to an inconsistency. Consider $\lambda x{:}\mathbf{Z} \bullet x{-}x$ and $\lambda x{:}\mathbf{Z} \bullet 0$. By extensionality, we conclude that they are equivalent, since $\forall x{:}\mathbf{Z} \bullet x{-}x \equiv 0$. But if now we apply each function to $\bot$ in turn, we can deduce that $\bot \equiv 0$! Similarly, we can deduce that $-1 \sqcap 0 \sqcap 1 \equiv 0$ by applying the two functions to $2 \sqcap 3$ in turn.

It is not fruitful to attack the anomaly by restricting the extensionality axiom, because the two functions in question are so simple that we expect them to be equivalent by extensionality, however restricted. The remedy we shall adopt is to restrict the β axiom to proper arguments. Now the anomalies disappear, since neither $\bot$ nor $2 \sqcap 3$ are proper.

Of course, we are left with the question of assigning a meaning to applications with improper arguments. We choose to make function application strict and distribute over choice. In symbols, $E\ \bot \equiv \bot$ and $E(F \sqcap G) \equiv E\ F \sqcap E\ G$. Now both example functions yield $\bot$ when applied to $\bot$, and 0 when applied to $2 \sqcap 3$.

We should not forget that function types also accommodate bottom and choice. For reasons of symmetry, we decide that function application should be strict and distributive in the function as well as in the argument, i.e., $\bot E \equiv \bot$ and $(E \sqcap F)G \equiv E\ G \sqcap F\ G$. These decisions are not controversial, since there is no reasonable alternative, but they do lead to a quandary, as we shall see.

The above is the first of several examples of inconsistencies that arise from naively applying the traditional laws of functions; ridding ourselves of them requires compromise. We choose our compromises to make life as pleasant as possible when we put our tools to their intended use. If our purposes change, then so might our compromises. For example, not insisting on strictness would be appropriate if the target program language is a lazy functional language such as Haskell [Peterson and Hammond 1997]; such a calculus is explored in Bunkenburg [1997]. In Hehner [1993; 1998], the decision as to whether function application distributes over choice is left open in general; it must be decided individually for every abstraction the developer writes.

3.2 Distribution and η

The following inconsistency was discussed but not solved satisfactorily in Meertens [1986]. Define $F \mathrel{\hat{=}} \lambda x{:}\mathbf{Z} \bullet x \sqcap 3$ and $G \mathrel{\hat{=}} (\lambda x{:}\mathbf{Z} \bullet x) \sqcap (\lambda x{:}\mathbf{Z} \bullet 3)$. Using η, we prove their equivalence by transforming one into the other (here and elsewhere, we omit type information to reduce syntactic clutter):

$$\begin{array}{ll} & G \\ \equiv & \\ & (\lambda x \bullet x) \sqcap (\lambda x \bullet 3) \\ \equiv & \qquad \eta \\ & \lambda x \bullet ((\lambda x \bullet x) \sqcap (\lambda x \bullet 3))\ x \\ \equiv & \qquad \text{ap./}\sqcap \\ & \lambda x \bullet (\lambda x \bullet x)\ x \sqcap (\lambda x \bullet 3)\ x \\ \equiv & \qquad \beta \\ & \lambda x \bullet x \sqcap 3 \\ \equiv & \\ & F. \end{array}$$

Now we can show that $3 \sqcap 4 \sqcap 5 \sqcap 6 \equiv 3 \sqcap 6$ by applying the higher-order function $\lambda h{:}\mathbf{Z}{\to}\mathbf{Z} \bullet h\ 1 + h\ 2$ to F and G in turn:

$$\begin{array}{ll} & (\lambda h \bullet h\ 1 + h\ 2)\ F \\ \equiv & \qquad \beta\text{, assuming } F \text{ is proper} \\ & F\ 1 + F\ 2 \\ \equiv & \qquad \beta \\ & (1 \sqcap 3) + (2 \sqcap 3) \\ \equiv & \\ & 3 \sqcap 4 \sqcap 5 \sqcap 6, \end{array}$$

whereas for G we get

$$\begin{array}{ll} & (\lambda h \bullet h\ 1 + h\ 2)\ ((\lambda x \bullet x) \sqcap (\lambda x \bullet 3)) \\ \equiv & \qquad \text{ap./}\sqcap \\ & (\lambda h \bullet h\ 1 + h\ 2)\ (\lambda x \bullet x) \sqcap (\lambda h \bullet h\ 1 + h\ 2)\ (\lambda x \bullet 3) \\ \equiv & \qquad \beta \\ & ((\lambda x \bullet x)\ 1 + (\lambda x \bullet x)\ 2) \sqcap ((\lambda x \bullet 3)\ 1 + (\lambda x \bullet 3)\ 2) \\ \equiv & \qquad \beta \\ & (1 + 2) \sqcap (3 + 3) \\ \equiv & \\ & 3 \sqcap 6. \end{array}$$

We definitely do not want $3 \sqcap 4 \sqcap 5 \sqcap 6 \equiv 3 \sqcap 6$. (In the calculation, we have assumed that $\lambda x \bullet x \sqcap 3$ is proper. We shall see later on that this is reasonable.)

The η-axiom also leads us to conclude that $\perp_{T\to U}$ and $\lambda x{:}T \bullet \perp_U$ are equivalent:

$$\begin{array}{ll} & \perp_{T\to U} \\ \equiv & \qquad \eta \\ & \lambda x{:}T \bullet (\perp_{T\to U}\ x) \\ \equiv & \qquad \text{ap. strict} \\ & \lambda x{:}T \bullet \perp_U. \end{array}$$

But this runs counter to practice in imperative and functional programming languages. In imperative languages, for example, the body of a function f supplied as an argument to a procedure *proc* is not inspected at the point of invoking *proc*, but only at the point of invoking f within the body of *proc*.

We will resolve these dilemmas by restricting the η-axiom to *proper* functions only, with $\bot_{T\to U}$ and such functions as $(\lambda x \bullet x) \sqcap (\lambda x \bullet 3)$ being improper. We will have much more to say about properness of functions later on.

3.3 Extensionality and Distribution

Postulating distribution of application has the consequence that extensional equivalence of functions breaks down in the presence of higher-order functions. Consider as before $F \mathrel{\hat{=}} \lambda x{:}\mathbf{Z} \bullet x \sqcap 3$ and $G \mathrel{\hat{=}} (\lambda x{:}\mathbf{Z} \bullet x) \sqcap (\lambda x{:}\mathbf{Z} \bullet 3)$: any proof of their equivalence must lead to a contradiction because $\lambda h \bullet h\ 1 + h\ 2$ can distinguish them. Previously, we proved them equivalent by η and distribution; now we prove them equivalent by extensionality:

$$\begin{array}{ll}
 & F \equiv G \\
\Leftrightarrow & \\
 & (\lambda x \bullet x \sqcap 3) \equiv (\lambda x \bullet x) \sqcap (\lambda x \bullet 3) \\
\Leftrightarrow & \qquad \text{extensionality} \\
 & \forall x \bullet (\lambda x \bullet x \sqcap 3)\ x \equiv ((\lambda x \bullet x) \sqcap (\lambda x \bullet 3))\ x \\
\Leftrightarrow & \qquad \sqcap/\text{ap.} \\
 & \forall x \bullet (\lambda x \bullet x \sqcap 3)\ x \equiv (\lambda x \bullet x)\ x \sqcap (\lambda x \bullet 3)\ x \\
\Leftrightarrow & \qquad \beta \\
 & \forall x \bullet x \sqcap 3 \equiv x \sqcap 3.
\end{array}$$

We will choose to resolve the inconsistency by having function equivalence require more than extensionality in general (it will turn out that in the case of proper functions, extensionality is sufficient). On reflection, this restriction should not be surprising because unrestricted extensionality identifies the functions $\bot_{T\to U}$ and $\lambda x{:}T \bullet \bot_U$, against our wishes.

3.4 Monotonicity and Distribution

It is not too difficult to deduce that if function application distributes over $\sqcap$ then function application is monotone with respect to $\sqsubseteq$, that is, $E \sqsubseteq E' \Rightarrow F\ E \sqsubseteq F\ E'$. It follows that we can introduce an inconsistency if we can construct a nonmonotone function. Such a function is $f : \mathit{Two}{\to}U$ such that $f\ u \equiv 1$ and $f\ v \equiv 2$ (recall Two has two propers u and v such that $u \sqsubset v$). Assuming a conditional expression whose first argument is a proposition is available, we can construct f thus: $\lambda x{:}\,\mathit{Two} \bullet \mathbf{if}\ x \equiv u\ \mathbf{then}\ 1\ \mathbf{else}\ 2$.

We shall avoid this anomaly by admitting only λ-abstractions $\lambda x{:}T \bullet E$ such that

$$\forall x, x' \bullet x \sqsubseteq x' \Rightarrow E \sqsubseteq E[x \backslash x'].$$

The restriction is not a hindrance in practice because it turns out that all operators used in programming languages, and just about all those used in specifications, are indeed monotone. The function f above is nonmonotone because $\equiv$ is a nonmonotone operator. Trivially, all functions on flat domains are monotone, even if their bodies employ nonmonotone constructs. It is possible to set down reasonable syntactic rules for forming λ-abstractions that guarantee monotonicity (although some acceptable λ-abstractions might be ruled out).

It is a little disappointing that when we refine the (monotone) body of a λ-abstraction, we retain the obligation to show monotonicity—it is not necessarily preserved by refinement. For example, **if** $x \equiv u$ **then** $1 \sqcap 2$ **else** 1, which is monotone, is refined by **if** $x \equiv u$ **then** 2 **else** 1, which is *not* monotone! Fortunately, such examples are rare (those we know of rely on the use of the strong $\equiv$ or $\sqsubseteq$ operators). However, it brings some theoretical surprises as we shall see later.

3.5 Monotonicity and Skolem

The Skolem axiom, $(\forall x{:}T \bullet \exists y{:}U \bullet P) \Rightarrow (\exists f{:}T \to U \bullet \forall x{:}T \bullet P[y \backslash f\ x])$, which promises the existence of certain functions, seems intuitively reasonable, but in the presence of the monotonicity requirement on functions, it may be promising the impossible.

Recall again type *Two* with propers u and v such that $u \sqsubset v$. Let $P \mathrel{\hat{=}} (x \equiv u \ \wedge \ y \equiv 1) \vee (x \not\equiv u \ \wedge \ y \equiv 2)$. Then clearly $\forall x{:}Two \bullet \exists y{:}\mathbf{Z} \bullet P$ holds. But the function f promised by the Skolem axiom would have to map u to 1 and v to 2, and this is not monotone. The essence of the problem is that with the restriction to monotone functions, the Skolem axiom promises the existence of a monotone mapping for every possibly nonmonotone mapping, and that promise cannot be kept.

We shall not postulate Skolem as an axiom, but will be able to derive it for "reasonable" P. For example, if the x in $P(x, y)$ is drawn from a flat type, then Skolem holds. We will return to this point later.

4. THE AXIOMATIZATION PROCESS

We axiomatize prescriptive and conditional expressions. We also explain our method for axiomatizing types and their operations, using pair-types as an example. This will serve as a model for the more complex function types to come. We assume that an axiomatization of a first-order logic and some base types has already been given. More about the assumed logic, and our choice of logic, can be found in Section 6.

4.1 Axiomatizing Language Constructs

Recall that τE is an abbreviation for $E \not\equiv \bot$. The following postulate captures the intuitive idea of refinement as described earlier:

$$E \sqsubseteq F \Leftrightarrow (\tau E \Rightarrow \tau F) \wedge (\forall x{:}T \bullet E \sqsubseteq x \Leftarrow F \sqsubseteq x). \tag{1}$$

(The conjunct $\tau E \Rightarrow \tau F$ is required to distinguish between $\bot_T$ and $(\Box x{:}T \,|\, true)$.) All properties of $\sqsubseteq$ mentioned earlier follow from this and the usual properties of implication. From this, and the antisymmetry of $\sqsubseteq$, we can deduce the following:

$$E \equiv F \Leftrightarrow (\tau E \Leftrightarrow \tau F) \wedge (\forall x{:}T \bullet E \sqsubseteq x \Leftrightarrow F \sqsubseteq x). \tag{2}$$

It follows that to determine whether a given expression E is refined by another expression F, or whether E is equivalent to F, all we need to know is whether or not E and F are bottom, and if not then what propers refine them. So for any language construct *FOO*, we need to be able to determine τFOO and $FOO \sqsubseteq x$ (fresh x), and in general our strategy will be to define these by axioms. For a simple example, see the axiomatization of $\sqcap$ in Figure 2.

$\sqcap\tau$	$\tau(E \sqcap F) \Leftrightarrow \tau E \wedge \tau F$
$\sqcap\sqsubseteq$	$E \sqcap F \sqsubseteq x \Leftrightarrow E \sqsubseteq x \vee F \sqsubseteq x$

Fig. 2. Axioms for $\sqcap$.

$$\tau(\Box x{:}T \mid P) \Leftrightarrow (\exists x{:}T \bullet P \equiv \mathit{true})$$
$$(\Box x{:}T \mid P) \sqsubseteq y \Leftrightarrow ((\exists x{:}T \bullet P \equiv \mathit{true}) \Rightarrow (\exists x{:}T \bullet P \equiv \mathit{true} \wedge x \sqsubseteq y))$$

Fig. 3. Axioms for prescriptive expressions.

if	$(\mathbf{if}\ P\ \mathbf{then}\ E\ \mathbf{else}\ F \sqsubseteq G) \Leftrightarrow (P \Rightarrow E \sqsubseteq G) \wedge (\neg P \Rightarrow F \sqsubseteq G)$
$\succ$	$(P \succ E \sqsubseteq F) \Leftrightarrow (P \Rightarrow E \sqsubseteq F)$

Fig. 4. Axioms for conditional and assertion expressions.

The defining axioms of prescriptive expressions are given in Figure 3. Every expression can be expressed as a prescription, $E \equiv (\Box x{:}T \mid \tau E \wedge E \sqsubseteq x)$, where x is fresh.

Aside for Logicians. In Figure 3, we write $P \equiv \mathit{true}$ to elevate the boolean term P to a proposition. For brevity, we will write just P, rather than $P \equiv \mathit{true}$, whenever it occurs as the antecedent of an implication. We will further discuss the relationship between booleans and propositions in section 6.

It is possible to axiomatize some language constructs FOO by defining $\mathit{FOO} \sqsubseteq E$ for arbitrary expression E. Then we can determine $\tau \mathit{FOO}$ by instantiating E with $\perp$, and we can determine $\mathit{FOO} \sqsubseteq x$ by instantiating E with x. See, for example, the axiomatizations of conditional and assertion expressions in Figure 4.

The *assertion expression* has the form $P \succ E$; it is equivalent to E if P is true, and otherwise it is $\perp$. Assertion expressions (called "assumptions" in Ward [1994], and discussed, for example, in Möller [1989], and increasingly becoming part of specification and programming languages, e.g. Eiffel [Meyer 1992]), serve to annotate expressions with a little knowledge that can be used in their further refinement. For example, finding a zero of a function on the integers might be specified as $\lambda f{:}\mathbf{Z}\to\mathbf{Z} \bullet (\Box x{:}\mathbf{Z} \mid f\ x = 0)$. If we now wish to inform the implementor that the function will only ever be invoked for monotonically increasing functions, then we can specify

$$\lambda f{:}\mathbf{Z}\to\mathbf{Z} \bullet (\forall x{:}\mathbf{Z} \bullet f\ x \le f\ (x+1)) \succ (\Box x{:}\mathbf{Z} \mid f\ x = 0).$$

4.2 Axiomatizing Types

For each new type that is introduced, we must provide axioms describing its propers. For the case of pair types, properness seems perfectly straightforward: all proper pairs are of the form (x, y) for proper constituents x and y, and for every proper x and y, (x, y) is a proper pair. This is captured in the first two axioms in Figure 5.

Figure 5 also gives the axiomatization of operations on pairs. Observe the shape: there are two axioms to describe each type constructor—here, just pair formation, and two axioms to describe each type destructor—here, the two projection functions (we just describe the left projection **fst**, the right projection **snd** is similar). The

$$
\begin{array}{l}
\forall z{:}T\times U \bullet \exists x{:}T, y{:}U \bullet z\equiv(x,y)\\
\forall x{:}T, y{:}U \bullet \exists z{:}T\times U \bullet z\equiv(x,y)\\
\\
\tau(E,F) \Leftrightarrow \tau E \wedge \tau F\\
(E,F)\sqsubseteq x \Leftrightarrow (\tau F \Rightarrow E \sqsubseteq \mathbf{fst}x)\wedge(\tau E \Rightarrow F \sqsubseteq \mathbf{snd}x)\\
\\
\tau(\mathbf{fst}E) \Leftrightarrow \tau E\\
\mathbf{fst}E \sqsubseteq x \Leftrightarrow (\exists y{:}U \bullet E \sqsubseteq (x,y))
\end{array}
$$

Fig. 5. Axioms for pair types.

third axiom states that pair formation is strict in both arguments. The fourth states that refinement is carried out componentwise; the formulation of the axiom is slightly cluttered by the need to state that refinement is guaranteed if either component is $\bot$. Because of the intimate relationship between refinement and choice, the import of this axiom is that pair-formation distributes over choice in both arguments. The axioms for **fst** are obvious. The final axiom in fact implies that **fst** distributes over choice.

5. AXIOMATIZING FUNCTIONS

We axiomatize function types, following the same strategy as for pairs, and keeping in mind the desired properties of functions collected earlier. Recall that the bodies of λ-abstractions must be monotone in the bound variable.

5.1 The Core Axioms

A partial axiomatization of specificational functions is given in Figure 6. Recall that we use lowercase letters for propers, and uppercase letters for arbitrary expressions. Consequently, the lowercase variables in Figure 6 can be universally quantified over (we have not done so to minimize syntactic clutter), but not the uppercase ones. We have also omitted types to avoid clutter. For example, the first axiom in full is $\forall f{:}T\to U \bullet f\equiv(\lambda x{:}T \bullet f\ x)$.

The first axiom is just familiar η-equivalence on proper functions and shows that every proper function can be written as a λ-abstraction. Taking pair types as our model, we should expect a companion axiom which determines which λ-abstractions are proper. It turns out that a lot can be said without committing to that axiom, so we postpone further consideration of it for a moment.

The second group of axioms defines λ-abstraction. Axiom $\lambda\tau$ ensures that our desire to distinguish $\bot$ and $\lambda x{:}T \bullet \bot$ is met. Axiom $\lambda\sqsubseteq$ implies our desired property of extensional equivalence for proper functions.

The remaining axioms define the single function destructor, viz., application, in effect asserting that function application is strict and distributive and that application can be reduced to substitution. The essential interplay between abstraction and application is captured in axiom $\beta\equiv$ (there is nothing corresponding to axiom $\beta\equiv$ in pair types). Axiom ap.τ states the three ways by which function application yields $\bot$: either the function is $\bot$, or the argument is $\bot$, or the application is normal but yields $\bot$. Observe in both ap.τ and ap.$\sqsubseteq$ that function application $E\ F$ is determined by considering all applications $e\ f$ where e and f are proper refinements of E and F, respectively.

η	$f \equiv (\lambda x \bullet f\ x)$
$\lambda\tau$	$\tau(\lambda x \bullet E)$
$\lambda\sqsubseteq$	$(\lambda x \bullet E) \sqsubseteq f \Leftrightarrow (\forall x \bullet E \sqsubseteq f\ x)$
ap.τ	$\tau(E\ F) \Leftrightarrow \tau E \wedge \tau F \wedge (\forall e \mid E \sqsubseteq e \bullet (\forall f \mid F \sqsubseteq f \bullet \tau(e\ f)))$
ap.$\sqsubseteq$	$E\ F \sqsubseteq x \Leftrightarrow (\exists e \mid E \sqsubseteq e \bullet (\exists f \mid F \sqsubseteq f \bullet e\ f \sqsubseteq x))$
$\beta\equiv$	$(\lambda x \bullet E)\ y \equiv E[x\backslash y]$

Fig. 6. Core axiomatization of functions.

ap./$\bot$	$E\ \bot$	$\equiv$	$\bot$	
ap./$\bot$	$\bot\ E$	$\equiv$	$\bot$	
ap./$\sqcap$	$E\ (F \sqcap G)$	$\equiv$	$E\ F \sqcap E\ G$	
ap./$\sqcap$	$(E \sqcap F)\ G$	$\equiv$	$E\ G \sqcap F\ G$	
ap.mon.	$E \sqsubseteq F$	$\Rightarrow$	$E\ G \sqsubseteq F\ G$	
ap.mon.	$G \sqsubseteq H$	$\Rightarrow$	$E\ G \sqsubseteq E\ H$	
ext.$\sqsubseteq$	$f \sqsubseteq g$	$\Leftrightarrow$	$(\forall x \bullet f\ x \sqsubseteq g\ x)$	
ext.$\equiv$	$f \equiv g$	$\Leftrightarrow$	$(\forall x \bullet f\ x \equiv g\ x)$	
λ mon.	$(\forall x \bullet E \sqsubseteq F)$	$\Leftrightarrow$	$(\lambda x \bullet E) \sqsubseteq (\lambda x \bullet F)$	
$\lambda\alpha$	$(\lambda x \bullet E)$	$\equiv$	$(\lambda y \bullet E[x\backslash y])$	x not in the free variables of E
$\lambda/\sqcap$	$(\lambda x \bullet E \sqcap F)$	$\sqsubseteq$	$(\lambda x \bullet E) \sqcap (\lambda x \bullet F)$	

Fig. 7. Some theorems which follow from the axioms of Figure 6.

Even though we have not yet fixed what abstractions are proper, the axioms of Figure 6 imply a large body of desired theorems, including those listed in Figure 7. In particular, the function application is strict, distributive, and monotone on either side. For proper functions, refinement and equivalence are extensional. A λ-abstraction can be refined by refining its body (assuming monotonicity is preserved). The bound variable in an abstraction can be renamed (this is assuming that the logic gives renaming of universally quantified variables). Finally, moving choice out of an abstraction is a refinement, not an equivalence as might be expected. To see this, observe that $\lambda x{:}\mathbf{Z} \bullet \mathbf{if}\ \mathit{even}\ x\ \mathbf{then}\ 2\ \mathbf{else}\ 3$ refines $\lambda x{:}\mathbf{Z} \bullet 2 \sqcap 3$, whereas $(\lambda x{:}\mathbf{Z} \bullet 2) \sqcap (\lambda x{:}\mathbf{Z} \bullet 3)$ has as refinements only itself, $\lambda x{:}\mathbf{Z} \bullet 2$ and $\lambda x{:}\mathbf{Z} \bullet 3$.

To illustrate the axioms, we prove λ mon. from the axioms and then deduce $\lambda/\sqcap$:

$$\begin{array}{ll}
 & (\lambda x \bullet E) \sqsubseteq (\lambda x \bullet F) \\
\Leftrightarrow & \quad (1) \\
 & (\tau(\lambda x \bullet E) \Rightarrow \tau(\lambda x \bullet F)) \wedge (\forall f \bullet (\lambda x \bullet E) \sqsubseteq f \Leftarrow (\lambda x \bullet F) \sqsubseteq f) \\
\Leftrightarrow & \quad \text{axioms } \lambda\tau,\ \lambda\sqsubseteq \\
 & \forall f \bullet (\forall x \bullet E \sqsubseteq f\ x) \Leftarrow (\forall x \bullet F \sqsubseteq f\ x) \\
\Leftarrow & \quad \text{logic, } \sqsubseteq \text{ transitive} \\
 & \forall x \bullet E \sqsubseteq F.
\end{array}$$

The other direction follows easily from ap.mon. and $\beta\equiv$. We deduce that

$$\begin{array}{ll}
 & (\lambda x \bullet E \sqcap F) \sqsubseteq (\lambda x \bullet E) \sqcap (\lambda x \bullet F) \\
\Leftrightarrow & \quad \sqcap \text{ is greatest lower bound with respect to } \sqsubseteq \\
 & (\lambda x \bullet E \sqcap F) \sqsubseteq (\lambda x \bullet E) \wedge (\lambda x \bullet E \sqcap F) \sqsubseteq (\lambda x \bullet F) \\
\Leftrightarrow & \quad \lambda\text{mon.} \\
 & (\forall x \bullet E \sqcap F \sqsubseteq E) \wedge (\forall x \bullet E \sqcap F \sqsubseteq F),
\end{array}$$

which is a property of choice.

The axioms of Figure 6 resolve all the anomalies of section 3. For example, recall functions $F \mathrel{\hat{=}} (\lambda x{:}\mathbf{Z} \bullet x \sqcap 3)$ and $G \mathrel{\hat{=}} (\lambda x{:}\mathbf{Z} \bullet x) \sqcap (\lambda x{:}\mathbf{Z} \bullet 3)$. We used the putative equivalence of these functions to construct difficulties with both η and extensionality. Although they remain extensionally equal, they are no longer equivalent:

$$\begin{array}{ll}
 & F \equiv G \\
\Leftrightarrow & \qquad (2) \\
 & (\tau F \Leftrightarrow \tau G) \wedge (\forall f \bullet F \sqsubseteq f \Leftrightarrow G \sqsubseteq f) \\
\Leftrightarrow & \qquad \tau F,\ \tau G,\ \text{arbitrary } f \\
 & F \sqsubseteq f \Leftrightarrow G \sqsubseteq f \\
\Leftrightarrow & \qquad G,\ \sqcap\sqsubseteq \\
 & F \sqsubseteq f \Leftrightarrow ((\lambda x{:}\mathbf{Z} \bullet x) \sqsubseteq f \vee (\lambda x{:}\mathbf{Z} \bullet 3) \sqsubseteq f) \\
\Leftrightarrow & \qquad F,\ \lambda\sqsubseteq \\
 & (\forall x{:}\mathbf{Z} \bullet x \sqcap 3 \sqsubseteq f\ x) \Leftrightarrow (\forall x{:}\mathbf{Z} \bullet x \sqsubseteq f\ x) \vee (\forall x{:}\mathbf{Z} \bullet 3 \sqsubseteq f\ x).
\end{array}$$

The final line does not in general hold, which can be seen by instantiating f with $\lambda x \bullet \mathbf{if}\ \ even\, x\ \ \mathbf{then}\ \ x\ \ \mathbf{else}\ \ 3$.

5.2 Proper Functions

We must now address the question "What functions are proper?" First, we definitely expect traditional functions to be proper, i.e., those abstractions $\lambda x{:}T \bullet E$ for which E is proper for all (proper) x.

Second, we have already ruled that $\perp_{T\to U}$ is not proper.

Third, consider those abstractions $\lambda x{:}T \bullet E$ for which E is $\perp$ for some or all values of x and proper for the remaining values of x, i.e., the traditional *partial* functions. Assume for a moment that we deem these to be improper, and consider the extreme example $\lambda x{:}T \bullet \perp$. Because $\lambda x{:}T \bullet \perp$ is improper and not $\perp$, it must be equivalent to the choice over its proper refinements. But that is impossible, because by our assumption all proper functions have proper outcomes. We conclude that we must include the partial functions among the propers.

Finally, we are left with the question of whether functions such as $\lambda x \bullet 2 \sqcap 3$ whose bodies contain an unresolvable choice are to be considered proper. There seem to be two reasonable views. The liberal view is to accept all abstractions as proper. The conservative one is to regard $\lambda x \bullet 2 \sqcap 3$ as improper, in which case it is equivalent to the choice over the (infinite) collection of proper functions that refine it, i.e., all functions whose application to any argument is equivalent to either 2 or 3 (but not $2 \sqcap 3$), including $\lambda x \bullet 2$, $\lambda x \bullet 3$, $\lambda x \bullet \mathbf{if}\ \ even\, x\ \ \mathbf{then}\ \ 2\ \ \mathbf{else}\ \ 3$ and many others. According to this view, an abstraction is proper if and only if its body is proper or bottom.

The liberal view can be adopted by postulating the axiom

$$\lambda\Delta \qquad \Delta(\lambda x{:}T \bullet E).$$

(Incidentally, axiom $\lambda\tau$ can be erased from Figure 6 in the presence of this axiom since it follows therefrom, assuming the underlying logic ensures $\forall x{:}T \bullet \tau x$.) It has the disadvantage that the proper functions include rather exotic elements which we will frequently have to exclude explicitly, cluttering up our specifications. On the other hand, the liberal view is often convenient calculationally. In particular, we can tell by appearance whether a function F is proper or not, and hence whether F can be the subject of an instantiation, and whether $\beta\equiv$ applies when F is the argument of a higher-order function.

The conservative view is embodied in the axiom

$$\lambda\Delta' \qquad \Delta(\lambda x{:}T \bullet E) \Leftrightarrow (\forall x{:}T \bullet \tau E \Rightarrow \Delta E).$$

It states, in effect, that the proper functions are the partial functions of mathematics. However, in calculations it may be hard to prove that a given abstraction is proper, or that there exists a proper function satisfying some property. For instance, to show that $\lambda x \bullet E$ has a proper refinement, i.e., $\exists f \bullet (\lambda x \bullet E) \sqsubseteq f$, we have to show $\exists f \bullet \forall x \bullet E \sqsubseteq f\ x$. But it is not enough to exhibit a mapping from x to y_x such that $\forall x \bullet E \sqsubseteq y_x$—we must find a *monotone* mapping.

Although the liberal and conservative views seem roughly equally attractive on the surface, the conservative view has to be rejected because of a subtle anomaly. Consider function $G \mathrel{\hat{=}} \lambda x{:}\mathit{Two} \bullet \textbf{if } x{\equiv}u \textbf{ then } 1 \sqcap 2 \textbf{ else } 1$. This would seem to have two refinements (other than itself), i.e., $G_1 \mathrel{\hat{=}} \lambda x{:}\mathit{Two} \bullet \textbf{if } x{\equiv}u \textbf{ then } 1 \textbf{ else } 1$ and $G_2 \mathrel{\hat{=}} \lambda x{:}\mathit{Two} \bullet \textbf{if } x{\equiv}u \textbf{ then } 2 \textbf{ else } 1$. However, G_2 has to be rejected because it is not monotone. We can now conclude with the help of law (2) that $G \equiv G_1$, in contradiction of our assumption that G_1 is proper while G is not! It seems that the only reasonable escape is to include G among the propers. Now (2) implies $G \not\equiv G_1$ because G has G itself as a proper refinement, whereas G_1 does not. The root cause of this anomaly is that certain seemingly natural refinements of functions are lost to the monotonicity requirement. Although this is true of both the liberal and conservative views, it is not significant in the former view because the "parent" function is retained among the proper refinements, and so no information is lost.

In conclusion, specificational functions are axiomatized by Figure 6, with axiom $\lambda\tau$ replaced by $\lambda\Delta$.

5.3 Total Functions

Function types bring a new subtlety to the notion of properness. Up to this point, we have had the property that if E is proper then it has no refinements other than itself, but this no longer holds once function types are introduced. For example, $(\lambda x{:}\mathbf{Z} \bullet 0{\le}x \succ 2) \sqsubseteq (\lambda x{:}\mathbf{Z} \bullet 2)$, although both functions are proper. Let us say that an expression E (of type T, say) is *total* and satisfies *total* E, if and only if it has no refinements other than itself, i.e., if and only if $\forall x{:}T \bullet E \sqsubseteq x \Rightarrow E \equiv x$. For example, $\lambda x{:}\mathbf{Z} \bullet 2$ is total, but not $\lambda x{:}\mathbf{Z} \bullet 0{\le}x \succ 2$. On flat types (and pair types derived from flat types), totality and properness are identical. On function types, however, totality is stronger than properness, and in fact specializes to the usual sense of the word.

Totality is obviously an important concept. We can use it to rescue the Skolem axiom which we have shown cannot hold for specificational functions in general. We can prove

$$(\forall x{:}T \mid \mathit{total}\ x \bullet \exists y{:}U \bullet P) \Rightarrow (\exists f{:}T{\rightarrow}U \bullet (\forall x{:}T \mid \mathit{total}\ x \bullet P[y\backslash f\ x])).$$

There still remains an irritation, however: although Skolem promises the existence of a function f satisfying a property P, it is not guaranteed that all the

$$\begin{array}{ll} \text{unfold} & (\mu f \bullet \lambda x \bullet E) \equiv (\lambda x \bullet E[f \backslash \mu f \bullet \lambda x \bullet E]) \\ \text{prefix} & (\lambda x \bullet E[f \backslash F]) \sqsubseteq F \Rightarrow (\mu f \bullet \lambda x \bullet E) \sqsubseteq F \end{array}$$

Fig. 8. Axioms for recursive functions.

refinements of f also satisfy P. For that, we should restrict ourselves to predicates $P(x, y)$ that are monotone in y in the sense $y \sqsubseteq y' \Rightarrow (P(x, y) \Rightarrow P(x, y'))$ for all x. This condition is automatically satisfied when the type of y is flat.

5.4 Recursive Functions

Recursively defined functions have the form $\mu f{:}T{\to}U \bullet \lambda x{:}T \bullet E$ where E is of type U and may contain free occurrences of f. This notation allows us to write a recursive function without naming it. More usually, recursive functions are simultaneously described and named by writing $f \mathrel{\hat{=}} \lambda x{:}T \bullet E$, where f may occur in E. The body $\lambda x{:}T \bullet E$ of the recursive function must be monotone in f to ensure that the fixpoint exists (more details of this are given in the section on denotational semantics below). We axiomatize recursion in the standard way as a fixpoint, and the least prefixpoint with respect to refinement; see Figure 8 (again, types have been omitted).

In program development, λ-abstractions are refined to recursive functions by *recursive refinement*. Suppose we set out to implement the specification $\lambda x \bullet E$ via recursion. Our modus operandi is to proceed through a series of refinements beginning with E and ending with a term of the form $F[\lambda x{\bullet}E]$ where $\lambda x{\bullet}E$ appears within F. Along the path, we will have transmuted nonalgorithmic notation into algorithmic substitutes. We can now *almost* conclude that $\mu f \bullet \lambda x \bullet F[f]$ is the recursive function we were after, i.e. $(\lambda x \bullet E) \sqsubseteq (\mu f \bullet \lambda x \bullet F[f])$. We say "almost" because we have not ensured that the recursion is well-founded, or in programming terms, that invocations of the recursive function terminate. As every programmer should to know (although not necessarily in formal terms), to ensure termination of a recursive function, the argument of each recursive call must be less than the 'incoming' arguments with respect to some well-ordering. This added requirement is captured in the formal statement of the recursion introduction theorem below:

THEOREM (μFUN). *If $E \sqsubseteq F[f \backslash \lambda y \bullet y{<}x \succ E[x \backslash y]]$ for all x, then $(\lambda x \bullet E) \sqsubseteq (\mu f \bullet \lambda x \bullet F[f])$, where $<$ is a well-order on the source type of f.*

PROOF. To begin:

$$\begin{array}{ll} & (\lambda x \bullet E) \sqsubseteq (\mu f \bullet \lambda x \bullet F) \\ \Leftrightarrow & \qquad \text{axm unfold} \\ & (\lambda x \bullet E) \sqsubseteq (\lambda x \bullet F[f \backslash \mu f \bullet \lambda x \bullet F]) \\ \Leftrightarrow & \qquad \text{thm } \lambda \text{ mon.} \\ & \forall x{:}T \bullet E \sqsubseteq F[f \backslash \mu f \bullet \lambda x \bullet F] \end{array}$$

We prove $\forall x{:}T \bullet E \sqsubseteq F[f \backslash \mu f \bullet \lambda x \bullet F]$ by well-founded induction

$$\frac{\forall x{:}T \bullet (\forall y{:}T \bullet y{<}x \Rightarrow P[y]) \Rightarrow P[x]}{\forall x{:}T \bullet P[x]},$$

where here $P[x] \mathrel{\hat{=}} E \sqsubseteq F[f \backslash \mu f \bullet \lambda x \bullet F]$. With arbitrary x of type T we have

$$\begin{array}{ll}
& E \sqsubseteq F[f\backslash \mu f \bullet \lambda x \bullet F] \\
\Leftarrow & \quad \text{assumption of the theorem, and } \sqsubseteq \text{ transitive} \\
& F[f\backslash \lambda y \bullet y{<}x \succ E[x\backslash y]] \sqsubseteq F[f\backslash \mu f \bullet \lambda x \bullet F] \\
\Leftarrow & \quad F \text{ mon. in expression variable } f \\
& (\lambda y \bullet y{<}x \succ E[x\backslash y]) \sqsubseteq (\mu f \bullet \lambda x \bullet F) \\
\Leftrightarrow & \quad \text{axm unfold} \\
& (\lambda y \bullet y{<}x \succ E[x\backslash y]) \sqsubseteq (\lambda x \bullet F[f\backslash \mu f \bullet \lambda x \bullet F]) \\
\Leftrightarrow & \quad \text{thm } \lambda\alpha \\
& (\lambda y \bullet y{<}x \succ E[x\backslash y]) \sqsubseteq (\lambda y \bullet F[f\backslash \mu f \bullet \lambda x \bullet F][x\backslash y]) \\
\Leftrightarrow & \quad \text{thm } \lambda\text{mon.} \\
& \forall y \bullet (y{<}x \succ E[x\backslash y]) \sqsubseteq F[f\backslash \mu f \bullet \lambda x \bullet F][x\backslash y] \\
\Leftrightarrow & \quad \text{axm } \succ \\
& \forall y \bullet y{<}x \Rightarrow (E[x\backslash y] \sqsubseteq F[f\backslash \mu f \bullet \lambda x \bullet F][x\backslash y]).
\end{array}$$

□

5.5 Example

We construct a small program to decompose a natural number into the sum of two squares. The starting specification is

$$ss \mathrel{\hat{=}} \lambda n{:}\mathbf{N} \bullet (\Box(i,j){:}\mathbf{N}\times\mathbf{N} \mid i^2 + j^2 = n) : \mathbf{N}\rightarrow\mathbf{N}\times\mathbf{N}.$$

We proceed in the standard way for such problems by choosing a finite search space containing an i and j such that $i^2 + j^2 = n$ (assuming there is such an i, j), and then repeatedly testing the pairs it contains. If the candidate (i, j) we test satisfies $i^2 + j^2 = n$ we are done, and if not we reduce the search space by removing (i, j) (and any other pairs we can eliminate at the same time). We choose as our search space all the (i, j)'s such that $lo \le i \le j \le hi$ where lo and hi are some naturals, initially 0 and $\lfloor\sqrt{n}\rfloor$, respectively. In summary,

$$ss \sqsubseteq \lambda n{:}\mathbf{N} \bullet ss_1(0, \lfloor\sqrt{n}\rfloor)$$

where

$$ss_1 \mathrel{\hat{=}} \lambda lo, hi{:}\mathbf{N} \bullet (\Box(i,j){:}\mathbf{N}\times\mathbf{N} \mid lo \le i \le j \le hi \wedge i^2 + j^2 = n) : \mathbf{N}\times\mathbf{N}\rightarrow\mathbf{N}\times\mathbf{N}.$$

Let E denote the body of ss_1. It is not difficult to conclude the following facts about E:

$$\begin{array}{l}
lo^2 + hi^2 = n \Rightarrow E \sqsubseteq (lo, hi) \\
lo^2 + hi^2 < n \Rightarrow E \equiv E[lo\backslash lo + 1] \\
lo^2 + hi^2 > n \Rightarrow E \equiv E[hi\backslash hi - 1].
\end{array}$$

Using elementary properties of **if** ... **then** ... **else** ... we infer

$$\begin{array}{ll}
E \sqsubseteq & \textbf{if}\, lo^2 + hi^2 = n\, \textbf{then}\, (lo, hi) \\
& \textbf{else if}\, lo^2 + hi^2 < n\, \textbf{then}\, ss_1(lo + 1, hi) \\
& \textbf{else}\, ss_1(lo, hi - 1).
\end{array}$$

For a termination argument, observe first that τE implies $lo \le hi$. Second, observe that in comparison with E, which is $ss_1(lo, hi)$, the absolute difference of the two arguments of $ss_1(lo + 1, hi)$ is less by 1, and similarly for $ss_1(lo, hi -$

1). Practitioners will recognize the ingredients for the well-founded ordering to show termination when τE holds; we omit the details. When τE does not hold, termination is not an issue. Hence, we have shown $ss1 \sqsubseteq f$ where

$$\begin{aligned} f \mathrel{\hat{=}} \lambda lo, hi{:}\mathbf{N}\bullet\ & \mathbf{if}\ lo^2 + hi^2 = n\ \mathbf{then}\ (lo, hi) \\ & \mathbf{else\ if}\ lo^2 + hi^2 < n\ \mathbf{then}\ f(lo+1, hi) \\ & \mathbf{else}\ f(lo, hi-1). \end{aligned}$$

6. BOOLEANS AND PROPOSITIONS

With the introduction of bottom and choice, the question arises as to what is the most appropriate logic for reasoning about specifications. There appear to be two reasonable approaches. On the one hand, we can take steps to keep bottom and choice out of the logic language, enabling us to employ traditional two-valued predicate logic. On the other hand, we might move to a many-valued logic that explicitly accommodates bottom and choice. In the latter approach, propositions need not necessarily be associated with truth and falsity, but might in addition be associated with bottom ("neither-true-nor-false") or $true \sqcap false$ ("possibly true, possibly false"). Our axiomatization of specificational functions does not depend on this choice. However, in the axiomatization of some language constructs, in particular those given in Figures 3 and 4, it does indeed matter. The reader need not be overly concerned with this, partly because there is a good axiomatization of these constructs no matter which route we choose, and partly because the constructs are peripheral to our main concern, namely functions. That said, our own preference is to employ a many-valued logic; we explain why.

The introduction of bottom and choice to the boolean type needs to be undertaken with special care. The two standard approaches are to define the boolean connectives to be strict and distributive (see for example Larsen and Hansen [1996]) or as left-to-right evaluating (see for example Partsch [1990]). However, these turn out to be seriously unattractive: many of the familiar laws of boolean algebra break down, and the booleans become awkward to handle calculationally. It turns out, however, that the boolean connectives can be extended to accommodate bottom and choice in quite a different way such that almost all the familiar laws of boolean algebra continue to hold (the main loss is the law of the excluded middle). This is described in Morris and Bunkenburg [1998a; 1998b].

Actually, because the language of a refinement calculus is used to make specifications (which are a superset of programs), the operations in every type will be about as rich as mathematics can provide. In particular, in specificational languages the boolean type includes arbitrary universal and existential quantifications, even over infinite domains. Again, it turns out that the quantifiers can be suitably extended to allow for bottom and choice while retaining just about all the laws of predicate calculus (the only serious loss is that instantiation can only be carried out with proper terms). Again, see Morris and Bunkenburg [1998b] for the details.

Because we have had to construct a formal many-valued logic to reason about boolean expressions, we have elected to avoid duplicated effort by adopting the same logic for reasoning about specifications in general. In short, we do not distinguish between propositions and boolean expressions, and hence our logic language propositions do double duty, serving also as object language expressions of type boolean.

This means, for example, that even something as exotic as $E \sqsubseteq F$ or $E \equiv F$ is a boolean expression (and hence a specification). There are advantages and disadvantages. One disadvantage is that formal reasoning in many-valued logics is not quite so convenient. An advantage is that it facilitates the smooth transition of specifications to programs, because propositions can migrate effortlessly into specifications/programs, as we often want them to when we formally extract programs from specifications. See Morris and Bunkenburg [1998b] for an example of this.

It turns out that in Morris and Bunkenburg [1998b] $P \Rightarrow Q$ is indeed equivalent to $(P \equiv \mathit{true}) \Rightarrow Q$; thus the convention adopted in this paper is compatible with Morris and Bunkenburg [1998b].

We repeat that our preference for many-valued logics and a fusion of booleans and propositions has no bearing on the axiomatization of specificational functions presented above.

7. MODEL THEORY

In this section, we provide a denotational semantics for the language constructs we have given. The model is parameterized by models of base types and a logic. We require that tt is the denotation of a theorem, and that $f\!f$ is the denotation of an antitheorem. The model of the logic may have denotations for further truth values like $\bot_{\mathbf{B}}$ and $\mathit{true} \sqcap \mathit{false}$.

Consistency of the calculus has been proven. For the proof, the calculus has been augmented by a logic as outlined in section 6 and described in detail in Morris and Bunkenburg [1998b]. However, the choice of logic is not crucial to the axiomatization and model of functions.

The grammar of the types is $\mathit{type} ::= B_0 \,|\, ... \,|\, B_k \,|\, \mathit{type} {\times} \mathit{type} \,|\, \mathit{type} {\rightarrow} \mathit{type}$, for some given base types $B_0, ... B_k$. Each type is interpreted by a complete partial order $([T], \leq^T)$. For each type T, abbreviate $[T] \cup \{\bot\}$ by $[T]_\bot$, where $\bot$ is a special element that none of the $[T]$ contain. For each T, we extend the partial order $([T], \leq^T)$ to $([T]_\bot, \leq_T)$ by defining $t \leq_T u$ as $t = \bot \vee t \leq^T u$.

The usual way of giving semantics to a functional programming language is to interpret every expression of type T as an element of $[T]_\bot$. However, this approach does not accommodate choice.

We can accommodate choice by interpreting every expression E as the subset of $[T]_\bot$ that intuitively contains interpretations of all the possible outcomes of E. To model recursion and refinement, we need an order on these sets. The order appropriate for a "total correctness" calculus (i.e., one in which $E \sqcap \bot \equiv \bot$) such as this one is the "Smyth" order $(\sqsubseteq_0)$ (see Plotkin [1976] and Smyth [1978]), defined by

$$A \sqsubseteq_0 B \;\hat{=}\; \forall b \in B \bullet \exists a \in A \bullet a \leq_T b,$$

where A and B are subsets of $[T]_\bot$. Inconveniently, $(\sqsubseteq_0)$ is not antisymmetric. We make it antisymmetric by restricting it to *upclosed* sets only.

The *upclosure* of a subset S of a partial order $(P, \leq)$ is written $S{\uparrow}_{\leq}$, defined by $\{p \in P \mid (\exists s \in S \bullet s \leq p)\}$. To avoid double subscripts, we abbreviate $S{\uparrow}_{\leq_T}$ by $S{\uparrow}_T$. Conveniently, for upclosed sets, $(\sqsubseteq_0)$ is identical with $(\supseteq)$. Therefore, we will interpret every expression of type T as an upclosed subset of $[T]_\bot$, and give meaning to recursion and refinement using $(\supseteq)$.

T	$[T]$	$t \leq^T u$
$\mathbf{Z}$	$\{..., -1, 0, 1, ...\}$	$t = u$
$T \times U$	$[T] \times [U]$	$(\pi_0 t \leq^T \pi_0 u) \wedge (\pi_1 t \leq^U \pi_1 u)$
$T \rightarrow U$	$[T] \overset{mon}{\rightarrow} \mathcal{P}_1^{\uparrow}[U]$	$\forall v \in [T] \bullet t\ v \supseteq u\ v$

Fig. 9. Interpretations of the types.

For every type T, we denote by $\mathcal{P}^{\uparrow}[T]$ the set $\{S \mid S \subseteq [T]_{\perp} \bullet S{\uparrow}_T\}$, that is, the upclosed subsets of $[T]_{\perp}$. We abbreviate $\mathcal{P}^{\uparrow}[T] \setminus \{\emptyset\}$ as $\mathcal{P}_1^{\uparrow}[T]$.

For each base type B_i, a set $[B_i]$ is given, and it will be ordered simply by equality. Pair types are interpreted as the product of the constituent types. The function type $T \rightarrow U$ is interpreted by the functions f from $[T]$ to $\mathcal{P}_1^{\uparrow}[U]$ that are monotone in the sense that $t \leq^T u \Rightarrow f\ t \supseteq f\ u$. The interpretation of base type $\mathbf{Z}$, and the interpretations of pair types and function types, are collected in Figure 9, where π_0 and π_1 denote the left and right projections from pairs to their constituents.

Construction of a universal domain D, which is a superset of $[T]$ for every type T, is clerical, using the definitions in Figure 9. Readers familiar with models of the untyped λ-calculus will note that we do not require $D \cong D \rightarrow D$, since we are dealing with a typed language.

An environment ρ is a mapping from each variable of type T to an element of $[T]$. In addition, it maps each recursion dummy f of type $T \rightarrow U$ to an element of $\mathcal{P}_1^{\uparrow}[T \rightarrow U]$.

Expressions are interpreted by induction on their structure. Every expression E of type T is interpreted in environment ρ by the set $[E]\rho$, which is an element of $\mathcal{P}_1^{\uparrow}[T]$.

For each constant c of base type T, we are given an element $[c]$ of $[T]$. Since base types are flat, $\{[c]\}$ is upclosed. For every operator symbol $\mathbf{f}$ of the base types we are given a *matching* function $[\mathbf{f}]$. By "matching" we mean that if the argument types of $\mathbf{f}$ are $T_0, ..., T_k$, and its result type is U, then $[\mathbf{f}]$ is a function in $[T_0] \rightarrow ... \rightarrow [T_k] \rightarrow [U]_{\perp}$. Since base types are flat, $[\mathbf{f}]$ is monotone. Alternatively, one could have interpreted operators simply as functional constants. In that case, the denotation of constants would have to be upclosed explicitly.

$[E\ F]\rho$ is upclosed, since unions of upclosed sets are upclosed and since functions types are interpreted by functions to upclosed sets.

We do not give a model for the logic. Rather we assume that every proposition P is interpreted in environment ρ as some element $[P]\rho$. We assume that *tt* is the interpretation of some theorem of the logic, and we assume that *ff* is the interpretation of some antitheorem of the logic. With two-valued logic, the domain of propositions would be $\{tt, ff\}$, the interpretations of true and false. We give interpretations for refinement, equivalence, and the classifiers τ and Δ in Figure 10.

Figure 11 gives the semantics of the typed expressions. It can easily be verified that for every expression E of type T, its interpretation $[E]\rho$ is a set upclosed with respect to $\leq_T$.

In the language of this article, recursion only makes sense for functions. In a more general setting, one could treat recursive expressions of the shape $\mu x{:}T \bullet E$. The recursive expression $\mu f \bullet \lambda x \bullet E$ is interpreted by the fixpoint of the functional F as defined in Figure 11. F acts on $\mathcal{P}^{\uparrow}[T \rightarrow U]$ which is a complete lattice under

proposition	interpretation in ρ
$E \sqsubseteq F$	if $[E]\rho \supseteq [F]\rho$ then *tt* else *ff*
$E \equiv F$	if $[E]\rho = [F]\rho$ then *tt* else *ff*
τE	if $\perp \notin [E]\rho$ then *tt* else *ff*
ΔE	if $[E]\rho$ has a non-$\perp$ minimum then *tt* else *ff*

Fig. 10. Interpretation of refinement and equivalence.

expression X	of type	its interpretation $[X]\rho$
$\perp$	T	$[T]_\perp$
$E \sqcap F$		$[E]\rho \cup [F]\rho$
$P \succ E$		if $[P]\rho = tt$ then $[E]\rho$ else $[\perp]\rho$
$(\Box x{:}T \mid P)$		if $S \neq \emptyset$ then $S{\uparrow_T}$ else $[\perp]\rho$
		where $S \mathrel{\hat{=}} \{t \mid t \in [T] \wedge [P]\rho[x \mapsto t] = tt\}$
if P **then** E **else** F		if $[P]\rho = tt$ then $[E]\rho$ else (if $[P]\rho = ff$ then $[F]\rho$ else $[\perp]\rho$)
c		$\{[c]\}$
$\mathbf{f}\ E_0 ... E_k$	T	$\{e_0, ..., e_k \mid e_0 \in [E_0]\rho \wedge ... \wedge e_k \in [E_k]\rho \bullet f\ e_0 ... e_k\}{\uparrow_T}$
		where $f\ e_0 ... e_k \mathrel{\hat{=}}$ if $(\exists i \bullet e_i = \perp)$ then $\perp$ else $[\mathbf{f}]\ e_0 ... e_k$
x	T	$\{\rho\ x\}{\uparrow_T}$
$E\ F$	T	$\bigcup\{e, f \mid e \in [E]\rho \wedge f \in [F]\rho \bullet ap(e, f)\}$
		where $ap(e, f) \mathrel{\hat{=}}$ if $e{=}\perp \vee f{=}\perp$ then $[T]_\perp$ else $e\ f$
$\lambda x \bullet E$	$T{\to}U$	$\{f \mid f \in [T{\to}U] \wedge (\forall v \in [T] \bullet f\ v = [E]\rho[x \mapsto v])\}{\uparrow_{T\to U}}$
$\mu f \bullet \lambda x \bullet E$	$T{\to}U$	μF, where
		$F : \mathcal{P}^{\uparrow}[T{\to}U] \overset{mon}{\to} \mathcal{P}^{\uparrow}[T{\to}U]$
		$F\ S \mathrel{\hat{=}} [\lambda x \bullet E]\rho[f \mapsto S]$
rec. dummy f		$\rho\ f$
(E, F)	$T \times U$	$\{e, f \mid e \in [E]\rho \wedge f \in [F]\rho \bullet pair(e, f)\}{\uparrow_{T\times U}}$
		where $pair(e, f) \mathrel{\hat{=}}$ if $e = \perp \vee f = \perp$ then $\perp$ else (e, f)
$\mathbf{f}stE$		$\{e \mid e \in [E]\rho \bullet$ if $e = \perp$ then $\perp$ else $\pi_0 e\}$

Fig. 11. Interpretation of the expressions.

the order $\supseteq$. By the language restriction that $\lambda x \bullet E$ must be refinement-monotone in f, we ensure that F is $\supseteq$-monotone, and therefore the generalized limit theorem [Hitchcock and Park 1972; Nelson 1989] ensures that a least fixpoint exists. The mathematics of this construction does not exclude the possibility, in principle, that μF is the empty set, or that μF is a set containing functions that map some arguments to the empty set. If this was the case, it would mean that empty choices (so-called *miracles*) could be created by the back-door of recursion. This would be as surprising as it would be undesirable, and should be impossible for any reasonable specification language. We have not been able to prove that it is indeed impossible for our language. However, we have done so under the mild restriction that the recursive body employ at most finite choice (the proof can then appeal to König's Lemma). In particular, every recursion written in the deterministic programming sublanguage is nonmiraculous.

Finally, we give some examples. The three expressions 2, $2 \sqcap 3$, and $\perp_{\mathbf{Z}}$ are interpreted as $\{2\}$, $\{2, 3\}$, and $\{\perp, 0, 1, -1, 2, -2, ..\}$. The abstraction $\lambda x{:}\mathbf{Z} \bullet 2$ is interpreted as the singleton set containing the function that maps every element of $[\mathbf{Z}]$ to $\{2\}$. The abstraction $\lambda x{:}\mathbf{Z} \bullet 2 \sqcap 3$ is interpreted as the set containing all those functions mapping each element of $[\mathbf{Z}]$ to a nonempty subset of $\{2, 3\}$.

8. CONCLUSION AND RELATED WORK

Specificational functions are made by combining regular functions with choice and bottom, taking special care to accommodate higher-order functions. We have shown examples of their usefulness both in making specifications and in extracting algorithmic functions from those specifications. The technical difficulties that stand in the way of establishing a consistent set of laws for reasoning about them are well known and have led to severe restrictions on their use in the past (see the commentary on related work below). We have picked our way around these difficulties to arrive at an axiomatic treatment which does not fall foul of the anomalies. The key elements in our approach are

(1) a richer notion of equivalence than extensionality, based on refinement;

(2) a restriction on quantifications to range over proper elements only, where properness is carefully defined for each type; and

(3) the imposition of a monotonicity requirement on functions.

Clearly, some compromises have had to be made, but we believe that practitioners could comfortably live with the compromises we have arrived at.

The two standard ways of combining functions and choice are to move to set-valued functions or to generalize functions to binary relations. Here we reject moving to set-valued functions, because in calculations it leads to frequent packing and unpacking of values in to and out of sets even though often the set is just a singleton set. The relational approach lends itself well to calculation (for example see Bird and de Moor [1997] and Brink and Schmidt [1996]). However, we would like to stick with functions because they capture better the directional nature of programs as input-to-output mappings. This decision means we do not lose the direct connection to current program languages which have functions, not relations.

Munich CIP [Bauer et al. 1985] is a development method based on a wide-spectrum language CIP-L; a comprehensive account of it is given in Partsch [1990]. CIP-L has similar concepts to our choice, equivalence ("strong equality"), refinement ("descendancy"), bottom, and properness. Functions are defined as (possibly recursive) abstractions. However, there are severe limitations placed on functions in CIP-L. Most importantly, choice and quantification over functions is forbidden. It is not clear to us how this restriction is enforced. In this article, bodies of λ-abstractions must be monotone in the bound variable. In CIP-L that is automatic, since every language construct is monotone anyway. All λ-abstractions are considered proper. The main transformation rules are α, $\beta\sqsubseteq$ (called "unfold"), and our theorem $E[x\backslash F] \sqsubseteq (\lambda x \bullet E)F$, if τF, but no concise axiomatization is given in Partsch [1990]; so it is not clear which "rules" are axioms and which theorems.

Calculation with functions is dealt with in Hoogerwoord [1989]. The language has underdetermined "where" clauses that specify a value by asserting a property it enjoys. This is closely related to Hilbert's ϵ, and so refinement is pretty much impossible.

Norvell and Hehner present a refinement calculus for expressions [1993], by giving a partial list of axioms for a language similar to the one discussed here. The main differences lie in the treatment of termination and the properness of functions. For termination of recursive programs they annotate the expressions with abstract execution times. It is not clear how refinement and timing relate. An abstraction $\lambda x \bullet E$ is a proper function (in their terminology "an element") if and only if E is proper for all x. Since the language has no bottom, that implies that only traditional functions are elements. It seems their intention to have flat types only, which would avoid the anomalies of our Sections 3.4 and 5.2. However, we believe their system is inconsistent, since there are λ-abstractions that are nonmiraculous refinements of proper functions—which implies that function types are *non*flat. See Bunkenburg and Morris [1999] for details.

In closely related work [Hehner 1993; 1998], a half-and-half approach to distributivity of functions over choice in their arguments is described. The author of that work distinguishes between those occurrences of the parameter in the body of the function for which distribution is appropriate and those for which direct substitution of the argument is appropriate. These occurrences can be syntactically decided, and of course are fixed for each function. Therefore, some λ-abstractions do distribute over choice in their arguments, and others do not. However, this approach seems to impose a host of trivial concerns on the programmer. It is not clear whether there are sufficient compensating gains.

Ward's thesis [1994] presents a functional specification and programming language, defined by a semantics that models demonic and angelic choice. However, the language is not given a proof theory, and there is no suggestion that the given refinement laws are sufficient in practice. New refinement laws can be generated by proving candidates sound using the semantics.

The language of VDM includes "loose let-expressions" of the form

$$\mathbf{let}\ x{:}T\ \mathbf{be\ s.t.}\ P\ \mathbf{in}\ E.$$

Its intended meaning is "E with x bound to an arbitrary value satisfying P." However, its axiomatization has proved elusive, and Bicarregui et al. [1994] suggest the approach taken by Larsen and Hansen [1996].

Larsen and Hansen [1996] present a denotational semantics for a functional language with underdeterminism. The type language is extended by comprehension types of the form $\{E|x{:}T \bullet P\}$, and the expression language is extended by **choice** T, underdeterministically selecting an element of the type T. However, which element is chosen depends on the whole environment, even on those variables that do not occur in T. The proof system is based on generalized type inference, with propositions of the form $E : T$, rather than equivalence and refinement relations. Indeed, it is hard to see how equivalence and refinement would fit in. Larsen and Hansen consider those lambda-abstractions proper that map propers to bottom or propers. Therefore the anomalies of Sections 3.4 5.2 could be reproduced, if strong operators such as $\equiv$ were added to the language.

Previous work by one of the present authors [Morris 1997] gives weakest-condition semantics to an expression language with choice and bottom. The style of semantics fits the weakest-precondition semantics of the imperative refinement calculus. However, no axioms or logic is given, and issues pertaining to functions are not addressed.

ACKNOWLEDGMENT

We thank David Watt and Richard Botting for reviewing drafts of this article. We thank the three anonymous referees for very thorough reviews.

REFERENCES

Bauer, F. L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., and Wössner, H. 1985. *The Munich Project CIP*. Lecture Notes in Computer Science, vol. 183. Springer Verlag, Berlin. ISBN 0-387-15187-7.

Bicarregui, J. C., Fitzgerald, J. S., Lindsay, P. A., Moore, R., and Ritchie, B. 1994. *Proof in VDM: A practioner's guide*. FACIT. Springer Verlag.

Bird, R. and de Moor, O. 1997. *Algebra of Programming*. Prentice Hall, London. ISBN 0-13-507245-X.

Brink, C. and Schmidt, G. 1996. *Relational Methods in Computer Science*. Springer Verlag. Supplemental volume of the journal *Computing*.

Bunkenburg, A. 1997. Expression Refinement. Ph.D. thesis, Computing Science Department, University of Glasgow.

Bunkenburg, A. and Morris, J. M. 1999. Inconsistent Theories of Non-deterministic Functions. *Information Processing Letters*. Submitted.

Hehner, E. 1993. *A practical theory of programming*. Springer Verlag, New York, London.

Hehner, E. C. R. 1998. Unified algebra. In preparation.

Hitchcock, P. and Park, D. 1972. Induction Rules and Termination Proofs. In *IRIA Conference on Automata, Languages, and Programming Theory*.

Hoogerwoord, R. R. 1989. The design of functional programs: a calculational approach. Ph.D. thesis, Technische Universiteit Eindhoven.

Larsen, P. G. and Hansen, B. S. 1996. Semantics of under-determined expressions. *Formal Aspects of Computing 8,* 1, 47–66.

Leisenring, A. C. 1969. *Mathematical Logic and Hilbert's ϵ-symbol.* MacDonald and Co., London.

Meertens, L. 1986. Algorithmics - Towards Programming as a Mathematical Activity. *Mathematics and Computer Science 1*, 1–46. CWI Monographs (J. W. de Bakker, M. Hazewinkel, J. K. Lenstra, eds.) North Holland, Publ. Co.

Meyer, B. 1992. *Eiffel: The Language.* Prentice Hall.

Möller, B. 1989. Applicative assertions. In *Mathematics of Program Construction*, J. Van de Snepscheut, Ed. Lecture Notes in Computer Science, vol. 375. Springer Verlag, Berlin, 348–362.

Morris, J. M. 1997. Non-deterministic expressions and predicate transformers. *Information Processing Letters 61*, 241–246.

Morris, J. M. and Bunkenburg, A. 1998a. E3: A logic for reasoning equationally in the presence of partiality. *Science of Computer Programming*. To appear.

Morris, J. M. and Bunkenburg, A. 1998b. Partiality and nondeterminacy in program proofs. *Formal Aspects of Computer Science 10*, 76–96.

Nelson, G. 1989. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems 11,* 4 (Oct.), 517–561.

Norvell, T. S. and Hehner, E. C. R. 1993. Logical specifications for functional programs. In *Proceedings of the Second International Conference on Mathematics of Program Construction, Oxford, 29 June - 3 July 1992.* Lecture Notes in Computer Science, vol. 669. Springer Verlag, 269–290.

Partsch, H. A. 1990. *Specification and Transformation of Programs.* Springer Verlag, New York.

Peterson, J. and Hammond, K., e. 1997. Report on the Programming Language Haskell. http://haskell.org/report/.

Plotkin, G. 1976. A Powerdomain Construction. *SIAM Journal of Computing 5,* 3, 452–487.

Smyth, M. B. 1978. Power domains. *Journal of Computer and System Sciences 16,* 1, 23–26.

Søndergaard, H. and Sestoft, P. 1992. Non-determinism in functional languages. *The Computer Journal 35,* 5, 514 – 523.

WARD, N. 1994. A refinement calculus for nondeterministic expressions. Ph.D. thesis, University of Queensland.

WIRTH, N. 1973. *Systematic Programming, an Introduction.* Prentice Hall.

Received August 1998; accepted January 1999