Partial Functions and Undefined Terms in Logic

References

- W.M. Farmer, "A partial functions version of Church's simple theory of types," *Journal of Symbolic Logic*, 55:1269-1291, 1990.
- Parnas, D.L., "Predicate logic for software engineering," *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, September 1993, pp. 856 – 862.
- J. Crow *et al.*, "A tutorial introduction to PVS," In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.
  Also available at:
  `http://www.csl.sri.com/papers/wift-tutorial/`
- J. Rushby, S. Owre, N. Shankar, "Subtypes for specifications: Predicate subtyping in PVS," *IEEE Transactions on Software Engineering*, Vol. 24, No. 9, September 1998, pp. 709–720.
  `http://www.csl.sri.com/papers/tse98/`

Outline

- Preliminaries
- Motivation
- A cautionary tale
- Methods of handling partial functions
- Comparison of methods
- Summary

Preliminaries: Partial & Total Functions

Let A and B be sets. Let $f \subset A \times B$ such that if $(a, b) \in f$ and $(a, b') \in f$ then $b = b'$. In this case we write $f : A \to B$ and call f a function.

We often do not make a distinction as to whether the function is defined for every possible argument (i.e. Is f totally defined for all of A or only partially defined?).

Def: Let $dom(f) = \{a \in A | \exists b \in B : f(a) = b\}$ be the *domain of* f. If $dom(f) = A$ we say that f is a *total function*, otherwise we say that f is a *partial function*.

E.g. Addition $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and multiplication $\cdot : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ are total functions but division $/ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is partial. (Why?)

Motivation:

In our definition of predicate logic:

- Only one "sort" of objects, those in our universe A.
- All functions are total: $f(a,b)$ is always some element of A
- All predicates are always defined: $P(f(a,b),c)$ is either true or false. I.e. $P : A^2 \to \{F,T\}$ is total.

Value of logical expressions containing undefined terms is undefined: $1/0 \leq 2/0$

Thus not "allowed" to reason about / on $\mathbb{R}$!

Problems with current logic:

1. Often don't care about all values.
2. Makes notation cumbersome.
3. Restricts what we can say.

Motivation:

Ex. 1 - Consider statement: "There is a student who has a passing mark in every course."

$$\exists x(S(x) \wedge \forall y(C(y {\to} P(m(x,y)))))$$

What is $m(x,x)$ or $m(y,y)$?

Ex. 2 - Dealing with arrays: An n element array f does not contain any duplicate elements:

$$\forall i \forall j(1 \leq i \wedge i \leq n \wedge 1 \leq j \wedge j \leq n$$
$$\wedge i \neq j \to f(i) \neq f(j))$$

or alternatively

$$\forall i \forall j(1 \leq i \wedge i \leq n \wedge 1 \leq j \wedge j \leq n$$
$$\wedge f(i) = f(j) \to i = j)$$

Ex. 3: Consider code:

```
if (x>=0)
  y=sqrt(x);
else y=sqrt(-x);
```

In PVS we could model this as:

```
P1: PROPOSITION IF x>=0 THEN y=sqrt(x)
                ELSE y=sqrt(-x) ENDIF
```

This is logically equivalent to low level spec:

$$(x \geq 0 \to y = \sqrt{x}) \wedge (x < 0 \to y = \sqrt{-x})$$

Problem: Contains undefined terms for every $x \neq 0$.

High level spec would be: $y = \sqrt{|x|}$

Partial functions in Logic Wish List

Partial functions are often used to specify software and are implemented in software.

For software engineering we need a way of specifying observed behavior of a program using logic that has:

1. Total predicates: Must have "yes" or "no" answer, not "maybe".
2. Concise notation: If it is too complicated, it will not be used (correctly) or understood.
3. Intuitive: Must capture engineer's intended meaning.
4. Consistent: Must not get "false positives" (must not be able to "prove" that programs satisfies a specification when it does not)

Methods for handling partial functions

a) Traditional analysis: Define consistent way of dealing with undefined terms

b) Traditional logic: Eliminate undefined terms by making all functions total through Types and Bounded Quantification

c) Three valued logic - True, False & Undefined

Method (c) makes predicates partial so we won't consider it.

A Cautionary Tale: Do formal "proof" of $1 = 2$.

Traditional Analysis Approach to Partial Functions and Undefinedness

Terms (expressions) may be undefined

- Constants, variables always defined
- Functions may be partial so their application might be undefined (e.g. $1/0, \sqrt{-1}$)
- application of function is undefined if any argument is undefined (e.g. $0 * 1/0$ is undefined!)

Once values are assigned to free variables, any formula must be either true or false.

How? Make predicates total by say that predicates (including $=$) are False if any argument is undefined.

Thus $1/0 \neq 1/0$

Traditional Analysis Approach:

Used in theorem prover IMPS and some practical software engineering approaches.

Main Idea: Any atomic predicate containing an undefined term is False!

Note: Ex. 3 now has intended meaning

$$(x \geq 0 \rightarrow y = \sqrt{x}) \wedge (x < 0 \rightarrow y = \sqrt{-x})$$

is equivalent to $y = \sqrt{|x|}$.

Caveat: $\neg(\sqrt{x} \leq \sqrt{y}) \not\Leftrightarrow \sqrt{x} > \sqrt{y}$

Restriction of Quantifiers

Often want to restrict ourselves to considering x's of certain *type*.

$$\forall x(P(x) \rightarrow Q(x))$$

$$\exists x(P(x) \wedge Q(x))$$

E.g. In Dilbert $\forall x(Manager(x) \rightarrow Idiot(x))$
$\exists x(Animal(x) \wedge \neg Glasses(x))$

What is the relationship between these two forms?

$$\neg\forall x(P(x) \rightarrow Q(x)) \text{ iff } \exists x(P(x) \wedge \neg Q(x))$$

Why?

Note: Other styles of quantification

$$(\forall x \in P)Q(x) \text{ or } \forall x \in P : Q(x)$$

mean same as $\forall x(Px \rightarrow Qx)$

$\exists x(Px \wedge Qx)$ is also written:

$$(\exists x \in P)Q(x) \text{ or } \exists x \in P : Q(x)$$

read "There exists an x in P such that Q(x) holds."

This starts to lead into Type Theory.

Bounded Quantification

Idea: Restrict quantification to values in domain of function E.g. $(\forall x \in dom(f))Q(f(x))$

Problem: Works for Traditional Analysis Approach where undefined terms allowed but not Traditional Logic Approach where all functions must be total. Why?

$$(\forall x \in dom(f))Qf(x) \text{ means}$$
$$\forall x(x \in dom(f) \rightarrow Qf(x))$$

Solution: Make *Bounded Quantification* a primitive operation and check that terms never undefined:

$(\forall x : P)Q(f(x))$ is a formula of a (strongly) typed logic if:

i) $P \subseteq dom(f)$ and

ii) $\{f(x)|x \in P\} \subseteq dom(Q)$
(Recall $Q : dom(Q) \rightarrow \{T, F\}$)

If (i) and (ii) hold then $(\forall x : P)Qf(x)$ is true in an interpretation structure iff for every $x \in P$, $f(x) \in Q$.

Traditional Logic Approach (Bounded Quantification):

Used by PVS and many formal mathematical logics.

Main idea: Universe divided into different "types". All functions have their domain restricted to the elements on which they are defined making all functions total.

E.g. In PVS prelude file

```
nonzero_real: NONEMPTY_TYPE = {r: real | r /= 0}
nzreal: NONEMPTY_TYPE = nonzero_real

+, -, *: [real, real -> real]
/: [real, nzreal -> real]
```

$$/ : \mathbb{R} \times \{r \in \mathbb{R} | r \neq 0\} \rightarrow \mathbb{R}$$

All function and predicate arguments are type checked to insure that no terms are undefined. Before reasoning about x/y, must prove $y \neq 0$.

Ex. 3 revisited

```
  sqrt: [nonneg_real -> nonneg_real]

  P1: PROPOSITION FORALL (x,y:real):
IF x>=0 THEN y=sqrt(x) ELSE y=sqrt(-x) ENDIF

  P2: PROPOSITION FORALL (x,y:real):
IF x>=0 THEN y=sqrt(x) ELSE y=sqrt(-x) ENDIF
      IFF (y=sqrt(abs(x)))
```

From PVS prelude file:

```
nonneg_real: NONEMPTY_TYPE = {x: real| x >= 0}
                CONTAINING 0

  m, n: VAR real
  abs(m): {n: nonneg_real | n >= m}
    = IF m < 0 THEN -m ELSE m ENDIF
```

Eliminating Undefined Terms by Type-checking

PVS forces you to prove that all terms are defined before you can conclude your proof is correct.

E.g. Taking $\sqrt{-x}$ in PROPOSITIONS P1 and P2 results in following proof obligation or "Type correctness condition":

```
% Subtype TCC generated (at line 13, column 53)
  % for -x
  % unchecked
P1_TCC1: OBLIGATION
(FORALL (x: real): NOT x >= 0 IMPLIES -x >= 0);
```


Another Comparison of Styles

Ex. 4a: "The value of x is found in array f"

$$\exists i(f(i) = x)$$

When undefined terms are allowed, the size of array, whether the index starts from 0 or 1 (or -39) does not matter. This will be true only if there is a matching value in the array.

In typed logic:

Define domain and range types and declare type of array

```
  index:TYPE
  T: NONEMPTY_TYPE
  f: [index->T]
  x: VAR T
  P3:PROPOSITION (EXISTS (i:index):f(i)=x)
```


Ex. 4b: "The value of x is found in the N element array f or all values in f are not equal to x"

$$\exists i(f(i) = x) \vee \forall i((1 \leq i \leq N) \rightarrow f(i) \neq x)$$

The above formula is used when undefined terms are allow. The predicate $(1 \leq i \leq N)$ is a necessary *guard condition*. Why?

In typed logic:

Define domain and range types and declare type of array before stating theorem.

```
  N:posnat
  index:TYPE={i:int| 1<=i & i<=N} CONTAINING 1
  T: NONEMPTY_TYPE
  f: [index->T]
  x: VAR T
  P4:PROPOSITION (EXISTS (i:index):f(i)=x) OR
(FORALL (i:index):NOT(f(i)=x))
```


Summary

Traditional Analysis Approach

Allows undefined terms & makes any **atomic predicate** applied to an undefined term False (i.e. $a = 1/0$ is False).

Advantages:

- Directly supports partial functions
- Concise
- Supports abstract, implementation independent specifications.

Disadvantages:

- Requires guard terms for universal quantifications
- Treatment of undefined terms leads to non-standard relationship among basic math operators e.g. $\neg(x < \sqrt{x})$ is not logically equivalent to $x \geq \sqrt{x}$ (Why?)

Summary

Traditional Logic Approach

Makes bounded quantification a primitive operation and then uses types to eliminate undefined terms, making all functions total.

Advantages:

- No guard terms for universal quantifications
- Normal relationship between standard math operators
- Typechecking provides tool for detecting errors

Disadvantages:

- Not as concise
- No direct support for partial functions - requires definition of domain to make function total
- Specification closer to implementation