Generalizations of Rice's Theorem, Applicable to Executable and Non-Executable Formalisms

Cornelis Huizing, Ruurd Kuiper and Tom Verhoeff

Eindhoven University of Technology, Dept. of Math. & Comp. Sc.,
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands
{c.huizing,r.kuiper,t.verhoeff}@tue.nl

Abstract

We formulate and prove two Rice-like theorems that characterize limitations on *nameability* of properties within a given *naming scheme* for partial functions. Such a naming scheme can, but need not be, an executable formalism. A programming language is an example of an executable naming scheme, where the program text names the partial function it implements. Halting is an example of a property that is not nameable in that naming scheme.

The proofs reveal requirements on the naming scheme to make the characterization work. Universal programming languages satisfy these requirements, but also other formalisms can satisfy them. We present some non-universal programming languages and a non-executable specification language satisfying these requirements. Our theorems have Turing's well-known Halting Theorem and Rice's Theorem as special cases, by applying them to a universal programming language or Turing Machines as naming scheme. Thus, our proofs separate the nature of the naming scheme (which can, but need not, coincide with computability) from the diagonal argument. This sheds further light on how far reaching and simple the 'diagonal' argument is in itself.

1 Introduction

Turing's Halting Theorem concerns the question whether a program, or another expression in an executable formalism such as a Turing Machine, halts. The Halting Theorem states that there exists no program that decides this question. Many students learn this theorem and are often left with an uneasy feeling. Since neither halting nor executability seems to be used extensively in the proof, they wonder what exactly the role of these notions is. A partial answer is given when they encounter Rice's Theorem, which states that any non-trivial property is undecidable. Partial, because the theorem is still about executable formalisms and because the typical proof invokes the undecidability of halting. The latter suggests that halting is still the "deciding" property: only for systems where halting is a semantic property, Rice's Theorem applies. In this paper, however, it turns out that neither executability nor halting is essential.

We formulate two generalized versions of Rice's Theorem, in which executability and halting are not essential. Furthermore, halting is not used in the proof at all.

One can view Rice's Theorem as an incompleteness result: it is not possible to decide any non-trivial property of computable functions with a computable function. Informally stated, our versions apply to any formalism that expresses a subdomain of (partial) functions. Given such a function formalism (satisfying a relatively weak notion of closedness) and any non-trivial property of these functions, this property cannot be expressed in the formalism. Rice's Theorem is a special case: take as subdomain the computable functions and as formalism a universal programming language or the formalism of Turing Machines and halting as property. Then the theorem states that halting cannot be expressed in the formalism, i.e., there is no computable function that decides halting. The theorem can be applied to other formalisms as well. For example, we introduce a relatively weak formalism that is a subdomain of imperative programs for which halting is decidable. It is, however, not expressible by a program in this weak formalism. At the other side of the spectrum, there are formalisms in which notions such as halting of (computable) programs can be expressed. In [6], Hehner suggests to consider halting in such a formalism (a specification language) in order to raise doubt about the halting theorem, since halting is expressible in such a formalism and nevertheless the proof of the halting theorem seems to go through. Our generalized theorem applies to these formalisms as well. Although halting for *programs* may be expressible, any non-trivial property of functions in *this*, larger, domain cannot be expressed by a function in that domain.

We speak here about formalisms and languages, but our results are, in fact, about a more general notion of *nameability*, which can mean any system to designate the functions in the domain. A (programming, specification) language to express functions is a common way to do this, but other systems can be used as well and they are not restricted to countable domains.

In Section 2, we formulate two generalizations of Rice's Theorem, one for properties about the value of functions with a certain input (the so-called pointwise theorem) and one for properties of functions as a whole (the so-called function-wise theorem). In Section 3, we apply both theorems to executable formalisms; in the proof of the function-wise theorem, a more sophisticated notion of closedness is needed and we show that executable formalisms such as universal imperative programming languages (in particular a subset of JavaScript) satisfy this closedness property. In Section 4, we apply the theorems to specification formalisms that are more powerful than executable formalisms. In Section 5, we elaborate on Hehner's position in relation to the Halting Problem and in Section 6, we draw our conclusions and propose further research.

2 General Rice Theorems

We present two generalized versions of Rice's Theorem. Both work with the same semantic domain of partial functions and are concerned with properties of such functions. The difference is in the nature of the properties. In §2.1, the properties involve the argument of the function, and are called *pointwise* properties. In §2.2, the properties do not involve the function argument, but concern the function as a whole, and are called *function-wise* properties. The proofs are structurally similar, but differ in how the diagonal argument is set up.

2.1 Pointwise

In this section, we consider the generalization of the halting function to recognizers for arbitrary *pointwise* properties of functions (pointwise Rice). Furthermore, we also generalize the notion of *computable* (expressible or definable in some computational formalism) to that of being *nameable*, in some, not necessarily computational, formalism, that we call a *naming scheme.*

First, we define the kind of functions we will be dealing with as semantics for the naming formalism.

Definition 2.1. *Let $PFUN = STRING \to STRING$ be the set of* ***partial functions*** *from STRING to STRING, where STRING is the set of finite sequences over some alphabet.*

Notes: The alphabet is not necessarily finite, or even countable. The set *STRING* could even be replaced by any *infinite* set (with a finite set, the argument breaks down). We could have used *total* functions, by extending the range with a special element to denote *undefined.*

However, when instantiating this general setting to that of computability and halting, we will use undefined for a non-terminating computation. Thus, undefined will receive special treatment.

Convention: String variables are in lower case Roman, and partial function variables in lower case Greek. Upper case Roman is used for meta entities.

Definition 2.2. *A **naming scheme** for (some elements of) PFUN is a (w.l.o.g. total) function $\mathcal{S}$ (for semantics) from STRING to PFUN, where $\mathcal{S}(n)$ denotes the partial function associated with name n. We call partial function ϕ **nameable** in $\mathcal{S}$ when there exists a name $n \in STRING$ with $\mathcal{S}(n) = \phi$.*

To be nameable does not necessarily mean that the function value can be effectively calculated for given arguments. The name only pinpoints the function.

Note: Because *PFUN* has a cardinality strictly larger than *STRING*, there exist partial functions that are not nameable in $\mathcal{S}$. This existence proof does not shed any light on what such unnameable partial functions can be. Our generalized Rice Theorems will explicitly reveal a class of unnameable partial functions, viz. those that recognize non-trivial properties, *provided* that the naming scheme satisfies a certain closedness property.

Definition 2.3. *A **pointwise property** P is a subset of STRING. We say that 'partial function ϕ has pointwise property P in point x' when $\phi(x) \in P$.*

The qualification 'pointwise' can be omitted, when clear from the context. Note that $\phi(x) \in P$ does not hold when $\phi(x)$ is undefined. Thus, the property $P = STRING$ requires exactly that ϕ is defined in the point at hand.

Definition 2.4. *Partial function $\phi \in PFUN$ **classifies** name n w.r.t. pointwise property P when for all points $x \in STRING$ we have*

$$\phi(n \,\&\, x) = \texttt{YES} \quad \text{if and only if} \quad \mathcal{S}(n)(x) \in P \tag{1}$$

where $n \,\&\, x$ denotes a reversible mechanism to encode a pair of strings as a single string, and $\texttt{YES} \in STRING$ is a specific string constant.

*We call ϕ a **recognizer** of property P when ϕ is* total *and ϕ classifies all names n w.r.t. P.*

*Property P is said to be **nameable** in $\mathcal{S}$ when it has a nameable recognizer.*

Definition 2.5. *Pointwise property P is **non-trivial** w.r.t. $\mathcal{S}$ when there exist names p and q, and strings s_p and s_q, such that $\mathcal{S}(p)(s_p) \in P$ and $\mathcal{S}(q)(s_q) \notin P$.*

*Otherwise, P is called **trivial** w.r.t. $\mathcal{S}$, that is, when*

$$\begin{aligned} &\text{either} \quad \mathcal{S}(n)(x) \in P, \text{ for all names } n \text{ and strings } x &(2)\\ &\text{or} \quad \mathcal{S}(n)(x) \notin P, \text{ for all names } n \text{ and strings } x &(3)\end{aligned}$$

We aim to prove the pointwise Rice Theorem:

Pointwise property P is nameable in $\mathcal{S}$ if and only if P is trivial w.r.t. $\mathcal{S}$.

Along the way, we discover what this requires of the naming scheme $\mathcal{S}$.

Proof Let P be any pointwise property.

Part 1 Right-hand side implies left-hand side.

Assume P is trivial w.r.t. $\mathcal{S}$. Goal is to prove that P is nameable in S.

To guarantee that P is nameable in S it suffices to exhibit two names, say t and f, such that $\mathcal{S}(t)$ and $\mathcal{S}(f)$ are total, and

$$\mathcal{S}(t)(x) \quad = \quad \texttt{YES}, \text{ for all } x \in \mathit{STRING} \tag{4}$$

$$\mathcal{S}(f)(x) \quad \neq \quad \texttt{YES}, \text{ for all } x \in \mathit{STRING} \tag{5}$$

Let us assume that $\mathcal{S}$ is such that these names t and f exist. Note that $\mathcal{S}(t)$ is a constant function, and that $\mathcal{S}(f)$ can also be a constant function, e.g., mapping to $\texttt{NO} \in \mathit{STRING}$ with $\texttt{NO} \neq \texttt{YES}$.

If (2) holds, then $\mathcal{S}(t)$ recognizes P. If (3) holds, then $\mathcal{S}(f)$ recognizes P. Either way, P is nameable in $\mathcal{S}$.

This reveals a first requirement for our naming scheme $\mathcal{S}$ to complete the proof, viz. the existence of names t and f above. It is a "weak" requirement, which we revisit later.

Part 2 Negation of right-hand side implies negation of left-hand side. Assume that P is non-trivial, i.e., let p and q be names, and s_p and s_q be strings such that

$$\mathcal{S}(p)(s_p) \quad \in \quad P \tag{6}$$

$$\mathcal{S}(q)(s_q) \quad \notin \quad P \tag{7}$$

Goal is to prove that P is not nameable in $\mathcal{S}$. For any name r, we shall show that it does not name a recognizer of P. For that purpose, it suffices to exhibit a name c such that $\mathcal{S}(r)$ fails to classify c in point c, i.e. such that

$$S(r)(c \,\&\, c) = \texttt{YES} \quad \text{is not equivalent to} \quad \mathcal{S}(c)(c) \in P \tag{8}$$

Assume that $\mathcal{S}(r)$ is total (otherwise, it certainly does not recognize P). Consider the partial function ϕ with[1]

$$\phi(x) \quad = \quad \mathcal{S}(q)(s_q) \textbf{ if } \mathcal{S}(r)(x \,\&\, x) = \texttt{YES} \textbf{ else } \mathcal{S}(p)(s_p) \tag{9}$$

The naming scheme $\mathcal{S}$ needs to be such that the partial function ϕ is nameable. Note that ϕ depends on r.

This reveals a second requirement on the naming scheme. It is a stronger requirement than the first one, but still no show stopper (also revisited later). Common computational formalisms satisfy these requirements (also see Sections 3 and 4).

Let c be a name for the partial function ϕ, that is

$$\mathcal{S}(c) \quad = \quad \phi \tag{10}$$

[1]We use a Hoare-inspired notation for conditional expressions, where the condition is in the middle, between **if** and **else** .

We now show (8), i.e. that $\mathcal{S}(r)$ does not classify c in point c, by calculating:

$$\begin{array}{ll}
& \mathcal{S}(c)(c) \in P \\
= & \quad \{ \text{ definition of } c \text{, i.e. (10) } \} \\
& \phi(c) \in P \\
= & \quad \{ \text{ definition of } \phi \text{, i.e. (9) } \} \\
& \Big(\mathcal{S}(q)(s_q) \ \mathbf{if}\ \mathcal{S}(r)(c \,\&\, c) = \texttt{YES}\ \mathbf{else}\ \mathcal{S}(p)(s_p)\Big) \in P \\
= & \quad \{ \text{ distribution, using that } \mathcal{S}(r) \text{ is total } \} \\
& \mathcal{S}(q)(s_q) \in P \ \mathbf{if}\ \mathcal{S}(r)(c \,\&\, c) = \texttt{YES}\ \mathbf{else}\ \mathcal{S}(p)(s_p) \in P \\
= & \quad \{ \text{ definition of } p \text{ and } q \text{, i.e. (6) and (7) } \} \\
& \mathit{false}\ \mathbf{if}\ \mathcal{S}(r)(c \,\&\, c) = \texttt{YES}\ \mathbf{else}\ \mathit{true} \\
= & \quad \{ \text{ logic } \} \\
& \mathcal{S}(r)(c \,\&\, c) \neq \texttt{YES}
\end{array}$$

This establishes (8), from which we conclude that P is not nameable in $\mathcal{S}$.
(End of Proof)

Thus, we have proven

Theorem 2.1. *(Pointwise Rice) Pointwise property P is nameable in $\mathcal{S}$ if and only if P is trivial w.r.t. $\mathcal{S}$, provided that $\mathcal{S}$ is closed in the sense that*

1. *there exist names t and f for total functions satisfying (4) and (5);*
2. *for all names p, q, r, s_p, and s_q, the partial function ϕ in (9) is nameable.*

Note that the second closedness requirement involves:

- the pairing of strings,
- the "application" of (the partial function named by) a name (here: p, q, and r), and
- a conditional clause.

2.2 Function-wise

We now want to generalize the result of the preceding section further, by considering properties of functions as a whole, and not just pointwise. It is interesting that this can still be done, but that the proof is different, imposing different requirements on the naming scheme. In particular, we now need an ability for names to refer to themselves, rather than a string-pairing operation. String-pairing is still needed to specialize the results in this section to the pointwise case of the preceding section.

Strings, partial functions, and naming schemes are as in the preceding section. We now consider properties in full generality. That way, also properties like whether ϕ satisfies $\phi(\texttt{"a"}) = \phi(\texttt{"b"})$ are possible.

Definition 2.6. *A **function-wise property** P of partial functions is a subset of PFUN.
Partial function ϕ has function-wise property P when $\phi \in P$.*

The qualification 'function-wise' can be omitted, when clear from the context.

Definition 2.7. *Partial function $\phi \in PFUN$* ***classifies*** *name n w.r.t. function-wise property P when*

$$\phi(n) = \texttt{YES} \quad \textit{if and only if} \quad \mathcal{S}(n) \in P \tag{11}$$

The notions of a partial function being a recognizer of a property, and a property being nameable are defined as before.

Definition 2.8. *Function-wise property P is* ***non-trivial*** *w.r.t. $\mathcal{S}$ when there exist names p and q, such that $\mathcal{S}(p) \in P$ and $\mathcal{S}(q) \notin P$.*

Otherwise, P is called ***trivial*** *w.r.t. $\mathcal{S}$, that is, when*

$$\begin{aligned} &\textit{either} \quad \mathcal{S}(n) \in P, \ \textit{for all names } n && (12)\\ &\textit{or} \quad \mathcal{S}(n) \notin P, \ \textit{for all names } n && (13)\end{aligned}$$

We aim to prove the pointwise Rice Theorem:

Function-wise property P is nameable in $\mathcal{S}$ if and only if P is trivial w.r.t. $\mathcal{S}$.

Along the way, we discover what this requires of the naming scheme $\mathcal{S}$.

Proof Let P be any function-wise property.

Part 1 Right-hand side implies left-hand side.

Assume P is trivial w.r.t. $\mathcal{S}$. The goal is to prove that P is nameable in S.

To guarantee that P is nameable in S it suffices to exhibit two names, say t and f, such that $\mathcal{S}(t)$ and $\mathcal{S}(f)$ are total, and

$$\begin{aligned} \mathcal{S}(t)(x) &= \texttt{YES}, \ \text{for all } x \in \mathit{STRING} && (14)\\ \mathcal{S}(f)(x) &\neq \texttt{YES}, \ \text{for all } x \in \mathit{STRING} && (15)\end{aligned}$$

Let us assume that $\mathcal{S}$ is such that these names t and f exist. If (12) holds, then $\mathcal{S}(t)$ recognizes P. If (13) holds, then $\mathcal{S}(f)$ recognizes P. Either way, P is nameable in $\mathcal{S}$.

This reveals a first requirement for our naming scheme $\mathcal{S}$ to complete the proof, viz. the existence of names t and f above. It is the same as in the pointwise case.

Part 2 Negation of right-hand side implies negation of left-hand side. Assume that P is non-trivial, i.e., let p and q be names such that

$$\begin{aligned} \mathcal{S}(p) &\in P && (16)\\ \mathcal{S}(q) &\notin P && (17)\end{aligned}$$

Goal is to prove that P is not nameable in $\mathcal{S}$. For any name r, we shall show that it does not name P, i.e., that $\mathcal{S}(r)$ does not recognize P. For that purpose, it suffices to exhibit a name c such that $\mathcal{S}(r)$ fails to classify c, i.e. such that

$$S(r)(c) = \texttt{YES} \quad \text{is not equivalent to} \quad \mathcal{S}(c) \in P \tag{18}$$

Assume that $\mathcal{S}(r)$ is total (otherwise, it certainly does not recognize P). Consider the partial function ϕ with

$$\phi(x) \quad = \quad \mathcal{S}(q)(x) \ \textbf{if} \ \mathcal{S}(r)(c) = \texttt{YES} \ \textbf{else} \ \mathcal{S}(p)(x) \tag{19}$$

where $\mathcal{S}(c) = \phi$. Note that the condition depends on r but not on x. Since $\mathcal{S}(r)$ is total, ϕ either equals $\mathcal{S}(q)$ or $\mathcal{S}(p)$:

$$\phi \quad = \quad \mathcal{S}(q) \ \textbf{if} \ \mathcal{S}(r)(c) = \texttt{YES} \ \textbf{else} \ \mathcal{S}(p) \tag{20}$$

The naming scheme $\mathcal{S}$ needs to be such that partial function ϕ is nameable.

This reveals a second requirement on the naming scheme. It is a stronger requirement than the first one, but still no show stopper.

Let c be a name for the partial function ϕ, that is

$$\mathcal{S}(c) \quad = \quad \phi \tag{21}$$

We now show (18), i.e. that $\mathcal{S}(r)$ does not classify c, by calculating:

$$\begin{array}{cl}
 & \mathcal{S}(c) \in P \\
= & \quad \{ \text{ definition of } c\text{, i.e. (21) } \} \\
 & \phi \in P \\
= & \quad \{ \text{ property of } \phi\text{, i.e. (20) } \} \\
 & (\mathcal{S}(q) \textbf{ if } \mathcal{S}(r)(c) = \texttt{YES} \textbf{ else } \mathcal{S}(p)) \in P \\
= & \quad \{ \text{ distribution, using that } \mathcal{S}(r) \text{ is total } \} \\
 & \mathcal{S}(q) \in P \textbf{ if } \mathcal{S}(r)(c) = \texttt{YES} \textbf{ else } \mathcal{S}(p) \in P \\
= & \quad \{ \text{ definition of } p \text{ and } q\text{, i.e. (16) and (17) } \} \\
 & \textit{false} \textbf{ if } \mathcal{S}(r)(c) = \texttt{YES} \textbf{ else } \textit{true} \\
= & \quad \{ \text{ logic } \} \\
 & \mathcal{S}(r)(c) \neq \texttt{YES}
\end{array}$$

This establishes (18), from which we conclude that P is not nameable in $\mathcal{S}$.
(End of Proof)

Thus, we have proven

Theorem 2.2. *(Function-wise Rice) Function-wise property P is nameable in $\mathcal{S}$ if and only if P is trivial w.r.t. $\mathcal{S}$, provided that $\mathcal{S}$ is closed in the sense that*

1. *there exist names t and f for total functions satisfying (14) and (15);*
2. *for all names p, q, and r, there exists a name c for the partial function ϕ in (19).*

Note that the latter requirement involves:

- the "application" of (the partial function named by) a name (here: p, q, and r),
- a conditional clause, and
- the ability of a name to refer to itself (ϕ "contains" c, where $\mathcal{S}(c) = \phi$).

This differs from the requirements revealed in the proof of Theorem 2.1. Universal computational formalisms satisfy these requirements (also see Sections 3 and 4).

Note that the proofs of Theorems 2.1 and 2.2 are almost isomorphic. Still, there is a significant difference. Compare the two definitions of ϕ. In (9), the condition depends on the argument, but in (19) it does not. However, in (19) we need to introduce a name (viz. c) for ϕ into its own definition, which is not the case in (9).

3 Executable Formalisms

The two Rice Theorems (pointwise and function-wise) of the preceding section can be applied to various concrete naming schemes. In this section, we consider three executable naming schemes, of increasing expressive power. We will not work out all technical details, since there are no new results. These applications serve to show that our general Rice Theorems can indeed be instantiated to obtain Turing's Halting Theorem and Rice's Theorem. Furthermore, they illustrate why it is useful to have a general version, because it now can also be applied to non-universal formalisms.

A naming scheme could just consist of a structureless mapping from names to partial function. The naming schemes in this section, however, are based on a syntactic structure, with a compositional semantics. That way, it is relatively straightforward to show that they satisfy the closedness requirements of the theorems.

First, we describe the three naming schemes, which are basically imperative programming languages, and then we consider the consequences of the two Rice Theorems. The simplest language $\mathcal{L}_1$ is a loop-free language having the following features:

1. a countably unbounded supply of string-valued (global) variables, with the empty string as initial value; variables `input` and `output` play a special role;
2. some operators to form string expressions and boolean expressions, including string constants, string comparison `=`, catenation `~`, the pairing operator `&`, its two corresponding projections;
3. an assignment statement that updates a variable based on an expression: *variable* `=` *expression*`;`;
4. a selection statement: `if (` *expression* `)` *statement* `else` *statement*;
5. sequential composition of statements into blocks: { *statement* … *statement* }.

The partial function ϕ associated with (syntactically valid) name (program text) n is defined by $\phi(x) = y$ where y is the final value of variable `output` when program n is executed after setting variable `input` to x. This is actually a total function. Syntactically invalid programs map to the all-undefined function. For instance, the program `output = input ~ input` 'squares' the input string. There exists no $\mathcal{L}_1$ program to decide whether the input is a square.

The second language $\mathcal{L}_2$ extends $\mathcal{L}_1$ by adding a *divergence* statement `div` that does not terminate. Whenever execution hits a divergence, the value of the named partial function is undefined.

The third language $\mathcal{L}_3$ extends $\mathcal{L}_1$ by adding a *repetition* statement: `while (` *expression* `)` *statement*. Note that `div` can now be programmed as an infinite loop; hence, $\mathcal{L}_3$ extends $\mathcal{L}_1$ semantically. In fact, $\mathcal{L}_3$ is universal. Therefore, being nameable in $\mathcal{L}_3$ is equivalent to being computable.

3.1 Pointwise Rice

By construction, the three languages satisfy the closedness requirements for our pointwise Rice Theorem. Function application deserves some attention. Given a name n and a string s, to use the result of applying the function named n to the given string s, insert a transformed version of n:

- rename all variables in n to fresh variables,

- prepend an assignment to initialize the (renamed) input variable to the argument s, and
- take the result from the (renamed) output variable.

Observe that all programs (names) in $\mathcal{L}_1$ terminate and define a total function. Thus, halting is a trivial property w.r.t. $\mathcal{L}_1$, and it can be (trivially) recognized by the $\mathcal{L}_1$ program (name) `output = "YES"`. Halting is non-trivial w.r.t. $\mathcal{L}_2$ and $\mathcal{L}_3$, because there exist terminating and non-terminating programs. Hence, there exists no $\mathcal{L}_2$ program that recognizes halting of $\mathcal{L}_2$ programs, nor does there exist an $\mathcal{L}_3$ program that recognizes halting of $\mathcal{L}_3$ programs. The latter statement is Turing's Halting Theorem.

Note that there does exist an $\mathcal{L}_3$ program that recognizes halting of $\mathcal{L}_2$ programs, that is, halting in $\mathcal{L}_2$ is decidable. This can be done by transforming the $\mathcal{L}_2$ program into an $\mathcal{L}_1$ program by

- prepending it with the statement `v = "YES"`, where `v` is a fresh variable,
- replacing every divergence by `v = "NO"`, and
- appending the statement `output = v`,

and then simulating execution of the resulting $\mathcal{L}_1$ program, which always terminates.

3.2 Function-wise Rice

For function-wise Rice, we only consider $\mathcal{L}_3$. Since $\mathcal{L}_3$ is universal, we know by Kleene's Second Recursion Theorem [2] that the closedness requirement for the function-wise Rice Theorem is satisfied. This instantiation gives us Rice's Theorem for (un)decidability of (semantic) properties of computable functions.

For students, this may be a long call, and it might be helpful to see that closedness is not such a magical thing. The Generalized Challenge in [9] provides an interactive guided tour to discover how to construct programs in a (universal) subset of *JavaScript* that process their own listing. Very briefly stated, the main idea is to embed an encoding of the program as a kind of blueprint, and to process this blueprint to initialize a variable with the exact listing of the entire program. The program can then proceed to process this listing through the initialized variable. Note that in (19), the function named by c (i.e., ϕ) does not apply c to some argument, but applies a given named function to c.

4 Specification Formalisms

In this section, we consider a non-executable naming scheme, or what one might call a specification language. This naming scheme is more 'powerful' than the executable naming schemes of the preceding section, in the sense that strictly more partial functions have a name. However, it names partial functions without telling one how to compute them.

We first describe this specification language, again without providing all technical details. Then we consider the consequences of the pointwise Rice Theorem.

A specification (name) is given as a predicate on *STRING* $\times$ *STRING* (input, output). Let P be such a predicate, then its semantics is defined as the partial function ϕ with

$$\phi(x) \quad = \quad \begin{cases} y \quad \text{if there exists a unique } y \text{ with } P(x,y) \\ \text{undefined, otherwise (there exists no } y\text{, or multiple } y \text{ exist)} \end{cases} \tag{22}$$

Predicates can involve the propositional operators (negations, conjunction, disjunction, implication), some string operators (equality, catenation, ...), and universal and existential quantification over string variables. Instead of saying that a partial function ϕ is nameable in this specification language, we also say that ϕ is specifiable.

Some examples of specifications:

- squaring: $P(x, y) \iff y = x \sim x$
- square root: $P(x, y) \iff x = y \sim y$ (not nameable in $\mathcal{L}_2$)

Here are some examples of combining specifications to obtain new specifications. Given two specifications P and Q, specifying partial function ϕ and ψ, specify the function $\rho(x) = \phi(x) \sim \psi(x)$ (catenation of ϕ and ψ):

$$R(x, y) \quad \iff \quad \exists y_1, y_2\, P(x, y_1) \wedge Q(x, y_2) \wedge y = y_1 \sim y_2$$

This is strict, that is, if either function is undefined, then the catenation also is.

Given two specifications P and Q, specifying partial functions ϕ and ψ, specify the function $\rho(x) = \phi(\psi(x))$ (composition):

$$R(x, y) \quad \Leftrightarrow \quad \exists y_1\, P(x, y_1) \wedge Q(y_1, y)$$

The halting function for the executable formalisms of the preceding section can be specified (in the specification language). That is, the recognizer for the pointwise property that a partial function ϕ, named in, say, $\mathcal{L}_3$, halts (i.e., $\phi(x)$ is defined) is specifiable.

The idea is that we can define, in predicates, whether string n is a syntactically valid program (in the executable formalism), and whether execution of n on input i terminates. The latter involves defining the notion of program state (encoded in a string, including such information as the program, to what (syntactic) point execution has progressed, and the values of all variables), what the effect is on the state of executing each statement type, the notion of an execution (encoded in a string) as a sequence of states, and the notion of halting (when execution runs off the end, or possibly we can introduce an explicit `stop` statement).

4.1 Pointwise Rice

Our specification language is closed for pointwise Rice. The conditional combination (cf. (9)) of two specifications P and Q based on a third specification R as guard can be achieved as follows:

$$(R(x\&x, \texttt{YES}) \Rightarrow Q(c_1, y)) \wedge (\exists s\, s \neq \texttt{YES} \wedge R(x\&x, s) \Rightarrow P(c_2, y)) \tag{23}$$

Consequently, recognizers for properties of specifications are specifiable (nameable in the specification language itself) if and only if the property is trivial.

4.2 Function-wise Rice

Although not crucial for our results, it would be nice if our specification language would be closed for function-wise Rice as well. Since the specification language is an extension of programming language $\mathcal{L}_3$, it is to be expected that the requirements of the theorem are satisfied there as well. We have not proved this yet.

5 Hehner's problems with the Halting Problem

In [4, 5, 6], Hehner considers the incomputability argument in the proof of the Halting Theorem. He claims that the inconsistency that is arrived at by constructing the self-referential program (corresponding to the partial functions ϕ in (9) and (19)) should not be interpreted as refuting the computability of the halting function, but as pointing to a problem with the definition of that function. Very briefly, his argument is as follows – for the detailed explanation we recommend Hehner's [6].

First, consider the classic proof of the Halting Theorem in the setting of a programming language with functions that as a parameter can take the name of a program text defined in a library, and at the call rather than instantiate the body that is provided in the library, analyses its code. Assume the existence of $\mathcal{H}$, the implementation of the halting function which returns true/false when applied to the text of a terminating/nonterminating program. Then define the text for the refuting program as:

$$\mathcal{D} \quad = \quad \text{“if } \mathcal{H}(\mathcal{D}) \text{ then while } \top \text{ do } x = 1 \text{ else } x = 0\text{”} \tag{24}$$

The inconsistency in the value of $\mathcal{H}(\mathcal{D})$ is interpreted as refuting the assumption that $\mathcal{H}$ exists.

Second, there is a very closely parallel argument for specifications. Hehner uses the setting of Unifying Theories of Programming [7, 3], where the set of programs is a subset of the set of specifications. Specifically, specifications can specify termination or nontermination. Furthermore, a program that (non)terminates also specifies, at least, (non)termination. Now assume the existence of h, the (not necessarily computable, but expressible in the specification formalism) function that returns true/false when applied to the text of a specification that specifies termination/nontermination. The text for the refuting specification $\mathcal{D}$ is, again, defined as:

$$\mathcal{D} \quad = \quad \text{“if } h(\mathcal{D}) \text{ then while } \top \text{ do } x = 1 \text{ else } x = 0\text{”} \tag{25}$$

The resulting inconsistency now cannot be blamed on the computability assumption, as such an assumption does not apply to specifications. Hehner proposes that this should be interpreted as that the definition of the halting function per se is the source of the inconsistency.

Because of the close parallels between the arguments, he argues that in both cases there is an inconsistency between the definition of the halting function and the refuting expression – and that in both cases the culprit is the halting function.

In [8], we argue that the conclusion that in the specification case h is not well-defined is not justified.

With the insights of the present paper, we know that the relevant question is not whether it is well-defined, but whether it is in the specification domain, that is, whether h has a name. Generalized Rice immediately shows that it is not and that this does not say anything about its well-definedness.

And the parallel between program domain and specification domain is that in the program case $\mathcal{H}$ is not a program and in the specification case h is not a specification.

6 Conclusions and Future Work

Inspired by Hehner's skeptical view on Turing's Halting Theorem [4, 5], we set out to clarify the situation [8], by carefully disentangling his chain of arguments. Hehner uses the UTP (Unified Theory of Programming) formalism to denote partial functions. UTP is a hybrid formalism that has both the features of an imperative programming language and a non-executable specification language. In this paper, we consider general mechanisms for denoting partial functions, that we have dubbed *naming schemes*. Both executable (programming) formalisms and non-executable (specification) formalisms can be viewed as naming schemes.

When such a naming scheme satisfies certain closedness requirements, a Rice-like theorem holds: a property of named partial functions can be recognized by a named partial function if and only if that property is trivial. We have distinguished two types of properties, viz. *pointwise*, which concern a given named partial function for a given argument, and *function-wise*, which concerns just a given named partial function.

We have illustrated this with three executable naming schemes, of increasing expressive power, all of which satisfy the conditions for the pointwise Rice Theorem. For the simplest naming scheme, halting is a trivial property, but for the other two it is not, and, hence, halting is not recognizable by a partial function named in that same naming scheme. This corresponds to Turing's Halting Theorem. The universal language also satisfies the conditions for the function-wise Rice Theorem, which then corresponds to Rice's Theorem (for computable functions).

We have also applied the pointwise Rice Theorem to a non-executable specification language. That language is more powerful still: it can specify (name) all computable functions, but also others, such as the halting function for the universal executable naming scheme. This also clarified the problems that Hehner raised with the Halting Theorem. Although our specification language, like UTP, is an extension of an executable formalism, it is not as rich as UTP. UTP has more non-determinism, in the sense that several partial functions can satisfy a single specification. It would be interesting to consider hybrid naming schemes, like UTP, that combine programming features and specification features in one language.

Thus, we have shown that Rice's Theorem can be generalized to languages more powerful than Turing complete languages, as well as languages that are less powerful. The proofs of these generalizations do not depend on halting anymore, as was to be expected, since halting is not applicable to non-executable languages, such as our specification language.

For future research, we would like to explore the relationship between the pointwise and the function-wise Rice Theorems: Under what conditions can one be proved from the other? For us it is also still an open question whether the simpler two executable languages and the specification language satisfy the closedness requirements for the function-wise Rice Theorem. A different interesting angle would be to approach the proofs of the Rice-like theorems not through counter example construction, but through complexity arguments. Complexity aspects for Rice's Theorem recently received attention, notably through [1].

References

[1] Andrea Asperti. The intensional content of Rice's theorem. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 113–119, New York, NY, USA, 2008. ACM.

[2] N.J. Cutland. *Computability: An introduction to recursive function theory.* Cambridge University Press, 1980.

[3] E.C.R. Hehner. *A Practical Theory of Programming.* Springer, 2004. Free download at `www.cs.utoronto.ca/~hehner/aPToP`.

[4] E.C.R. Hehner. Retrospective and prospective for unifying theories of programming. In Steve Dunne and Bill Stoddart, editors, *Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 2006.

[5] E.C.R. Hehner. Problems with the halting problem. In *COMPUTING2011 Symposium on 75 years of Turing Machine and Lambda-Calculus*. Karlsruhe Germany, invited, october 20–21 2011.

[6] E.C.R. Hehner. Problems with the halting problem. `http://www.cs.toronto.edu/~hehner/PHP.pdf`. Revised version, 12 March 2012.

[7] C.A.R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[8] Cornelis Huizing, Ruurd Kuiper, and Tom Verhoeff. Halting still standing – programs versus specifications. In Shengchao Qin, editor, *Unifying Theories of Programming*, volume 6445 of *Lecture Notes in Computer Science*, pages 226–233. Springer Berlin / Heidelberg, 2010.

[9] T. Verhoeff. Tom's JavaScript machine. `http://www.win.tue.nl/~wstomv/edu/javascript` (accessed 15 April 2012).