THE DENOTATIONAL BASIS FOR SOFTWARE EXECUTION TRACING

Leighton Brough, Paul Bailes
School of Information Technology and Electrical Engineering
The University of Queensland QLD 4072 Australia
brough@itee.uq.edu.au, paul@itee.uq.edu.au

ABSTRACT
Software execution tracing is a popular and effective technique, used to support a range of software engineering activities. Nevertheless execution tracing has rarely been the focus of study in its own right, although many case studies and applications have been described. Existing formal notions of trace in computer science are shown to be based implicitly, and then explicitly, on an operational semantic basis, and the limitations of the resulting trace monoid structure for many practical tracing tasks is highlighted. Justified by the category theoretic notion of duality, this paper introduces the denotational basis for software execution tracing, to address these practical limitations. An example of a language with denotational trace added is presented: a simple program in that language, its denotational traces, and their subsequent analysis are shown to support complex tasks such as specification recovery and reasoning about correctness, as well as simple, informal proofs of space and time complexity – techniques and results of immediate usefulness to the practicing software engineer. While both the theory of denotational trace and its consequences are in an early stage of development, initial results suggest uniquely useful, practical results can be derived from denotational traces.

KEY WORDS
Execution tracing, denotational semantics, trace analysis

1. INTRODUCTION
Software execution tracing is a practical technique whereby an automatic record is made of the execution of a program, for analysis by a programmer. By providing a history of the execution of a program, a trace aids the programmer in comprehending the relationship between the program source code and the corresponding execution behaviour. Consequently almost all existing execution tracing systems are source-oriented (as is generally agreed should be the case, e.g., [19]), in as much as the details recorded in a trace directly relate to source-level constructs in the program. In this, tracing differs from logging, which is usually intended to provide user-level information rather than insights into the execution of the program source code itself. Execution tracing is clearly a very commonly used practical technique, which many if not most experienced software engineers will have used for one or many of the wide range of activities it supports including software design, development, tuning, debugging, testing and profiling [2]. Various ingenious execution tracing systems exist such as the popular DTrace system [1], and anecdotal reports and personal experience suggest that one-off, bespoke tracing systems are a part of many software engineering projects.

Existing, practical tracing systems - which are ubiquitous - are designed, implemented and used in an ad-hoc fashion. Existing theoretical approaches to tracing, being grounded in automata theory, do not seem to be of practical use for the most common, day-to-day needs of tracing system users, as evidenced by the lack of formal trace theory used in practical tracing system1. In the next section we explain how this is due to the limitations of the trace monoid structure.

This lack of thorough and effective, theoretical grounding results in practical problems at several levels:

• Designers of programming languages have no guidance as to how to include a tracing system as part of a mature tool-chain for their language - in particular what information traces can and should include and how they should be structured.

• Language implementers and tracing system designers have no guidance as to how to build sound, practical tracing implementations, or the ability to know (i.e., prove) that what they have built is correct; instead they are guided by experience and intuition.

• Users of ad-hoc, practical tracing systems have no guarantees (proofs) that the system they are using is correct or accurately reflects the semantics of the program.

• Users of practical tracing systems have at best ad-hoc (and possibly erroneous) techniques and tools for reasoning about the contents of traces with ad-hoc structure and organisation, again relying only on experience and intuition.

• Lacking a solid theoretical basis, it is difficult to implement generic, correct, useful tools for the specification or implementation of tracing systems, or to support reasoning about traces (as evidenced by the lack of such tools).

The denotational (vs. operational) theory of tracing introduced here offers a conceptually-integrated solution to all of these practical problems, based in formal mathematics, justified by the category-theoretic concept of duality:

\[\text{With some important exceptions, see section 2.2.}\]
• For the first time, an explanation can be offered for the success vs. failure of existing formal trace methods for specific applications such as concurrency, versus some more day-to-day uses for practical tracing. Denotational trace is introduced, specifically to address the practical failures of existing theory.

• The concept of denotational, full and faithful traces is introduced to provide a language designer or implementer with the full set of information available for tracing, including how it is structured and precisely how it relates to a (denotational semantic) specification of the language. The question as to what traces may contain, and how they are structured, is answered. This in turn provides users of these traces a clear and precise definition of how they relate to the semantics of a program of interest.

• A practical example of a simple language and the derived, denotational trace is provided, to illustrate how denotational trace can support complex, high-value software engineering tasks such as the recovery of specification from trace – something of considerable usefulness to an engineer engaged in the maintenance of poorly or incorrectly documented code.

• Given such a mathematical basis for practical tracing, the user of a denotational tracing system has access to proven, reliable tools such as proof by induction. Two examples are given of back-of-the-envelope proofs from traces using induction – for the space and time complexity of a program – techniques of great usefulness to anyone needing to analyse or debug the performance of real programs.

• Formal, mathematical techniques offer the potential for easy automation, thereby providing better tool support for trace users, examples are provided.

2. TRACING AND SEMANTICS

Any attempt to discuss tracing with both precision and clarity requires a less vague definition of what trace is than a “history of the execution of a program”. Immediate questions arise such as: “what exactly should be traced?” and “what is the structure of a trace?” etc. This leads directly to the requirement for a clear specification of the behaviour or meaning of a program, based on the semantics of the language in use.

2.1 Tracing in Theory

Formal methods in computer science have had a clear answer as to what a ‘trace’ constitutes for many years. One early use of the term ‘trace’ in the literature is that of Scott [21]: “the operational meaning will generally provide a trace of the history of the computation.” Here Scott uses the term “operational” to refer explicitly to the understanding of semantics as a transition system, as distinct from the “mathematical” semantics he was developing, which eventually became known as denotational semantics [22]. The notion of a transition system is deeply rooted in computer science and directly linked to computation theory; both the Turing machine and its computational equivalent the lambda calculus are transition systems, being examples of a machine and term rewriting system respectively, which move through a series of states (or a reduction sequence), and may or may not terminate (or reach normal form). Scott’s statement above observes that a record of the states (or reductions) of any such “operational” understanding of behaviour constitutes a “trace”. This notion of trace seems pervasive, e.g., Jacobs and Rutten ([10]) also gives an example of how this sort of trace arises automatically from a transition system.

In line with this thinking, it has become customary when using formal, mathematical approaches to model traces as monoids: abstract, algebraic structures which, aside from a few other basic properties, support a single, associative binary operation, i.e., a sequencing operation which allows abstract trace events to be ordered, and is inhabited by only one type. A concrete example of a monoid is the familiar list structure; in fact a list is a free monoid (i.e. a monoid unencumbered by any properties other than those essential to its definition). The trace monoid is simply a list of the transitions made by an operational semantic system, extended with the idea of partial commutativity, i.e., the possibility of reordering sub-traces, which has been used fruitfully in the study of concurrent systems and nondeterminism [3].

2.2 Tracing in Practice

Developers of practical tracing systems are almost always concerned with source-orientation and usually with function (or procedure) argument values on entry and return values on exit. This is the case for essentially all of the case studies of tracing systems in the academic literature – although an exhaustive review is well beyond the scope of this paper. This pragmatic focus on function application corresponds directly to the source-oriented needs of any programmer attempting to work with simple questions of correctness, such as why a function returns the wrong value. Function entry and exit points in relation to each other are of particular interest – the nested structure, which reflects the nested structure of the source as it unfolds at runtime. The trace monoid, being a list inhabited by a single type, provides too simple a structure to encode this nested structure, and thus much of the information of interest.

3. OPERATIONAL AND DENOTATIONAL SEMANTICS ARE DUALS

The operational semantic notion of a transition system has been most abstractly captured by the idea of final coalgebra semantics, as has been explicitly observed by Turi and Plotkin [24]. Intuitively, the idea here is that final coalgebra semantics captures the sense in which all transition systems are abstractly the same (or more formally, isomorphic).

Final coalgebra semantics has a mathematical dual, which is initial algebra semantics, and this turns out to be a natural structure for expressing denotational semantics, a point made explicit by Rutten and Turi [20]. Initial algebra semantics is already a familiar mathematical structure as
this describes things that are data-type-like, an observation made clearly by Gougen [5].

So in a deep sense, operational and denotational semantics are dual to each other. Jacobs and Rutten provide an overview of the relationship between these dual structures, and the associated tools which come with them (bisimilarity/coinduction vs equality/induction) [10].

Having established that the denotational semantic perspective is dual to that of operational semantics, and that operational semantics has an associated concept of trace (the trace monoid), we now ask what trace might correspond to denotational semantics?

Notably, coinduction/bisimilarity have been useful in exploring concurrency [20], and recent work has exploited the connections between the trace monoid, coalgebraic structures and coinduction [8]. Duality suggests here we should find that denotational traces are good for questions of correctness (via equality), will be compositional, and that induction should be a natural proof technique. Similarly we get some hints as to what denotational traces might not be so good for – precisely those domains where operationally-based trace has traditionally been the focus, e.g., concurrent systems. The reality is that the world of programming is inhabited by both species of structure, and they are equally useful for different applications.

The concepts and their dual discussed in this paper are summarised in the following table:

<table>
<thead>
<tr>
<th>familiar example:</th>
<th>data type</th>
<th>state machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>object:</td>
<td>initial algebra</td>
<td>final coalgebra</td>
</tr>
<tr>
<td>recursion strategy:</td>
<td>fold</td>
<td>unfold</td>
</tr>
<tr>
<td>morphism:</td>
<td>catamorphism</td>
<td>anamorphism</td>
</tr>
<tr>
<td>sameness:</td>
<td>equality</td>
<td>bisimilarity</td>
</tr>
<tr>
<td>proof technique:</td>
<td>induction</td>
<td>coinduction</td>
</tr>
<tr>
<td>semantics:</td>
<td>denotational</td>
<td>operational</td>
</tr>
</tbody>
</table>

Table 1. Duality Cheat-Sheet

4. DENOTATIONAL TRACE

In the very early stages, denotational semantics did not have a suitable, mathematical structure with which to model the semantic domain, and Strachey proceeded on the basis that something suitable would be found in due course [23]. Indeed shortly thereafter Scott developed a class of suitable structures, now known as Scott domains [21]. One deficiency of Scott domains as a denotational model is that the resulting semantics is not modular. Subsequently, various other mathematical structures have been used to model the semantic domain, for this reason as well as others.

Here we use the monadic approach introduced by Moggi, specifically for the purpose of allowing modular semantics [16]. Modularity in the semantics allows us to retain a clear separation between basic semantics for a language of interest and semantics augmented with trace generation. Further, this separation of concerns makes it possible to investigate how tracing can be added to any denotational semantics consistently, as we show in the next section. Moggi’s ideas have been successful in the practical exploration of modular (denotational) semantics, such as the work of Liang, Hudak and Jones on modular interpreters [13]. In particular they use monad transformers to add features to an existing semantics, a practice we follow. One additional, advantageous feature of the denotational approach – which we exploit here – is that the semantic equations can be expressed directly in a functional programming language, as a (monadic) interpreter for the language of interest, and a (monad) transformer which can be used to transform this interpreter into one which automatically produces execution trace, in addition to the original semantics.

Nothing here is intended to imply this implementation of a denotational tracing system is optimal for any particular application – most practical systems would be quite optimised to the problem at hand – although here this is well suited to the exploration of the relationship between semantics and tracing.

4.1 Full and Faithful Traces

In order to explore the question of what information can potentially be provided in a denotational trace, we would like to have a 1:1 correspondence between trace objects and the corresponding semantics for each element of syntax. The mathematical term for this relationship is a bijection, meaning it is both injective (each semantic function has at most one corresponding trace object), and surjective (every semantic function has a corresponding trace object). Because we are using a monadic model for the denotational semantics here, this tells us that a bijective monad transformer (which adds tracing to a denotational, monadic specification of semantics) is therefore a full and faithful functor [25].

So by tracing all of the denotational semantic functions, we produce what we call a full and faithful trace since it contains all semantically relevant information. A full and faithful trace therefore represents a maximal bound with respect to the execution tracing information that can be collected, given a denotational semantic specification. A concrete example of a full and faithful trace is given in section 6.

It should be emphasised that full and faithful traces are unlikely to be optimally useful in unmodified form. For example, profiling activities will only focus on timing information and execution counts, which require only a subset of information available in the full trace, i.e. a derived trace. In fact it is an essential feature of many practical uses of tracing that full and faithful traces will not be used directly, because appropriate selection of the essential information needing to be included in a derived trace is key to enabling the associated activities effectively.²

Our purpose here is to highlight that full and faithful traces represent the maximal set of execution information available, derived via a completely straightforward transformation applied to the language semantics, and that many (and perhaps all) source-oriented, trace-based

² In fact almost all extant tracing systems produce what would here be called derived traces, directly.
activities can be enabled by traces derived from full and faithful, denotational traces.

5. A DENOTATIONAL TRACER
In order demonstrate how a trace can be derived from the denotational semantic specification of a language, and thereby explore the potential usefulness of denotational traces, we deliberately choose a very simple sequential language to trace. Firstly this is because simple sequential programming problems are of fundamental importance to most programming tasks, even those involving elements of concurrency. Secondly, as was discussed in section 3, denotational semantics has traditionally been better suited to sequential programming languages than concurrent ones, so we expect useful results may be found here.

The language we use here is based on Scott’s LCF and Plotkin’s PCF, which have a long history of supporting useful research into programming language semantics (and in particular the exploration of the correspondence between denotational and operational semantics [18]). We add named declarations to the language (i.e. a ‘let’ or ‘while’ declaration). Without this, recursive programs (and therefore loops) cannot be expressed without the use of a recursion combinator, severely impacting readability of source code. The execution order is applicative (aka eager/strict evaluation), the familiar order of argument evaluation found in most mainstream languages today. All functions are curried (i.e., have exactly one argument) as this can be easily used to model n-ary functions with any number of arguments. The concrete syntax for the language is a small subset of Haskell, although this is of little consequence here as we are really only interested in the abstract syntax. The meta-language used to implement the denotational semantics here is also Haskell [14].

5.1 Value Domain
The semantic domain supports a few basic types of values. Lambda function values are associated with the environment prevailing at the time of their declaration (i.e. the environment is bound statically as is usual in almost all languages).

```haskell
data Value = ValInteger Integer
            | ValBoolean Bool
            | ValLambda String EnvId (() -> Meaning)
```

The meaning of a program or expression is the value it denotes, augmented by both an environment and trace information.

```haskell
type Means a = Env a
```

The details of Env monad are not included here in the interests of brevity. It can be thought of as a type modifier applied to the value type, that associates appropriate environment bindings with semantic values. We will only be applying Means to Value, and this will appear many times in subsequent type declarations, so we define a convenient type alias for the semantic value domain:

```haskell
type Meaning = Means Value
```

5.2 Syntax
The concrete syntax for the language is a Haskell-like variant of the lambda calculus. The abstract syntax is very simple. A program consists of an expression, and an optional list of declarations. Expressions may include literals, identifiers, conditional expressions, function applications and lambda expressions:

```haskell
data Prog = Prog Expr [Decl]
data Decl = Decl String Prog
data Expr = Number Integer
            | Boolean Bool
            | Ident String
            | Lambda String Expr
            | IfStmt Expr Expr Expr
            | Apply Expr Expr
```

Because denotational semantics provides an explicit mapping from each abstract syntax element to a meaning in the semantic domain, composed if necessary from the meanings of syntactic sub-components, it is natural to express such semantics as structural induction over the data types describing the abstract syntax, such as those above. The pattern of recursion involved is known as ‘fold’ to functional programmers [9], and recognised as a functional programming design pattern known as a catamorphism [15]. At this point the semantics are explicitly structured as initial algebra semantics, and we see how naturally this arises in the context of denotational semantics, in particular due to source-orientation and compositionality.

Using these ideas we define the type of the fold algebra for the abstract syntax of the essential functional language, in particular we use updateable fold algebras as described in [12]:

```haskell
data SynCataAlg uProg uDecl uExpr = SynCataAlg
  { fProg :: uExpr -> [uDecl] -> uProg,
    fDecl :: String -> uProg -> uDecl,
    fBoolean :: Bool -> uExpr,
    fNumber :: Integer -> uExpr,
    fIdent :: String -> uExpr,
    fLambda :: String -> uExpr -> uExpr,
    fIfStmt :: uExpr -> uExpr -> uExpr -> uExpr,
    fApply :: uExpr -> uExpr -> uExpr }
```

The fold operation (i.e. pattern of recursion) is then defined for the abstract syntax algebra:

```haskell
class Fold alg t a where
  fold :: alg -> t -> a
instance Fold (SynCataAlg uProg uDecl uExpr) Prog uProg
  where fold alg (Prog e ds) = fProg alg (fold alg e) (map (fold alg) ds)
instance Fold (SynCataAlg uProg uDecl uExpr) Decl uDecl
  where fold alg (Decl i p) = fDecl alg i (fold alg p)
...
```

... etc.

The instances for the fold class corresponding to the remaining abstract syntax structures are derived similarly, as explained in [12].
5.3 Semantics

We can now express the meaning of a program as an instance of the syntax fold algebra:

evalA :: (Meaning -> Meaning -> Meaning) -> SynCataAlg Meaning Meaning Meaning

evalA = SynCataAlg declare
  bind
  (return . ValBoolean)
  (return . ValInteger)
  find
  abstract
  choose
  apply

Each of the operations specified in the algebra above defines the semantics for the corresponding item of syntax:

declare :: Meaning -> [Meaning] -> Meaning

declare expr decls = local $ do sequence_ decls
  expr

abstract :: String -> Meaning -> Meaning

abstract arg expr = do
  env <- getEnv
  return $ ValLambda arg env $ \_ -> expr

choose :: Meaning -> Meaning -> Meaning -> Meaning

choose p e1 e2 = do ValBoolean b <$> p
  if b then e1 else e2

apply :: Meaning -> Meaning -> Meaning

apply fn expr = do
  ValLambda arg env str lambda <- fn
  e <$> expr
  withEnv env $ do bind arg $ return e

A declaration, or a 'program', is simply the meaning of an expression, evaluated in the context of a "local" environment with an optional sequence of associated bindings. Lambda abstractions are evaluated in the context of the environment that prevailed at the time of definition - the function "getEnv" simply returns an identifier for the current environment state. Application of lambdas occurs in the context of the environment at the time of abstraction (using "withEnv"), extended with the additional binding of the argument value to the lambda argument variable identifier. Function application is strict: first the function expression is evaluated to derive a lambda value or error, then the argument is evaluated and bound to the argument in the environment, before finally the body of the lambda is evaluated. Choice, provided by the "if" statement, evaluates a (Boolean) condition then evaluates the "then" or "else" consequent. Binding and finding of identifiers via the bind and find functions in the algebra operate exactly as would be expected and are not interesting to include here.

Denotational semantics usually uses Scott brackets, to provide an explicit mapping from (abstract) source constructs to semantic domain values. We do not have access to these special symbols in Haskell, however the transliteration between the two is trivial and purely notational.

More importantly, in effect what we have here are denotational semantics structured as initial algebra semantics, i.e. a fold applied to the associated algebra, gives us an interpreter which can be applied to any abstract syntax element:

interpret = fold evalA

5.4 Trace Structures

For the purposes of this paper, the answer to the question, “what should be traced?” is answered with, “everything,” i.e., we want a full and faithful record of the semantics, because we are interested in the limits to the information available on this basis. The trace structure thus directly reflects the compositional structure of the semantic equations in 1:1 correspondence:

data Trace
  = Prog [Trace] -- bindings
  | Trace -- execution
  | Bind Identifier Trace Value
  | Boolean Bool
  | Number Integer
  | Choose Trace -- condition
  | Trace -- consequent
  | Find Identifier Value
  | Abstract Identifier Value
  | Apply Trace -- function
  | Trace -- argument
  | Trace -- binding
  | Trace -- execution
  | Value -- result

5.5 Trace Generation

Tracing is added to our semantic domain as follows:

type Means' a = (TraceT Means) a

TraceT is a can be thought of as a type modifier applied to our existing value domain, which associates appropriate state information with semantic values, being the trace information collected thus far during execution. Once again we define a convenient type alias:

type Meaning' = Means' Value

Full and faithful tracing as specified above is implemented here by defining a transformation (‘traceT’ - a higher-order function) that can be applied to the usual semantic fold algebra for the essential functional language:

traceT :: SynCataAlg Meaning Meaning Meaning

- SynCataAlg Meaning' Meaning' Meaning'

traceT alg = SynCataAlg
  (\ expr decls -> trace "prog" []
    $ fProg alg expr decls)
  (\ ident prog -> trace "bind" [ident]
    $ fDecl alg ident prog)
  (\ bool -> trace "boolean" [show bool]
    $ fBoolean alg bool)
  (\ num -> trace "number" [show num]
    $ fNumber alg num)
  (\ ident -> trace "find" [ident]
    $ fIdent alg ident)
  (\ arg thunk str -> trace "abstract" [arg, str]
    $ fLambda alg arg thunk str)
  (\ p e1 e2 -> trace "choose" []
    $ fIfStmt alg p e1 e2)
  (\ fn expr -> trace "apply" []
    $ fApply alg fn expr)
This allows us to define a modified interpreter, which augments the original one by collecting trace information as well:

```
interpret' = fold $ traceT evalA
```

The practical motivations for tracing often involve programs that are long-running or do not terminate, or programs that terminate with an exception or error. This suggests that traces should be structured as an ordered, potentially infinite collection of abstract trace events.

At the lowest level we model traces similarly here, using the inbuilt Haskell list type:

```
type Trace = [Event]
```

Further, a list allows us to have “begin” events which do not have a corresponding “end” event, which provides a convenient description for programs which terminate in error, with one or more semantic steps incomplete. While this model of a sequence of events provides a suitable low-level basis, it does not describe richer, compositional structure relating directly to syntax. Without events indicating both the start and end of semantic operations, their exact hierarchical relationship cannot be recorded. Thus each trace event must encode the details of the associated semantic function beginning (inputs) or ending (outputs) and the timestamp.

In practice all that the “traceT” transformer does is to wrap each function in the existing algebra in a call to “trace” which has the side-effect of logging begin and end trace events associated with that function; the semantics described by the original algebra are not altered.

```
trace str details action = do bgn
                                  result <- action
                                  end $ printVal result
                                  return result

where bgn = log Bgn $ args details
end = log End
```

The “log” function records a trace message, with “args” formatting a list of arguments appropriately.

```
function records a trace message, with “args” formatting a list of arguments appropriately.
```

When working with these low-level traces we will first be parsing them into the trace structures defined in the previous section – note that the trace structures could have been generated directly in principle.

### 6. A SIMPLE EXAMPLE

Having defined full and faithful tracing for the example language, we can now explore the concept of full and faithful traces in practice with an example program.

Consider the function “f” (factorial) defined by:

```
f (n) = ff (1,n)              (1)
ff (m,0) = m                  (2)
ff (m,n+1) = ff (m*n,n)       (3)
```

This function “f” is in turn defined as a tail-recursive version corresponding to the familiar iteration often seen in introductory programming texts. Restating “f” in the concrete syntax of our simple language\(^3\), we have:

```
f = \n -> ff 1 n;
ff = \m -> \n -> if eq n 0 then m 
   else ff (mul m n) (sub n 1);
```

NB: here we use two quite different notations: numbered equations in an italic font for specification/reasoning by the programmer, and a fixed-width font for program code and trace outputs. This is to highlight the interaction between mathematical, equational reasoning by the programmer, and machine-readable source and machine-written traces and their derivatives.

Because full and faithful traces are inherently and indeed maximally verbose, a small and simple example was chosen deliberately here for reasons of space. Executing the expression “f 0” results in the following trace:

```
> prog
  > bind f
    > prog
    > abstract n (ff 1 n)
    > abstract = \n -> ff 1 n
    > prog = \n -> ff 1 n
    > bind = \n -> ff 1 n
    > bind ff
      > prog
      > abstract m (\n -> ...)
      > abstract = \n -> \n -> ...
      > prog = \m -> \n -> ... 
      > bind = \m -> \n -> ...
    > apply
      > find f
      > find = \n -> ff 1 n
      > number 0
      > number = 0
      > bind n
      > bind = 0
    > apply
      > apply
      > find ff
      > find = \m -> \n -> ...
      > number 1
      > number = 1
      > bind m
      > bind = 1
      > abstract n (if ...)
      > abstract = \n -> ...
      > apply = \n -> ...
    > find n
    > find = 0
    > bind n
    > bind = 0
    > choose
      > apply
      > apply
      > find eq
      > find = \v1 -> eq v1
      > find n
      > find = 0
      > bind v1
      > bind = 0
      > apply = \v2 -> eq v0 v2
```

\(^3\) The ‘standard library’ for the simple language includes just three inbuilt, primitive functions: “eq”, “mul” and “sub”, being equality, multiplication and subtraction respectively.
The "..." in the trace messages to the right is an elision of longer messages, for readability, also the timestamp and environment identifier have been elided from each trace event as they are not used here; otherwise the trace is exactly as it is produced in full form. The indentation is a simple presentation device to help visualise and highlight the nested structure.

6.1 Specification Recovery

In order to show how traces can be useful for reasoning about correctness, we explore an example of specification recovery, where the specification for a program is recovered from its traces. This example is directly applicable to a programmer engaged in software maintenance, attempting to understand a program by reading (possibly undocumented) source code and the corresponding execution traces – a very common scenario in industry.

Consider the evaluation of "f3":

```
> f 3
  > ff 3 1
    eq 3 0 = False
    mul 1 3 = 3
    sub 3 1 = 2
    > ff 3 2
      eq 2 0 = False
      mul 3 2 = 6
      sub 2 1 = 1
      > ff 6 1
        eq 1 0 = False
        mul 6 1 = 6
        sub 1 1 = 0
        > ff 6 0
          eq 0 0 = True
          < = 6
          < = 6
          < = 6
          < = 6
```

The trace above consists of just the "apply" events. Nothing is reordered; some information that is irrelevant to the task at hand has been removed. Laying out this information a little differently, we can see the execution tracing of "f" with a sample selection of inputs (arguments) n = 0, 1, 3, etc. will yield respective information:

```
f 0 = ff 1 0
    = 1
f 1 = ff 1 1
    = ff (mul 1 1) (sub 1 1)
    = 1
```

The great value of these execution traces is that from them may now be inferred an invariant for "ff". While invariant analysis is usually presented in iterative procedural contexts, it’s equally applicable in tail-recursive functions. For initial argument values to ff of m=M0 and n=N0, then for any subsequent invocation of ff we hypothesise:

```
m * fact (n) = M0 * fact (N0) (4)
```

where

```
fact (0) = 1 (5)
fact (n+1) = (n+1) * fact (n) (6)
```

Note that this “fact”, while potentially executable, plays the role of a specification rather than an implementation artefact in this scenario.

Subsequently, this invariant hypothesis can be separately validated and used in the discovery of specifications of other components. Deductive validation can be achieved using correctness proof techniques (e.g. fold-unfold program transformations [4] involving factorial). The specification for ff can be derived from conjoining the invariant with the termination condition ff(m,0)= m, thus

```
ff(M0, N0) = M0 * fact (N0) (7)
```

The specification for f is then derived directly

```
f(X) = ff(1, X) = 1 * fact (X) = fact (X) (8)
```

6.2 Time Complexity

Because a faithful trace contains a record of all semantic steps, the length of such a trace can be used as a direct measure of the time taken to execute a program. Other measures of time complexity derived from the full and faithful trace are certainly possible, for example varying time costs could be associated with each syntax element/semantic production.

---

4 A simple program was built to generate this derived trace.

5 A small program was built to compute this – an example of induction on the trace structure.
For the factorial example above, we can tabulate the number of semantic steps involved in executing the “f” function:

<table>
<thead>
<tr>
<th>N</th>
<th>steps (f n)</th>
<th>steps (f (n - 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>56</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>84</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>112</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 2. Time Complexity of “f”

Because the number of steps increases by the same amount with each increment of “n” we can infer the relationships:

\[
\begin{align*}
\text{steps } (f(n+1)) &= \text{steps } (f(n)) + 28 \\
\text{steps } (f(n+1)) &= 28 + n \times 28
\end{align*}
\]  

The time complexity of “f” for these inputs at least is obviously linear with respect to “n”, or in the familiar “big-oh” notation: O(n).

Because we have access to full and faithful traces associated with the execution of “f” for each value of “n” above, a stronger result is possible: in fact we can prove via basic induction that the time complexity of “f” is O(n), because the faithful traces in this case provide complete coverage of the “if” branches in the code. There is no other “if” branch in the source code which we have not already seen executed, which could result in a count of semantic steps differing from the pattern identified above, given any value of “n” (for the natural numbers, at least). Note that we do not even need to refer to the source here or collect any further information – the faithful trace is sufficient in this case to show us that there are no branches that have not already been executed.

### 6.3 Space Complexity

Given the simple nature of the example language - which has no heap-allocated space, just a call stack, and where all functions have precisely one argument – a measure of stack space usage can be derived from the faithful trace by counting the depth of nesting of “apply” semantic steps. As for time complexity, other measures of space complexity could also be computed, but are not considered here.

Tabulating the maximum stack depth for various values of “n” we have:

<table>
<thead>
<tr>
<th>N</th>
<th>space (f n)</th>
<th>space (f (n - 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3. Space Complexity of “f”

Now we can infer:

\[\text{space } (f(n+1)) = \text{space } (f(n)) + 1\]  

The space complexity is linear, or O(n). As before, we can go further – given that the full and faithful traces provide full coverage of the “if” branch – and prove via induction that this is the case.

### 7. RELATED WORK

While most tracing frameworks have an application-driven, ad hoc design to the structure and details of their trace, the work of Kishon et. al. provides a generalized treatment of what they call “execution monitoring” – in effect a generalization of dynamic instrumentation [11]. Their implementation method is similar to the example used in this paper, in that altered, ‘monitoring’ semantics are derived from the denotational semantics for a functional language. Although some examples of monitors are presented including a very basic tracer, no systematic treatment is given to the topic of tracing per se, or how it relates to a given, arbitrary semantic specification [11]. Nilsson and Sparud developed a structure called the “evaluation dependence tree” (EDT) in an attempt to provide a precise definition of what a trace structure is in the context of a lazy, functional language [17], but this was not generalized to other language contexts.

### 8. CONCLUSION

The ad-hoc design of existing, practical tracing systems, and lack of effective formal theory, results in real, practical problems for programming language designers, implementers and users. The denotational (vs. operational) theory of tracing offers a conceptually-integrated solution to these practical problems, based in formal mathematics, justified by the category-theoretic concept of duality:

- For the first time an explanation has been offered of the practical success vs. failure of the trace monoid for specific applications such as concurrency, versus some more day-to-day uses for tracing (see sections 2 & 3). Denotational tracing has been introduced to directly address the practical applications where trace monoids fail to be useful (see section 4).
- The concept of denotational, full and faithful traces provides a language designer or implementer with the full set of information available for tracing, including how it is structured and precisely how it relates to a (denotational semantic) specification of the language. The question as to what traces may contain, and how they are structured, is answered (see section 4.1).
- An example of denotational, full and faithful traces was provided in section 5, and for the example of a simple program shown to support the recovery of program specification from trace – a high-value software engineering activity of considerable usefulness to an engineer engaged in the maintenance of poorly or incorrectly documented code (see section 6.1). Note that this result is not possible using existing theoretical techniques where traces are structured as trace monoids since monoids lack the structure required to encode the necessary information.

6 Once again, simple, structural induction on the trace was used to build a small tool to compute this.
• Given this solid, mathematical basis for practical tracing, the user of a denotational tracing system now has access to proven, reliable tools such as proof by induction. Back-of-the-envelope proofs from traces of the space and time complexity of programs - as was shown in the examples in this paper (see sections 6.2 & 6.3) - are of great usefulness to anyone needing to analyse or debug the performance of real programs. Note that these results would also not be possible if traces were structured as monoids.

• Formal, mathematical techniques often allow for easy automation, thereby providing better tool support for trace users; three examples of simple tools to support reasoning from traces are described in this paper (see section 6).

Existing results in computer science which have established full abstraction between operational and denotational semantics under various conditions [18], suggest that it may be possible to both prove the correctness of and/or automatically derive, an (operational) implementation of a language including denotational tracing, given a (denotational) specification of semantics. Further research is required to explore the full potential of these ideas, and their practical implications.

REFERENCES