Oracle Types

Akiva Leffert Jason Reed

March 19, 2021

Abstract

We present Oracle Types, a new type-theoretic primitive for Typescript, which permits user-customizable extensions to the type language and type-checking algorithms. This enables several applications: arithmetic constraint checking, a dynamic live-update ORM, type-safe localization of multi-language data, and a rich, 'mobile-first' interactive type-checking experience.

1 Introduction

Although advanced type systems are commonly available, even for languages such as Javascript, the unquenchable thirst for new type-theoretic features exceeds the ability of even the most diligent implementors to keep up. But that hasn't stopped Typescript from trying. Pursuing their mission of typing even the most outlandish of Javascript programs, the Typescript team has taken the 'yes and' approach to theoretical constructs to new heights, introducing concepts such as conditional types and string template types.

Still, despite their best attempts, the designers of Typescript have failed to achieve apotheosis.

What is needed, clearly, is *extensibility*, so that programmers can use whichever type theoretic features are most important to their domain, defying the limits of the system's creators. We employ a well-understood theoretical device to make things sort of work out in some arbitrary fashion despite the shackles of previously established premises ([LAL04]) and defer to an *external oracle*.

In order to do this in a principled way we introduce *Oracle Types* and proceed to demonstrate their utility by adding them into the Typescript compiler.

In Section 2.1, we show how oracle types can be used to attach annotations to numeric types which express arithmetic properties of them, which can be checked at compile time with constraint solvers. In Section 2.2, we use oracle types to automate localization of data structure fields. In Section 2.3, a dynamic ORM based on oracle types automatically ensures that the types of the primitives provided by the ORM reflect the current schema of the database, without requiring any explicit schema recompilation step. In Section 2.4, we discuss an application that oracle types are uniquely suited for: using modern

mobile technology, we can tap the type-theoretic expertise of individual human beings in real-time to contribute interactively to the type-checking process.

Our implementation is available at

`https://github.com/bovik-labs/oracle-types`

2 Examples

2.1 Arithmetic Constraints

An example of a *refinement type* of numbers is one that constrains a number to have a certain arithmetic property. For example, we can introduce the type

`type LessEq<T extends number>` = ···

so that, for example, `LessEq<5>` is the type of all numbers that are less than or equal 5. Similarly, we have type operators such as

`type Plus<T extends number, U extends number>` = ···

So that if 2 and 3 are understood as the singleton refinements consisting of only the number 2 (and 3, respectively), then naturally `Plus<2,3>` is the singleton type consisting of only the number 5. Using Oracle Types, we can automate inference of semantically entailed subtyping relationships, by appealing to the constraint solver Z3 [dMB08] to do the actual inference. For example, in the following code:

```
1  import { Plus, LessEq, infer } from './z3';
2
3  function test_cases(x: LessEq<5>) {
4
5    //// Error, because <= x (+ 2 3) is not the same as <= x 5!
6    { const bad: LessEq<Plus<2, 3>> = x; }
7
8    // However:
9    { const good: LessEq<Plus<2, 3>> = infer(x); }
10
11   //// Error, because not sound to infer <= (+ 2 2) from <= 5!
12   { const bad: LessEq<Plus<2, 2>> = infer(x); }
13
14   // <= 5 implies <= 6
15   { const good: LessEq<Plus<2, 4>> = infer(x); }
16
17 }
```

the `infer` function allows subtyping from one `LessEq` constraint to another, so long as the entailment is valid over the ordered monoid natural numbers under addition. The programmer must remember to insert enough `infer` calls to mediate between syntactically non-identical constraint types, but due to Typescript's own type inference, the annotation burden is fairly mild: the type indexes on infer need not be specified in the example above, because they can be inferred from the return constraint type.

2.2 Dynamic Translantion

Ah, language, a true wonder of human ingenuity. And yet, most programming is done in English. APIs are typically designed in English. This hardly seems fair to the 94% of the people of the world whose first language is not English. Good software is localized. Evidently programming languages are not good software. Even were one to localize a language itself, it would have to interact with English language names for libraries, API payloads, etc.

Oracle types give us a solution. Consider a function

$$\texttt{Localize} : \texttt{string} \times \texttt{string} \times \tau \to \tau$$

that would localize the fields of a record. For example, suppose we want to write a calendaring app, and we have a record with fields for each day of the week.

```
type Schedule = {
  'Sunday': string,
  'Monday': string,
  'Tuesday': string,
  'Wednesday': string,
  'Thursday': string,
  'Friday': string,
  'Saturday': string
}
```

Look at those garbage English names! *¿Y si quisiéramos programar en español?* With the `Localized` type constructor, we can easily solve this problem. Simply wrap your type in it:

```
type Calendario = Localized<'en', 'es', Schedule>

// Equivalent to
type Calendario = {
  'Domingo': string,
  'Lunes': string,
  'Martes': string,
  'Miercoles': string,
  'Jueves': string,
  'Viernes': string,
  'Sabado': string
}
```

2.3 Dynamic ORM

An inconvenience of most ORM (Object-Relational Model) systems is that the user needs to explicitly represent the database schema in some way that the ORM can consume it, and present an API to the user that is type-correct with respect to that schema. With Oracle Types, we can avoid this explicit step, and instead consult the database itself at type-check time, and type the ORM API functions accordingly.

Here is a small example to demonstrate its use. We imagine a database with three tables, `users`, `papers`, and `reviews`, to model a set of research papers and reviews thereof. Its schema is the following:

```
CREATE TABLE users (id int PRIMARY KEY NOT NULL, name text);
CREATE TABLE papers (id int PRIMARY KEY NOT NULL, title text, author int REFERENCES users(id));
CREATE TABLE reviews (
  id int PRIMARY KEY NOT NULL,
  score int,
  author int REFERENCES users(id),
  paper int REFERENCES papers(id)
);
```

Using this database we can write the following typescript code:

```
import { getModels } from './orm';

const connection = <const>{
  db: 'postgres',
  user: 'postgres',
  host: 'database',
  port: 5432
}

async function go() {
  const models = await getModels(connection);
  const User = models.get('users');
  const Paper = models.get('papers');
  const Review = models.get('reviews');

  const papers = await Paper.findAll();
  for (let i = 0; i < papers.length; i++) {
    const paper = papers[i];
    const author = (await paper.author()).name;
    console.log(`paper id ${paper.id} title ${paper.title} author ${author}`);
  }

  const users = await User.findAll();
  users.forEach(user => {
    const id = user.id;
    const name = user.name;
    console.log(`their name is ${name} and id is ${id}`);
  });

  // Transitive foreign key traversals should work
  const review = (await Review.findAll())[0];
  const { author, id, paper, score } = review;
  const opaper = await paper();
  const oauthor = await author();
  console.log(`the review had score ${score} and was written by ${oauthor.name}`);
  console.log(`the paper it was about was by ${(await opaper.author()).name}`);

  process.exit(0);
}
```

On line 3, we set up the connection information required to connect to the database. Line 11 uses this information to obtain a single object that contains models for all tables of the database. Lines 12-14 get individual tables out of the models object — enough type information is present that the programmer can autocomplete on the names of the tables upon entering the argument of the `get` method.

On line 19 we can see that we can get the author of a paper in a natural way, because the `paper` model object obtained from line 16 has a method that is automatically populated from the foreign key relationship between the `papers` table and the `users` table.

Lines 31-36 demonstrate that traversing multiple hops through the foreign key graph is scarcely more difficult than one hop.

The principal advantage of this design is that if the database schema changes in such a way that the code is no longer semantically valid, that invalidity is immediately realized as a type error, without any intermediate step being required to regenerate the host-language representation of the schema.

2.4 Mobile-First Interactive Typechecking

We take inspiration from other systems research work, in which a failure to answer a query by normal means can be remediated by a technique (pioneered in [PBK+99]) called 'Phone-a-Friend'.

Our realization of this protocol works as follows. The library we provide offers a type constructor `PhoneAFriend<PhoneNumber, Query>`, which uses an external SMS API service to relay the query to the human with the given phone number. The human makes a response, which the SMS service sends to a previously configured HTTP endpoint on a webhook server running in the cloud. The typechecker makes an http request to the webhook server, waiting on a response, which is parsed into a type by our library.

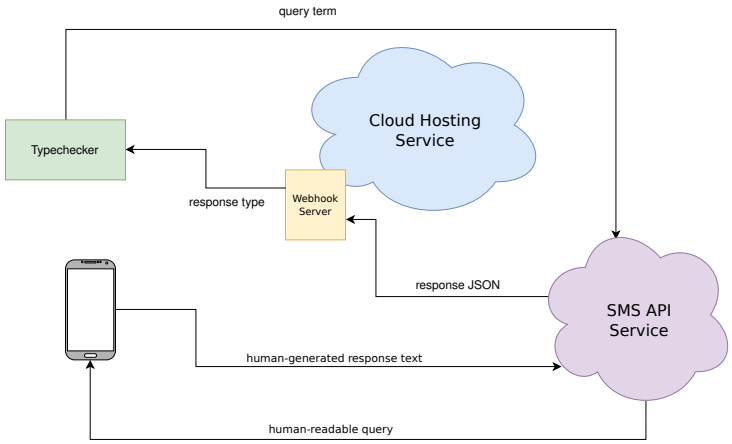

Figure 1: Interactive Type-Checking Network Architecture

3 Implementation

The key implementation technique that enables all the above applications, which, to the best of the authors' knowledge, has somehow been overlooked by decades of programming language research, is allowing a type operator to have the ability, as a side-effect, to run an arbitrary shell command dervied from a type argument.

3.1 Modifying the Typescript Compiler

Typescript 4.1 [Mic20] added several new `IntrinsicTypeKind` type operators which allow manipulation of types extending `string`. For example, `Uppercase<'foo' | 'bar'>` reduces to the type `'FOO' | 'BAR'`. By analogy with these types, we introduce `Shell<T extends string>`, whose semantics is defined as follows: Any strings in the disjunctive expansion of `T` are executed as shell subprocesses, and whatever they write to the `stdout`

file descriptor is collected and yielded as the result type. The implementation is quite simple; the crux of the change required is to extend the function `applyStringMapping` in `TypeScript/src/compiler/checker.ts` like so:

```
1         function applyStringMapping(symbol: Symbol, str: string) {
2             switch (intrinsicTypeKinds.get(symbol.escapedName as string)) {
3                 case IntrinsicTypeKind.Uppercase: return str.toUpperCase();
4                 case IntrinsicTypeKind.Lowercase: return str.toLowerCase();
5                 case IntrinsicTypeKind.Capitalize: return str.charAt(0).toUpperCase() + str.slice(1);
6                 case IntrinsicTypeKind.Uncapitalize: return str.charAt(0).toLowerCase() + str.slice(1);
7                 case IntrinsicTypeKind.Shell: {
8                     const exec = require('child_process').execSync;
9                     return exec(str).toString();
10                }
11            }
12            return str;
13        }
```

The remaining changes are mere plumbing to ensure that `IntrinsicTypeKind.Shell` is well-defined in the same way as `Uppercase`, `Lowercase`, etc.

3.2 Implementing Arithmetic Constraints

Given the type primitive `Shell<T>`, it is relatively straightforward to interface with Z3 to provide constraint solving in the type system. We work throughout with template string literal types to manipulate sexpressions in SMT-LIB [BFT16] format.

```
1  // Strip trailing newline from a string literal type
2  type StripNl<T extends string> = T extends `${infer S}\n` ? S : T;
3
4  // Given a string type containing an sexp expressing a z3 program,
5  // return 'sat' or 'unsat'
6  type SolverResult<Z3 extends string> =
7    StripNl<Shell<`echo '${Z3}' | z3 -in`>>;
8
9  // A phantom type used to express constraints about integer values
10 type Constr<T> = { constr: T };
11
12 // An integer value so constrained
13 type ConstrNum<T> = number & Constr<T>;
14
15 // Generate a Z3 assertion for constraint T
16 type GenAssert<T> = T extends string ? `(${T})` : 'false';
17
18 // Generate Z3 code that checks whether T implies U.
19 // Z3 will return 'unsat' if the implication *does* hold,
20 // because T && !U will be false.
21 type GenZ3<T, U> = `
22 (declare-const x Int)
23 (assert ${GenAssert<T>})
24 (assert (not ${GenAssert<U>}))
25 (check-sat)
26 `;
27
28 // If T => U, yields the appropriate result type for constraint U, otherwise unknown.
29 type InferCond<T, U> = SolverResult<GenZ3<T, U>> extends 'unsat' ? ConstrNum<U> : unknown;
30
31 // Convert x from one constraint type to another
32 export function infer<T, U>(x: ConstrNum<T>): InferCond<T, U> {
33   return x as any;
34 }
35
36 type strish = string | number;
37 export type Plus<T extends strish, U extends strish> = `(+ ${T} ${U})`;
38 export type LessEq<T extends strish> = ConstrNum<`<= x ${T}`>;
```

The type constructors `Plus` and `LessEq` (lines 37-38) build up sexpressions representing addition and the boolean less-than constraint, stringifying

numerical constants as necessary. These can be used to build up instances of the type `ConstrNum<T>` (line 13), which represents the refinement of the type `number` which must satisfy constraint `T`. The type `GenZ3` (line 21) converts two assertions T and U into a complete Z3 query which tries to determine the satisfiability of $T \wedge \neg U$. This is the negation of the implication $T \Rightarrow U$, so if the query returns an answer of **un**satisfiable, we know the implication holds. for this reason, the type `InferCond<T, U>` (line 29) returns a `ConstrNum<U>` only if the `GenZ3` query returns the string `unsat`, and returns `unknown` otherwise, inducing a type error, in the case that the attempted subtyping coercion fails.

3.3 Implementing Localized Types

To implement localization, we can simply shell out to a script that calls into the Google Translate API, or uses a local dictionary on the filesystem.

3.4 Implementing the Dynamic ORM

In order to implement the dynamic ORM, we first of all must have a way of getting the schema out of the database. Using the well-known open-source postgres database engine, this is not difficult: the schema (and foreign key relationships) are themselves stored in metadata tables, and are easily obtained with standard SQL queries.

A standalone script named `get_schema.js` is implemented, which can be called by the Oracle Type invocation like so:

```
1  // A type representing postgres connection information
2  type Conn = {
3    db: string,
4    user: string,
5    host: string,
6    port: number
7  };
8
9  // Given connection `C`, calls the `get_schema` script to get the
10 // schema of a postgresql database and returns a string literal type
11 // containing it as json.
12 type SchemaTxt<C extends Conn> =
13   Shell<`node get_schema.js ${C['host']} ${C['port']} ${C['user']} ${C['db']}`>;
```

Template literal types are used to interpolate the database connection information into the command-line arguments. The output of `get_schema.js` is a JSON string representing an object containing a description of the database schema, and so it must be parsed to obtain an actual object type.

Fortunately, parsing JSON at the TypeScript type level is easily accomplished via well-understood and sound engineering practices [Kyl20]. From there, the implementation of our ORM is straightforward. The function `getModels` uses the same schema-getting code (only now at run-time) to build a `Models` object from the schema:

```
1  export async function getModels<C extends Conn>(conn: C): Promise<ModelsI<SchemaOf<C>>> {
2    const client = get_client(conn);
3    return new Models(client, await get_schema(client)) as any;
4  }
```

The `Models` class so invoked simply constructs an instance of `Model` for the appropriate table:

```
// Given a DbSchema, returns a class with getters for individual tables
class Models<DB extends DbSchema> {
  constructor(public client: DbClient, public schema: DB) { }

  get<K extends string & keyof DB['table_schemas']>(tableName: K): TableModel<DB, DB['table_schemas'][K]> {
    return new Model<DB, DB['table_schemas'][K]>(this, tableName);
  }
}
```

We can see here specifically how IDE auto-completion of table names functions; `DbSchema`'s field `table_schemas` is a map whose keys are table names, so the type of the only argument of `get` becomes a disjunction type of all table names, and the TypeScript language server can communicate exactly this set to the programmer.

Finally, the heart of the implementation is the `Model` class:

```
// Implement the utility class for a model
class Model<DB extends DbSchema, S extends TableSchema> implements TableModel<DB, S> {
  constructor(public models: Models<DB>, public name: string) { }

  proxyForeignKeys(row: RowModel<S>): RowOfDb<DB, S> {
    return new Proxy(row, {
      get: (target, prop) => {
        const found = this.models.schema.table_schemas[this.name].cols
          .find(col => col.column_name === prop && is_foreign(col.data_type));
        if (found && is_foreign(found.data_type)) {
          const dt = found.data_type;
          return async () => {
            const formatted = format(
              'select * from %I tgt where tgt.%I = %L',
              dt.target_table, dt.target_col, (row as any)[found.column_name]
            );
            const res = await this.models.client.query(formatted);
            return this.proxyForeignKeys(res.rows[0]);
          }
        }
        else {
          return (target as any)[prop];
        }
      }
    }) as any;
  }

  async findAll(): Promise<RowOfDb<DB, S>[]> {
    const res = await this.models.client.query(format('select * from %I', this.name));
    return res.rows.map(row => this.proxyForeignKeys(row)) as any;
  }
}
```

The `findAll` method actuall executes a query against the database, asking for all rows of a table. Instead of just returning the rows as they are, we modify them with the method `proxyForeignKeys`. The effect of that is to patch field accesses to the row, using the `Proxy` class, so that columns that are foreign keys become auto-populated method calls which perform further database queries to get the corresponding row of the foreign table.

All of this proxying we have described takes place at run-time; a corresponding rewriting must also take place at type-checking time for the types to appear correct to the programmer. This takes place in the definition of the types that lead up to `TableModel`:

```
// Convert from a string describing a pg type to an appropriate typescript type
type Inhab<K> =
  K extends 'text' ? string :
```

```
  K extends 'integer' ? number :
  K;

// Given a single column type, return the type that should be the value
// part of the row record for that column.
type MakeColumn<T extends Column> = Inhab<T['data_type']>;

// Given a disjunction of columns, return the object type that is one row.
type DisjunctToRow<T extends Column> =
  { [K in T['column_name']]: MakeColumn<Extract<T, { column_name: K }>> };

// Given a single table's schema, return the typescript type of one row of that table.
type RowModel<S extends TableSchema> = DisjunctToRow<S['cols'][number]>;

type AsyncThunk<T> = () => Promise<T>;

type LookupStub<TS extends TableSchemas, TABLE> =
  TABLE extends keyof TS ? AsyncThunk<ProxiedRowModel<TS, TS[TABLE]>> : "couldn't find table reference";

type LookupStubs<TS extends TableSchemas, ROW extends { [k: string]: any }> =
  { [K in keyof ROW]: ROW[K] extends schema.Stub<infer TGT> ?
    LookupStub<TS, TGT> : ROW[K] };

// Given a single table's schema, return the typescript type of one row of that table,
// with proxies for foreign keys
type ProxiedRowModel<TS extends TableSchemas, S extends TableSchema> =
  LookupStubs<TS, RowModel<S>>;

// Given a database schema (for recursive lookups), and single table's
// schema, return the typescript type of one row of that table, with
// proxies for foreign keys.
type RowOfDb<DB extends DbSchema, S extends TableSchema> =
  ProxiedRowModel<DB['table_schemas'], S>;

// Given a single table's schema, return the typescript type of the utility class for that model
interface TableModel<DB extends DbSchema, S extends TableSchema> {
  findAll: () => Promise<RowOfDb<DB, S>[]>
}
```

Note the use in `LookupStubs` of conditional types with inference, in that we can find the target table (`infer TGT`) of a stubbed foreign key in the schema.

3.5 Implementing Mobile-First Interactive Typechecking

The client-server architecture that enables SMS-based interactive typechecking is fairly simple. The client looks something like this:

```
import * as http from 'http';
import * as tiny from 'tiny-json-http';
import * as fs from 'fs';

const twilio = require('twilio');

const accountSid = ... ;
const authToken = ... ;
const client = twilio(accountSid, authToken);
const server = ... ;
const url = `http://${server}/listen`;
const args = process.argv.slice(2);

async function send_msg(msg: string): Promise<any> {
  return client.messages
    .create({
      from: ... ,
      to: ... ,
      body: msg,
    });
}

async function go() {
  const msg = args[0];
  if (msg !== undefined && msg !== "") {
    await send_msg(msg);
  }
  // receive response
  const res = await tiny.get({ url });
  console.log(res.body.message);
}

go().catch(x => console.log(JSON.stringify({ error: x })));
```

This script sends a message based on the commandline arguments, and makes a request to a `/listen` callback, expecting to receive a message corresponding to the user-returned type.

The server this client communicates with is build with the Twilio API:

```
const fs = require('fs');
const accountSid = ... ;
const authToken = ... ;
const client = require('twilio')(accountSid, authToken);

const http = require('http');
const express = require('express');

const MessagingResponse = require('twilio').twiml.MessagingResponse;

const app = express();
app.use(express.urlencoded({extended: false}));

// listener: undefined | (x : string) => void
let listener = undefined;

function notify(message) {
  if (listener !== undefined) {
    listener(message);
    listener = undefined;
  }
}

app.post('/sms', (req, res) => {
  console.log('got a message');
  console.log(req.body);
  const message = req.body.Body;
  notify(message);
   res.writeHead(200, {'Content-Type': 'text/xml'});
   res.end('<?xml version="1.0" encoding="UTF-8"?><Response></Response>');
});

app.get('/listen', (req, res) => {
  listener = (message) => { res.json({message}); }
});

http.createServer(app).listen(1234, () => {
  console.log(Relay server listening on port 1234');
});
```

It establishes a webhook callback at `/sms` for Twilio to notify it that an inbound SMS message has arrived, and notifies a listener waiting for a response on `/listen`. The service only supports a single user for the sake of a prototype, but we expect it would be a fruitful exercise to scale this server up to many concurrent users.

To use the client script `client.js` from a typescript program, one needs only write code such as

```
type Interpret<Msg extends string> = Lowercase<Msg> extends "number\n" ? number
  : Lowercase<Msg> extends "string\n" ? string
  : Msg;

export type PhoneAFriend<Query extends string> =
  Interpret<Lowercase<Shell<`client.js '${Query}'`>>>;

const x: PhoneAFriend<'What type is 42?'> = 42;
```

When this code is typechecked, the user will be texted “What type is 42?” and they will have the opportunity to respond, and their response will be converted to the appropriate type.

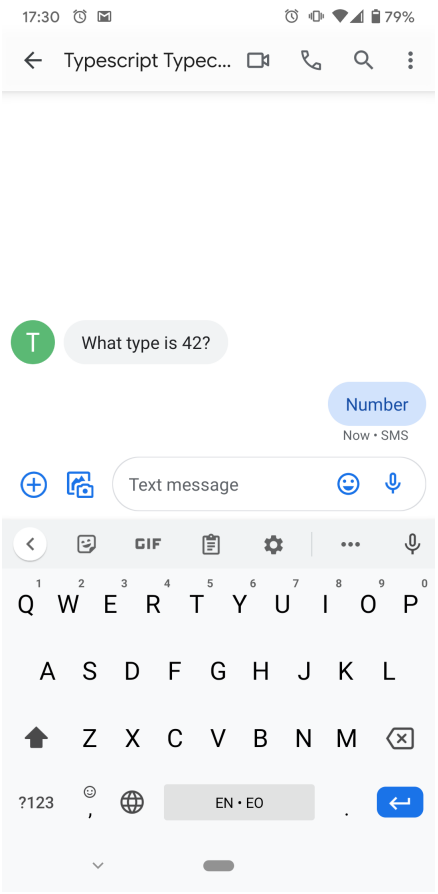

Figure 2: Interactive SMS Type-Checking

4 Conclusion

As our example applications show, opening the door to arbitrary shell computation at the type level leads to a variety of useful applications. Petty concerns about 'determinacy' or 'security' or 'why am I being charged $0.0075' or 'where did all my files go, I just tried to type-check some sketchy code I found on the Dark Web' are clearly the purview of regressive, hide-bound curmudgeons who don't think programming should be fun.

References

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Kyl20] Jamie Kyle. JSON parser in TypeScript. `https://github.com/jamiebuilds/json-parser-in-typescript-very-bad-idea-please-dont-use`, 2020.

[LAL04] Jeffrey Lieber, J. J. Abrams, and Damon Lindelof. *Lost*, 2004.

[Mic20] Microsoft. Typescript 4.1. `https://github.com/microsoft/TypeScript`, 2020.

[PBK+99] Regis Philbin, David Briggs, Steven Knight, Mike Whitehill, and Michael Davies. *Who Wants to be a Millionaire?*, 1999.