Abcird

Language and Compiler

R. Kent Dybvig August 29, 2022

Abcird language

Overview

- Typescript-like language suitable for reduction to a provable circuit
- Functional core + `assert`
  - External calls may have effects
- Strongly and statically typed
- Data structure sizes fixed at compile time
- Loops (maps and folds) are bounded by (fixed) data-structure sizes

Abcird Language

Datatypes

- Field: natural number limited by the maximum value of a field
- Boolean: true or false
- Bytes[*k*]: vector containing a constant number *k* of bytes
- Vector[*k*, *t*]: vector containing a constant number *k* of values of type *t*
- Struct S {*x*: *t*, ...} : nominally typed collection of named fields {*x*, ...} of types {*t*, …}
- Enums E {x_1, x_2, ...} : nominally typed set of named values E.x_1 = 0, E.x_2 = 1, etc.,
- Opaque[*x*]: a nominally typed value of some unknown type *x*

Abcird Language

Programs

- At present, an Abcird program consists of a single file
- The file must contain zero or more program elements:
  - circuit definitions
  - external declarations
  - structure declarations
  - enum declarations
- At least one circuit should be marked as an export
- Each export and external must be uniquely named

Abcird Language

Circuit definitions

```
[export] circuit C(x: Field, y: Boolean): Vector[3, Field] {
  return [x + 1, x + 2, y ? x + 3 : x - 3];
}
```

Abcird Language

External declarations

```
circuit C(x: Field, y: Boolean): Vector[3, Field];

witness W(): Field;

statement S(bv: Bytes[256]): Field;
```

Abcird Language

Structure declarations

```
structure S {x: Field, y: Array[3, Boolean]}

export circuit q(s: S): Field {
  return s.x;
}

export circuit p(): S {
  return new S(17, [true, true, false]);
}
```

Abcird Language

Enum declarations

```
enum color { red, green, blue }

circuit paint(c: color): Boolean;

export circuit go(): {
  assert paint(color.green) ‘oops’;
}
```

Abcird Language

Module definitions

```
module M {
  statement up(x: Bytes[10]): Boolean;
  statement down(x: Bytes[10]): Boolean;
  export circuit run(b: Boolean, x: Bytes[10]): Boolean {
    if (b) up(x) else down(x);
}

import M;
export {run}
```

Abcird Language

Type parameterization

```
module M[n] {
  statement up(x: Bytes[n]): Boolean;
  statement down(x: Bytes[n]): Boolean;
  export circuit run(b: Boolean, x: Bytes[n]): Boolean {
    if (b) up(x) else down(x);
}

import M[10];
import M[20];
```

Abcird Language

Type parameterization (cont.)

```
module M[n, t] {
  statement up(x: Bytes[n]): t;
  statement down(x: Bytes[n]): t;
  export circuit run(b: Boolean, x: Bytes[n]): t {
    if (b) up(x) else down(x);
}

import M[10, Boolean];
import M[20, Array[5, Field]];
```

Abcird Language

Type parameterization (cont.)

```
module M[n] {
  statement up[t](x: Bytes[n]): t;
  statement down[t](x: Bytes[n]): t;
  export circuit run[t](b: Boolean, x: Bytes[n]): t {
    if (b) up[t](x) else down[t](x);
}
import M[20];

export circuit go(): Void { run[Field](true, 'secret code'); }
```

Abcird Language

Function overloading

```
circuit f(z: Field): Field;
witness f(a: Boolean, b: Field): Boolean;

circuit f(a: Boolean): Field; {
  return f(a, 7) ? f(11) : 0;
}

export circuit go(): Void {
  assert f(false, f(f(true))) ‘oops’;
}
```

Abcird Language

Function overloading (cont.)

```
module M[n] {
  circuit add(a: Field, n: Field): Field { return a + n; }
  export circuit f(v: Vector[n, Field]): Field; {
    return fold add 0 over v;
  }
}
import M[3];
import M[4];
export circuit go(): Field; {
  return f([1, 2, 3]) + f([5, 6, 7, 8]);
}
```

Abcird Language

Statements

expr ;	*expression as statement*
const *id* = *expr* ;	*variable binding*
const *id* : *type* = *expr* ;	*variable binding with explicit type*
assert *expr message* ;	*assert*
return ;	*return (only for functions with return type Void)*
return *expr* ;	*return value of expression*
if (*expr*) *stmt*	*one-armed conditional*
if (*expr*) *stmt* else *stmt*	*two-armed conditional*
{ *expr* . . . *expr* }	*sequence / scope block*

Abcird Language

Expressions (cont.)

map *fun* [over *expr*] . . .	*map over vector*
fold *fun expr* [over *expr*] . . .	*fold over vector*
expr ? *expr* : *expr*	*conditional*
expr || *expr*	*logical or*
expr && *expr*	*logical and*
expr == *expr*	*equivalence*
expr as *type*	*cast*
expr + *expr*	*addition*
expr - expr	*subtraction*
expr * *expr*	*multiplication*

Abcird Language

Expressions (cont.)

! *expr*	*logical not*
expr [*expr*, . . ., *expr*]	*array reference*
expr . *expr*	*struct field or enum value reference*
id . *id*	*enum value reference*
fun (*expr*, . . ., *expr*)	*function (circuit, witness, statement) call*
new *id* (*expr*, . . ., *expr*)	*structure creation*
[*expr*, . . ., *expr*]	*array creation*
(*expr*, . . ., *expr*)	*sequence*
id	*variable reference*
true | false | field | string	*literal: string literal is converted into a Bytes value*

Abcird Language

Functions

id	*reference to named function*
circuit (x : t, . . .) : t { stmt . . . stmt }	*inline function*
(fun)	*parenthesized function*

Abcird Language

Correspondence of Abcird and Typescript/Javascript types

Abcird Type	Typescript Type
Field	bigint i, 0 <= i <= field-max
Boolean	boolean
Vector[n, t]	Array<t> of length n
Bytes[n]	Uint8Array of length n
struct name {x: t, …}	object with properties x … of types t …
enum name { x_1, …, x_n }	number i, 0 <= i < n
Void (return type)	void

Abcird Language

Correspondence of Abcird and Typescript/Javascript types (cont.)

- A struct created by Abcird always has the specified fields and no others
- An object passed to Abcird must have the specified fields, may have others
- Structs are nominally typed within Abcird, structurally typed at the boundary
- A Uint8Array passed to Abcird as a Bytes must have the specified length
- An Array passed to Abcird as a Vector must have the specified length, and each element must have the specified type
- Fields and enums passed to Abcird must fall within the required limits

Abcird Compilation

Current targets

- Typescript
- Zkir

Abcird Compilation

Passes for Typescript target

parse-file

report-unreachable

hoist-local-variables

reject-duplicate-bindings

eliminate-statements

eliminate-boolean-connectives

expand-modules-and-types

infer-types

discard-unused-functions

reject-multiply-defined-exports

recognize-let

print-typescript

Abcird Compilation

Passes for Zkir target

parse-file

report-unreachable

hoist-local-variables

reject-duplicate-bindings

eliminate-statements

eliminate-boolean-connectives

expand-modules-and-types

infer-types

discard-unused-functions

reject-multiply-defined-exports

recognize-let

~~print-typescript~~

Abcird Compilation

Passes for Zkir target (cont.)

replace-enums

unroll-loops

inline-circuits

reduce-to-circuit

flatten-datatypes

optimize-circuit

print-unbound

print-zkir

Abcird Compiler

t.acd

```
witness w(n: Field): Boolean;

circuit foo(a: Field, n: Field): Field {
  assert !(n == 0) "oops";
  const t = a * 16;
  return w(n) ? t + n : t;
}

export circuit bar(a: Vector[2, Field]): Field {
  return fold foo 0 over a;
}
```

Abcird Compiler

t.ts

```
// function w(n: bigint): boolean; // witness
function _w_0(n: bigint): boolean
{
  const t : boolean = w(n);
  if (!(typeof(t) === 'boolean'))
     __abcird_type_error('w', 'return value', 't.acd line 1, char 1', 'Boolean', t);
  return t;
}

function _foo_0(a: bigint, n: bigint): bigint
{
  __abcird_assert(!(n === 0n), "oops");
  const t: bigint = a * 16n;
  if (_w_0(n)) {
    return t + n;
  } else {
    return t;
  }
}
```

Abcird Compiler

t.ts (cont.)

```
function bar(a: Array<bigint>): bigint
{
  if (!(Array.isArray(a) && a.length == 2 && a.every((t) => typeof(t) === 'bigint')))
     __abcird_type_error('bar', 'argument 1', 't.acd line 9, char 1', 'Vector[2, Field]', a);
  return _bar_0(a);
}
function _bar_0(a: Array<bigint>): bigint
{
  return _folder_0(_foo_0, 0n, a);
}

function _folder_0(f: (x: bigint, x1: bigint) => bigint, x: bigint, a0: Array<bigint>): bigint
{
  for (let i = 0; i < 2; i++) x = f(x, a0[i]);
  return x;
}
```

Abcird Compiler

final IR representation of t.acd

```
(witness %w.6 ((%n.37 (tfield))) ((tfield 1)))
(circuit %bar.0 ((%a.39 (tfield)) (%a.38 (tfield))) ((tfield))
  (= %t.41 (== %a.39 0))
  (= %t.43 (select %t.41 0 1))
  (assert %t.43 "oops")
  (= (%t.47) (call %w.6 1 %a.39))
  (= %a.50 (select %t.47 %a.39 0))
  (= %t.52 (== %a.38 0))
  (= %t.54 (select %t.52 0 1))
  (assert %t.54 "oops")
  (= %t.15 (* %a.50 16))
  (= (%t.58) (call %w.6 1 %a.38))
  (= %t.23 (+ %t.15 %a.38))
  (= %t.61 (select %t.58 %t.23 %t.15))
  (%t.61)))
```

Abcird Compiler

t.zkir

```
{ “inputs": 2,
  "gates": [
    [ "load_imm", "le_bytes:00" ],
    [ "test_eq", 0, 2 ],
    [ "load_imm", "le_bytes:01" ],
    [ "cond_select", 3, 2, 4 ],
    [ "assert", 5, "oops" ],
    [ "call_extern", 0, 0 ],
    [ "constrain_to_boolean", 6 ],
    [ "cond_select", 6, 0, 2 ],
    [ "test_eq", 1, 2 ],
    [ "cond_select", 8, 2, 4 ],
    [ "assert", 9, "oops" ],
    [ "load_imm", "le_bytes:10" ],
    [ "pure_gate_out", 7, 10, "i128:1", "i128:0", "i128:0", "i128:0" ],
    [ "call_extern", 0, 1 ],
    [ "constrain_to_boolean", 12 ],
    [ "pure_gate_out", 11, 1, "i128:0", "i128:1", "i128:1", "i128:0" ],
    [ "cond_select", 12, 13, 11 ],
    [ "output", 14 ]
  ]}
```

Abcird Compiler

Running the compiler

Producing typescript:

abcirdc generate-typescript *pathname*.acd # produces *pathname*.ts

Producing zkir:

abcirdc generate-zkir *pathname*.acd # produces *pathname*.zkir

Displaying external symbols:

abcirdc external-bindings *pathname*.acd # prints externals to stdout

To see additional options:

abcirdc —help

Abcird language

Possible TODO

- match typescript type syntax more closely
- import or include
- nominal type aliases, e.g., type feet = Field; type meters = Field
- tuples
- casts from enum and Boolean to Field
- serialize values to Fields or Bytes + more Bytes operations (ref, map, fold)
- support for discriminated unions
- limited function overloading for exports and externals
- see also compiler/TODO

Abcird compiler and toolchain

Possible Todo

- modular Field arithmetic
- print-zkir support for casts
- “header” file defining Abcird types for Typescript, e.g., type Field = bigint
- tie error messages in with VSC
- allow typescript generation, zkir generation, and external printing in same run
- language reference and compiler user’s guide
- always check coverage

Abcird Language and Compiler

For more information

Auto-generated Abcirdc grammar:

github.com/input-output-hk/abcirdc/blob/master/markdown/Program.md

Abcird compiler description:

github.com/input-output-hk/abcirdc/blob/master/compiler.md

FINI

Abcird Compiler

election add_voter: Abcird source code

```
circuit add_voter(pk: Bytes[24]): Void {

  assert !context$eligible_voters$path_of(pk).is_some "Attempted to add a voter twice";

  const sk = private$secret_key();

  const apk = public_key(sk);

  assert (apk == public$authority()) "Attempted to add a voter without authorization";

  // We probably want to support enums here?

  assert (public$state() == PublicState.setup) "Attempted to add a voter after setup
phase";

  public$eligible_voters$add(pk);

}
```

Abcird Compiler

election add_voter: Typescript output

```
function add_voter(pk: Uint8Array): void
{
  if (!(pk.buffer instanceof ArrayBuffer && pk.BYTES_PER_ELEMENT == 1 && pk.length == 24))
pk); __abcird_type_error('add_voter', 'argument 1', 'examples/election.acd line 122, char 1', 'Bytes[24]',
  return _add_voter_0(pk);
}

function _add_voter_0(pk: Uint8Array): void
{
  __abcird_assert(!_context$eligible_voters$path_of_0(pk).is_some, "Attempted to add a voter twice");
  const sk: Uint8Array = _private$secret_key_0();
  const apk: Uint8Array = _public_key_0(sk);
  __abcird_assert(_equal_6(apk, _public$authority_0()), "Attempted to add a voter without authorization");
  __abcird_assert(_public$state_0() === 0, "Attempted to add a voter after setup phase");
  _public$eligible_voters$add_0(pk);
  return undefined;
}
```

Abcird Compiler

election add_voter: Zkir output


```
{
  "inputs": 1,
  "gates": [
    [ "constrain_bits", 0, 192 ],
    [ "load_imm", "le_bytes:01" ],
    [ "call_extern", 20, 0 ],
    [ "constrain_to_boolean", 2 ],
    [ "constrain_bits", 3, 192 ],
    [ "constrain_to_boolean", 5 ],
    [ "constrain_to_boolean", 7 ],
    [ "constrain_to_boolean", 9 ],
    [ "constrain_to_boolean", 11 ],
    [ "constrain_to_boolean", 13 ],
    [ "constrain_to_boolean", 15 ],
    [ "constrain_to_boolean", 17 ],
    [ "constrain_to_boolean", 19 ],
    [ "constrain_to_boolean", 21 ],
    [ "constrain_to_boolean", 23 ],
    [ "load_imm", "le_bytes:00" ],
    [ "cond_select", 2, 24, 1 ],
    [ "assert", 25, "Attempted to add a voter twice" ],
    [ "call_extern", 15 ],
    [ "constrain_bits", 26, 192 ],
    [ "load_imm", "le_bytes:6C617265733A656C656374696F6E3A706B3A" ],
    [ "poseidon_compress", 27, 26 ],
    [ "mod_power_of_two", 28, 192 ],
    [ "call_extern", 0 ],
    [ "declare_pub_input", 1 ],
    [ "declare_pub_input", 30 ],
    [ "test_eq", 29, 30 ],
    [ "assert", 31, "Attempted to add a voter without authorization" ],
    [ "call_extern", 3 ],
    [ "declare_pub_input", 1 ],
    [ "declare_pub_input", 32 ],
    [ "test_eq", 32, 24 ],
    [ "assert", 33, "Attempted to add a voter after setup phase" ],
    [ "call_extern", 9, 0 ],
    [ "declare_pub_input", 1 ],
    [ "declare_pub_input", 0 ]
  ]
}
```