darlang Documentation

Release 0.0.1

Andrew Comminos

Jun 14, 2018

Contents:

CHAPTER 1

Basics

This document is very incomplete, perhaps some meaning can be derived from it.

1.1 Declarations

```
func(arg_1, arg_2, ..., arg_n) -> expr
```

All functions in darlang are implicitly typed. Both the argument types and the return value are derived through unification by the callee and implementation, respectively.

1.2 Binding

```
id | expr; ...
```

Binding is the process of assigning an expression to a new identifier in the current scope. All references to the id on the left hand side of the expression will be substituted with the associated expression.

1.3 Conditionals

```
{
  case_1 : expr_1;
  case_2 : expr_2;
  ...
  case_n : expr_n;
       * : expr_wildcard;
}
```

Darlang uses a singular guard-like construct to provide control flow.

Cases are evaluated from top to bottom- the first true case expression will cause the associated expression to be returned.

All branching constructs must have a "wildcard" case, to be executed when no cases are satisfied.

1.4 Example

Let's get started with a program that computes the classic *Euclidean algorithm*.

```
euclid(a, b) ->
  {
    is(b, 0) : a;
           * : euclid(b, mod(a, b))
  }

main() -> euclid(15, 10)
```

CHAPTER 2

Types

2.1 Primitives

2.2 Product types

2.3 Disjoint unions

CHAPTER 3

Specialization

Specialization is the mechanism through which darlang provides parametric function polymorphism. It can be seen as an analogue to implicitly-generated C++ templates, implementation-wise.

3.1 Specialization at work

Suppose the existence of an under-constrained function *tuplify*;

```
tuplify(a) -> (a, a)
```

Such a function can be observed to impose no constraints on the type of the argument *a*. Since the set of potential types for *tuplify* is infinite, the darlang compiler skips code generation for it in the absence of invocations.

Consider, however, if we had the following invocations:

```
main() ->
  ds | tuplify("hello");
  is | tuplify(5);
  0
```

Two **specializations** of *tuplify* are generated by the compiler- one with signature *string -> (string, string)*, and one with signature *i64 -> (i64, i64)*. This reduces the need for runtime typing logic, reducing instruction count at the expense of code size.

Note: Specializations of a function are populated by the darlang compiler by traversing the call graph starting from top-level exports, which possess known types, and enumerating all required implementations.

IR is generated for each possible specialization of a function, each of which possess a unique symbol name derived from the function's name and its solved argument types.

CHAPTER 4

Quirks

- All values of a recursive type are enforced to be heap allocated.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search