1 The Binomial No-Arbitrage Pricing Model

1.1 One-Period Binomial Model

Definition 1.1. Time zero The beginning of the time period.

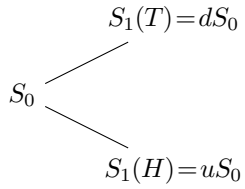
Definition 1.2. Time one The end of the time period.

Lemma 1.1. *Let S_0 be the value of the stock S at time zero.*

Lemma 1.2. *Let $S_1(H)$ be the value of the stock S at time one in the "heads" case.*

Lemma 1.3. *Let $S_1(T)$ be the value of the stock S at time one in the "tails" case.*

Lemma 1.4. *Let p be the probability of S having value $S_1(H)$ at time one, and $q=1-p$ be the probability of S having value $S_1(T)$ at time one.*



Lemma 1.5. *Let u be the up factor, and d be the down factor.*

$$u=\frac{S_1(H)}{S_0},d=\frac{S_1(T)}{S_0}$$

1a ⟨*factor* 1a⟩≡

```
typedef double (*factor)(void *context, double s_time_zero, double s_time_one);
double one_period_factor(void *context, double s_time_zero, double s_time_one) {
  return s_time_one / s_time_zero;
}

typedef struct {
  factor up;
  factor down;
} factor_pair;

factor_pair default_one_period_factor = { .up = one_period_factor, .down = one_period_factor };
```

Trivial example with a simple OO tuple...

1b ⟨*one-period.h* 1b⟩≡

```
#pragma once
#include "pair.h"
```

1c ⟨*one-period.c* 1c⟩≡

```
#include "one_period.h"
#include <stdio.h>

int main() {
  puts("Hello World!");
  Pair *p = pair_new((void *)1, (void *)2);
  printf("%d\n", (int)pair_first(p));
}
```

2 Pair

A pair is a two-element immutable tuple. We start by forward declaring the pair struct in the header. This way we can hide the implementation of the tuple.

2a ⟨*typedef-pair-h* 2a⟩≡ (2g)
```
typedef struct _Pair Pair;
```

We also need to declare the struct in the implementation file, but the details can only be accessed from the implementation file.

2b ⟨*typedef-pair-c* 2b⟩≡ (2h)
```
struct _Pair {
  void *first, *second;
};
```

Next we need a way to create new pairs given two items. We accept **void *** so we can hold elements of any type, including NULL to simulate an empty pair.

2c ⟨*pair-new-h* 2c⟩≡ (2g)
```
Pair *pair_new(void *, void *);
```

And the corresponding pair implementation. In this instance we are using the built-in malloc for simplicity.

2d ⟨*pair-new-c* 2d⟩≡ (2h)
```
Pair *pair_new(void *first, void *second) {
  Pair *t = malloc(sizeof(Pair));
  t->first = first;
  t->second = second;
  return t;
}
```

We also need a couple of accessors so we can get at the elements of the pair. Here I decided to allow the mutation of the items from the pair by default, a truly immutable pair wouldn't allow the mutation of the elements inside the pair either.

2e ⟨*pair-get-h* 2e⟩≡ (2g)
```
void *pair_first(Pair *);
void *pair_second(Pair *);
```

To implement we just return the references to the internal pair items.

2f ⟨*pair-get-c* 2f⟩≡ (2h)
```
void *pair_first(Pair *self) {
  return self->first;
}
void *pair_second(Pair *self) {
  return self->second;
}
```

Finally we have the full pair implementation.

2g ⟨*pair.h* 2g⟩≡
```
#pragma once
⟨typedef-pair-h 2a⟩
⟨pair-new-h 2c⟩
⟨pair-get-h 2e⟩
```

2h ⟨*pair.c* 2h⟩≡
```
#include "pair.h"
#include <stdlib.h>
⟨typedef-pair-c 2b⟩
⟨pair-new-c 2d⟩
⟨pair-get-c 2f⟩
```