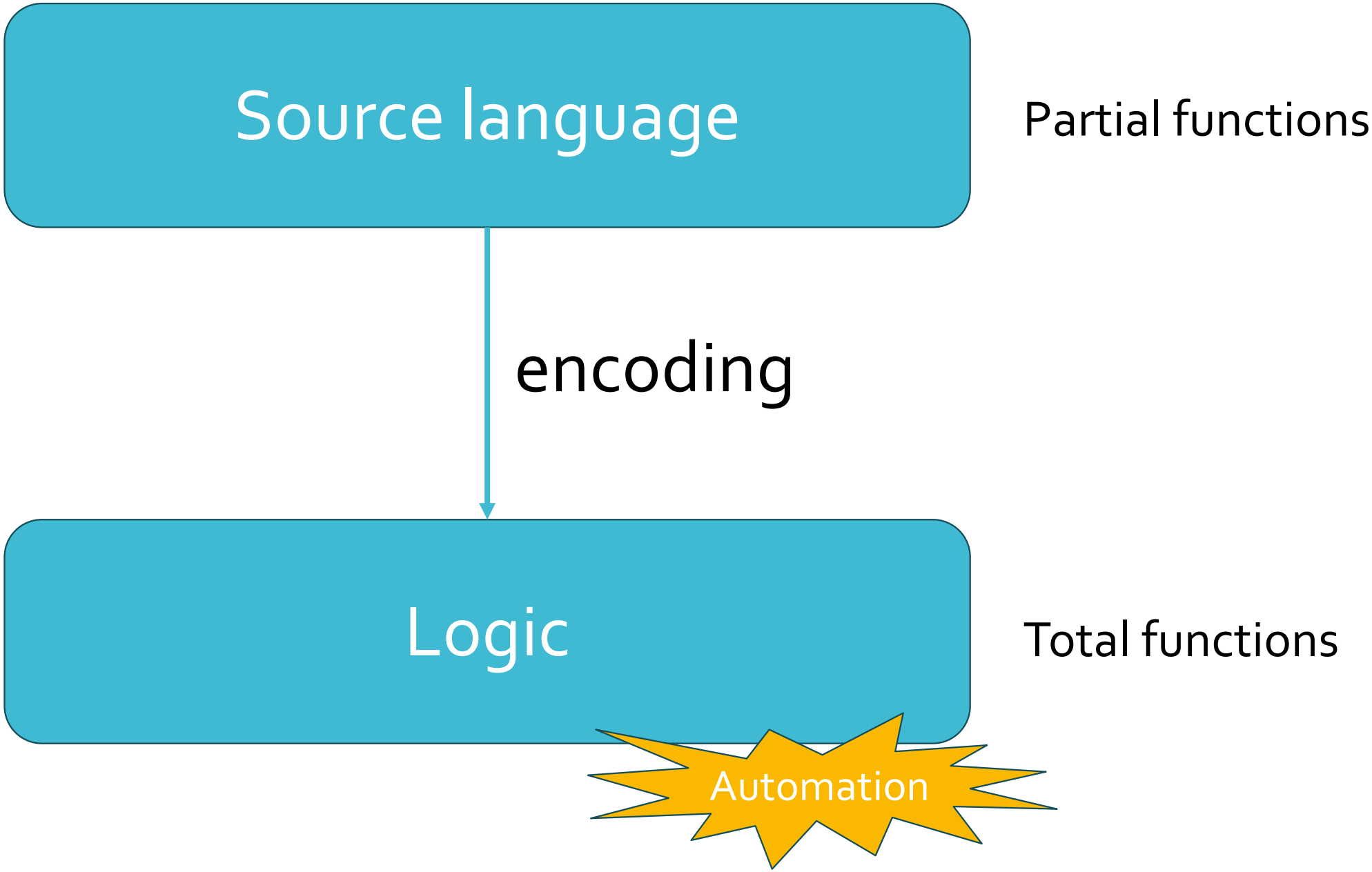
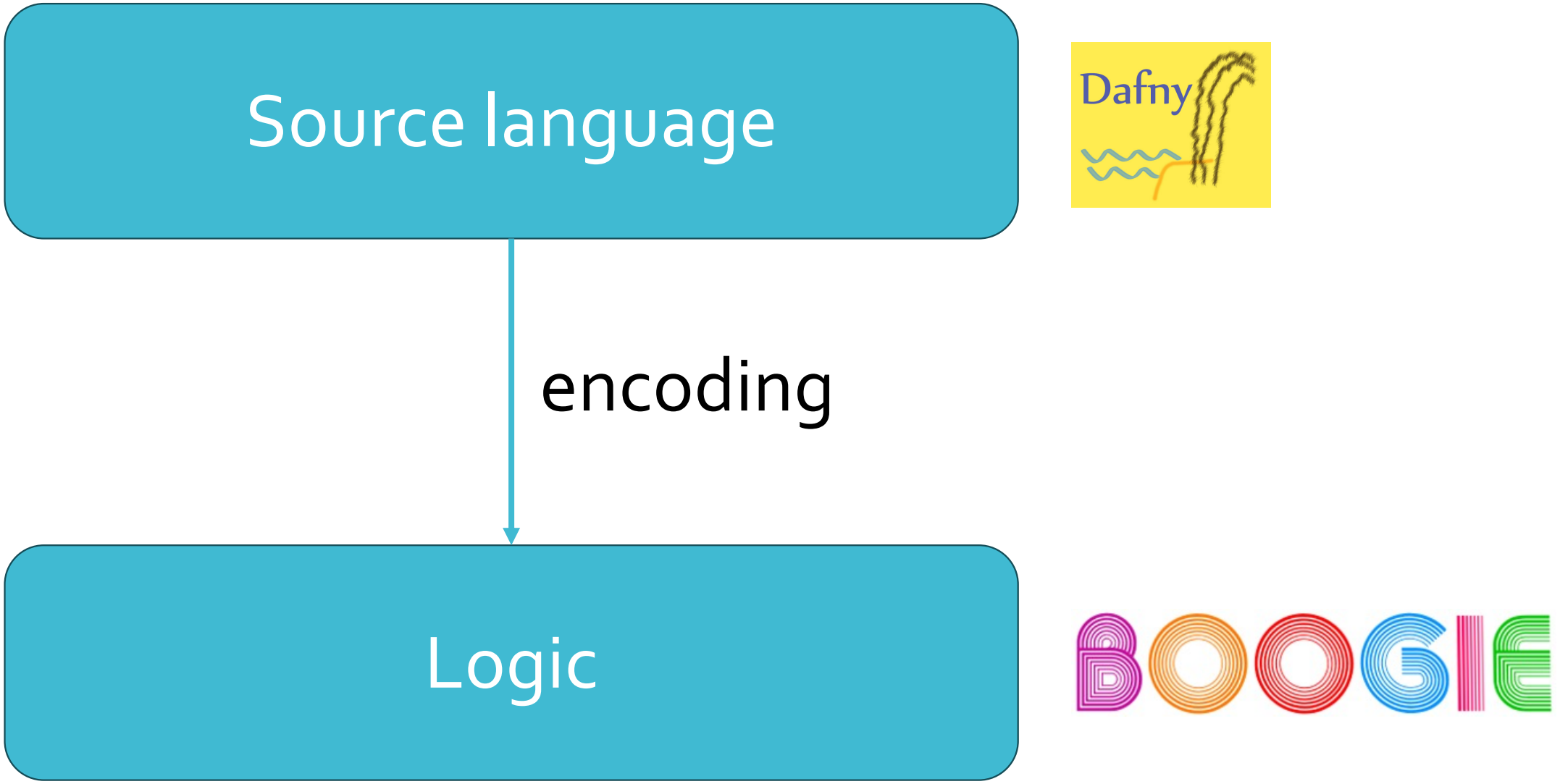
Encoding partial functions into a total logic

K. Rustan M. Leino

9 May 2024
IFIP WG 2.3 meeting 68
Princeton, NJ, USA


Source language
Partial functions
encoding
Logic
Total functions
Automation


Source language
encoding
Logic
Dafny
BOOGIE

Well-formedness

Source `program`

```
x := y / z
```

Correctness verification and well-formedness checks

```
assert z != 0
x := Div(y, z)
```

Total functions

Source program

```
function F(x: int): int {
  if x < 20 then 18 else 100/x
}
```

Well-formedness checks

```
if x < 20 {
} else {
  assert x != 0
}
```

Definition axiom for F

```
forall x: int ::
  F(x) == if x < 20 then 18 else Div(100, x)
```

Termination

Source program

```
function F(x: int): int {
  if x > 0 then 2 + F(x-1) else 9
}
```

Well-formedness checks

```
if x > 0 {
  assert x-1 < x && 0 <= x
} else {
}
```

Definition axiom

```
forall x: int ::
  F(x) == if x > 0 then 2 + F(x-1) else 9
```

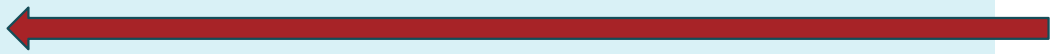
When is definition axiom available?

Source program

```
function F(x: int): int {
  if x > 0 then 1 + 100 / F(x-1) else 9
}
```

Well-formedness checks

```
if x > 0 {
  assert x-1 < x && 0 <= x
  assert F(x-1) != 0
} else {
}
```



Need information about F here

Definition axiom

```
forall x: int ::
  F(x) == if x > 0 then 1 + 100 / F(x-1) else 9
```

Partial functions

Source program, function definition

```
function F(x: int): int
  requires x != 0
{
  100 / x
}
```

Source program, function call

```
F(a + b)
```

Well-formedness checks, function body

```
assume x != 0
assert x != 0
```

Well-formedness checks, caller

```
assert a + b != 0
```

Definition axiom

```
forall x: int ::
  x != 0
  ==>
  F(x) == 100/x
```

Axiom must use precondition

Source program, function definition

```
function F(x: int): int
  requires 0 <= x
{
  if x == -3 then 1 + F(x) else 5
}
```

Well-formedness checks

```
assume 0 <= x
if x == -3 {
  assert x < x && 0 <= x
} else {
}
```

Definition axiom

```
forall x: int ::
  F(x) == if x == -3 then 1 + F(x) else 5
```

Double proof problem

Source program, function definition

```
function F(x: int): int
  requires …complicated…x…
{
  E
}
```

Source program, function call

```
c := F(a + b)
assert c == E
```

Well-formedness checks

```
assert …complicated…a+b…
c := F(a + b)
assert c == E
```

Definition axiom

```
forall x: int ::
  …complicated…x…
  ==>
  F(x) == E
```

CanCall predicates

Source program, function definition

```
function F(x: int): int
  requires …complicated…x…
{
  E
}
```

Source program, function call

```
c := F(a + b)
assert c == E
```

Well-formedness checks

```
assert …complicated…a+b…
assume CanCallF(a + b)
c := F(a + b)
assert c == E
```

Definition axiom

```
forall x: int ::
  CanCallF(x)
  ==>
  F(x) == E
```

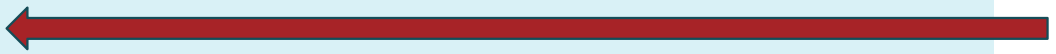
When is axiom available? (again)

Source program

```
function F(x: int): int {
  if x > 0 then 1 + (100 / F(x-1))^2 else 9
}
```

Well-formedness checks

```
if x > 0 {
  assert x-1 < x && 0 <= x
  assume CanCallF(x-1)
  assert F(x-1) != 0
} else {
}
```



Information about this call to F follows from the CanCallF

Definition axiom

```
forall x: int ::
  CanCallF(x)
==>
  F(x) == if x > 0 then 1 + (100 / F(x-1))^2 else 9
```

Another example use of CanCall

Source program, definitions

```
function F(x: int): int {
  x + 3
}

method M(x: int) returns (r: int)
  ensures r == F(x)
```

Source program, method call

```
c := M(a + b)
assert c == a + b + 3
```

Definition axiom

```
forall x: int ::
  CanCallF(x)
  ==>
  F(x) == x + 3
```

Correctness checks

```
{ // call to M:
  var x := a + b
  var r := *
  assume CanCallF(x)
  assume r == F(x)
  c := r
}
assert c == a + b + 3
```

Using CanCall in definition axiom (?)

Source program

```
function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Well-formedness checks

```
assume 0 <= n
if n == 0 {
} else {
  assert 0 <= n-1
  assert n-1 < n && 0 <= n
  assume CanCallTriangle(n-1)
}
```

Definition axiom

```
forall n: int ::
  CanCallTriangle(n)
  ==>
  Triangle(n) == if n == 0 then 0 else n + Triangle(n-1)
```

Source program, example caller

```
c := Triangle(12)
assert c == 78
```

Correctness checks

```
assert 0 <= 12
assume CanCallTriangle(12)
c := Triangle(12)
assert c == 78
```

Using CanCall in definition axiom (?)

Source program

```
function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Well-formedness checks

```
assume 0 <= n
if n == 0 {
} else {
  assert 0 <= n-1
  assert n-1 < n && 0 <= n
  assume CanCallTriangle(n-1)
}
```

Definition axiom

```
forall n: int ::
  CanCallTriangle(n)
  ==>
  (n != 0 ==> CanCallTriangle(n-1)) &&
  Triangle(n) == if n == 0 then 0 else n + Triangle(n-1)
```

Source program, example caller

```
c := Triangle(12)
assert c == 78
```

Correctness checks

```
assert 0 <= 12
assume CanCallTriangle(12)
c := Triangle(12)
assert c == 78
```

Using CanCall in definition axiom (?)

Source program

```
function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Well-formedness checks

```
assume 0 <= n
if n == 0 {
} else {
  assert 0 <= n-1
  assert n-1 < n && 0 <= n
  assume CanCallTriangle(n-1)
}
```

Definition axiom

```
forall n: int ::
  CanCallTriangle(n)
  ==>
  (n != 0 ==> CanCallTriangle(n-1)) &&
  Triangle(n) == if n == 0 then 0 else n + Triangle(n-1)
```

Source program, example caller

```
c := Triangle(a*a)
assert c == 77
```

Correctness checks

```
assert 0 <= a*a
assume CanCallTriangle(a*a)
c := Triangle(a*a)
assert c == 77
```

Use of quantifiers

Some axiom

```
forall x: int ::
  0 <= F(x) < 2 * G(x)
```

Source code, example 0

```
if G(a) == 10 {
  assert F(a) < 20
}
```

Source code, example 1

```
assert 1 <= G(a)
```

Quantifiers and automation: Matching patterns

Some axiom

```
forall x: int ::
  0 <= F(x) < 2 * G(x)
```

Source code, example 0

```
if G(a) == 10 {
  assert F(a) < 20
}
```

Source code, example 1

```
assert 1 <= G(a)
```

Matching pattern for definition axiom

Source program

```
function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Source program, example caller

```
c := Triangle(a*a)
assert c == 77
```

Correctness checks

```
assert 0 <= a*a
assume CanCallTriangle(a*a)
c := Triangle(a*a)
assert c == 77
```

Definition axiom

```
forall n: int ::
  CanCallTriangle(n)
  ==>
  Triangle(n) == if n == 0 then 0 else n + Triangle(n-1)
```

Matching pattern for definition axiom

Source program

```
function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Source program, example caller

```
c := Triangle(a*a)
assert c == 77
```

Correctness checks

```
assert 0 <= a*a
assume CanCallTriangle(a*a)
c := Triangle(a*a)
assert c == 77
```

Definition axiom

```
forall n: int ::
  CanCallTriangle(n)
  ==>
  Triangle(n) == if n == 0 then 0 else n + Triangle(n-1)
```

Matching pattern for definition axiom

Source program

```
function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Source program, example caller

```
c := Triangle(a*a)
assert c == 77
```

Correctness checks

```
assert 0 <= a*a
assume CanCallTriangle(a*a)
c := Triangle(a*a)
assert c == 77
```

Definition axiom

```
forall n: int ::
  CanCallTriangle(n)
  ==>
  (n != 0 ==> CanCallTriangle(n-1)) &&
  Triangle(n) == if n == 0 then 0 else n + Triangle(n-1)
```

Fuel

Source program

```
function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Definition axiom

```
forall n: int, ¥: Fuel ::
  CanCallTriangle(n, S(¥))
  ==>
  (n != 0 ==> CanCallTriangle(n-1, ¥)) &&
  Triangle(n) ==
    if n == 0 then 0 else n + Triangle(n-1)
```

Source program, example caller

```
c := Triangle(12)
d := Triangle(11)
assert c == d + 12
```

Correctness checks

```
assert 0 <= 12
assume CanCallTriangle(12, S(Z))
c := Triangle(12)
assert 0 <= 11
assume CanCallTriangle(11, S(Z))
d := Triangle(11)
assert c == d + 12
```

Fuel 2

Source program

```
function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Definition axiom

```
forall n: int, ¥: Fuel ::
  CanCallTriangle(n, S(¥))
  ==>
  (n != 0 ==> CanCallTriangle(n-1, ¥)) &&
  Triangle(n) ==
    if n == 0 then 0 else n + Triangle(n-1)
```

Source program, example caller

```
c := Triangle(12)
d := Triangle(10)
assert c == d + 23
```

Correctness checks

```
assert 0 <= 12
assume CanCallTriangle(12, S(S(Z)))
c := Triangle(12)
assert 0 <= 10
assume CanCallTriangle(10, S(S(Z)))
d := Triangle(10)
assert c == d + 23
```

Opaque functions

Source program

```
opaque function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Definition axiom

```
forall n: int, ¥: Fuel ::
  CanCallTriangle(n, S(¥), true)
  ==>
  (n != 0 ==> CanCallTriangle(n-1, ¥, true))
&&
  Triangle(n) ==
    if n == 0 then 0 else n + Triangle(n-1)
```

```
const TriangleAvailable: bool
```

Source program, example caller

```
c := Triangle(12)
d := Triangle(11)
assert c == d + 12
```

Correctness checks

```
assert 0 <= 12
assume CanCallTriangle(12, S(Z),
         TriangleAvailable)
c := Triangle(12)
assert 0 <= 11
assume CanCallTriangle(11, S(Z),
         TriangleAvailable)
d := Triangle(11)
assert c == d + 12
```

Encoding due to Mikaël Mayer

Revelations

Source program

```
opaque function Triangle(n: int): int
  requires 0 <= n
{
  if n == 0 then 0 else n + Triangle(n-1)
}
```

Definition axiom

```
forall n: int, ¥: Fuel ::
  CanCallTriangle(n, S(¥), true)
  ==>
  (n != 0 ==> CanCallTriangle(n-1, ¥, true))
&&
  Triangle(n) ==
    if n == 0 then 0 else n + Triangle(n-1)
```

```
const TriangleAvailable: bool
```

Source program, example caller

```
c := Triangle(12)
d := Triangle(11)
reveal Triangle
assert c == d + 12
```

Correctness checks

```
assert 0 <= 12
assume CanCallTriangle(12, S(Z),
         TriangleAvailable)
c := Triangle(12)
assert 0 <= 11
assume CanCallTriangle(11, S(Z),
         TriangleAvailable)
d := Triangle(11)
assume TriangleAvailable == true
assert c == d + 12
```

Encoding due to Mikaël Mayer

Questions about fuel

- Should fuel be used at all?
  That is, should fuel be set to 1 at call sites and not used in definition axioms?
- Should fuel be used with non-recursive functions?
- How should fuel propagate from one function to another?

```
function F(x: X): int {
  … G(…) … F(…) …
}
```

```
forall x: X, ¥: Fuel ::
  CanCallF(x, S(¥))
  ==>
  (… CanCallG(…, S(¥)) … CanCallF(…, ¥) …) &&
  F(x) == … G(…) … F(…) …
```

Encoding into just logic?

- As shown (and currently used in all Boogie front ends), everything is encoded as logical formulas (plus matching patterns)
- Would it instead be possible (and desirable) to do the encoding as formulas + directives?

Notes about meta proof of soundness

- Meta proof is on induction over context height
- Definition of CanCallF(x) is that it implies
  - Current context height implies decreases clause of F(x)
  - Precondition of F(x)
- Fuel argument is irrelevant to logic (but limits instantiation)
- Opaqueness is irrelevant (but limits instantiation)