Formalization of Termination of Recursive Functions

César A. Muñoz[1]

NASA Langley Research Center

International School of Rewriting
July 30 - August 3, 2018
Pontificia Universidad Javeriana de Cali, Colombia

[1]The formalization presented in this lecture was done in collaboration with Ariane A. Almeida (U. of Brasilia), Mauricio Ayala-Rincón (U. of Brasilia), Andrea B. Avelar (U. of Brasilia), Aaron Dutle (NASA), Thiago M. Ferreira (U. of Brasilia), Mariano Moscato (NIA), and Anthony Narkawicz (NASA).

Termination

- In computer science, *termination* is the is the quintessential example of a property that is undecidable.
- In 1939, Turing proved that it is impossible to construct an algorithm that decides whether or not another algorithm terminates on a given input [Tur37].
- Turing's proof applies to algorithms written as Turing machines, but the proof extends to other formalisms for expressing computations: λ-calculus, rewriting systems, computer programs.

Termination in Rewriting Systems

- Termination is a fundamental property of rewriting systems, e.g., confluence is decidable in terminating systems.
- Termination is undecidable even when a rewrite system consists of only one rule.
- Several syntactic and semantic techniques are available to prove termination of rewriting systems.
- This lecture focuses on termination of recursive functions specified in proof assistants.

Termination in Proof Assistants

- Termination is a meta-theoretical property in most interactive theorem provers, e.g.,
  - Termination is guaranteed for well-typed functions.
  - Termination is guaranteed for functions satisfying some constraints.
  - Termination is guaranteed for functions satisfying some semantic conditions.
- Once a definition of a function f is accepted by a proof assistant, the statement "For every value a the computation of $f(a)$ terminates" is assumed to hold.
- Our proof assistant of choice: SRI's Prototype Verification System (PVS).[2]

[2] `https://pvs.csl.sri.com`.

PVS

- PVS is an interactive theorem prover based on classical higher-order logic.
- PVS provides a strongly-typed specification language that supports predicate sub-typing, dependent types, inductive data types, parametric theories, etc.
- PVS is extensively used at NASA in the verification of safety-critical and mission-critical systems.[3]
- The NASA PVS Library consists of more than 20K lemmas (including the formalization presented in this lecture).[4]

[3] `https://shemesh.larc.nasa.gov/people/cam/FM`.

[4] `https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/`.

Recursive Functions in PVS

```
factorial(n:nat): RECURSIVE nat =
  IF n = 0 THEN 1
  ELSE n*factorial(n-1)
  ENDIF
MEASURE M BY R
```

where M is a measure, i.e., a function from the domain of *factorial* into a type T, and R is a well-founded relation on T, e.g.,

- $M =$ `LAMBDA(n:nat):n`,
- $R =$ `LAMBDA(n,m:nat):n < m`.

Recursive Functions in PVS

The well-founded relation R is usually "$<$" on natural numbers. In this case, it can be written

```
factorial(n:nat): RECURSIVE nat =
  IF n = 0 THEN 1
  ELSE n*factorial(n-1)
  ENDIF
MEASURE n
```

PVS produces the following Termination Correctness Condition:[5]

```
factorial_TCC2: OBLIGATION
  FORALL (n: nat): n /= 0 IMPLIES n - 1 < n
```

[5] This TCC corresponds to the so called Turing Termination Criterion.

Recursive Functions in PVS

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
MEASURE ?
```

Recursive Functions in PVS

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
MEASURE lex2(m,n)
```

In this case,

- T, the range of `lex2` is `ordinal`.
- R, the well-founded relation, is $<$ on ordinals.

Termination Correctness Conditions for *gcd*

```
gcd_TCC2: OBLIGATION
  FORALL (m, n: nat):
    n >= m AND NOT m = 0 AND NOT n = 0 IMPLIES
    lex2(m, n - m) < lex2(m, n)

gcd_TCC3: OBLIGATION
  FORALL (m, n: nat):
    NOT n >= m AND NOT m = 0 AND NOT n = 0 IMPLIES
    lex2(n, m) < lex2(m, n)
```

- `factorial_TCC2`, `gcd_TCC2`, and `gcd_TCC3` are automatically discharged by PVS.
- In general, the user has to provide the measure, the well-founded relation, and prove the TCCs.

Research Objectives

- Formalize in PVS different termination criteria and prove their equivalence, e.g., Turing termination [Tur89], size change principle [LJB01, TG03, KST$^+$11], calling context graphs [MV06], matrix-weighted graphs [Ave15], and dependency pairs [Art96, YSTK16, AG00].
- Use these criteria to specify terminating recursive functions in PVS and automatically discharge Termination Correctness Conditions.
- Study meta-theoretical properties related to termination and computability of PVS recursive functions.

PVS0: A Simple Computational Model

- PVS0 is a deep embedding of first-order PVS functions of type $T \to T$, where T is a parametric type.
- PVS0 functional expressions consists of
  - Constant values of type T.
  - A variable symbol of type T.
  - Unary and binary "built-in" operators.
  - If-then-else expressions.
  - Recursive calls.
- PVS0 is simple, but not minimal. In particular, it enables the use of arbitrary PVS functions of types $T \to T$ and $T \times T \to T$ as built-in atomic operators.

PVS0 Expressions

- PVS0 expressions e have the following form:

  $\texttt{cnst}(v) \mid \texttt{vr} \mid op1(n, e) \mid op2(n, e, e) \mid rec(e) \mid \texttt{ite}(e, e, e)$,

  where v is a value of type T and $n \in \mathbb{N}$.
- In PVS, it is defined using the following abstract data type.

```
PVS0Expr[T:TYPE+] : DATATYPE
BEGIN
  cnst(get_val:T) : cnst?
  vr : vr?
  op1(get_op:nat,get_arg:PVS0Expr) : op1?
  op2(get_op:nat,get_arg1,get_arg2:PVS0Expr) : op2?
  rec(get_arg:PVS0Expr) : rec?
  ite(get_cond,get_if,get_else:PVS0Expr) : ite?
END PVS0Expr
```

PVS0 Programs

Given a concrete type Val, which instantiates T, a PVS0 program with values in Val is a 4-tuple of the form $(O_1, O_2, \bot, e)$, where

- O_1 is a list of PVS functions of type $Val \to Val$, where $O_1(i)$, i.e., the i-th element of the list O_1, interprets the unary operator indexed by i in the constructor op1,

PVS0 Programs

Given a concrete type Val, which instantiates T, a PVS0 program with values in Val is a 4-tuple of the form $(O_1, O_2, \bot, e)$, where

- O_1 is a list of PVS functions of type $Val \to Val$, where $O_1(i)$, i.e., the i-th element of the list O_1, interprets the unary operator indexed by i in the constructor op1,
- O_2 is a list of PVS functions of type $Val \times Val \to Val$, where $O_2(i)$, i.e., the i-th element of the list O_2, interprets the binary operator indexed by i in the constructor op2,

PVS0 Programs

Given a concrete type Val, which instantiates T, a PVS0 program with values in Val is a 4-tuple of the form $(O_1, O_2, \bot, e)$, where

- O_1 is a list of PVS functions of type $Val \to Val$, where $O_1(i)$, i.e., the i-th element of the list O_1, interprets the unary operator indexed by i in the constructor `op1`,
- O_2 is a list of PVS functions of type $Val \times Val \to Val$, where $O_2(i)$, i.e., the i-th element of the list O_2, interprets the binary operator indexed by i in the constructor `op2`,
- $\bot$ is a constant of type Val representing the Boolean value false in the conditional construction `ite`, and

PVS0 Programs

Given a concrete type Val, which instantiates T, a PVS0 program with values in Val is a 4-tuple of the form $(O_1, O_2, \bot, e)$, where

- O_1 is a list of PVS functions of type $Val \to Val$, where $O_1(i)$, i.e., the i-th element of the list O_1, interprets the unary operator indexed by i in the constructor `op1`,
- O_2 is a list of PVS functions of type $Val \times Val \to Val$, where $O_2(i)$, i.e., the i-th element of the list O_2, interprets the binary operator indexed by i in the constructor `op2`,
- $\bot$ is a constant of type Val representing the Boolean value false in the conditional construction `ite`, and
- e is a `PVS0Expr`$[Val]$, which is the syntactic representation of the program itself.

Example 1: Factorial

```
factorial(n:nat): RECURSIVE nat =
  IF n /= 0 THEN n*factorial(n-1)
  ELSE 1
  ENDIF
```

Let $f \in \texttt{PVSO}[Val] = (O_1, O_2, \bot, e_f)$, where

- $Val = \texttt{nat}$

Example 1: Factorial

```
factorial(n:nat): RECURSIVE nat =
  IF n /= 0 THEN n*factorial(n-1)
  ELSE 1
  ENDIF
```

Let $f \in \texttt{PVS0}[Val] = (O_1, O_2, \bot, e_f)$, where

- $Val = \texttt{nat}$
- $\bot = 0$

Example 1: Factorial

```
factorial(n:nat): RECURSIVE nat =
  IF n /= 0 THEN n*factorial(n-1)
  ELSE 1
  ENDIF
```

Let $f \in \texttt{PVS0}[Val] = (O_1, O_2, \bot, e_f)$, where

- $Val = \texttt{nat}$
- $\bot = 0$
- $\text{op1}(0, n) = \max(0, n - 1)$

Example 1: Factorial

```
factorial(n:nat): RECURSIVE nat =
  IF n /= 0 THEN n*factorial(n-1)
  ELSE 1
  ENDIF
```

Let $f \in \texttt{PVS0}[Val] = (O_1, O_2, \bot, e_f)$, where

- $Val = \texttt{nat}$
- $\bot = 0$
- $\text{op1}(0, n) = \max(0, n - 1)$
- $\text{op2}(0, n, m) = n * m$

Example 1: Factorial

```
factorial(n:nat): RECURSIVE nat =
  IF n /= 0 THEN n*factorial(n-1)
  ELSE 1
  ENDIF
```

Let $f \in \texttt{PVS0}[Val] = (O_1, O_2, \bot, e_f)$, where

- $Val = \texttt{nat}$
- $\bot = 0$
- $\text{op1}(0, n) = \max(0, n - 1)$
- $\text{op2}(0, n, m) = n * m$
- $e_f = \texttt{ite(vr,op2(0,vr,rec(op1(0,vr))),cnst(1))}$

Example 2: GCD

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
```

Let $g \in \texttt{PVS0}[\mathit{Val}] = (O_1, \texttt{null}, \bot, e_g)$, where

- $\mathit{Val} = [\texttt{nat}, \texttt{nat}]$

Example 2: GCD

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
```

Let $g \in \texttt{PVS0}[\mathit{Val}] = (O_1, \texttt{null}, \bot, e_g)$, where

- $\mathit{Val} = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$

Example 2: GCD

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
```

Let $g \in \texttt{PVS0}[\mathit{Val}] = (O_1, \texttt{null}, \bot, e_g)$, where

- $\mathit{Val} = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\texttt{op1}(0,(m,n)) = \texttt{IF } m = 0 \texttt{ OR } n = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$

Example 2: GCD

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
```

Let $g \in \texttt{PVS0}[\mathit{Val}] = (O_1, \texttt{null}, \bot, e_g)$, where

- $\mathit{Val} = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\texttt{op1}(0,(m,n)) = \texttt{IF } m = 0 \texttt{ OR } n = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$
- $\texttt{op1}(1,(m,n)) = \texttt{IF } n >= m \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$

Example 2: GCD

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
```

Let $g \in \texttt{PVS0}[\mathit{Val}] = (O_1, \texttt{null}, \bot, e_g)$, where

- $\mathit{Val} = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\texttt{op1}(0,(m,n)) = \texttt{IF } m = 0 \texttt{ OR } n = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$
- $\texttt{op1}(1,(m,n)) = \texttt{IF } n >= m \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$
- $\texttt{op1}(2,(m,n)) = (m+n,0)$

Example 2: GCD

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
```

Let $g \in \texttt{PVS0}[\mathit{Val}] = (O_1, \texttt{null}, \bot, e_g)$, where

- $\mathit{Val} = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\texttt{op1}(0,(m,n)) = \texttt{IF } m = 0 \texttt{ OR } n = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$
- $\texttt{op1}(1,(m,n)) = \texttt{IF } n >= m \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$
- $\texttt{op1}(2,(m,n)) = (m+n, 0)$
- $\texttt{op1}(3,(m,n)) = (m, \max(0, n-m))$

Example 2: GCD

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
```

Let $g \in \texttt{PVS0}[Val] = (O_1, \texttt{null}, \bot, e_g)$, where

- $Val = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\texttt{op1}(0,(m,n)) = \texttt{IF } m = 0 \texttt{ OR } n = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$
- $\texttt{op1}(1,(m,n)) = \texttt{IF } n >= m \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$
- $\texttt{op1}(2,(m,n)) = (m+n, 0)$
- $\texttt{op1}(3,(m,n)) = (m, \max(0, n-m))$
- $\texttt{op1}(4,(m,n)) = (n,m)$

Example 2: GCD

```
gcd(m,n:nat) : RECURSIVE nat =
  IF m = 0 OR n = 0 THEN m + n
  ELSIF n >= m THEN gcd(m,n-m)
    ELSE gcd(n,m)
  ENDIF
```

- $e_g =$

```
ite(op1(0,vr),
    op1(2,vr),
    ite(op1(1,vr),
        rec(op1(3,vr)),
        rec(op1(4,vr))))
```

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

Let $a \in \texttt{PVS0}[Val] = (O_1, O_2, \bot, e_a)$, where

- $Val = [\texttt{nat}, \texttt{nat}]$

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

Let $a \in \texttt{PVS0}[Val] = (O_1, O_2, \bot, e_a)$, where

- $Val = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

Let $a \in \texttt{PVS0}[Val] = (O_1, O_2, \bot, e_a)$, where

- $Val = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\texttt{op1}(0,(m,n)) = \texttt{IF } m = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

Let $a \in \texttt{PVS0}[Val] = (O_1, O_2, \bot, e_a)$, where

- $Val = [\texttt{nat}, \texttt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\texttt{op1}(0,(m,n)) = \texttt{IF } m = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$
- $\texttt{op1}(1,(m,n)) = \texttt{IF } n = 0 \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ ENDIF}$

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

Let $a \in \mathtt{PVS0}[Val] = (O_1, O_2, \bot, e_a)$, where

- $Val = [\mathtt{nat}, \mathtt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\mathtt{op1}(0,(m,n)) = \mathtt{IF}\ m = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$
- $\mathtt{op1}(1,(m,n)) = \mathtt{IF}\ n = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$
- $\mathtt{op1}(2,(m,n)) = (n+1,0)$

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

Let $a \in \mathtt{PVS0}[Val] = (O_1, O_2, \bot, e_a)$, where

- $Val = [\mathtt{nat}, \mathtt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\mathrm{op1}(0,(m,n)) = \mathtt{IF}\ m = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$
- $\mathrm{op1}(1,(m,n)) = \mathtt{IF}\ n = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$
- $\mathrm{op1}(2,(m,n)) = (n+1,0)$
- $\mathrm{op1}(3,(m,n)) = (\max(0,m-1),1)$

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

Let $a \in \mathtt{PVS0}[Val] = (O_1, O_2, \bot, e_a)$, where

- $Val = [\mathtt{nat}, \mathtt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\mathrm{op1}(0,(m,n)) = \mathtt{IF}\ m = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$
- $\mathrm{op1}(1,(m,n)) = \mathtt{IF}\ n = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$
- $\mathrm{op1}(2,(m,n)) = (n+1,0)$
- $\mathrm{op1}(3,(m,n)) = (\max(0,m-1),1)$
- $\mathrm{op1}(4,(m,n)) = (m,\max(0,n-1))$

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

Let $a \in \mathtt{PVS0}[Val] = (O_1, O_2, \bot, e_a)$, where

- $Val = [\mathtt{nat}, \mathtt{nat}]$
- $\bot = (0,0).\top = (1,0)$
- $\mathrm{op1}(0,(m,n)) = \mathtt{IF}\ m = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$
- $\mathrm{op1}(1,(m,n)) = \mathtt{IF}\ n = 0\ \mathtt{THEN}\ \top\ \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}$
- $\mathrm{op1}(2,(m,n)) = (n+1,0)$
- $\mathrm{op1}(3,(m,n)) = (\max(0,m-1),1)$
- $\mathrm{op1}(4,(m,n)) = (m,\max(0,n-1))$
- $\mathrm{op2}(0,(m,n),(i,j)) = (\max(0,m-1),i)$

Example 3: Ackermann

```
ackermann(m,n:nat) : RECURSIVE nat =
  IF m = 0 THEN n+1
  ELSIF n = 0 THEN ackermann(m-1,1)
    ELSE ackermann(m-1,ackermann(m,n-1))
  ENDIF
```

- $e_a =$

```
ite(op1(0,vr),
    op1(2,vr),
    ite(op1(1,vr),
        rec(op1(3,vr)),
        rec(op2(0,vr,rec(op1(4,vr))))))
```

Semantic Evaluation Relation ε

Given a PVS0 program $pvso = (O_1, O_2, \bot, e_{pvso})$ of type $\texttt{PVS0}[Val]$, the predicate ε holds when the semantic evaluation of an expression e of type $\texttt{PVS0Expr}[Val]$ on the input value v_i results in the value v_o.

$$
\begin{array}{rcl}
\varepsilon(pvso)(e, v_i, v_o) & := & \texttt{CASES } e \texttt{ OF} \\
\texttt{cnst}(v) & : & v_o = v; \\
\texttt{vr} & : & v_o = v_i; \\
\texttt{op1}(j, e_1) & : & j < |O_1| \ \wedge \exists\, v' \in Val : \\
 & & \varepsilon(pvso)(e_1, v_i, v') \wedge v_o = O_1(j)(v'); \\
\texttt{op2}(j, e_1, e_2) & : & j < |O_2| \ \wedge \exists\, v', v'' \in Val : \\
 & & \varepsilon(pvso)(e_1, v_i, v') \ \wedge \\
 & & \varepsilon(pvso)(e_2, v_i, v'') \ \wedge \\
 & & v_o = O_2(j)(v', v''); \\
\texttt{rec}(e_1) & : & \exists\, v' \in \mathcal{V}al : \varepsilon(pvso)(e_1, v_i, v') \ \wedge \\
 & & \varepsilon(pvso)(e_{pvso}, v', v_o) \\
\texttt{ite}(e_1, e_2, e_3) & : & \exists\, v' : \varepsilon(pvso)(e_1, v_i, v') \ \wedge \\
 & & \texttt{IF } v' \neq \bot \texttt{ THEN } \varepsilon(pvso)(e_2, v_i, v_o) \\
 & & \texttt{ELSE } \varepsilon(pvso)(e_3, v_i, v_o) \texttt{ ENDIF } .
\end{array}
$$

The Relation ε is Deterministic

Lemma 1.
Let pvso be a PVS0 program of type `PVS0`$[Val]$. *For any expression e of type* `PVS0Expr`$[Val]$ *and all values* $v_i, v_o', v_o'' \in Val$,

$$\varepsilon(pvso)(e, v_i, v_o') \text{ and } \varepsilon(pvso)(e, v_i, v_o'') \text{ implies } v_o' = v_o''.$$

The proof of this lemma uses the induction schema generated for the inductive relation ε.

Semantic Evaluation Function χ

- Given a PVS0 program $pvso = (O_1, O_2, \bot, e_{pvso})$ of type $\texttt{PVS0}[Val]$ and a natural number n, representing a maximum number of recursive calls, the function χ evaluates an expression e of type $\texttt{PVS0Expr}[Val]$ on the input value v_i. The function returns either the undefined value $\diamondsuit$ or a value of type Val.
- The function χ is recursively defined on the structure of e.

The Function χ

$$
\begin{array}{rcl}
\chi(pvso)(e, v_i, n) & := & \texttt{IF } n = 0 \texttt{ THEN } \diamondsuit \texttt{ ELSE CASES } e \texttt{ OF} \\
\texttt{cnst}(v) & : & \quad v; \\
\texttt{vr} & : & \quad v_i; \\
\texttt{op1}(j, e_1) & : & \quad \texttt{IF } j < |O_1| \texttt{ THEN} \\
 & & \qquad \texttt{LET } v' = \chi(pvso)(e_1, v_i, n) \texttt{ IN} \\
 & & \qquad \texttt{IF } v' = \diamondsuit \texttt{ THEN } \diamondsuit \texttt{ ELSE } O_1(j)(v') \texttt{ ENDIF} \\
 & & \quad \texttt{ELSE } \bot \texttt{ ENDIF ;} \\
\texttt{op2}(j, e_1, e_2) & : & \quad \texttt{IF } j < |O_2| \texttt{ THEN} \\
 & & \qquad \texttt{LET } v' = \chi(pvso)(e_1, v_i, n), \\
 & & \qquad\quad v'' = \chi(pvso)(e_2, v_i, n) \texttt{ IN} \\
 & & \qquad \texttt{IF } v' = \diamondsuit \vee v'' = \diamondsuit \texttt{ THEN } \diamondsuit \texttt{ ELSE } O_2(j)(v', v'') \texttt{ ENDIF} \\
 & & \quad \texttt{ELSE } \bot \texttt{ ENDIF ;} \\
\texttt{rec}(e_1) & : & \quad \texttt{LET } v' = \chi(pvso)(e_1, v_i, n) \texttt{ IN} \\
 & & \quad \texttt{IF } v' = \diamondsuit \texttt{ THEN } \diamondsuit \texttt{ ELSE } \chi(pvso)(e_{pvso}, v', n-1) \texttt{ ENDIF ;} \\
\texttt{ite}(e_1, e_2, e_3) & : & \quad \texttt{LET } v' = \chi(pvso)(e_1, v_i, n) \texttt{ IN} \\
 & & \quad \texttt{IF } v' = \diamondsuit \texttt{ THEN } \diamondsuit \\
 & & \quad \texttt{ELSIF } v' \neq \bot \texttt{ THEN } \chi(pvso)(e_2, v_i, n) \\
 & & \quad \texttt{ELSE } \chi(pvso)(e_3, v_i, n) \texttt{ ENDIF ;} \\
 & & \texttt{ENDIF}
\end{array}
$$

Equivalence of ε and χ Evaluation

Theorem 2.
Let pvso be a PVS0 program of type `PVS0`$[Val]$. *For any* $v_i \in Val$ *and e of type* `PVS0Expr`$[Val]$,

$$\varepsilon(pvso)(e, v_i, v_o) \text{ if and only if } v_o = \chi(pvso)(e, v_i, n),$$

for some n, where $v_o \neq \diamondsuit$.

Factorial, GCD, and Ackermann Lemmas

- The PVS0 program f computes the PVS function factorial, i.e., for any $n, k \in \texttt{nat}$,

$$\mathit{factorial}(n) = k \text{ if and only if } \varepsilon(f)(e_f, n, k).$$

- The PVS0 program g computes the PVS function gcd, i.e., for any $n, m, k \in \texttt{nat}$,

$$\mathit{gcd}(m, n) = k \text{ if and only if } \varepsilon(g)(e_g, (m, n), (k, i)),$$

  for some i.

- The PVS0 program a computes the PVS function ackermann, i.e., for any $n, m, k \in \texttt{nat}$,

$$\mathit{ackermann}(m, n) = k \text{ if and only if } \varepsilon(a)(e_a, (m, n), (k, i)),$$

  for some i.

ε-Termination

- The PVS0 program $pvso \in \mathtt{PVS0}[Val]$ ε-terminates for an input $v_i \in Val$ if the following predicate holds

$$T_\varepsilon(pvso, v_i) \equiv \exists\, v_o \in Val \;:\; \varepsilon(pvso)(e_{pvso}, v_i, v_o).$$

- The PVS0 program $pvso$ is *ε-terminating* if for all $v_i \in Val$, $T_\varepsilon(pvso, v_i)$ holds.

χ-Termination

- The PVS0 program $pvso \in \texttt{PVS0}[Val]$ χ-terminates for an input $v_i \in Val$ if the following predicate holds

$$T_\chi(pvso, v_i) \equiv \exists\, n \in \texttt{nat} \,:\, \chi(pvso)(e_{pvso}, v_i, n) \neq \diamondsuit.$$

- The PVS0 program $pvso$ is χ-*terminating* if for all $v_i \in Val$, $T_\chi(pvso, v_i)$ holds.

Equivalence of T_ε and T_χ

Theorem 3.
Let pvso be a PVS0 program of type `PVS0`[*Val*]. *The following conditions hold:*

1. *For any* $v_i \in Val$, $T_\varepsilon(pvso, v_i)$ *if and only if* $T_\chi(pvso, v_i)$.
2. *pvso is* ε*-terminating if and only if pvso is* χ*-terminating.*

A PVS0 program *pvso* that is ε-terminating (or equivalently, χ-terminating) is said to be *terminating*, denoted `terminating`(*pvso*).

Existence of Non-terminating PVS0 Programs

Lemma 4.
Let $\Delta = (O_1, O_2, \bot, \texttt{rec(vr)})$. *For any* $v \in \mathit{Val}$, $\neg T_\chi(\Delta, v)$.

Proof.

1. Define $\mu(\mathit{pvso}, v)$, for any *pvso* and v such that $T_\chi(\mathit{pvso}, v)$, as the minimum n that satisfies

$$\chi(\mathit{pvso})(e_{\mathit{pvso}}, v, n) \neq \diamondsuit.$$

2. By contradiction, assume that $\chi(\Delta)(e_\Delta, v, n) \neq \diamondsuit$ for some n.
3. Therefore, $\chi(\Delta)(e_\Delta, v, \mu(\Delta, v)) \neq \diamondsuit$, by definition of μ.
4. By definition of Δ and χ, it is also the case that $\chi(\Delta)(e_\Delta, v, \mu(\Delta, v) - 1)$.
5. This is a contradiction because $\mu(\Delta, v) - 1 < \mu(\Delta, v)$.

□

Evaluation of Terminating PVS0 Programs

Let $pvso = (O_1, O_2, \bot, e_{pvso})$ be a terminating PVS0 program in $\mathtt{PVS0}[Val]$, i.e., $\mathtt{terminating}(pvso)$. The evaluation of $pvso$ on v can be defined as

$\mathtt{pvs0\_eval}(pvso)(v) \equiv \mathtt{pvs0\_eval\_expr}(pvso)(e_{pvso}, v)$, where

$$
\begin{array}{rcl}
\mathtt{pvs0\_eval\_expr}(pvso)(e, v_i) & := & \mathtt{CASES}\ e\ \mathtt{OF} \\
\mathtt{cnst}(v) & : & v; \\
\mathtt{vr} & : & v_i; \\
\mathtt{op1}(j, e_1) & : & \mathtt{IF}\ j < |O_1|\ \mathtt{THEN} \\
 & & \quad O_1(j)(\mathtt{pvs0\_eval\_expr}(pvso)(e_1, v_i)) \\
 & & \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}\ ; \\
\mathtt{op2}(j, e_1, e_2) & : & \mathtt{IF}\ j < |O_2|\ \mathtt{THEN} \\
 & & \quad O_1(j)(\mathtt{pvs0\_eval\_expr}(pvso)(e_1, v_i), \\
 & & \qquad \mathtt{pvs0\_eval\_expr}(pvso)(e_2, v_i)) \\
 & & \mathtt{ELSE}\ \bot\ \mathtt{ENDIF}\ ; \\
\mathtt{rec}(e_1) & : & \mathtt{pvs0\_eval\_expr}(pvso)(e_{pvso})(\mathtt{pvs0\_eval\_expr}(pvso)(e_1, v_i)) \\
\mathtt{ite}(e_1, e_2, e_3) & : & \mathtt{LET}\ v' = \mathtt{pvs0\_eval\_expr}(pvso)(e_1, v_i)\ \mathtt{IN} \\
 & & \mathtt{IF}\ v' \neq \bot\ \mathtt{THEN}\ \mathtt{pvs0\_eval\_expr}(pvso)(e_2, v_i) \\
 & & \mathtt{ELSE}\ \mathtt{pvs0\_eval\_expr}(pvso)(e_3, v_i)\ \mathtt{ENDIF}\ .
\end{array}
$$

Correctness of `pvs0_eval`

Theorem 5.
For all terminating programs $pvso \in \mathtt{PVS0}[Val]$ *and* $v_i, v_o \in Val$, $\varepsilon(pvso)(e_{pvso}, v_i, v_o)$ *if and only if* $v_o = \mathtt{pvs0\_eval}(pvso)(v_i)$.

A Syntactic Termination Criterion

- T_ε and T_χ are impractical as termination criteria since they require case by case analysis on the input to the function.
- Turing Termination Criterion: If there is a measure on a well-founded relation that strictly decreases at every recursive call, the function is terminating.
- This criterion, which implements a simple static analysis, requires the formalization of several syntactic elements:
  - Recursive calls.
  - Conditions.
  - Paths.

PVS0 Calling Context

A PVS0 Calling Context is triple (r, P, C), where

- r is a `PVS0Expr` of the form `rec`(e), where e is a `PVS0Expr`.
- P is a path, i.e., a list of natural numbers.
- C is a list of Boolean expressions, i.e., a `PVS0Expr` or a negation of a `PVS0Expr`.

A (r, P, C) is a valid calling context of $e \in$ `PVS0Expr` if

- r is a subexpression of e.
- P is the path of r in e,
- C is the set of accumulated conditions for the path P.

The set of valid calling contexts of e are denoted `pvs0_tcc_valid_cc`(e).

Valid Calling Contexts of Ackermann

```
IF m = 0 THEN n+1
ELSIF n = 0 THEN ackermann(m-1,1)
ELSE ackermann(m-1,ackermann(m,n-1))
ENDIF
```

$$\texttt{ite}_{\langle 0\rangle}(\texttt{op1}_{\langle 00\rangle}(\texttt{0},\texttt{vr}_{\langle 000\rangle}),\texttt{op1}_{\langle 10\rangle}(\texttt{2},\texttt{vr}_{\langle 010\rangle}),$$
$$\texttt{ite}_{\langle 20\rangle}(\texttt{op1}_{\langle 020\rangle}(\texttt{1},\texttt{vr}_{\langle 0020\rangle}),\texttt{rec}_{\langle 120\rangle}(\texttt{op1}_{\langle 0120\rangle}(\texttt{3},\texttt{vr}_{\langle 00120\rangle})),$$
$$\texttt{rec}_{\langle 220\rangle}(\texttt{op2}_{\langle 0220\rangle}(\texttt{0},\texttt{vr}_{\langle 00220\rangle},$$
$$\texttt{rec}_{\langle 10220\rangle}(\texttt{op1}_{\langle 0\ldots\rangle}(\texttt{4},\texttt{vr}_{\langle 00\ldots\rangle}))))))$$

- $cc_1 =$
  $(\texttt{rec(op1(3,vr))}, \langle 120\rangle, \{\texttt{op1(1,vr)}, \texttt{!(op1(0,vr))}\})$.

Valid Calling Contexts of Ackermann

```
IF m = 0 THEN n+1
ELSIF n = 0 THEN ackermann(m-1,1)
ELSE ackermann(m-1,ackermann(m,n-1))
ENDIF
```

$$\texttt{ite}_{\langle 0\rangle}(\texttt{op1}_{\langle 00\rangle}(0,\texttt{vr}_{\langle 000\rangle}),\texttt{op1}_{\langle 10\rangle}(2,\texttt{vr}_{\langle 010\rangle}),$$
$$\texttt{ite}_{\langle 20\rangle}(\texttt{op1}_{\langle 020\rangle}(1,\texttt{vr}_{\langle 0020\rangle}),\texttt{rec}_{\langle 120\rangle}(\texttt{op1}_{\langle 0120\rangle}(3,\texttt{vr}_{\langle 00120\rangle})),$$
$$\texttt{rec}_{\langle 220\rangle}(\texttt{op2}_{\langle 0220\rangle}(0,\texttt{vr}_{\langle 00220\rangle},$$
$$\texttt{rec}_{\langle 10220\rangle}(\texttt{op1}_{\langle 0\ldots\rangle}(4,\texttt{vr}_{\langle 00\ldots\rangle}))))))$$

- $cc_1 =$
  $(\texttt{rec(op1(3,vr))}, \langle 120\rangle, \{\texttt{op1(1,vr)}, \texttt{!(op1(0,vr))}\})$.
- $cc_2 =$
  $(\texttt{rec(op2(0,vr,...))}, \langle 220\rangle, \{\texttt{!(op1(1,vr))}, \texttt{!(op1(0,vr))}\})$.

Valid Calling Contexts of Ackermann

```
IF m = 0 THEN n+1
ELSIF n = 0 THEN ackermann(m-1,1)
ELSE ackermann(m-1,ackermann(m,n-1))
ENDIF
```

$$\texttt{ite}_{\langle 0\rangle}(\texttt{op1}_{\langle 00\rangle}(0,\texttt{vr}_{\langle 000\rangle}),\texttt{op1}_{\langle 10\rangle}(2,\texttt{vr}_{\langle 010\rangle}),$$
$$\texttt{ite}_{\langle 20\rangle}(\texttt{op1}_{\langle 020\rangle}(1,\texttt{vr}_{\langle 0020\rangle}),\texttt{rec}_{\langle 120\rangle}(\texttt{op1}_{\langle 0120\rangle}(3,\texttt{vr}_{\langle 00120\rangle})),$$
$$\texttt{rec}_{\langle 220\rangle}(\texttt{op2}_{\langle 0220\rangle}(0,\texttt{vr}_{\langle 00220\rangle},$$
$$\texttt{rec}_{\langle 10220\rangle}(\texttt{op1}_{\langle 0\ldots\rangle}(4,\texttt{vr}_{\langle 00\ldots\rangle}))))))$$

- $cc_1 =$ $(\texttt{rec(op1(3,vr))}, \langle 120\rangle, \{\texttt{op1(1,vr)}, \texttt{!(op1(0,vr))}\})$.
- $cc_2 =$ $(\texttt{rec(op2(0,vr,...))}, \langle 220\rangle, \{\texttt{!(op1(1,vr))}, \texttt{!(op1(0,vr))}\})$.
- $cc_3 =$ $(\texttt{rec(op1(0,vr))}, \langle 10220\rangle, \{\texttt{!(op1(4,vr))}, \texttt{!(op1(0,vr))}\})$.

Evaluation of Conditions

$$
\begin{array}{rcl}
\texttt{eval\_conds}(pvso)(C, v_i) & := & \texttt{CASES } C \texttt{ OF} \\
\texttt{null} & : & \texttt{TRUE}; \\
\texttt{cons}(e, C') & : & (\exists\, v_o \in Val : \varepsilon(pvso)(e, v_i, v_o) \wedge v_o \neq \bot) \wedge \\
 & & \texttt{eval\_conds}(pvso)(C', v_i); \\
\texttt{cons}(!(e), C') & : & (\exists\, v_o \in Val : \varepsilon(pvso)(e, v_i, v_o) \wedge v_o = \bot) \wedge \\
 & & \texttt{eval\_conds}(pvso)(C', v_i).
\end{array}
$$

Termination Correctness Condition

- A PVS0 program $pvso \in \texttt{PVS0}[Val]$ satisfies the predicate $\texttt{pvs0\_tcc\_termination}(pvso)$ for a type M if and only
- There exists a function m from Val into M and a well-founded relation $<$ on M such that for all $v_i, v_o : Val$ and $(rec(e), P, C) \in \texttt{pvs0\_tcc\_valid\_cc}(e_{pvso})$,
  - $\varepsilon(pvso)(e, v_i, v_o)$ and
  - $\texttt{eval\_conds}(pvso)(C, v_i)$

  implies $m(v_o) < m(v_i)$.

TCC Correctness

Theorem 6.
Let pvso be a PVS0 program of type `PVS0`[*Val*]. *The predicate* `pvs0_tcc_termination`(*pvso*) *holds for a type M if and only if* `terminating`(*pvso*), *i.e., for all* $v : Val$, $T_\varepsilon(pvso, v)$ *(or, equivalently,* $T_\chi(pvso, v)$*)* .

- The direction "$\leftarrow$" uses $M = \texttt{nat}$ and the well-founded order $<$ on natural numbers.
- The proof of this statement uses the definition of

$$\Omega_m(v) := \min(\{n : \mathbb{N}^+ \mid \forall\, v' \in V \,:\, \neg(m(v) >^n m(v'))\}).$$

- Intuitively, $\Omega_m(v)$ is the length of the longest path downwards starting from $m(v)$.

Ω and μ

The following lemma states a relation between μ and Ω.

Lemma 7.
Let pvso be a PVS0 program that satisfies `pvs0_tcc_termination`*(pvso) for a well-founded relation* $<$ *over* M *and a measure function* m*. For any value* $v \in Val$*,* $\mu(pvso, v) \leq \Omega_m(v)$.

The PVS0 Ackermann Program is Terminating

Theorem 8.
The PVS0 Ackerman program ▸ a *satisfies* $\mathit{terminating}(a)$.

- By Theorem 6, it suffices to check $\texttt{pvs0\_tcc\_termination}(a)$.
- $\texttt{pvs0\_tcc\_termination}(a)$ can be checked for all the PVS0 calling contexts of a, i.e., ▸ $\{cc_1, cc_2, cc_3\}$ using $M = [\texttt{nat}, \texttt{nat}]$ and $(a, b) < (c, d) \equiv a < c \texttt{ OR } (a = c \texttt{ AND } b < d)$.
- The proofs of these conditions correspond to the proofs of the actual termination correctness conditions generated by the PVS Type Checker for the PVS function *ackermann*.

The Size-Change Principle (SCP)

SCP Termination Criterion

A program terminates on all inputs if *every infinite call sequence* (following program control flow) would cause an infinite descent (over a well-founded relation) in some data values. [LJB01].

Infinite Sequence of Computations

Let $pvso \in \texttt{PVS0}[Val]$, $\mathbf{cc} = (\texttt{rec}(e_0), P_0, C_0), \ldots$ be an infinite sequence of calling contexts of $pvso$, and $\mathbf{V} = v_0, \ldots$ be an infinite sequence of values in Val such that the following predicate holds.

$$\texttt{infinite\_seq\_ccs}(\mathbf{cc}, \mathbf{V}) \equiv \forall(i : nat) : (\texttt{eval\_conds}(pvso)(C_i, v_i)\ \texttt{AND}\ \varepsilon(pvso)(e_i, v_i, v_{i+1})).$$

Let $<$ be a well-founded relation over Val, $\texttt{SCP}_<(pvso)$ holds if for all infinite sequence $\mathbf{cc}$ of calling contexts of $pvso$ and infinite sequence $\mathbf{V}$ of values in Val that satisfy $\texttt{infinite\_seq\_ccs}(\mathbf{cc}, \mathbf{V})$, $v_{i+1} < v_i$ for all i.

SCP Termination Criterion

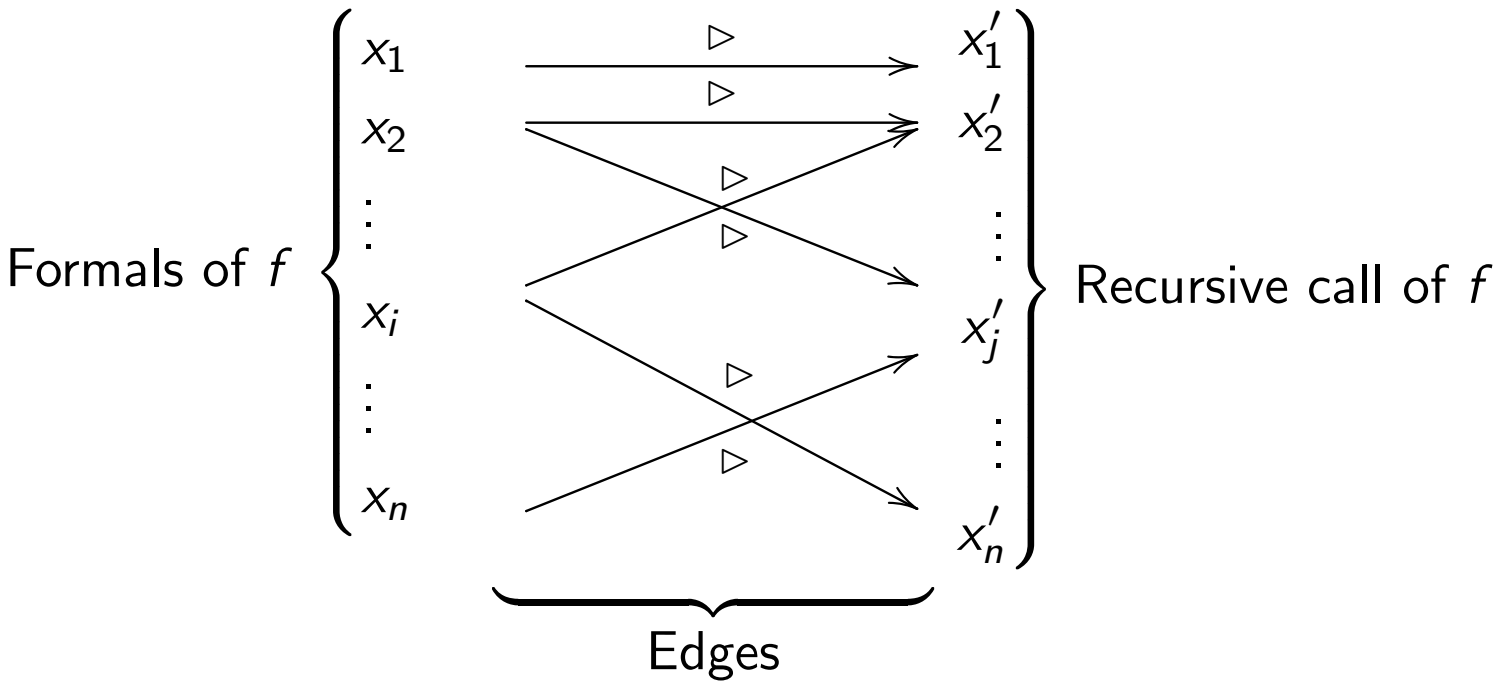
`scp_termination`(*pvso*) holds if there are no infinite sequence **cc** of calling contexts of *pvso* and infinite sequence **V** of values in *Val* that satisfy `infinite_seq_ccs`(**cc**, **V**).

Theorem 9.
For all pvso $\in$ `PVSO`[*Val*], *terminating*(*pvso*) *if and only if* *scp_termination*(*pvso*).

Theorem 10.
For all pvso $\in$ `PVSO`[*Val*], *scp_termination*(*pvso*) *if and only if* $SCP_{<}$(*pvso*) *for a well-founded relation* $<$ *over Val*.

Implementing SCP: Size Change Graph (SG)

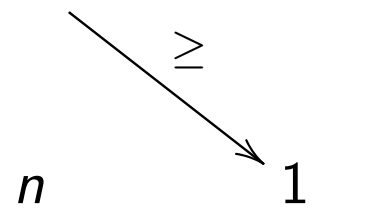
To every call c associated to a function call $f(x_1, ..., x_n) \xrightarrow{c} f(x'_1, ..., x'_m)$, the graph $\mathcal{G}_c$ is defined such that there is an edge $x_i \xrightarrow{\triangleright} x'_j$ if $x_i \triangleright x'_j$, where $\triangleright \in \{>, \geq\}$:



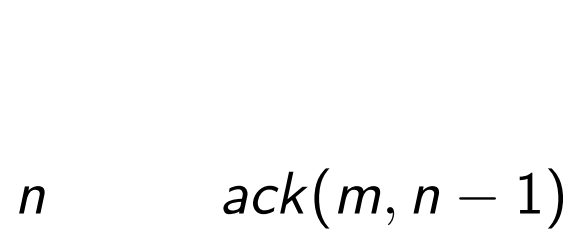
SCGs for Ackermann

$$ack(m,n) = ite(m = 0, n+1, ite(n = 0, \underbrace{ack}_{1}(m-1,1), \underbrace{ack}_{2}(m-1, 3 : \underbrace{ack}_{3}(m, n-1))))$$

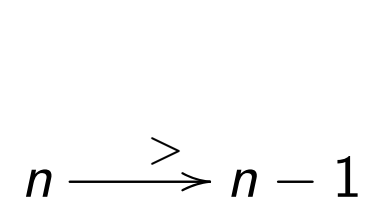
G_1 :



G_2 :

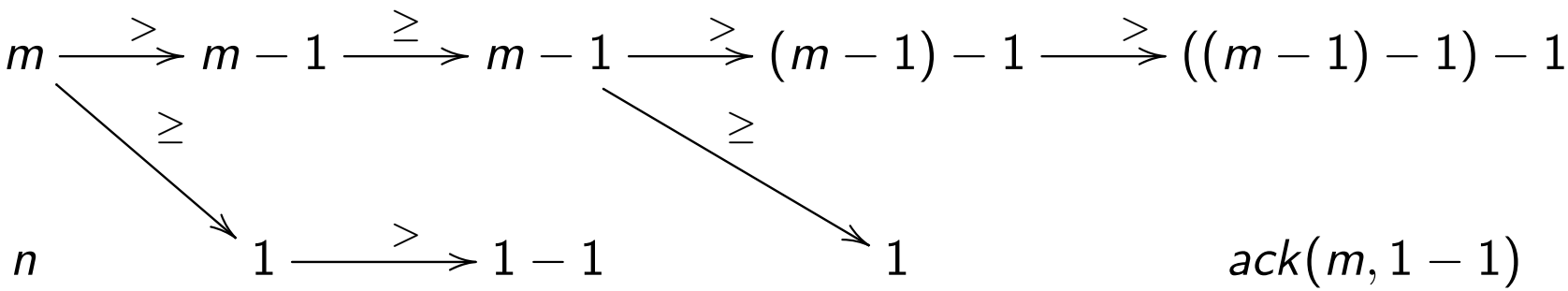


G_3 :



Multipaths and Threads

- A multipath is a sequence, potentially infinite, $G_{c_1}, G_{c_2}, \ldots$ of SCGs. A multipath can be seen as a concatenated graph.
- E.g., the Ackermann sequence of calls $c_1 c_3 c_1 c_2$ yields the multipath G_1, G_3, G_1, G_2:



Threads

- A thread in a multipath is a connected path of arcs $r_1 \xrightarrow{\rhd_1} r_2 \xrightarrow{\rhd_2} \dots$.
- A thread is descending if at least one $\rhd_i$ is $>$. The thread is infinitely descending if it contains infinitely many occurrences of $>$.
- Size Change Principle: A program terminates if every infinite call sequence yields an infinitely descending thread (over a well-founded order $<$).

Calling Context Graphs (CCG)

- CCG a termination analysis technique based on the size-change principle [MV06].
- SCG Multipaths are represented by:
  - A directed graph of calling contexts,
  - where edges are labelled using a family of measures.

Calling Contexts (Reminder)

Calling contexts are a representation of recursive calls and their governing conditions.

$$ack(m, n) = ite(m = 0, n + 1, \\ ite(n = 0, ack(m - 1, 1), \\ ack(m - 1, ack(m, n - 1))))$$

- $cc_1 = (ack(m - 1, 1), m \neq 0 \wedge n = 0)$

Calling Contexts (Reminder)

Calling contexts are a representation of recursive calls and their governing conditions.

$$ack(m, n) = ite(m = 0, n + 1, \\ ite(n = 0, ack(m - 1, 1), \\ ack(m - 1, ack(m, n - 1))))$$

- $cc_1 = (ack(m - 1, 1), m \neq 0 \wedge n = 0)$
- $cc_2 = (ack(m - 1, ack(m, n - 1)), m \neq 0 \wedge n \neq 0)$

Calling Contexts (Reminder)

Calling contexts are a representation of recursive calls and their governing conditions.

$$ack(m, n) = ite(m = 0, n + 1, \\ ite(n = 0, ack(m - 1, 1), \\ ack(m - 1, ack(m, n - 1))))$$

- $cc_1 = (ack(m - 1, 1), m \neq 0 \wedge n = 0)$
- $cc_2 = (ack(m - 1, ack(m, n - 1)), m \neq 0 \wedge n \neq 0)$
- $cc_3 = (ack(m, n - 1), m \neq 0 \wedge n \neq 0)$

Calling Context Graph

Given a PVS0 program $pvso \in \texttt{PVS0}[Val]$, a directed graph G of is built, where

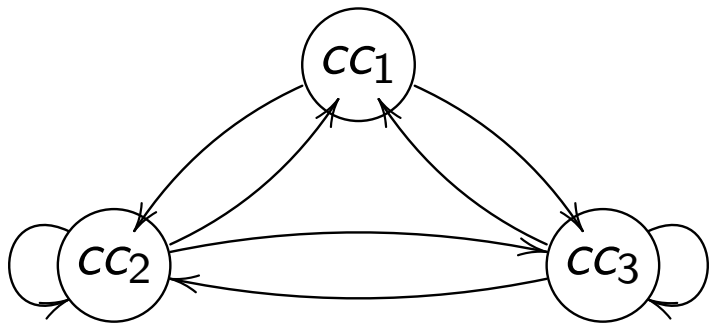
- the nodes of G are the calling contexts in $\texttt{pvs0\_tcc\_valid\_cc}(e_{pvso})$,
- there is an edge between calling contexts $(\texttt{rec}(e_a), P_a, C_a)$ and $(\texttt{rec}(e_b), P_b, C_b)$ if

$$\begin{aligned}\exists(v_a, v_b : Val) :&\texttt{eval\_conds}(pvso)(C_a, v_a) \ \texttt{AND} \\ &\varepsilon(pvso)(e_a, v_a, v_b) \ \texttt{AND} \\ &\texttt{eval\_conds}(pvso)(C_b, v_b).\end{aligned}$$

- Remark: The above is not an "if-and-only-if" condition. Hence, a fully connected graph of calling contexts satisfies this condition.

A CCG for Ackermann

- $cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
- $cc_2 = (ack(m-1, ack(m,n-1)), m \neq 0 \wedge n \neq 0)$
- $cc_3 = (ack(m,n-1), m \neq 0 \wedge n \neq 0)$



- There is no edge between cc_1 and itself because

$$\nexists(m,n:\mathtt{nat}) : (m \neq 0 \ \wedge \ n = 0) \ \wedge \ (m-1 \neq 0 \ \wedge \ 1 = 0)$$

- There is an edge between cc_2 and cc_1 even although

$$\nexists(m,n:\mathtt{nat}) : (m \neq 0 \wedge n \neq 0) \wedge (m-1 \neq 0 \wedge ack(m,n-1) = 0)$$

Walks, Circuits, and Cycles

Let G_{pvso} be a CCG of a PVS0 program *pvso* in PVS0[*Val*].

- A walk of G_{pvso} is a sequence $cc_{i_1}, \ldots, cc_{i_n}$ of calling contexts such that for all $1 \leq j < n$ there is an edge between cc_{i_j} and $cc_{i_{j+1}}$.
- A circuit is a walk $cc_{i_1}, \ldots, cc_{i_n}$, with $n > 1$, where $cc_{i_1} = cc_{i_n}$.
- A cycle is an elementary circuit, i.e., a circuit $cc_{i_1}, \ldots, cc_{i_n}$ where the only repeating nodes are cc_{i_1} and cc_{i_n}.

Measure Combination

Let $\mathcal{M}$ be a family of N measures $\mu_k : \mathit{Val} \to M$, with $1 \leq k \leq N$, and $<$ be a well-founded relation over M.

- A measure combination of a walk $cc_{i_1}, \ldots, cc_{i_n}$ is a sequence of natural numbers $k_1, \ldots, k_n$, with $1 \leq k_j \leq N$ representing measure μ_{k_j}, such that for all $1 \leq j < n$, $v, v' \in \mathit{Val}$,

$$\texttt{eval\_conds}(\mathit{pvso})(C_j, v) \ \texttt{AND}\ \varepsilon(\mathit{pvso})(e_j, v, v') \\ \texttt{IMPLIES}\ \mu_{k_j}(v) \ \rhd_j \ \mu_{k_{j+1}}(v'),$$

  where $cc_{i_j} = (C_j, P_j, \texttt{rec}(e_j))$ and $\rhd_j \in \{>, \geq\}$.
- A measure combination is descending if at least one $\rhd_j$ is $>$.

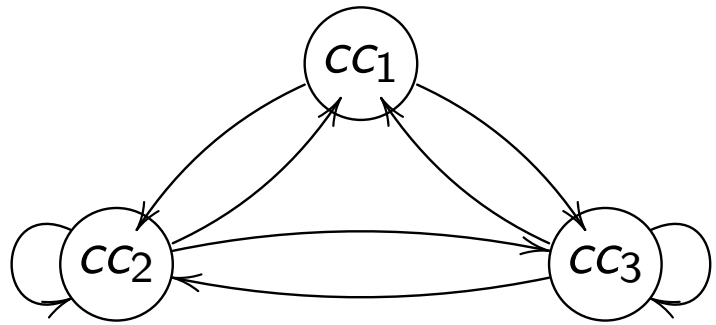
CCG Termination Criterion

- Let G_{pvso} be a CCG of a PVS0 program *pvso* in $\texttt{PVS0}[Val]$ and $\mathcal{M}$ be a family of N measures for a well-founded relation $<$ over a type M.
- $\texttt{ccg\_termination}_{\mathcal{M}}(G_{pvso})$ holds if for all circuits $cc_{i_1}, \ldots, cc_{i_n}$ in G_{pvso} there is a descending measure combination $k_1, \ldots, k_n$, with $k_1 = k_n$.

Finding Descending Measure Combinations

$$\mu_1(m,n) = m$$
$$\mu_2(m,n) = n$$

$\mu_1\mu_1$
$\mu_1\mu_2$
$\mu_2\mu_1$
$\mu_2\mu_2$



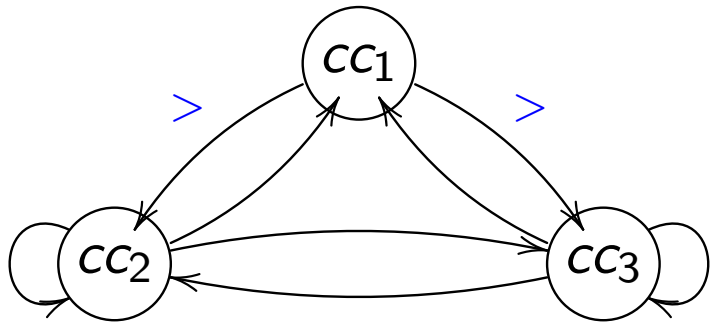
$ack(m,n) \equiv \ldots$

$cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
$cc_2 = (ack(m-1, ack(m,n-1)), m \neq 0 \wedge n \neq 0)$
$cc_3 = (ack(m,n-1), m \neq 0 \wedge n \neq 0)$

Finding Descending Measure Combinations

$$\mu_1(m,n) = m$$
$$\mu_2(m,n) = n$$

$$\mu_1\mu_1$$



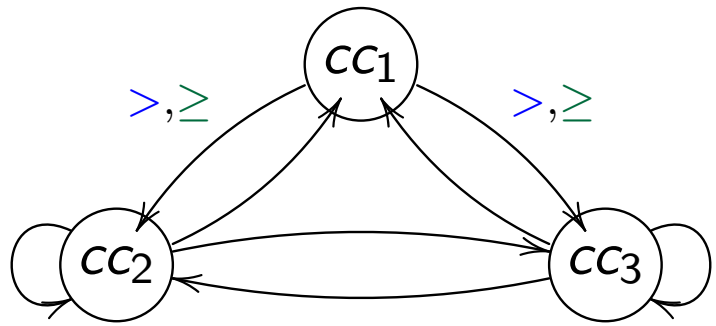
$ack(m,n) \equiv \ldots$

$cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
$cc_2 = (ack(m-1, ack(m,n-1)), m \neq 0 \wedge n \neq 0)$
$cc_3 = (ack(m,n-1), m \neq 0 \wedge n \neq 0)$

Finding Descending Measure Combinations

$$\mu_1(m,n) = m$$
$$\mu_2(m,n) = n$$

$$\mu_1\mu_2$$



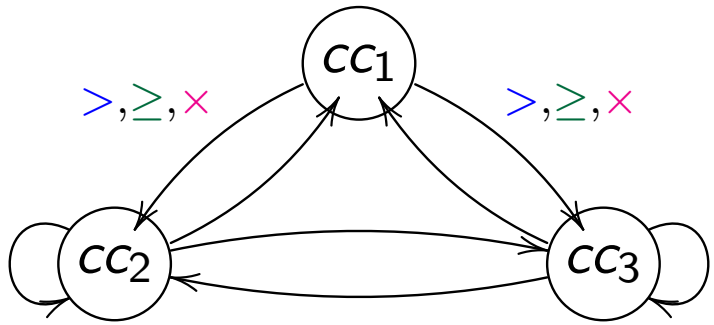
$ack(m,n) \equiv \ldots$

$cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
$cc_2 = (ack(m-1, ack(m,n-1)), m \neq 0 \wedge n \neq 0)$
$cc_3 = (ack(m,n-1), m \neq 0 \wedge n \neq 0)$

Finding Descending Measure Combinations

$$\mu_1(m,n) = m$$
$$\mu_2(m,n) = n$$

$$\mu_2\mu_1$$



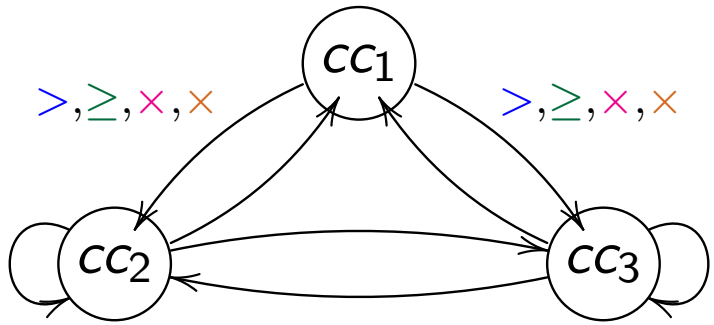
$ack(m,n) \equiv \ldots$

$cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
$cc_2 = (ack(m-1, ack(m,n-1)), m \neq 0 \wedge n \neq 0)$
$cc_3 = (ack(m,n-1), m \neq 0 \wedge n \neq 0)$

Finding Descending Measure Combinations

$$\mu_1(m,n) = m$$
$$\mu_2(m,n) = n$$

$$\mu_2\mu_2$$



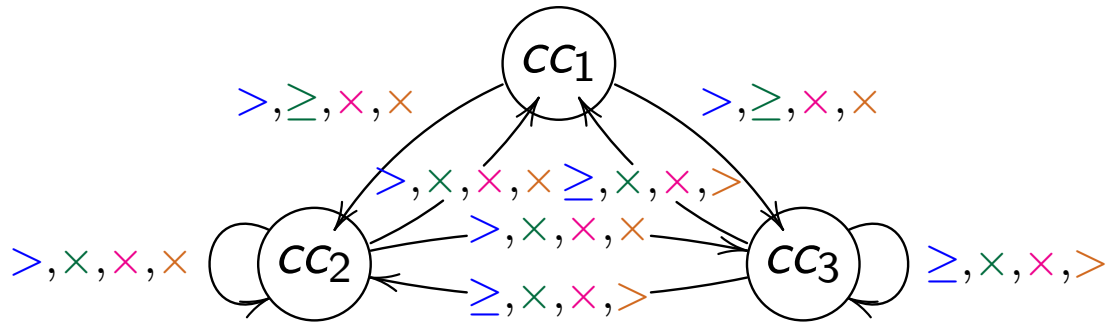
$ack(m,n) \equiv \ldots$

$cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
$cc_2 = (ack(m-1, ack(m,n-1)), m \neq 0 \wedge n \neq 0)$
$cc_3 = (ack(m,n-1), m \neq 0 \wedge n \neq 0)$

Finding Descending Measure Combinations

$$\mu_1(m,n) = m$$
$$\mu_2(m,n) = n$$

$\mu_1\mu_1$
$\mu_1\mu_2$
$\mu_2\mu_1$
$\mu_2\mu_2$

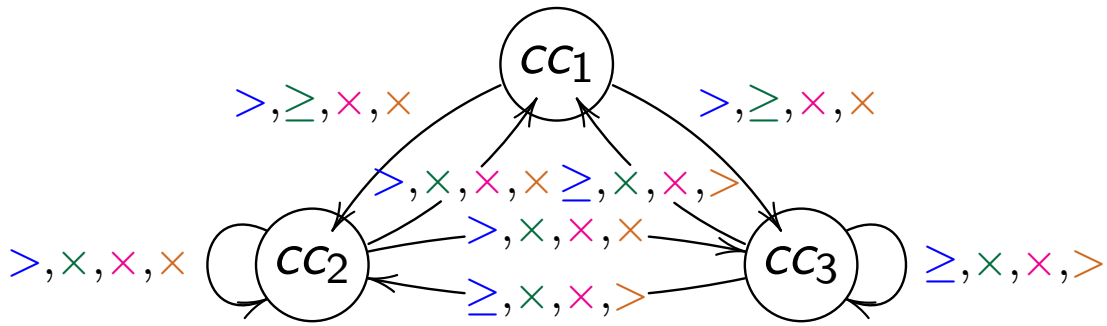


$ack(m,n) \equiv \ldots$

$cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
$cc_2 = (ack(m-1, ack(m,n-1)), m \neq 0 \wedge n \neq 0)$
$cc_3 = (ack(m,n-1), m \neq 0 \wedge n \neq 0)$

Finding Descending Measure Combinations

$\mu_1\mu_1$ $\mu_1\mu_2$
$\mu_2\mu_1$ $\mu_2\mu_2$



- Consider the circuit $cc_1, cc_3, cc_2, cc_2, cc_1$, the measure combination $1,1,1,1,1$, i.e., $>, \geq, >, >,$ is descending.
- Consider the circuit cc_3, cc_3, the measure combination $2,2$, i.e., $>$, is descending.
- The measure μ_1 is not enough to prove termination because $2,2$ is the only measure combination that is descending for cc_3, cc_3.

CCG Termination Correctness

Theorem 11.
For all pvso $\in \mathtt{PVS0}[Val]$, $scp\_termination(pvso)$ *if and only if* $ccg\_termination_{\mathcal{M}}(G_{pvso})$ *for some CCG* G_{pvso} *of pvso and family* $\mathcal{M}$ *of measures.*

- The number of cycles in a graph is finite. However, the number of circuits is potentially infinite.
- It's not enough to check for decreasing measure combinations in cycles (see [Ave15]).

Matrix Weighted Graphs

- Matrix Weighted Graphs is an effective technique to check for descending measure combinations in a CCG using an algebra over matrices [Ave15].
- Every edge in a CCG is labeled with a N$\times$N-matrix of values in $\{-1, 0, 1\}$, where N is the number of measures in $\mathcal{M}$.

Matrix Weighted Graph (MWG)

- Let G_{pvso} be a CCG of a PVS0 program *pvso* in $\mathtt{PVS0}[Val]$ and $\mathcal{M}$ be a family of N measures.
- A matrix weighted graph (MWG) W_{pvso} of a PVS0 program *pvso* in $\mathtt{PVS0}[Val]$ consists of a CCG G_{pvso} whose edges are correctly labeled by $\mathbb{N}\times\mathbb{N}$-matrices of values $\{-1, 0, 1\}$.

Correct Labels

$\mathbf{M}_{ab}$ is a correct label of an edge cc_a, cc_b in G_{pvso} if and only if for all $1 \leq i, j \leq N$ one of the following cases holds:

- $\mathbf{M}_{ab}(i,j) = 1$ only if for all $v_a, v_b \in Val$,

$$\texttt{eval\_conds}(pvso)(C_a, v_a) \ \texttt{AND} \ \varepsilon(pvso)(e_a, v_a, v_b) \ \texttt{IMPLIES} \ \mu_i(v_a) > \mu_j(v_b).$$

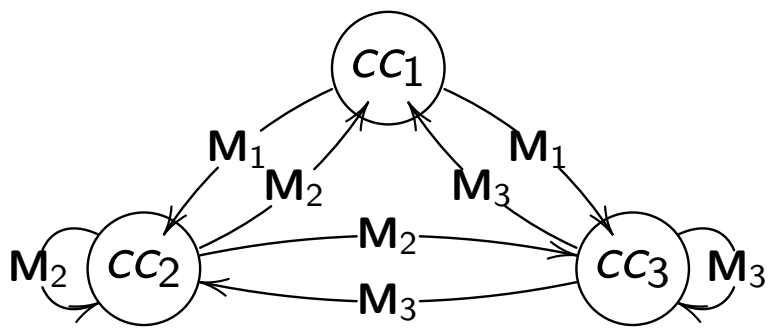
- $\mathbf{M}_{ab}(i,j) = 0$ only if for all $v_a, v_b \in Val$,

$$\texttt{eval\_conds}(pvso)(C_a, v_a) \ \texttt{AND} \ \varepsilon(pvso)(e_a, v_a, v_b) \ \texttt{IMPLIES} \ \mu_i(v_a) \geq \mu_j(v_b).$$

- $\mathbf{M}_{ab}(i,j) = -1$.

A MWG for Ackermann

- $cc_1 = (ack(m-1,1), m \neq 0 \wedge n = 0)$
- $cc_2 = (ack(m-1, ack(m, n-1)), m \neq 0 \wedge n \neq 0)$
- $cc_3 = (ack(m, n-1), m \neq 0 \wedge n \neq 0)$
- $\mu_1(m,n) = m$
- $\mu_2(m,n) = n$



$$\mathbf{M}_1 = \begin{bmatrix} 1 & 0 \\ -1 & -1 \end{bmatrix} \quad \mathbf{M}_2 = \begin{bmatrix} 1 & -1 \\ -1 & -1 \end{bmatrix}$$

$$\mathbf{M}_3 = \begin{bmatrix} 0 & -1 \\ -1 & 1 \end{bmatrix}$$

Basic Operations

- Let x, y be in $\{-1, 0, 1\}$.

$$x \times y = \begin{cases} -1 & \text{if } x = -1 \vee y = -1 \\ 1 & \text{if } x = 1 \vee y = 1 \\ 0 & \text{otherwise.} \end{cases}$$

$$x + y = \max(x, y).$$

- Matrix multiplication of values in $\{-1, 0, 1\}$ is defined as usual where addition and multiplication of values is defined as above.

Weight of Walk

- The weight of a walk $cc_{i_1}, \ldots, cc_{i_n}$ of a matrix weighted graph W_{pvso} is defined as $\Pi_{j=1}^{n-1} \mathbf{M}_{i_j i_{j+1}}$.
- A weight $\mathbf{M}$ is positive is there exists $1 \leq j \leq N$ such that $\mathbf{M}(i, i) > 0$.

MWG Termination Criterion

- Let W_{pvso} be a MWG of a PVS0 program *pvso* in $\texttt{PVS0}[Val]$ and $\mathcal{M}$ be a family of N measures for a well-founded relation $<$ over a type M.
- $\texttt{mwg\_termination}_{\mathcal{M}}(W_{pvso}$ holds if all circuits in W_{pvso} have a positive weight.
- Dutle's Procedure:
  - A sound and complete effective procedure to decide positive weight of all circuits in a CCG.
  - Formally verified in PVS.

Theorem 12.
For all pvso $\in \texttt{PVS0}[Val]$, $\texttt{ccg\_termination}_{\mathcal{M}}(G_{pvso})$ *for some CCG* G_{pvso} *if and only if* $\texttt{mwg\_termination}_{\mathcal{M}}(W_{pvso})$ *for some MWG* W_{pvso}.

Termination Analysis by CCG

- How to build a CCG?
- Which well-founded relation?
- Which family of measures?
- How to build the matrices?
- How to check for positive weight of all circuits in a CCG?
- How is CCG integrated into PVS?

How to Build a CCG

- The set of calling contexts is finite and can be extracted from the program by syntactic analysis.
- A fully connected CCG is sound (of course, the more edges the more inefficient the method).
- The theorem prover can be used to soundly remove edges from the graph, i.e., an edge cc_a, cc_b can be removed if

$$\begin{aligned} \vdash \forall(v_a, v_b : Val) : & \texttt{eval\_conds}(pvso)(C_a, v_a) \ \texttt{AND} \\ & \varepsilon(pvso)(e_a, v_a, v_b) \ \texttt{IMPLIES} \\ & \texttt{NOT eval\_conds}(pvso)(C_b, v_b). \end{aligned}$$

  can be discharged.

Which well-founded relation? Which family of measures?

- The order relation $<$ over natural numbers is a good starting point.
- Since *CCG* allows for a family of measures, it is sound to add as many measures as possible (of course the more measures the more inefficient the method).
- Predefined functions can be used, e.g., parameter projections (in the case of natural numbers), natural size of parameters (in the case of data types), maximum/minimum of parameters, etc. More complex recursions may need heuristics based on static analysis.
- Manolios and Vroon report that "[CCG] was able to automatically prove termination for over 98% of the more than 10,000 functions in the regression suite [of ACL2s]" [MV06].

How to Build the Matrices

- All edges starting in a given a calling context cc_a are labelled with the same matrix $\mathbf{M}_a$.
- To build a matrix $\mathbf{M}_a$ for the edges starting in cc_a, it is sound to set all its entries to -1.
- The theorem prover can be used to soundly reset the entries in $\mathbf{M}_a(i,j)$ to either 0 or 1 as follows,
  - If

$$\vdash \forall(v_a, v_b : Val) : \texttt{eval\_conds}(pvso)(C_a, v_a) \ \texttt{AND} \ \varepsilon(pvso)(e_a, v_a, v_b) \ \texttt{IMPLIES} \ \mu_i(v_a) > \mu_j(v_b),$$

    can be discharged then set $M_a(i,j)$ to 1.
  - If

$$\vdash \forall v_a, v_b : Val) : \texttt{eval\_conds}(pvso)(C_a, v_a) \ \texttt{AND} \ \varepsilon(pvso)(e_a, v_a, v_b) \ \texttt{IMPLIES} \ \mu_i(v_a) \geq \mu_j(v_b),$$

    can be discharged, then set $M_a(i,j)$ to 0.

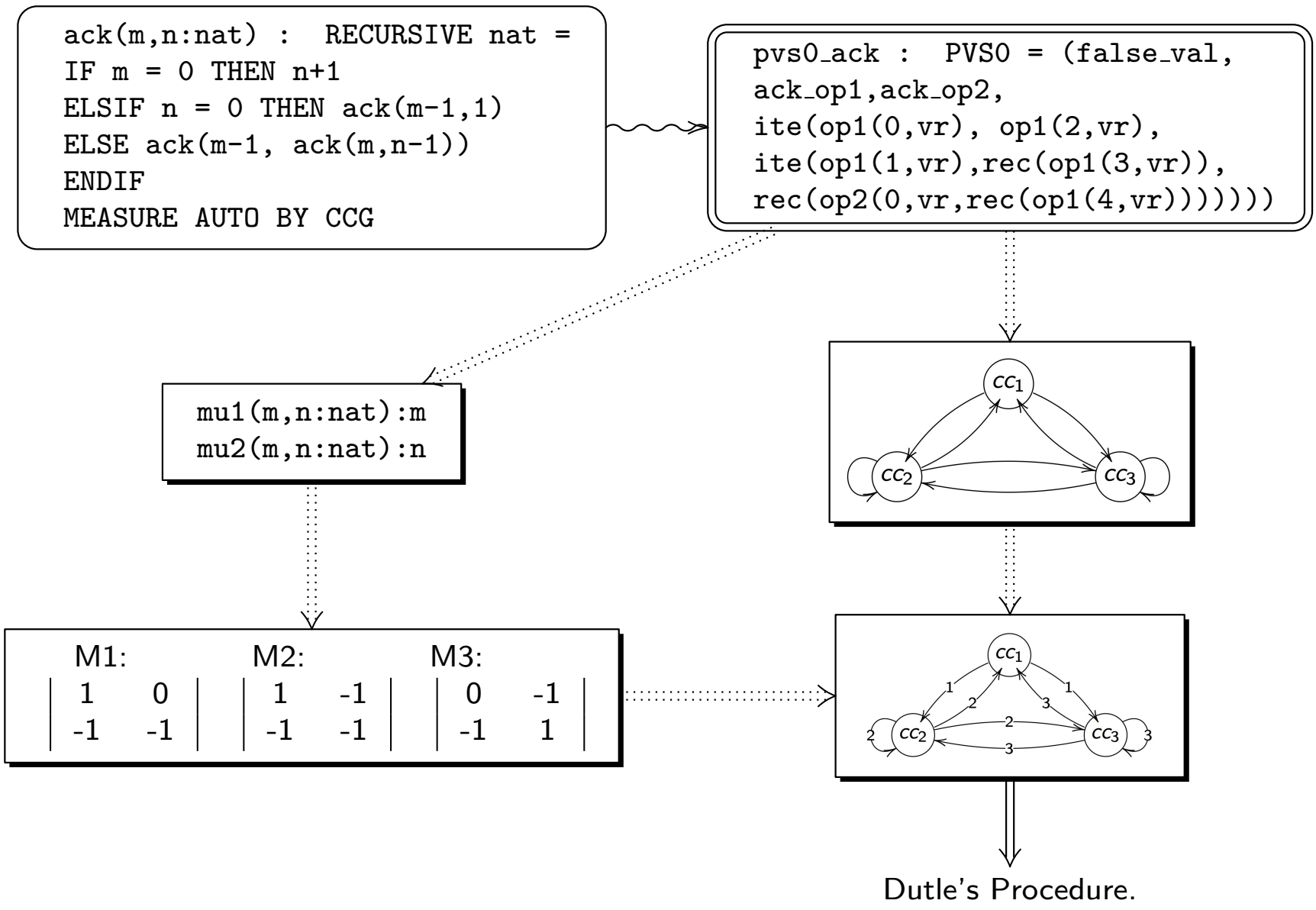
How to check for positive weight of all circuits in a CCG

Use Dutle's procedure!

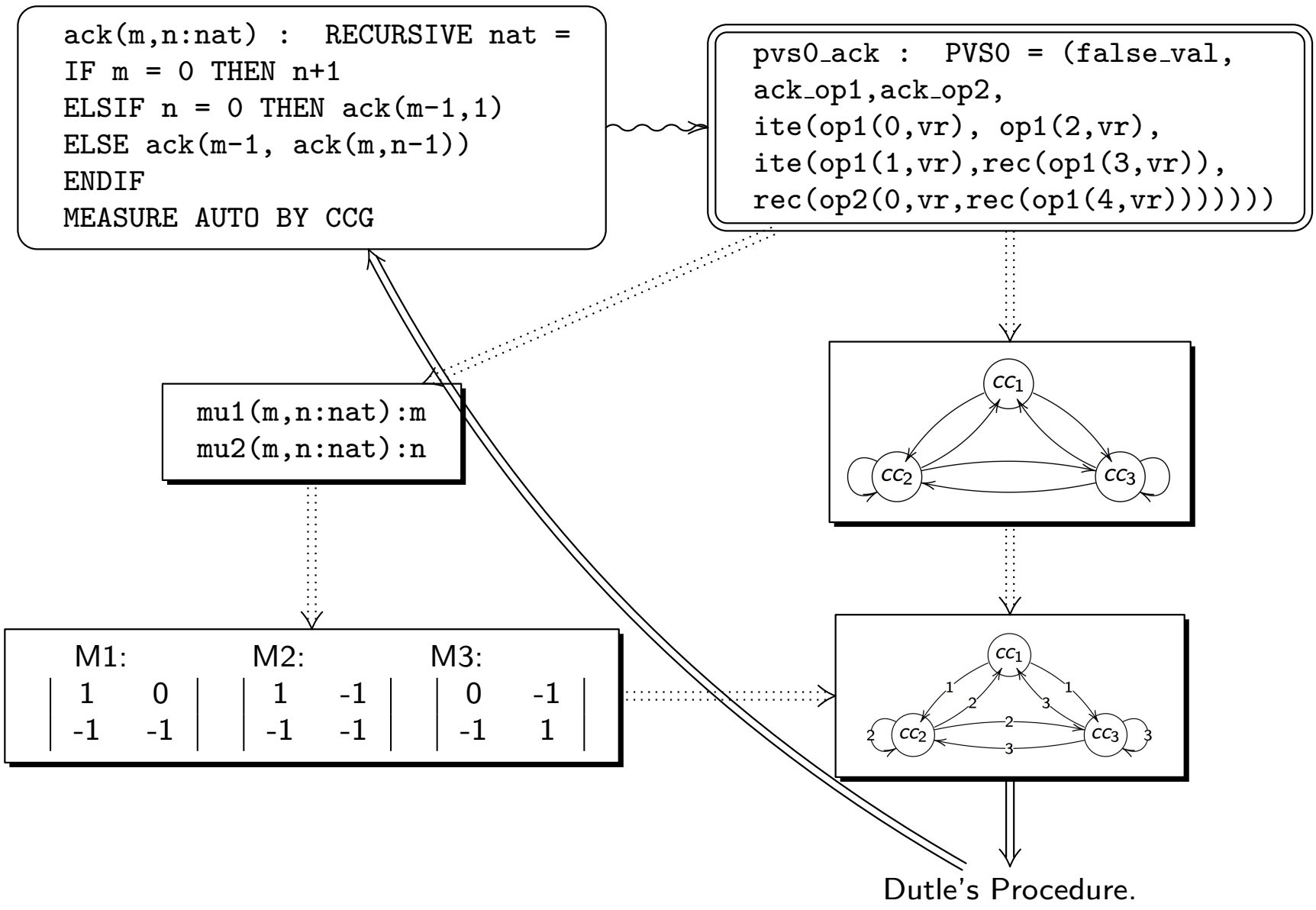
PVS Development

- The development presented in this lecture is fully formalized in PVS (including Dutle's procedure).
- For reference, see `CCG` and `PVS0` in NASA PVS Library (`https://github.com/nasa/pvslib`).
- In particular, it's formally verified that the following termination criteria are all equivalent: ε-termination, χ-termination, `pvs0_tcc_termination`, $\text{SCP}_{<}$, `scp_termination`, $\texttt{ccg\_termination}_{\mathcal{M}}$, $\texttt{mwg\_termination}_{\mathcal{M}}$, and Dutle's procedure.

Howe is CCG Being Integrated into PVS



Howe is CCG Being Integrated into PVS



PVS $\subseteq$ PVS0

By design, the PVS0 language can directly encode any PVS function f of type $T \to T$, where T is an arbitrary PVS type.

Lemma 13.
Let f be a PVS function of type $T \to T$. The program $mk\_pvs0(f) = (O_1, O_2, \bot, e_f)$ of type `PVS0`*, where $e_f = \texttt{op1}(0, \texttt{vr})$ and $O_1(0)(t) = f(t)$, satisfies the following properties:*

- $\mathit{terminating}(mk\_pvs0(f))$*, i.e., for any $t \in T$, $T_\varepsilon(mk\_pvs0(f), t)$.*
- $f = \mathit{pvs0\_eval}(mk\_pvs0(f))$*, i.e., for any $t \in T$, $f(t) = \mathit{pvs0\_eval}(mk\_pvs0(f))(t)$.*

PVS0 $\not\subseteq$ PVS

Some PVS0 programs cannot be embedded as PVS functions.

Lemma 14.
Let T be non-empty type. There is no PVS function f of type $T \to T$ such that for any $t \in T$, $\varepsilon(\Delta)(e_\Delta, t, f(t))$, where Δ is the PVS0 program defined in Lemma 4.

PVS0 $\subseteq$ PVS $\cup\{\diamondsuit\}$

However, any PVS0 program, even non-terminating ones, can be encoded as a PVS function of type $T \to T \cup \{\diamondsuit\}$.

Lemma 15.
Let pvso be a, possibly non-terminating, program of type `PVS0` *and* e_{pvso} *the* `PVS0` *expression of pvso. The PVS function*

$$f(t) := \texttt{IF } T_\chi(pvso, t) \texttt{ THEN } \chi(pvso)(e_{pvso}, t, \mu(pvso, t)) \\ \texttt{ELSE } \diamondsuit \texttt{ ENDIF },$$

satisfies the following property for any $t_i, t_o \in T$*:*

$$\varepsilon(pvso)(e_{pvso}, t_i, t_o) \text{ if and only if } f(t_i) = t_o.$$

An Oracle for PVS0 Programs

It is possible to define an oracle of type $\texttt{PVS0}[\texttt{PVS0}[\mathit{Val}]]$ that decides if a program of type $\texttt{PVS0}[\mathit{Val}]$ is terminating or not.

Theorem 16.
The program $\mathit{Oracle} = (O_1, O_2, \bot, e)$ *of type* $\texttt{PVS0}[\texttt{PVS0}[\mathit{Val}]]$, *where*

$$e = \mathit{mk\_pvs0}(\texttt{LAMBDA}(\mathit{pvso} : \texttt{PVS0}[\mathit{Val}]) : \texttt{IF } \mathit{terminating}(\mathit{pvso}) \texttt{ THEN } \top \texttt{ ELSE } \bot \texttt{ENDIF }),$$

with $\bot \neq \top$, *has the following properties.*

- *Oracle is a terminating* $\texttt{PVS0}[\texttt{PVS0}[\mathit{Val}]]$ *program, i.e., forall pvso of type* $\texttt{PVS0}[\mathit{Val}]$, $T_\varepsilon(\mathit{Oracle}, \mathit{pvso})$.
- *Oracle decides termination of any* $\texttt{PVS0}[\mathit{Val}]$ *program, i.e., for all pvso of type* $\texttt{PVS0}[\mathit{Val}]$,

$$\chi(\mathit{Oracle})(e, \mathit{pvso}, \mu(\mathit{Oracle}, \mathit{pvso})) = \top \text{ if and only if } T_\varepsilon(\mathit{pvso}).$$

An Oracle for PVS0 Programs

- This counterintuitive result is possible because PVS, in contrast to proof assistants based on constructive logic, allows for the definition of total functions that are non-computable, e.g.,

$$\texttt{LAMBDA}(\mathit{pvso} : \texttt{PVS0}[\mathit{Val}]) :$$
$$\texttt{IF terminating}(\mathit{pvso})\ \texttt{THEN}\ \top\ \texttt{ELSE}\ \bot\texttt{ENDIF}\ .$$

- These non-computable functions can be used in the construction of terminating PVS0 programs through the built-in operators.

Partial Recursive PVS0 Programs

Formalizing the notion of partial recursive functions in PVS0 requires to restrict the way in which programs are built.

1. The parametric type T is set to $\mathbb{N}$, i.e., $Val = \mathbb{N}$, where the number 0 represents the value false, i.e., $\bot = 0$. Any value different from 0 represents a true value, in particular $\top = 1$.
2. The built-in operators used in the construction of programs are restricted by a hierarchy of levels:
   - Operators in the first level can only be defined using
     - projections ($\Pi_1(x, y : \mathbb{N}) := x$ and $\Pi_2(x, y : \mathbb{N}) := y$),
     - successor ($succ(x : \mathbb{N}) : x + 1$), and
     - greater or equal than functions.
   - Operators in higher levels can only be constructed using programs from the previous level.

Partial Recursive PVS0 Programs

A hierarchy of PVS0[*nat*] program levels is formalized by the following predicate.

$$
\begin{array}{l}
\mathit{pvs0\_level}(n)(O_1, O_2, \bot, e) := \\
\quad \texttt{IF } n = 0 \texttt{ THEN } O_1 = \langle \mathit{succ} \rangle \ \wedge O_2 = \langle \Pi_1, \Pi_2, \mathit{ge} \rangle \\
\quad \texttt{ELSE } (\ \ \exists p' \in \texttt{PVS0}[\mathbb{N}] \ : \ \mathit{pvs0\_level}(n-1)(p') \ \wedge \\
\qquad\qquad \texttt{LET } ({O_1}', {O_2}', \bot', e') = p', \ {l_1}' = |{O_1}'| \texttt{ IN} \\
\qquad\qquad\ |O_1| = {l_1}' + 1 \ \wedge \\
\qquad\qquad\ (\ \forall i \in \mathbb{N} : i < {l_1}' \Rightarrow O_1(i) = {O_1}'(i) \ ) \wedge \\
\qquad\qquad\ (\ \forall v \in \mathbb{N} : \varepsilon(p')(e', v, O_1({l_1}')(v)) \ ) \ ) \wedge \\
\qquad\quad (\ \ \exists p' \in \texttt{PVS0}[\mathbb{N}] : \ \mathit{pvs0\_level}(n-1)(p') \ \wedge \\
\qquad\qquad \texttt{LET } ({O_1}', {O_2}', \bot', e') = p', {l_2}' = |{O_2}'| \texttt{ IN} \\
\qquad\qquad\ |O_2| = {l_2}' + 1 \ \wedge \\
\qquad\qquad\ (\ \forall i \in \mathbb{N} : i < {l_2}' \Rightarrow O_2(i) = {O_2}'(i) \ ) \wedge \\
\qquad\qquad\ (\ \forall v_1, v_2 \in \mathbb{N} : \varepsilon(p')(e', \kappa_2(v_1, v_2), O_2({l_2}')(v_1, v_2)) \ ) \ ),
\end{array}
$$

where the function κ_2 is an encoding of pairs of natural numbers onto natural numbers.

Undecidability of the Halting Problem

- The type `PartialRecursive` is defined to be a subtype of `PVS0`$[\mathbb{N}]$ containing all the programs *pvso* such that there is a natural n for which *pvs0_level*(n)(*pvso*) holds.
- `Computable` is a subtype of `PartialRecursive` containing those elements that are also terminating.
- `PartialRecursive`, and thus `Computable`, are enumerable.

Theorem 17.
There exists a PVS function of type $\mathbb{N} \to$ `PartialRecursive` *that is surjective.*

- The inverse of this surjective function, denoted as κ_P, is an injective function of type `PartialRecursive` $\to \mathbb{N}$.

Undecidability of the Halting Problem

Theorem 18.
There is no program oracle $= (O_1, O_2, \bot, e_o)$ *of type* `Computable` *such that for all pvso* $= (O_1', O_2', \bot, e)$ *of type* `PartialRecursive` *and for all* $n \in \mathbb{N}$,

$$T_\varepsilon(pvso, n) \text{ if and only if } \neg\varepsilon(oracle)(e_o, \kappa_2(\kappa_P(pvso), n), \bot).$$

Proof.
See [RMAR+18]. □


Thomas Arts and Jürgen Giesl.
Termination of term rewriting using dependency pairs.
Theor. Comput. Sci., 236(1-2):133–178, 2000.


Thomas Arts.
Termination by absence of infinite chains of dependency pairs.
In *Proceedings Trees in Algebra and Programming - CAAP 1996, 21st International Colloquium*, volume 1059 of *Lecture Notes in Computer Science*, pages 196–210. Springer, 1996.


Andréia B. Avelar.
Formalização da automação da terminação através de grafos com matrizes de medida.
PhD thesis, Universidade de Brasília, Departamento de Matemática, Brasília, Distrito Federal, Brasil, 2015.
In Portuguese.


Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl.
Termination of Isabelle Functions via Termination of Rewriting.

In *Proceedings Interactive Theorem Proving - Second International Conference, ITP 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2011.


Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram.
The size-change principle for program termination.
In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2001.


Panagiotis Manolios and Daron Vroon.
Termination analysis with calling context graphs.
In *Lecture Notes in Computer Science*, volume 4144, pages 401–414. Springer, 2006.


Thiago Mendonça Ferreira Ramos, César Muñoz, Mauricio Ayala-Rincón, Mariano Moscato, Aaron Dutle, and Anthony Narkawicz.
Formalization of the undecidability of the halting problem for a functional language.

In Lawrence S. Moss, Ruy de Queiroz, and Maricarmen Martinez, editors, *Logic, Language, Information, and Computation*, volume 10944 of *Lecture Notes in Computer Science*, pages 196–209, Oxford, UK, July 2018. Springer Berlin Heidelberg.

René Thiemann and Jürgen Giesl.
Size-change termination for term rewriting.
In Robert Nieuwenhuis, editor, *Proceedings Rewriting Techniques and Applications, 14th International Conference, RTA 2003*, volume 2706 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2003.

Alan M. Turing.
On computable numbers, with an application to the Entscheidungsproblem.
Proc. of the London Mathematical Society, 42(1):230–265, 1937.

Alan M. Turing.
Checking a large routine.

In Martin Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

Akihisa Yamada, Christian Sternagel, René Thiemann, and Keiichirou Kusakari.
AC dependency pairs revisited.
In *Proceedings 25th EACSL Annual Conference on Computer Science Logic, CSL 2016*, volume 62 of *LIPIcs*, pages 8:1–8:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.