Compiling Untyped λ-calculus to Lower-level Code by Game Semantics and Partial Evaluation

- **Daniil Berezun**
  State University of St. Petersburg
- **Neil D. Jones**
  DIKU, University of Copenhagen (prof. emeritus)

NORMALISATION BY TRAVERSAL OF SIMPLY-TYPED λ-CALCULUS

- **Implicit** in PCF research (Ong/Abramsky/. . . 1990s)
- **explicit** in ong [1]:
  1. Convert typed λ-expression M into long form M^{lf}
  2. Traverse the syntax nodes of M^{lf}:
  3. Traversal builds a history h of the normalisation of M
  4. $h \in H = (Subexp(M) \times H)^*$

Origins: research on full abstraction for PCF.

A PROGRAMMING PERSPECTIVE

The game semantics for PCF amounts to an executable implementation of PCF, i.e., a PCF interpreter.

An observation: this implementation uses none of the usual machinery:

parameters by closures or thunks; bindings by environments.

(Instead, all is done by tokens and back pointers).

A traversal is a

- **sequence of subexpressions of M. This is a finite set, whose elements we will call tokens**

  (think: M = program, tokens = program points)

- **each token in a traversal may have a back pointer (aka. justifier).**

ONG'S NORMALISATION PROCEDURE ONP

- applies to simply-typed λ-expressions
- begins by translating M into η-long form
- effect: head linear reduction of M, one step at a time
- Correctness: proven by game semantics and category theory. Strongly based on M's types.

Properties of the normalisation procedure:

Uses no β-reduction: just take a walk through subexpressions of M.

While running, ONP does not use the types of M at all.

OUR WORK

- Extend Ong [1] to the untyped λ-calculus. We use two kinds of back pointers.

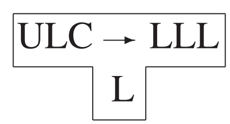
- Call the this algorithm UNP. Concretely, UNP can be programmed in HASKELL or SCHEME.

  Partial evaluation: we construct low-level code for λ-expression$M$ by partial evaluation:

$$[\![spec]\!](UNP, M) = \text{Target code for } M$$

- More: one can generate a compiler from UNP by partial evaluation:

$$[\![cogen]\!](UNP) \in$$

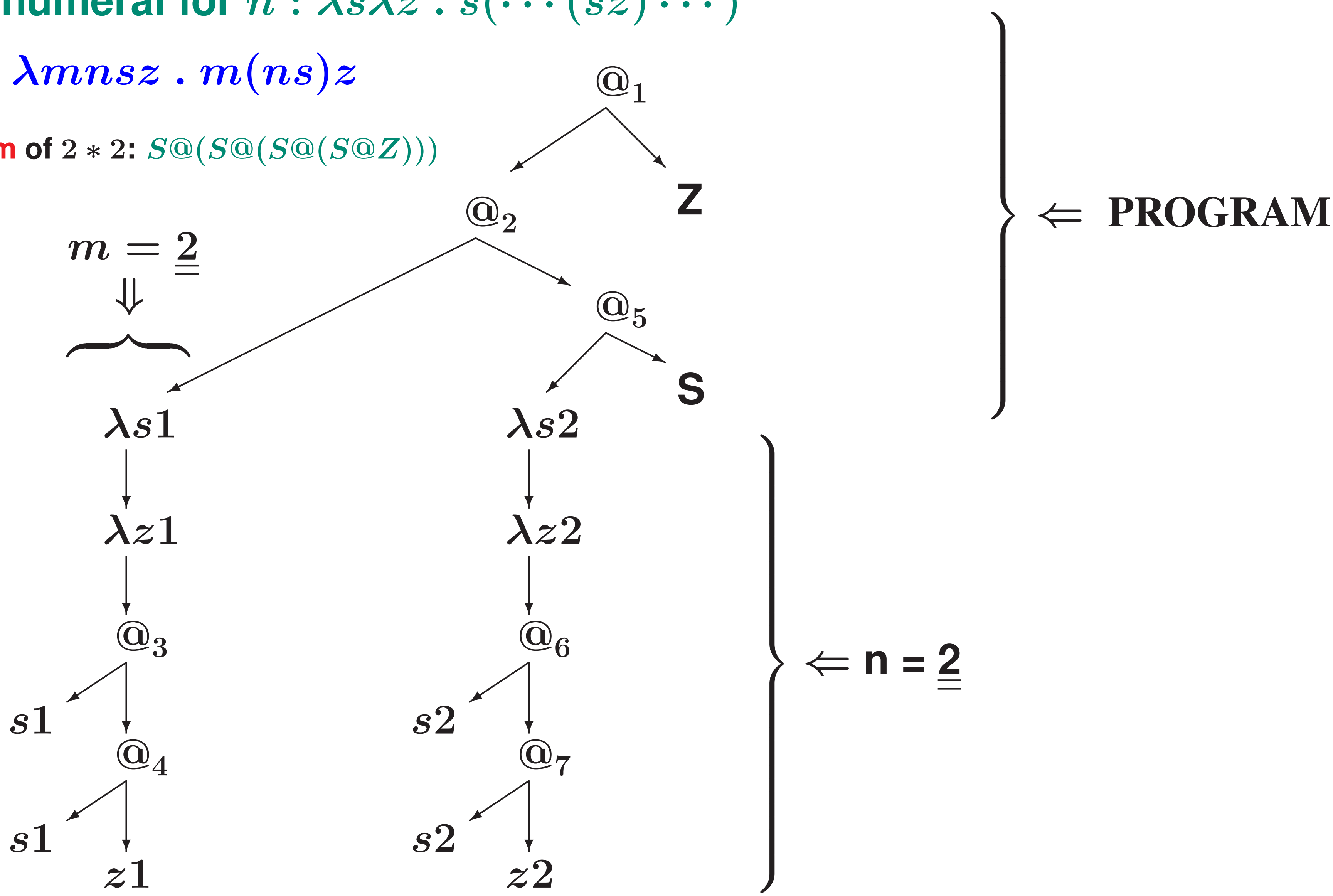


MULTIPLYING CHURCH NUMERALS: $2 * 2 = \underline{2}(\underline{2}S)Z$

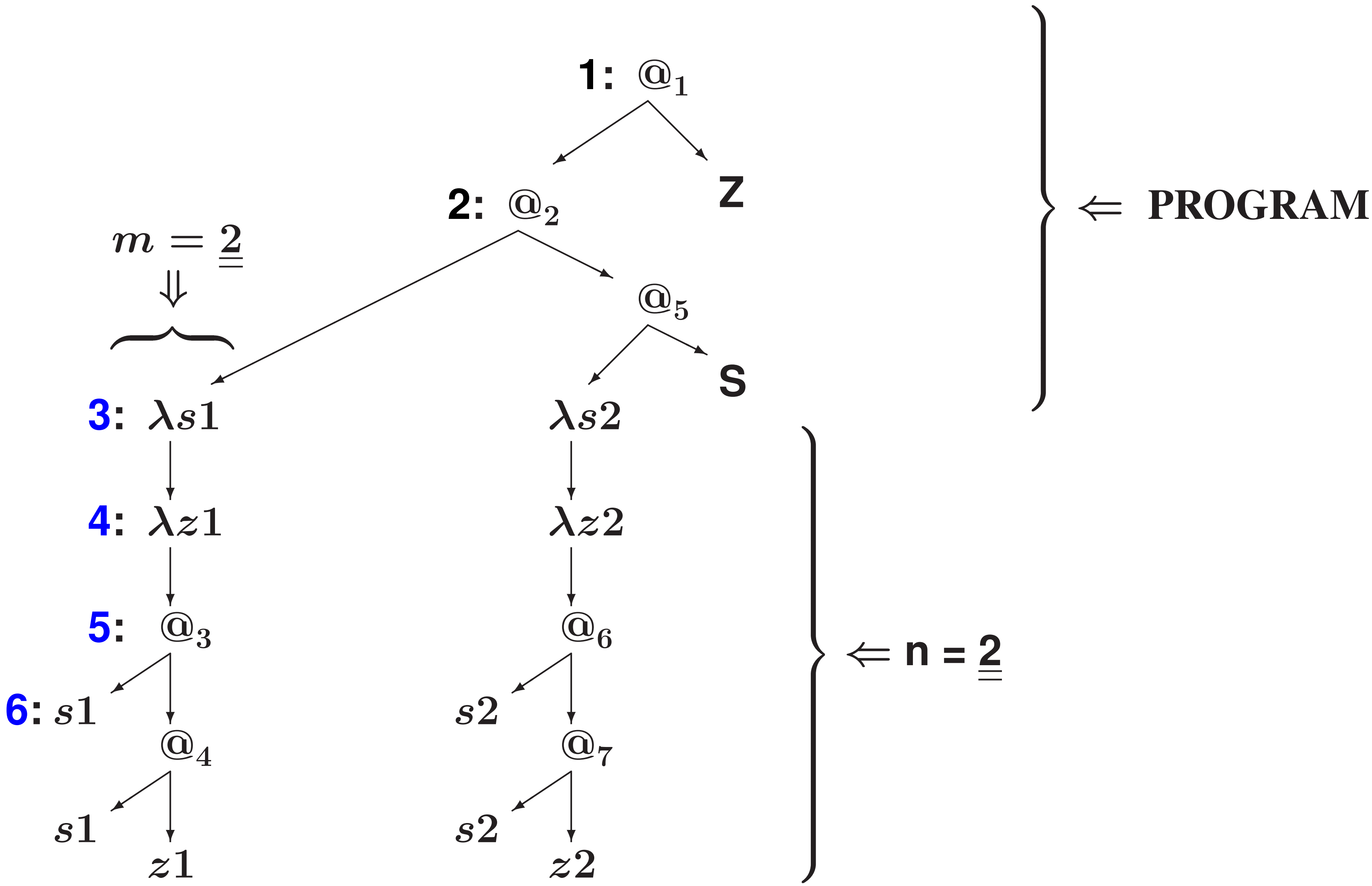
Church numeral for $n : \lambda s \lambda z \,.\, s(\cdots(sz)\cdots)$

$mul = \lambda mnsz \,.\, m(ns)z$

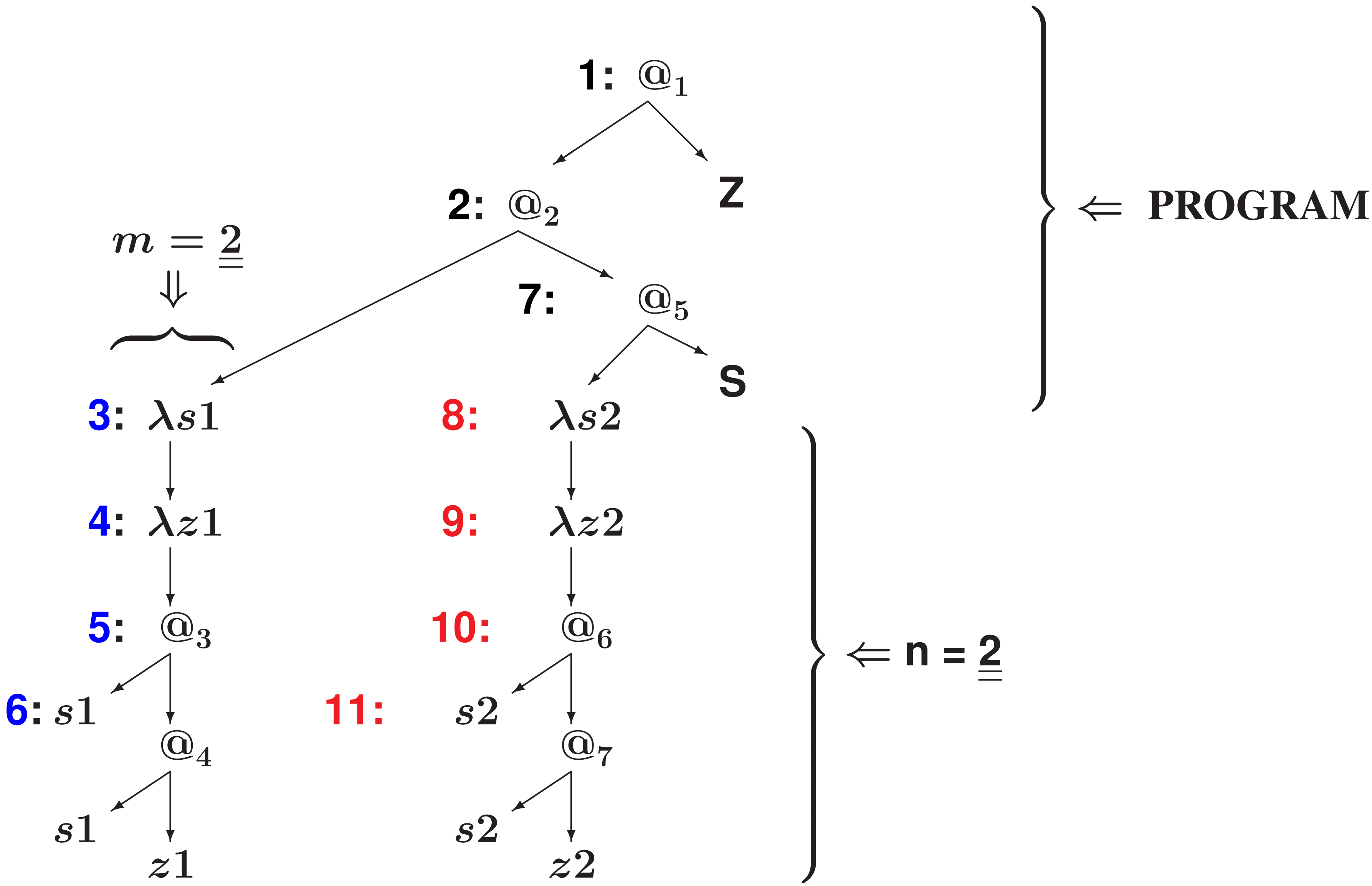
Normal form of $2 * 2$: $S@(S@(S@(S@Z)))$



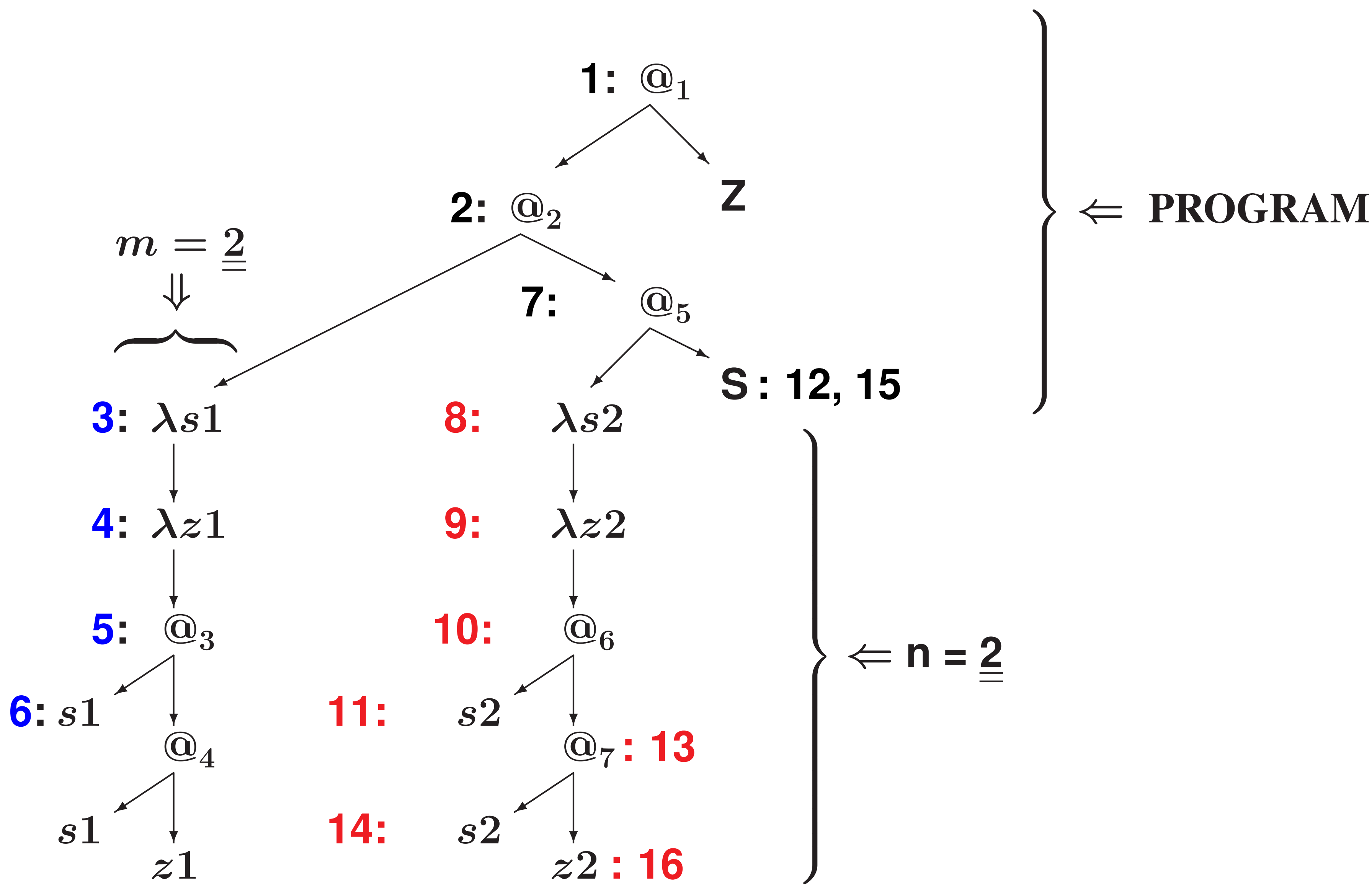
GAME: DATA $m = \underline{2}, n = \underline{2}$ VERSUS PROGRAM: STEPS 1–6



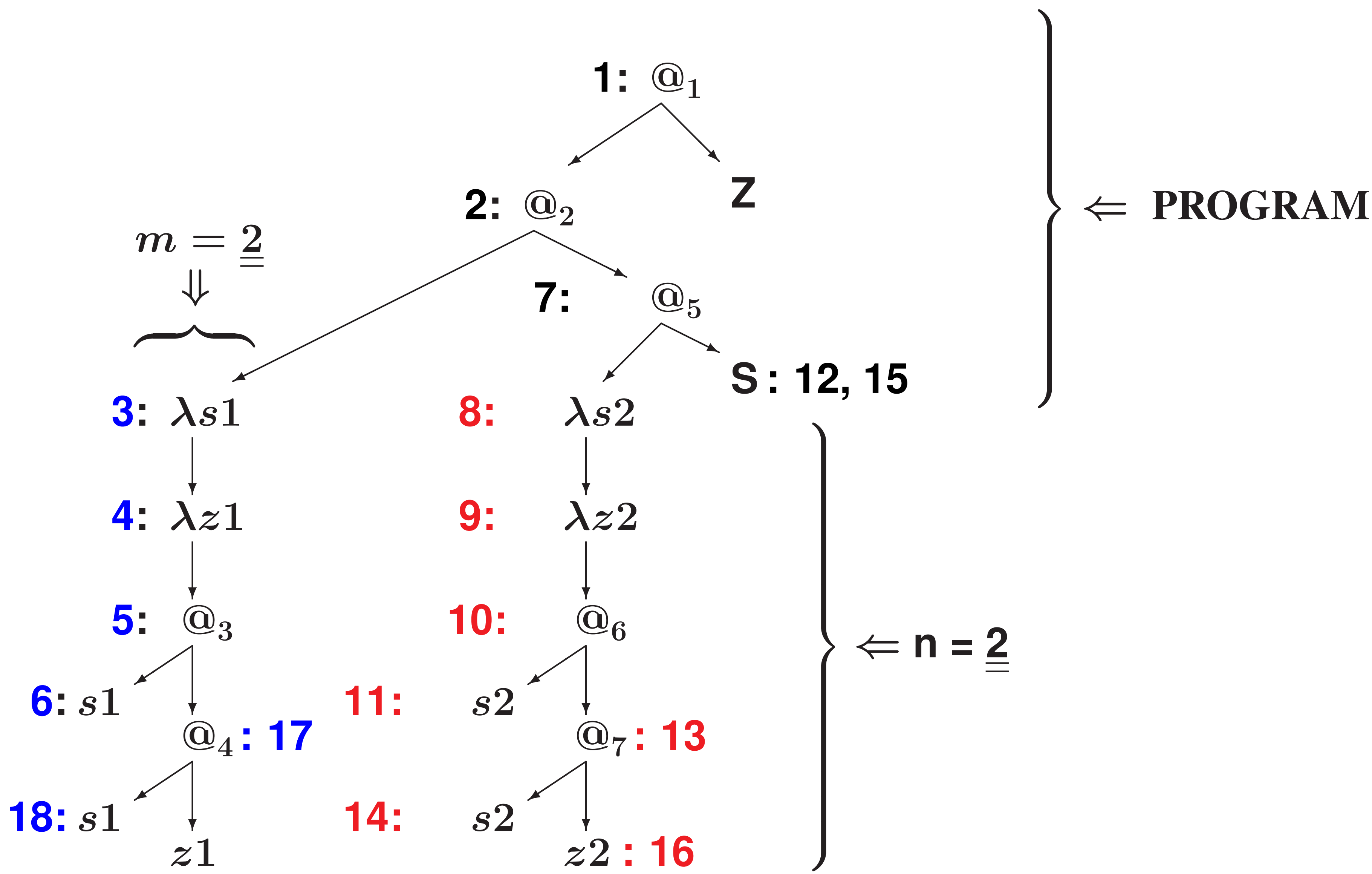
TRAVERSAL OF $2 * 2 = \underline{2}(\underline{2}S)Z$: STEPS 7–11



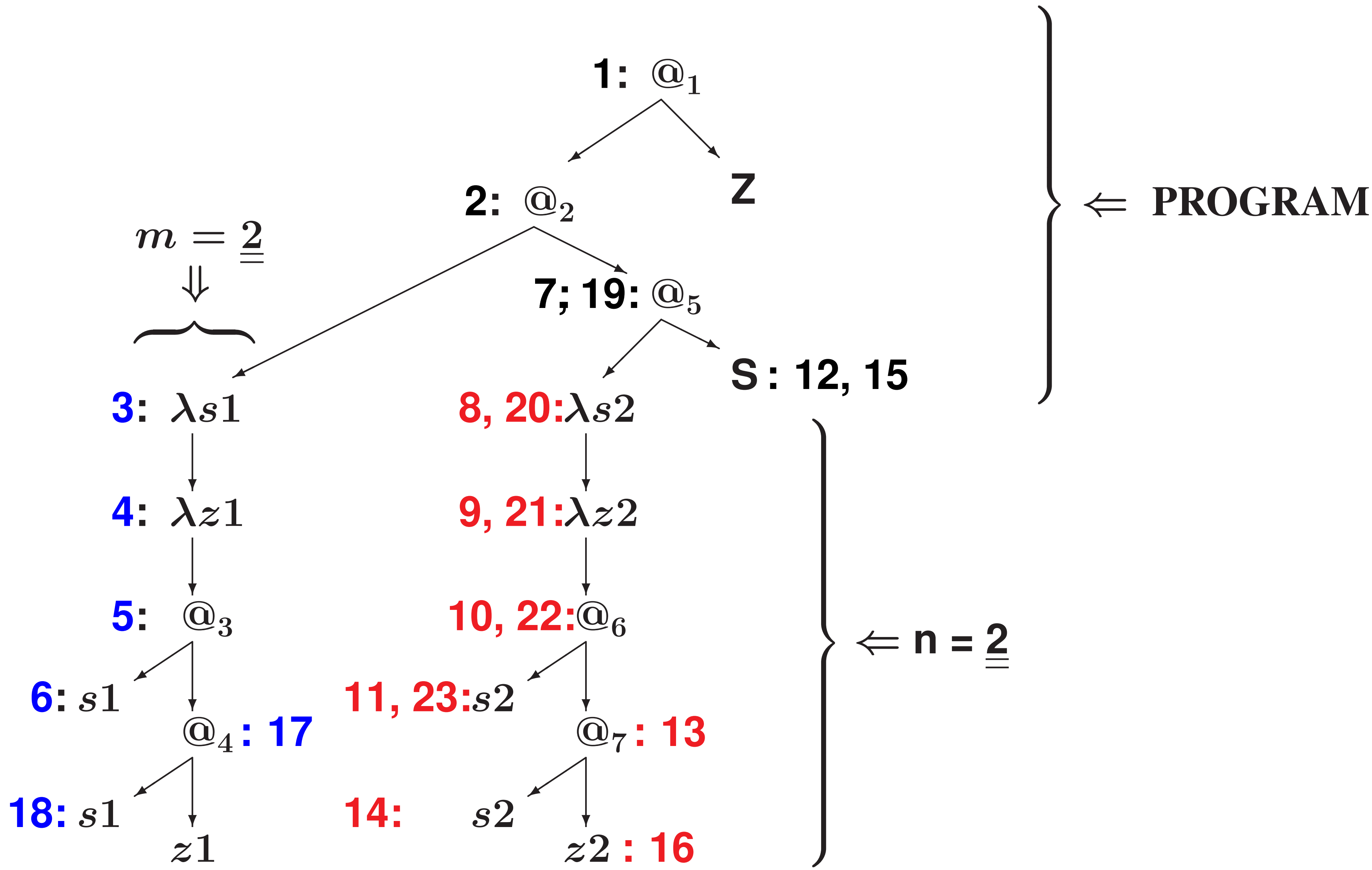
TRAVERSAL OF $2 * 2 = \underline{2}(\underline{2}S)Z$: STEPS 12–16



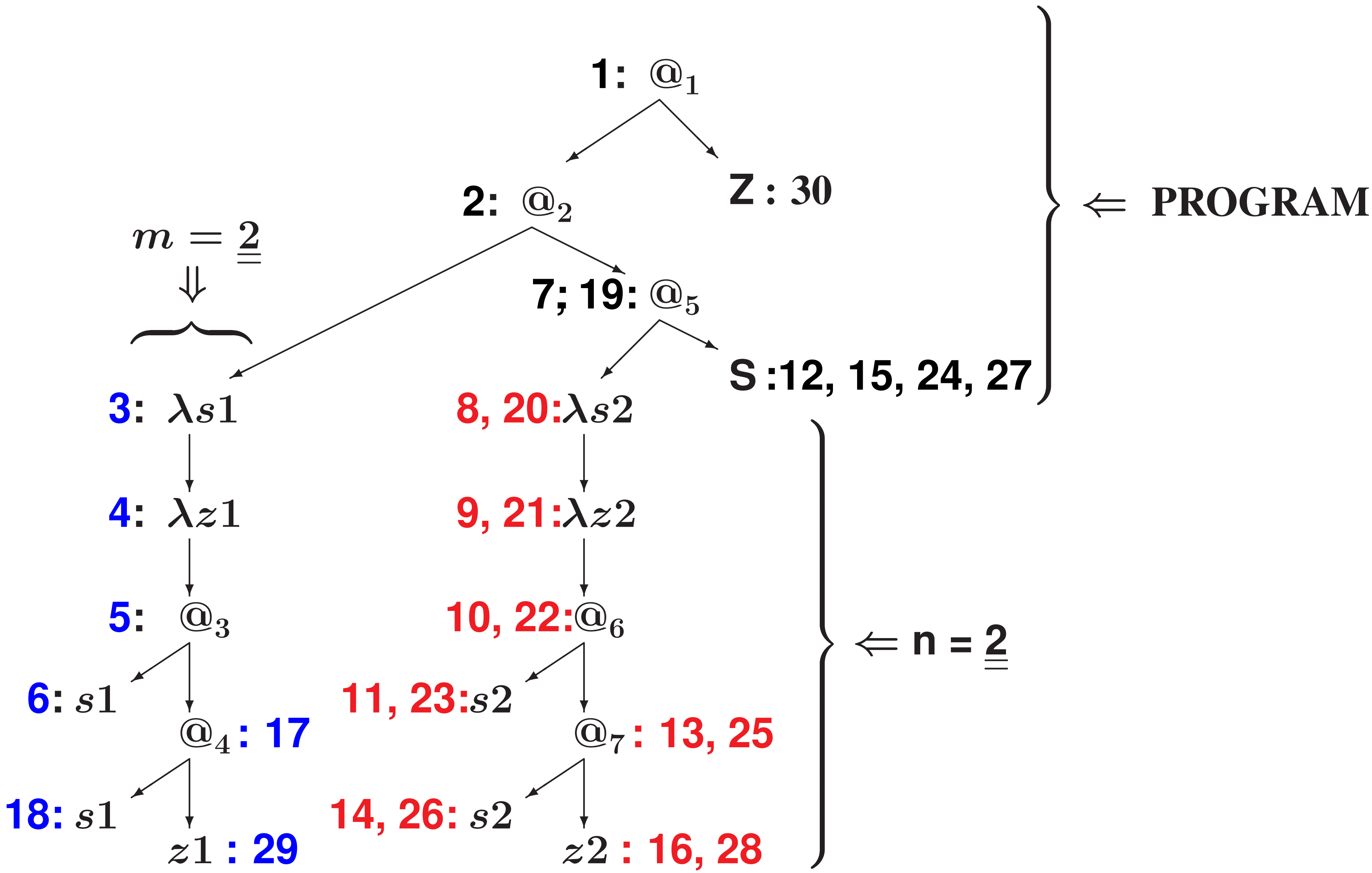
TRAVERSAL OF $2 * 2 = \underline{2}(\underline{2}S)Z$: STEPS 17–18



TRAVERSAL OF $2 * 2 = \underline{2}(\underline{2}S)Z$: STEPS 19–23



TRAVERSAL OF $2 * 2 = \underline{2}(\underline{2}S)Z$: STEPS 24–30



HOPS, SKIPS AND JUMPS: A CANONICAL TRAVERSAL ORDER

How on earth did we select the right node visit sequence ?

There are many possibilties, mostly wrong!

We develop several semantics.

- **Semantics 1 is classical β-reduction (a deterministic version)**
- **Semantics 5 resembles Ong's, with no environments, thunks, etc. but two kinds of back pointers.** **Leftmost head linear reduction**
- **All traverse subexpressions of M in the same order**

All the semantics achieve the canonical traversal order.

How is it defined? Mark the subexpression occurrences in M. Then trace their order during the complete leftmost head β-reduction.

STEPWISE DEVELOPMENT OF UNP

Semantics 1: A classical β-reduction semantics.

Semantics 2: An environment semantics as in functional programming.

Semantics 3: Environment-based but tail recursive. Realise nested evaluator calls by data structures.

Semantics 4: First history semantics. Implement the control data by back pointers into the computational history.

Semantics 5: Final history semantics. Implement the environments by back pointers into the computational history.

This history records the normaliser calls done until now (with argument values). Net effect: Semantics 5 is

$$UNP \in \boxed{\begin{array}{c}\Lambda\\ L\end{array}}$$

UNP is a first-order program.

A LITTLE MORE DETAIL

- **Classical reduction: needs a flag to avoid reducing e_0 twice in an application $(\lambda x.e_0)@e_2$.**
- **Environment semantics: $\rho \in \mathit{Env} = \mathit{Variable} \rightharpoonup \mathit{Exp} \times \mathit{Env}$. Two excerpts:**

$$[\![x]\!]\rho \quad = \quad \text{let } (e_0, \rho_0) = \rho(x) \text{ in } [\![e_0]\!]\rho_0$$

$$[\![e_1@e_2]\!]\rho \quad = \quad \text{let } (\lambda x.e_0, \rho_0) = [\![e_1]\!]\rho \text{ in } [\![e_0]\!]\rho_0[x \mapsto (e_2, \rho)]$$

- **Environment semantics is not compositional, but it is semi-compositional. This means:**

  in any call $[\![e]\!]\rho$ that occurs while evaluating λ-expression M, argument e will be a subexpression of M.

(This is good for compilation and partial evaluation.)

CONTINUATIONS AND DEFUNCTIONALISATION

Goal: Semantics 3 = tail-recursive version of Semantics 2. Techniques: well-known, e.g. John Reynolds' Definitional interpreters paper.

- **Continuations: modify Semantics 2 to have linear control flow.**

  Defunctionalisation: then replace the continuation functions by data structures.

- **Example of net effect: replace**

  $$[\![e_1 @ e_2]\!]^2 \rho \; = \; \text{let } (\lambda x.e_0, \rho_0) = [\![e_1]\!]^2 \rho \text{ in } [\![e_0]\!]^2 \rho_0[x \mapsto (e_2, \rho)]$$

  by:

  $$[\![e_1 @ e_2]\!]^3 \, \rho \, k \; = \; [\![e_1]\!]^3 \, \rho \, \langle Kapp \; e_2 \, \rho \, k \rangle$$

  plus:

  $$applycont \, \langle Kapp \; e_2 \, \rho \, k \rangle \; e_0 \, \rho_0 = [\![e_0]\!]^3 \, \rho_0[x \mapsto (e_2, \rho)] \; k$$

AND THE REST IS HISTORY...

Semantics 4:

- Replace the continuation argument k by a history h.

- h is a accumulative trace that remembers

  which semantic functions were called with which arguments?.

$$h \in H = (\mathit{Exp} \times \mathit{Env} \times H)^*$$

- What's the point? We can replace a continuation data structure such as $\langle \mathit{Kapp}\; e_2\; \rho\; k \rangle$ by a pointer to the time at which it was created (call it t).

  If you are given a back pointer as value of t, you can find the parts that $\langle \mathit{Kapp}\; e_2\; \rho\; k \rangle$ was built from in the history.

- Effect: save the time and space needed to build the continuation data.

- However this has a cost: keeping the history available for access.

THE LAST STEP

Semantics 5:

- Replace the environment ρ in Semantics 4 by a back pointer into the history h.
- Same idea, but a separate pointer is needed.
- A difference from Semantics 2-3-4:

  The value of a variable x is found,

  - not by applying a single function ρ, but
  - by following a chain of back pointers, to locate the place where x was last bound.
- Effect: all of the normaliser's arguments are now first-order.

PARTIAL EVALUATION, BRIEFLY

A partial evaluator is a **program specialiser**. Defining property of $spec$:

$$\forall p \in Programs \, . \, \forall s, d \in Data \, . \, [\![[\![spec]\!](p, s)]\!](d) = [\![p]\!](s, d)$$

- Program speedup by **precomputation**. Applications: **compiling**, and **compiler generation** (from an **interpreter**, and by **self-applying** $spec$).
- Given program p and "**static**" data s, $spec$ builds a ***residual program*** $p_s \stackrel{def}{=} [\![spec]\!](p, s)$.
- When run on any remaining "**dynamic**" data d, residual program p_s computes **what p would have computed on both** data inputs s and d.
- Net effect: a ***staging transformation***: $[\![p]\!](s, d)$ is a **1 stage** computation; but $[\![[\![spec]\!](p, s)]\!](d)$ is a **2 stage** computation.
- **Well-known in recursive function theory, as the S-1-1 theorem.**
- **Partial evaluation = engineering the S-1-1 theorem on real programs.**

THE LOW-LEVEL LANGUAGE LLL

LLL is a tiny tail recursive first-order functional language. Essentially a machine language with a heap. Functional version of WHILE in book:

Computability and Complexity from a Programming Perspective

SYNTAX

```
program ::=  f1 x = e1   ...   fn x = en

e       ::=  x      | f e
         |   token | case e of token1 -> e1 ... tokenn -> en
         |   (e,e) | case e of (x,y) -> e
         |   []     | case e of [] -> e x:y -> e
x       ::= variable
token   ::= an atomic symbol (from a fixed alphabet)
```

Variables have SIMPLE TYPES (not depending on M!):

```
tau ::=  Token   |  tau x tau   |   [ tau ]
```

A token, or a product type, has a static structure, fixed for any one LLL program. A list type `[tau]` (dynamic) has constructors `[]` and `:`.

HOW TO PARTIALLY EVALUATE NP (IN PROGRAM FORM) WITH RESPECT TO STATIC λ-EXPRESSION M ?

1. Annotate parts of NP as either static or dynamic. Variables ranging over
   (a) tokens are static, i.e., λ-expressions (subexpressions of M);
   (b) back pointers are dynamic;
   (c) so the traversal being built is dynamic too.
2. Classify data 1a as static (there are only finitely many)
3. Classify data 1b, 1c as dynamic (there are unboundedly many)
4. Computations in NP are either unfolded (done at PE time) or residualised (runtime code is generated to do them at stage 2)
   - Perform fully static computations at partial evauation time.
   - Operations to build or test a traversal: generate residual code.

THE RESIDUAL PROGRAM NP$_M = [\![spec]\!]$ NP M

If NP is semi-compositional:

Any recursive NP call has a substructure of M as argument.

Then:

- The partial evaluator can do, at specialisation time,

  all of the NP operations that depend only on M

- NP$_M$ contains “residual code”:
  - operations to extend the traversal; and
  - operations to follow back pointers

- NP$_M$ performs no operations at all on lambda expressions (!)

- Subexpressions of M will appear, but are only used as tokens:

  Tokens are indivisible, only used for equality comparisons with other tokens

AN OLD DREAM:
SEMANTICS-DIRECTED COMPILER GENERATION

(Just a wild idea for now, needs much more thought and work.)

Idea: specify the semantics of a subject programming language

(e.g., call-by-value λ-calculus, imperative languages, etc.)

by mapping source programs into LLL.

A "gedankeneksperiment", to get started:

Express the semantics of Λ by semi-compositional semantic rules without variable environments, thunks, etc:

$$[\![\ ]\!]^{\Lambda} : \Lambda \rightarrow \text{LLL}$$

Expectations/hopes:

- Reasonably many programming languages can be specified this way
- A generalising framework: compiling, optimisation,... tasks can all be reduced to questions and algorithms concerning LLL programs

TOWARDS SEPARATING PROGRAMS FROM DATA IN Λ

1. An idea: formalise a computation of λ-expression M on input d as a two-player game between the LLL-codes for M and d.

2. An example: `mul`, usual λ-calculus definition on Church numerals.

3. Loops appear from out of nowhere:
   - Neither `mul` nor the data contain loops;
   - but `mul` is compiled into an LLL-program with two nested loops.
   - Expect: can do the computation entirely without back pointers.

4. Current work: express such program-data games in a *communicating* version of LLL. A lead: apply traditional methods for compiling *remote function calls*.

5. Next step: optimise LLL. Remove all inessential bits of the traversal.

6. Think about complexity and data-flow analysis of such programs.

SOME RELATED WORK

References

[1] Luke Ong. Normalisation by traversals. *CoRR*, abs/1511.02629, 2015.

[2] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

[3] William Blum and Luke Ong. A concrete presentation of game semantics. In *Galop:Games for Logic and Programming Languages*, 2008.

[4] R. P. Neatherway, S. J. Ramsay, and C.-H. Luke Ong. A traversal-based algorithm for higher-order model checking. In *ICFP*, 2012.

[5] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 2000.

[6] Neil D. Jones, editor. *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*. Springer, 1980.