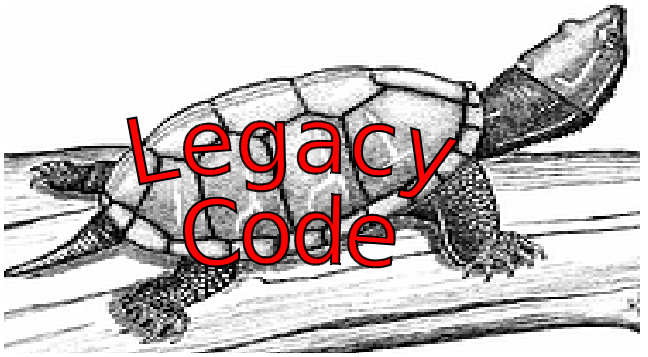
Semantics-based reverse engineering of data models from programs

Komondoor V Raghavan
IBM India Research Lab

(with G. Ramalingam, J. Field, et al)

Understanding legacy software

- Common scenario
  - huge existing legacy code base
  - building on top of existing code
  - transforming existing code
  - integrating legacy systems



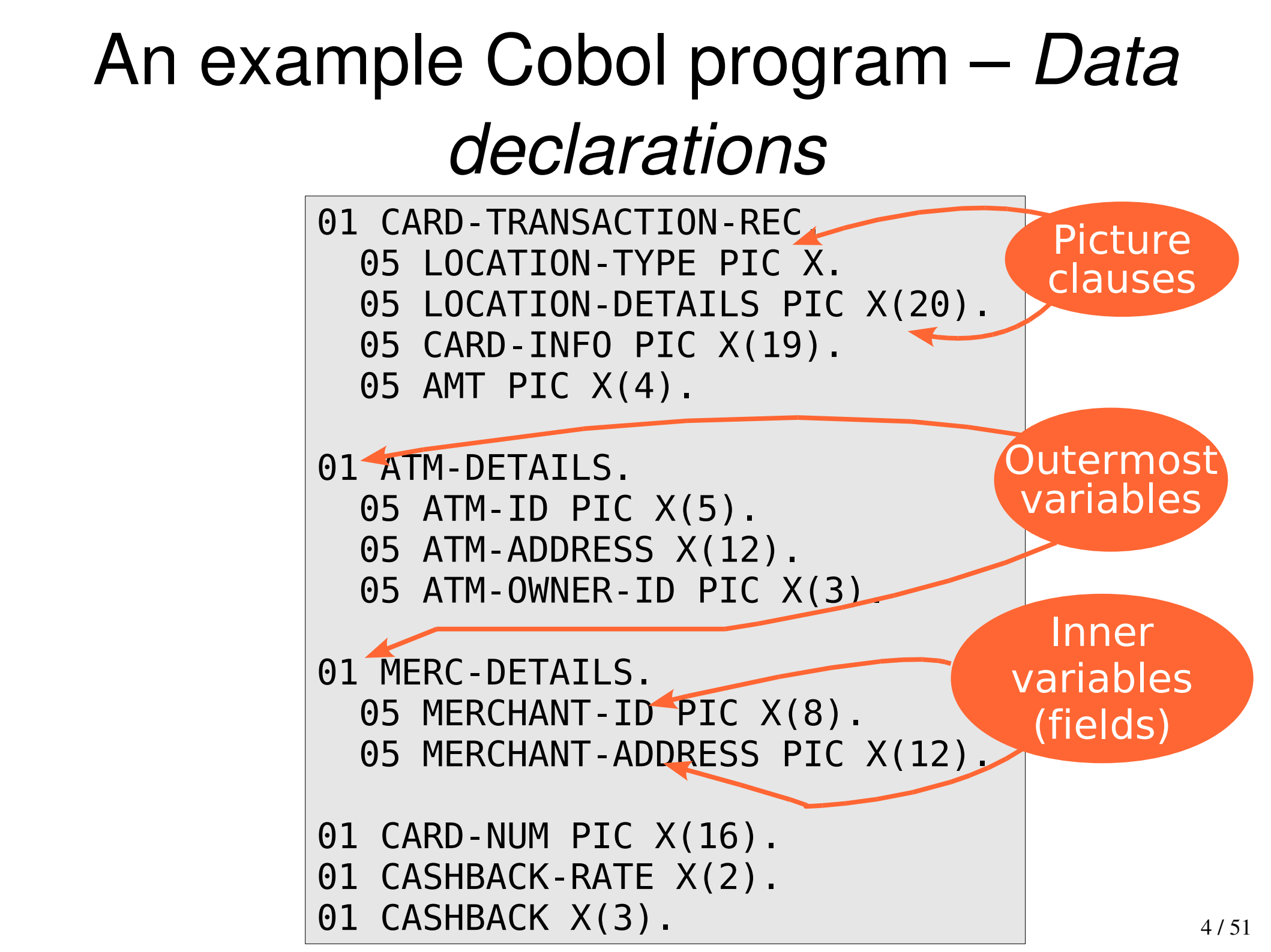
- Legacy code can be surprisingly hard to work with
  - lack of documentation and understanding of existing code
- Need tools to help understand legacy code

Reverse engineering data models

- Goal: Reverse engineer a *logical data model* of a given (legacy) program
  - or Type Inference
  - focused on weakly-typed languages like Cobol
- Understanding logical structure of data is key to program understanding
- A logical data model can assist in common legacy transformation and maintenance tasks

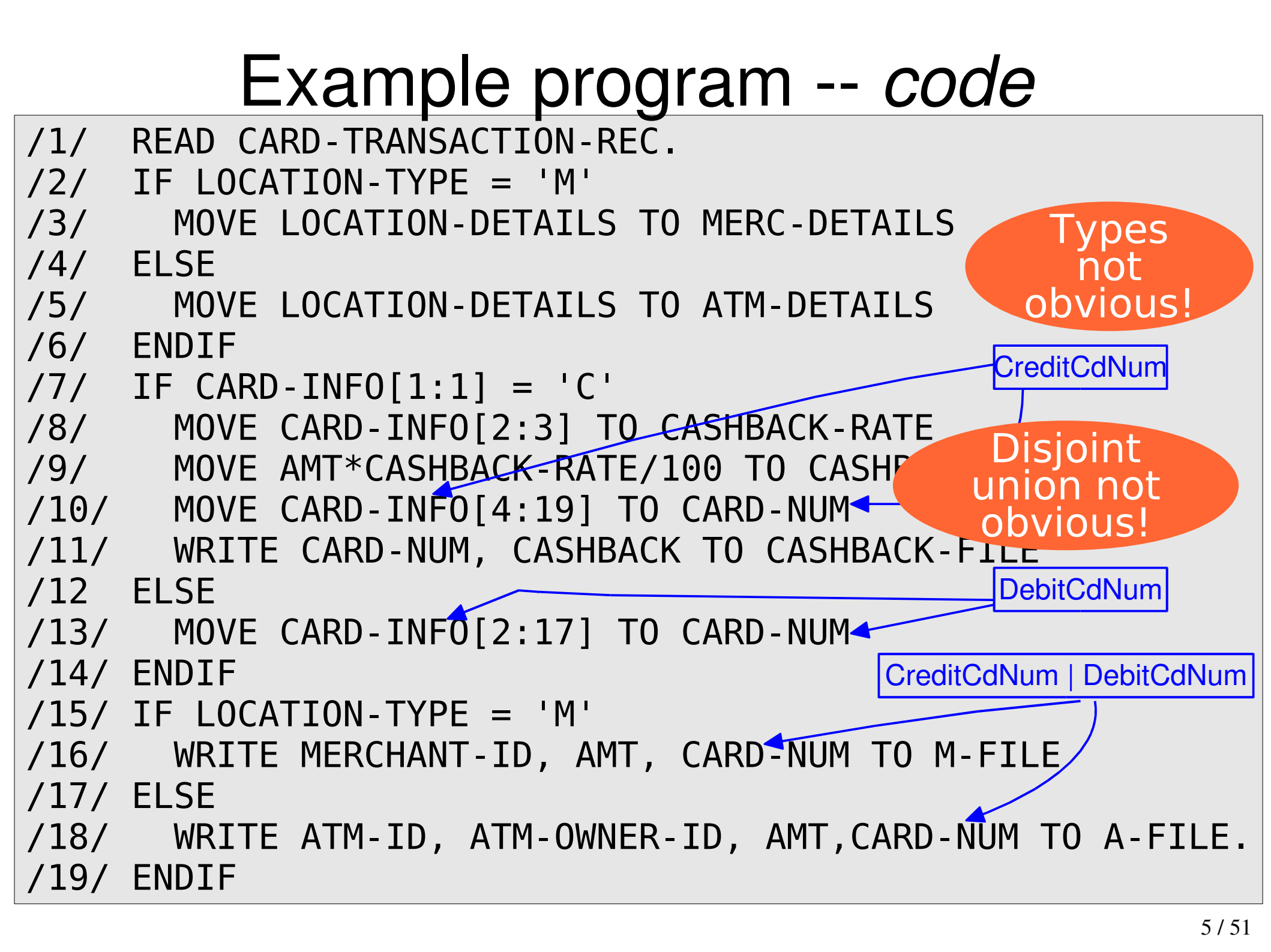
An example Cobol program – *Data declarations*

```
01 CARD-TRANSACTION-REC.
   05 LOCATION-TYPE PIC X.
   05 LOCATION-DETAILS PIC X(20).
   05 CARD-INFO PIC X(19).
   05 AMT PIC X(4).

01 ATM-DETAILS.
   05 ATM-ID PIC X(5).
   05 ATM-ADDRESS X(12).
   05 ATM-OWNER-ID PIC X(3).

01 MERC-DETAILS.
   05 MERCHANT-ID PIC X(8).
   05 MERCHANT-ADDRESS PIC X(12).

01 CARD-NUM PIC X(16).
01 CASHBACK-RATE X(2).
01 CASHBACK X(3).
```



Example program -- *code*

```
/1/  READ CARD-TRANSACTION-REC.
/2/  IF LOCATION-TYPE = 'M'
/3/    MOVE LOCATION-DETAILS TO MERC-DETAILS
/4/  ELSE
/5/    MOVE LOCATION-DETAILS TO ATM-DETAILS
/6/  ENDIF
/7/  IF CARD-INFO[1:1] = 'C'
/8/    MOVE CARD-INFO[2:3] TO CASHBACK-RATE
/9/    MOVE AMT*CASHBACK-RATE/100 TO CASHB
/10/   MOVE CARD-INFO[4:19] TO CARD-NUM
/11/   WRITE CARD-NUM, CASHBACK TO CASHBACK-FILE
/12  ELSE
/13/   MOVE CARD-INFO[2:17] TO CARD-NUM
/14/ ENDIF
/15/ IF LOCATION-TYPE = 'M'
/16/   WRITE MERCHANT-ID, AMT, CARD-NUM TO M-FILE
/17/ ELSE
/18/   WRITE ATM-ID, ATM-OWNER-ID, AMT,CARD-NUM TO A-FILE.
/19/ ENDIF
```



An example Cobol program – *Data declarations*

Implicit aggregate structure!

```
01 CARD-TRANSACTION-REC.
   05 LOCATION-TYPE PIC X.
   05 LOCATION-DETAILS PIC X(20).
   05 CARD-INFO PIC X(19).
   05 AMT PIC X(4).

01 ATM-DETAILS.
   05 ATM-ID PIC X(5).
   05 ATM-ADDRESS X(12).
   05 ATM-OWNER-ID PIC X(3).

01 MERC-DETAILS.
   05 MERCHANT-ID PIC X(8).
   05 MERCHANT-ADDRESS PIC X(12).

01 CARD-NUM PIC X(16).
01 CASHBACK-RATE X(2).
01 CASHBACK X(3).
```

AtmID ; OwnerID

MerchantID

'C':CreditTag ; CashBkRate ; CreditCdNum

!{'C'}:DebitTag ; DebitCdNum ; Unused

Algorithm 1 [TACAS '05]

- A “guarded” (dependent) type system, involving guarded type variables, records (concatenation), and unions
  - Example: (‘E’:α^1 ; β^7 ; γ^4 ; δ^2) |
    (!{‘E’}: ε^1 ; ϕ^9 ; η^4)

Algorithm 1 [TACAS '05]

- A “guarded” (dependent) type system, involving guarded type variables, records (concatenation), and unions
  - Example: (‘E’:Emp^{1} ; EId^{7} ; $Salary^{4}$; $Unused^{2}$) |
    (!{‘E’}:Vis^{1} ; SSN^{9} ; $Stipend^{4}$)

Meaningful names for clarity

- Formal characterization of a correct typing solution for a program
- Path-sensitive type inference algorithm
  - Improved accuracy; program-point specific types
  - Computed solution helps in constructing class diagram

Applications of guarded type system

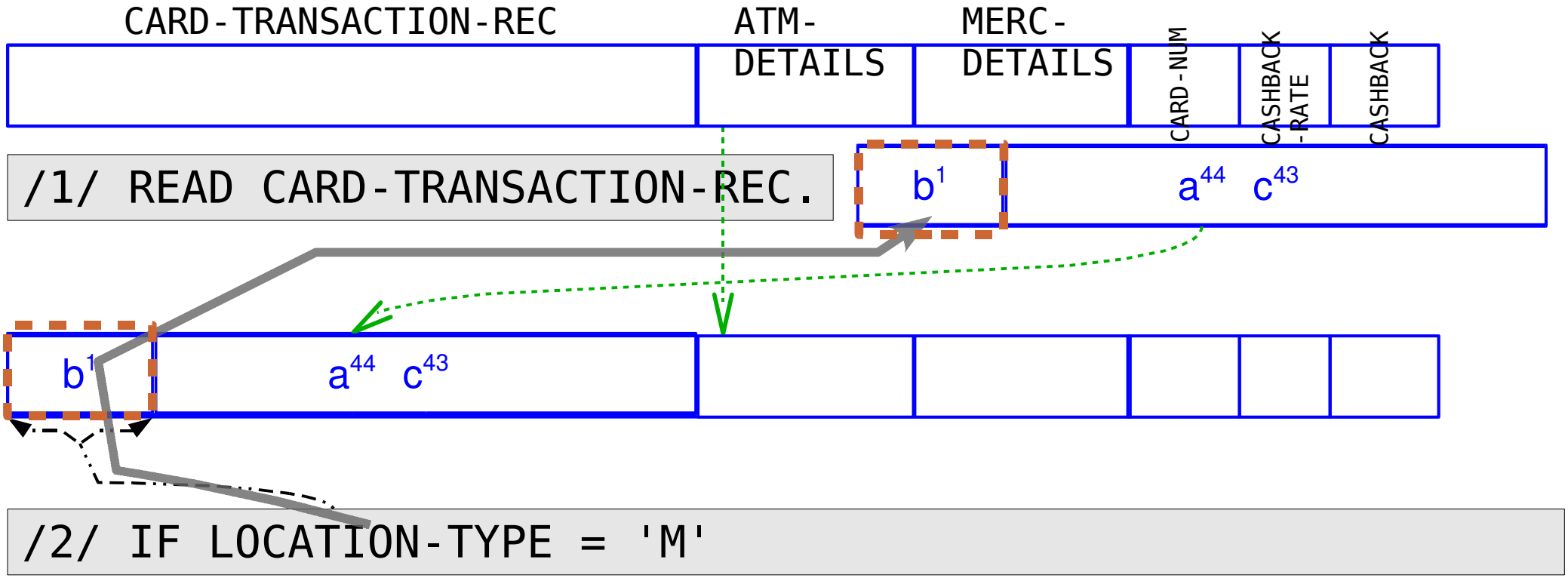
- Program understanding
- Understanding impact of changes
- Program transformation
  - Field expansion (e.g., Y2K expansion)
  - Porting from weakly-typed languages to object-oriented languages
  - Refactoring data declarations to make them better reflect logical structure

Key features of algorithm

- Based on dataflow analysis
  - Dataflow fact at each point is a *type* for the *entire memory*
  - Each *origin statement* (READ, MOVE literal TO var) gets a unique type variable
- Interprets predicates of the form

`var == literal, var != literal`

- Two key operations:
  - **Split:** Replace α^i by concatenation $\beta^j;\gamma^k$, **i = j + k.**
  - **Specialize:** Replace α^i by union $\beta^i \mid \gamma^i$.

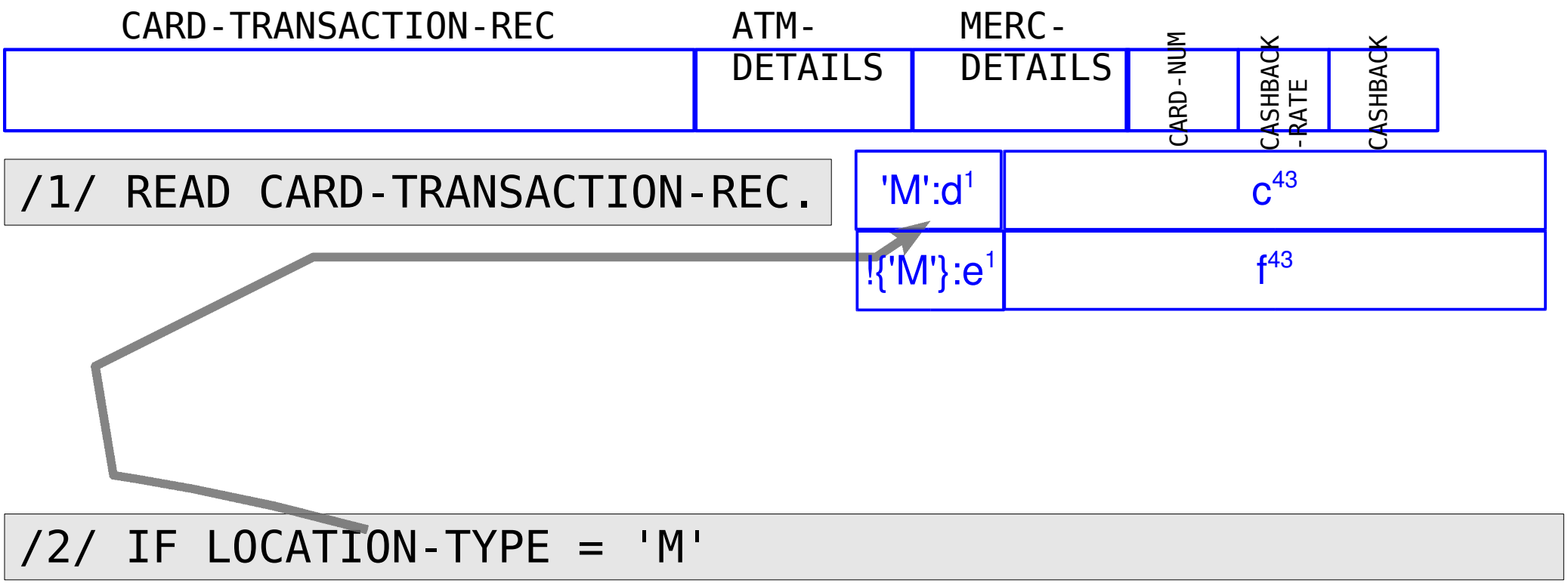


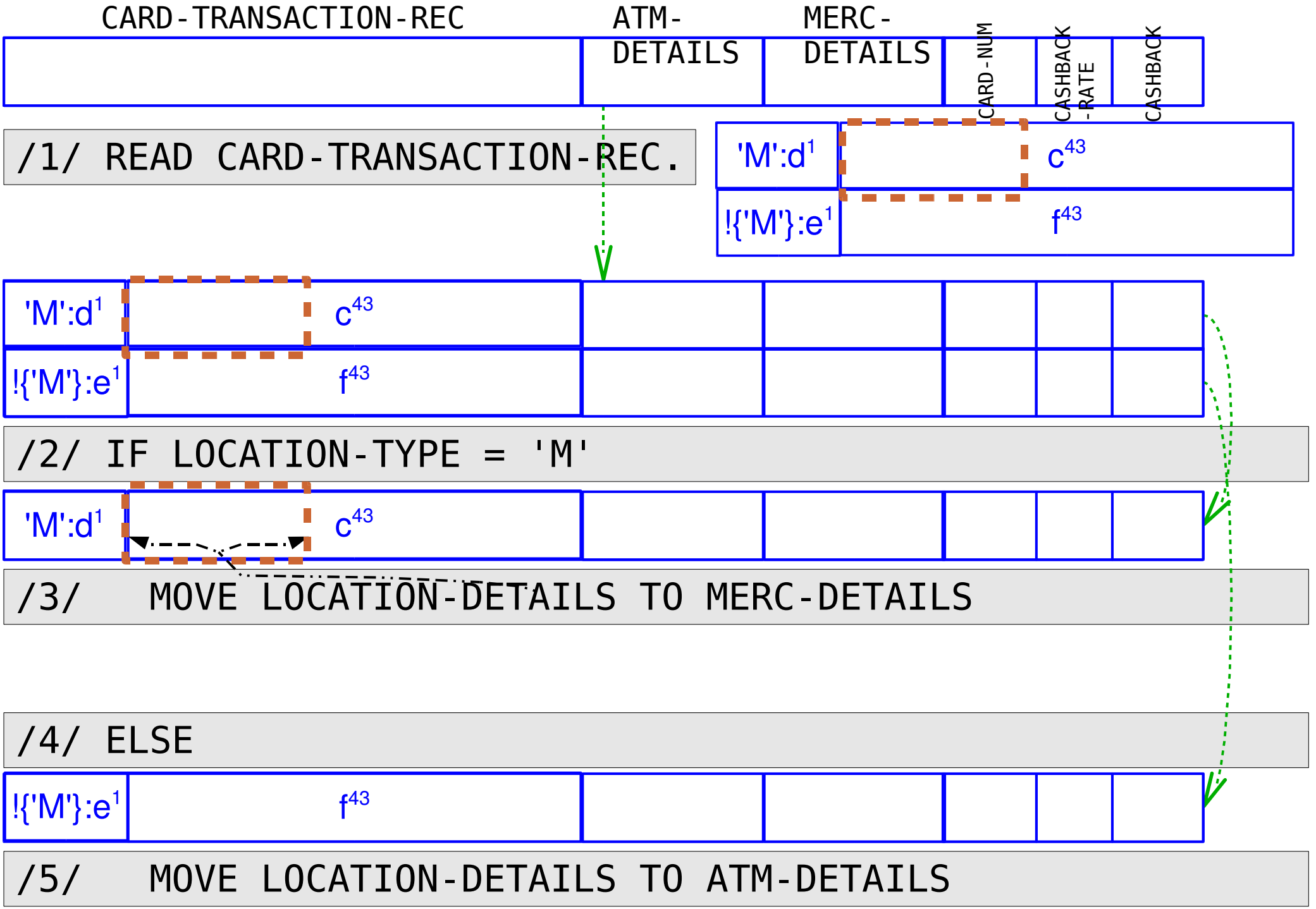
Split

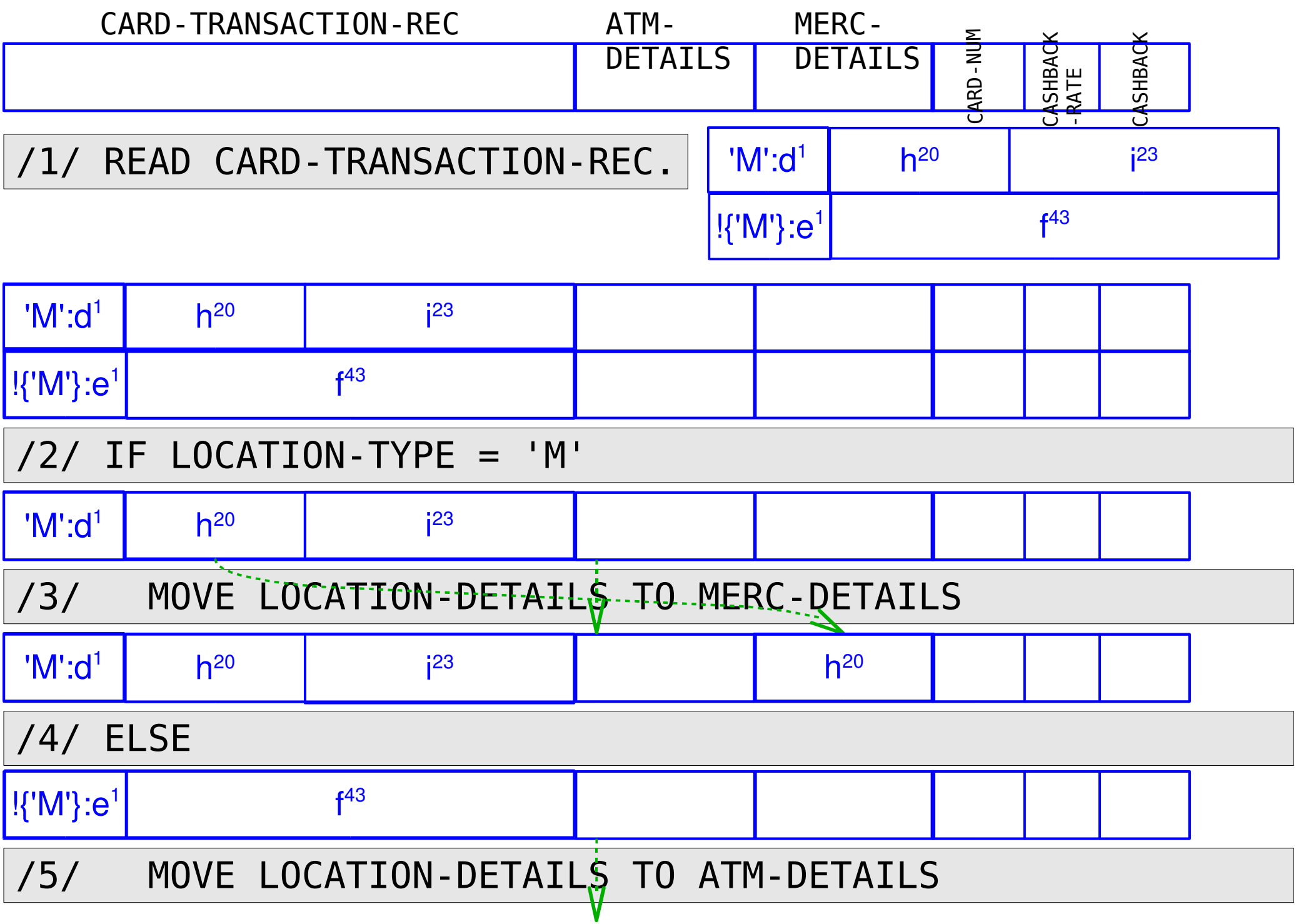
$a^{44} \rightarrow b^1 ; c^{43}$

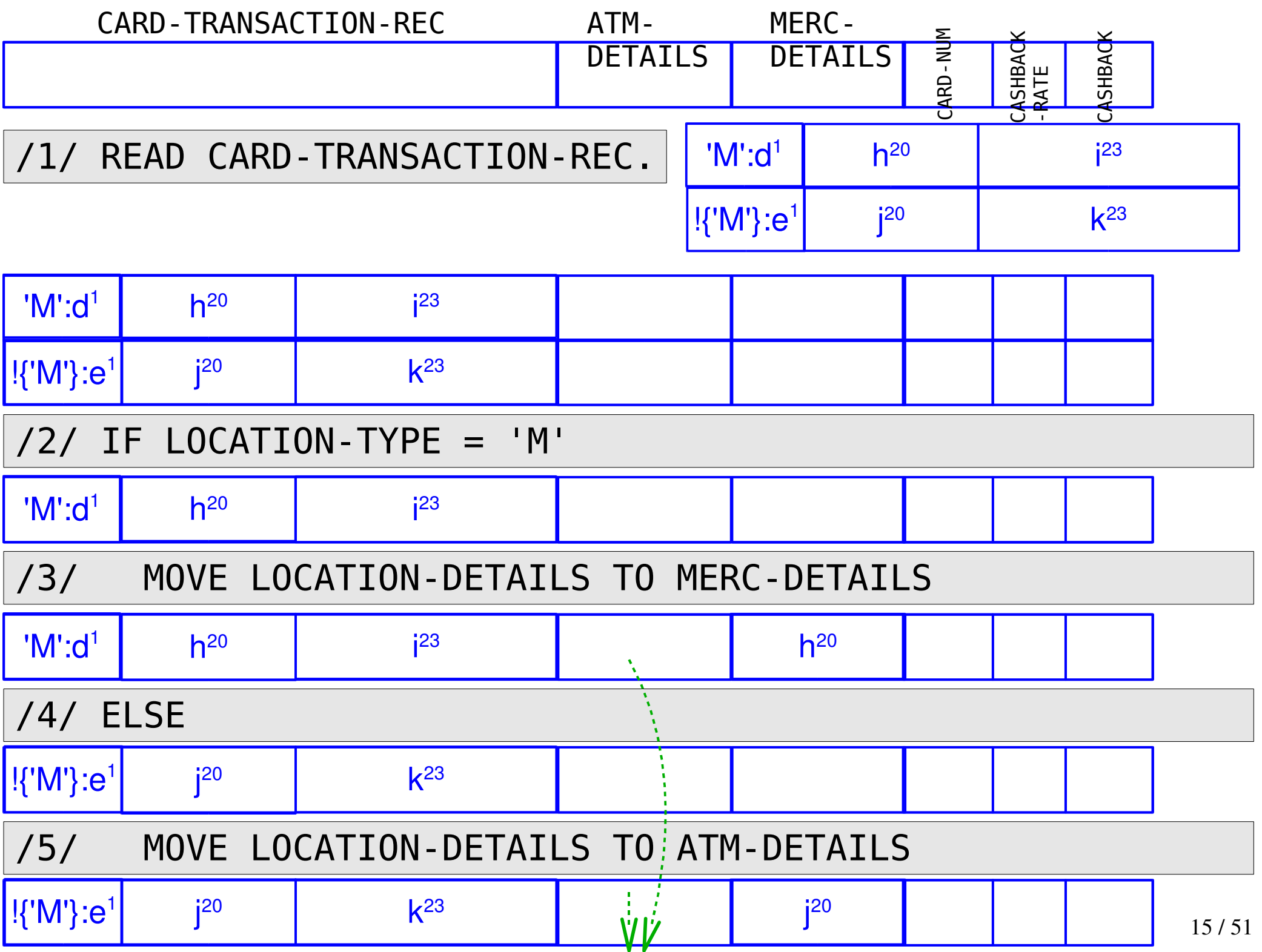
Specialize

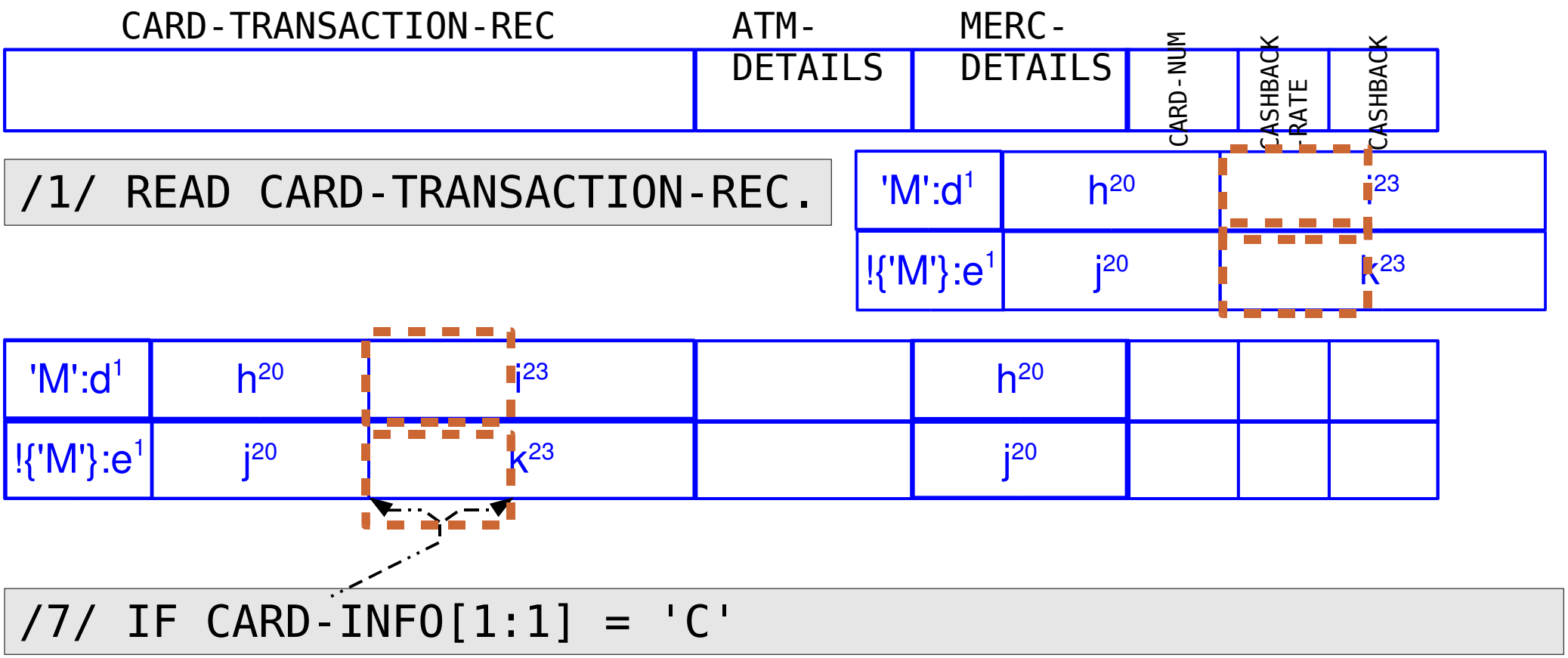
$b^1 \rightarrow$ 'M':d^1 | !{'M'}:e^1

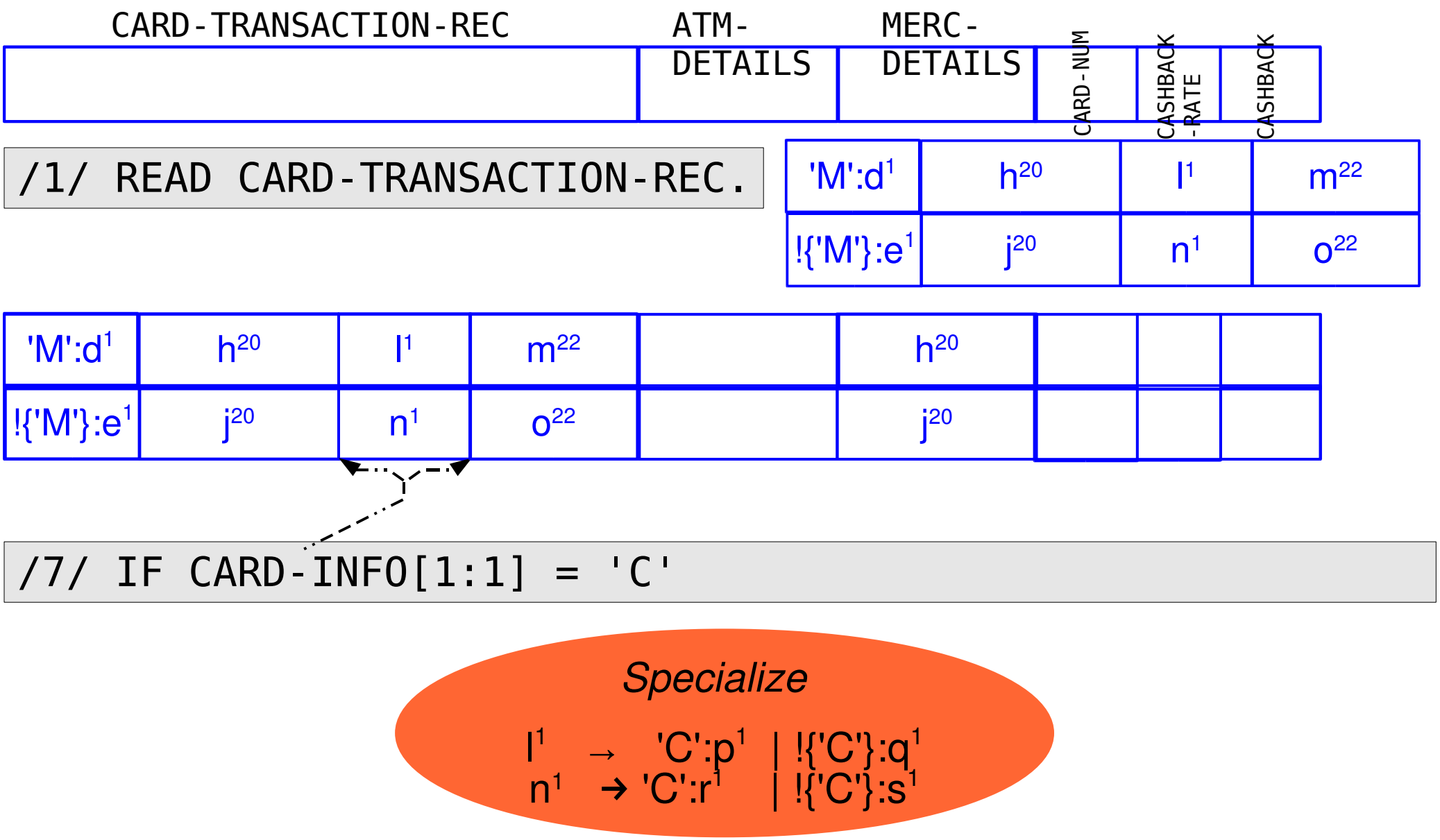

CARD-TRANSACTION-REC
ATM-
DETAILS
MERC-
DETAILS
CARD-NUM
CASHBACK
-RATE
CASHBACK
/1/ READ CARD-TRANSACTION-REC.
'M':d1
c43
!{'M'}:e1
f43
/2/ IF LOCATION-TYPE = 'M'

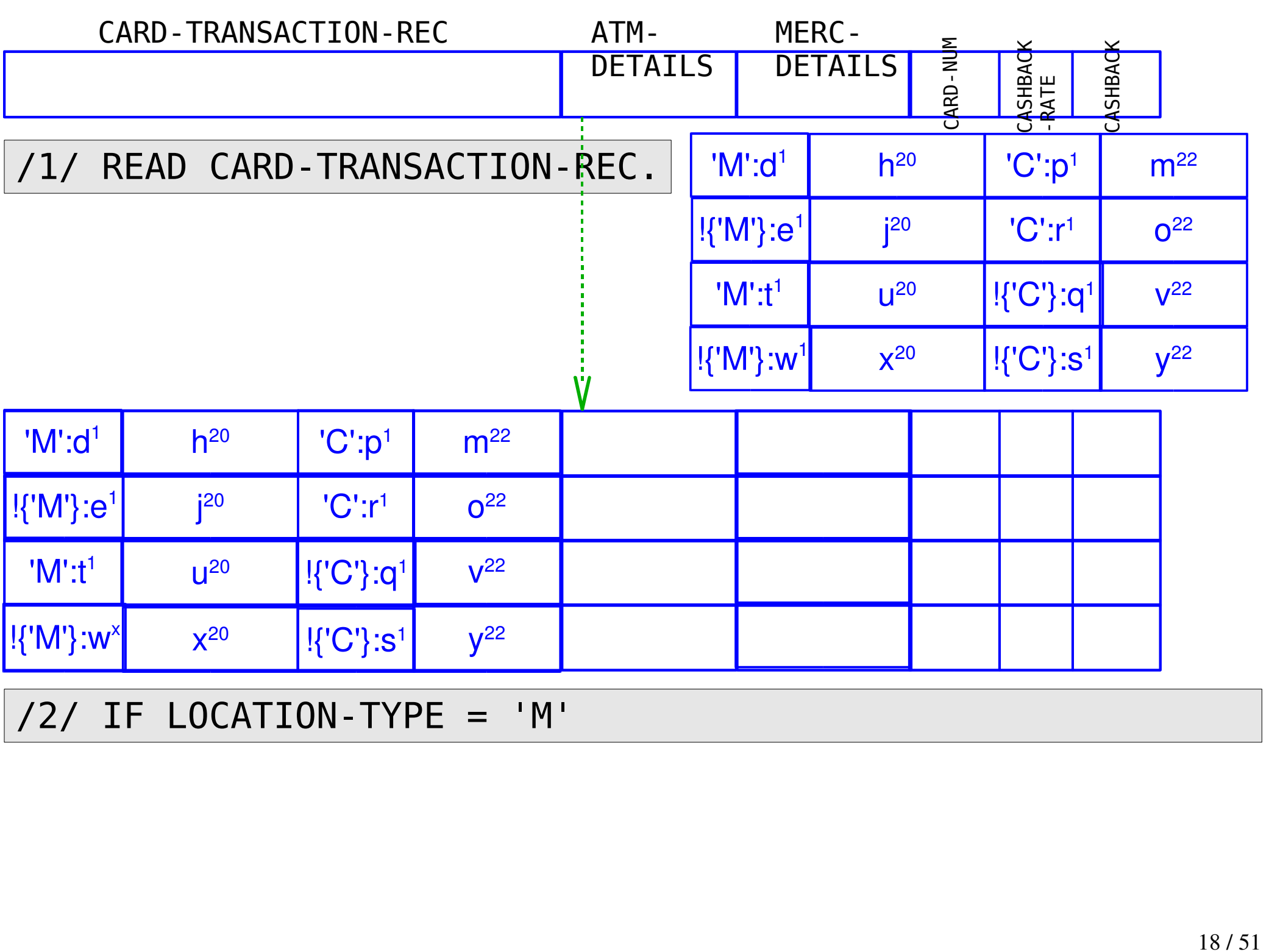

CARD-TRANSACTION-REC
ATM-
DETAILS
MERC-
DETAILS
CARD-NUM
CASHBACK
-RATE
CASHBACK
/1/ READ CARD-TRANSACTION-REC.
'M':d1
c43
!{'M'}:e1
f43
'M':d1
c43
!{'M'}:e1
f43
/2/ IF LOCATION-TYPE = 'M'
'M':d1
c43
/3/ MOVE LOCATION-DETAILS TO MERC-DETAILS
/4/ ELSE
!{'M'}:e1
f43
/5/ MOVE LOCATION-DETAILS TO ATM-DETAILS

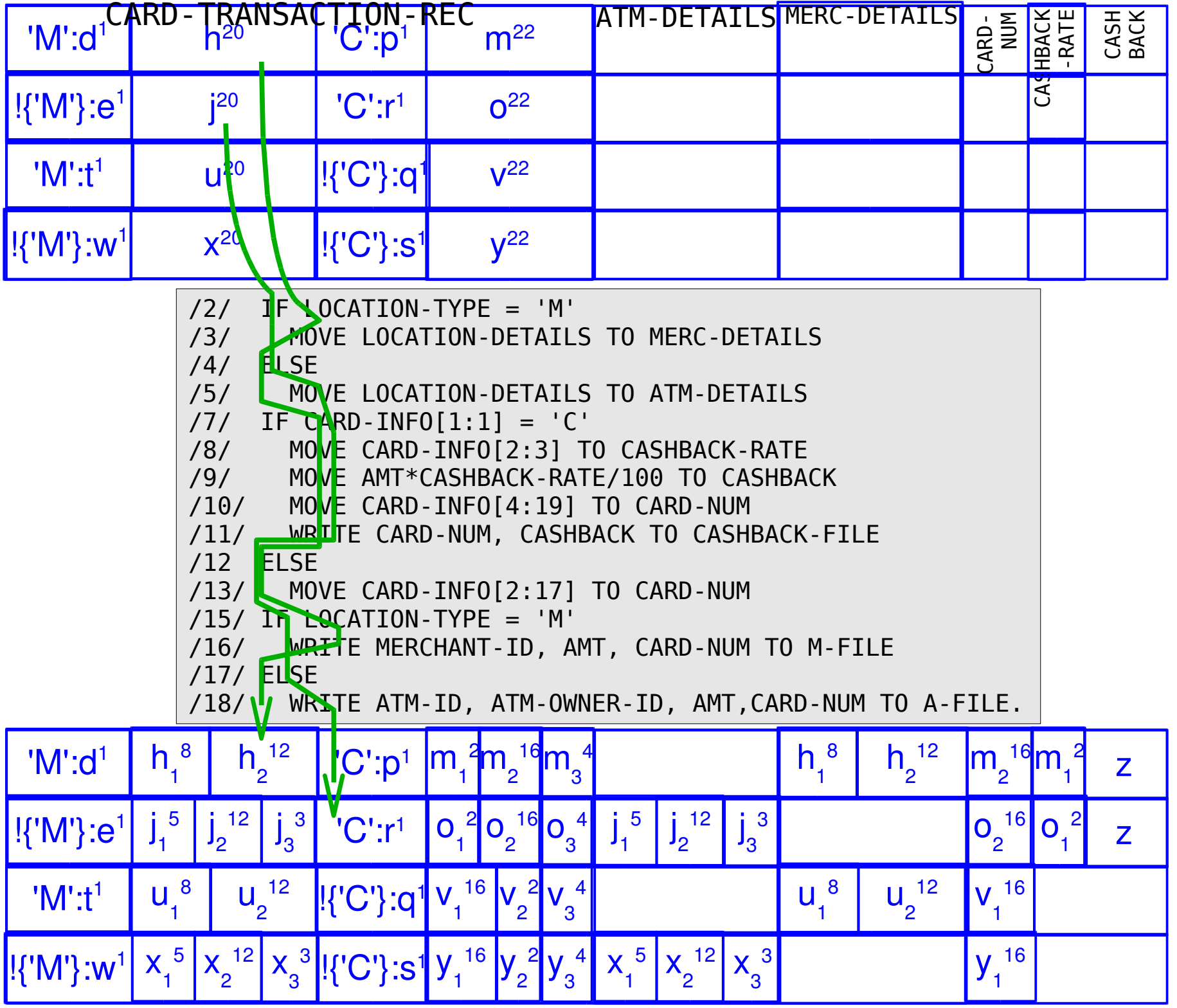

CARD-TRANSACTION-REC
ATM-DETAILS
MERC-DETAILS
CARD-NUM
CASHBACK-RATE
CASHBACK
/1/ READ CARD-TRANSACTION-REC.
'M':d1
h20
i23
!{'M'}:e1
f43
'M':d1
h20
i23
!{'M'}:e1
f43
/2/ IF LOCATION-TYPE = 'M'
'M':d1
h20
i23
/3/ MOVE LOCATION-DETAILS TO MERC-DETAILS
'M':d1
h20
i23
h20
/4/ ELSE
!{'M'}:e1
f43
/5/ MOVE LOCATION-DETAILS TO ATM-DETAILS


CARD-TRANSACTION-REC
ATM-DETAILS
MERC-DETAILS
CARD-NUM
CASHBACK-RATE
CASHBACK
/1/ READ CARD-TRANSACTION-REC.
'M':d1
h20
i23
!{'M'}:e1
j20
k23
/2/ IF LOCATION-TYPE = 'M'
/3/ MOVE LOCATION-DETAILS TO MERC-DETAILS
/4/ ELSE
/5/ MOVE LOCATION-DETAILS TO ATM-DETAILS

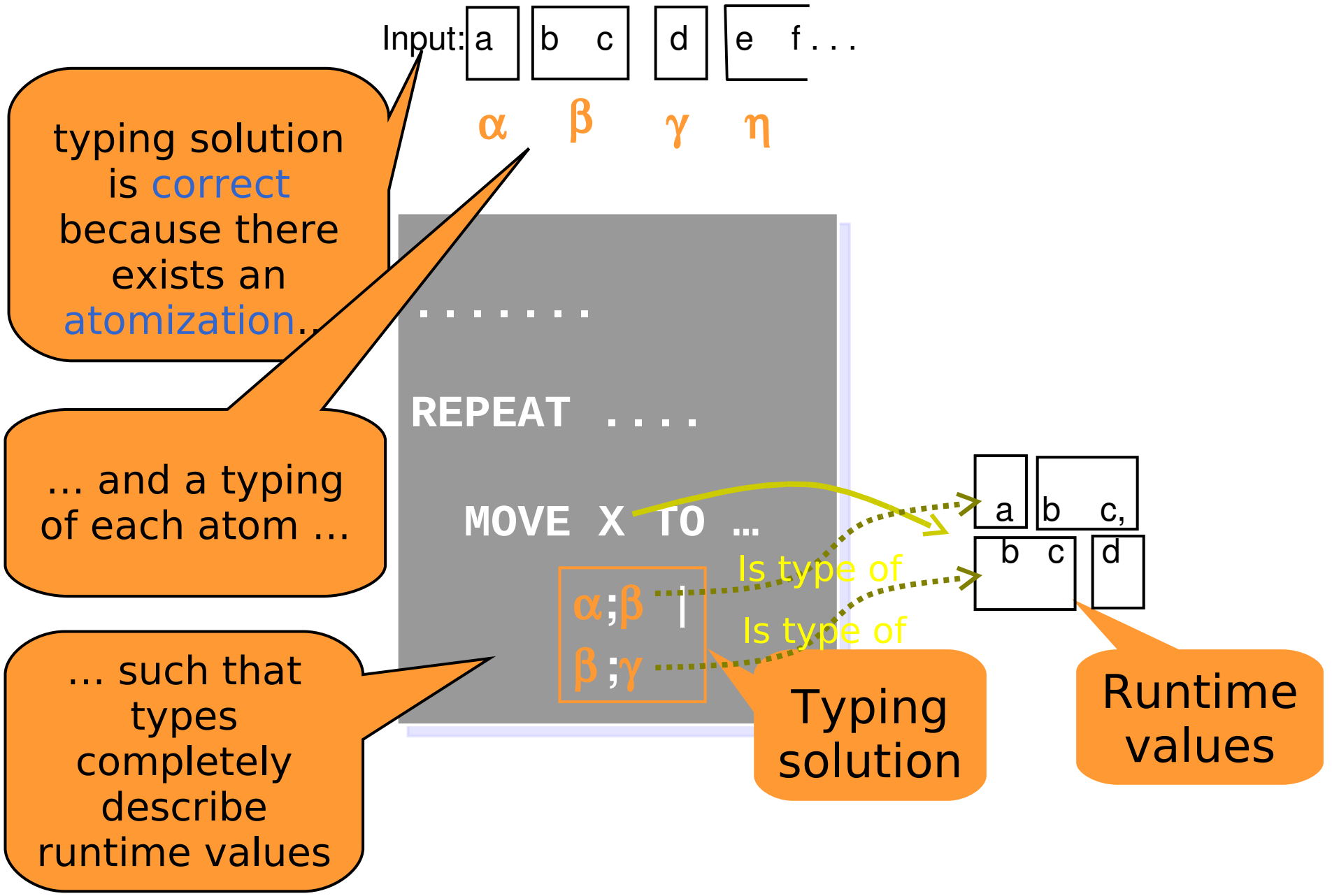

CARD-TRANSACTION-REC
ATM-DETAILS
MERC-DETAILS
CARD-NUM
CASHBACK RATE
CASHBACK
/1/ READ CARD-TRANSACTION-REC.
'M':d1
h20
i23
!{'M'}:e1
j20
k23
'M':d1
h20
i23
h20
!{'M'}:e1
j20
k23
j20
/7/ IF CARD-INFO[1:1] = 'C'


CARD-TRANSACTION-REC
ATM-DETAILS
MERC-DETAILS
CARD-NUM
CASHBACK-RATE
CASHBACK
/1/ READ CARD-TRANSACTION-REC.
'M':d1
h20
l1
m22
!{'M'}:e1
j20
n1
o22
'M':d1
h20
l1
m22
h20
!{'M'}:e1
j20
n1
o22
j20
/7/ IF CARD-INFO[1:1] = 'C'
Specialize
l1 → 'C':p1 | !{'C'}:q1
n1 → 'C':r1 | !{'C'}:s1


CARD-TRANSACTION-REC
ATM-DETAILS
MERC-DETAILS
CARD-NUM
CASHBACK-RATE
CASHBACK
/1/ READ CARD-TRANSACTION-REC.
'M':d1 h20 'C':p1 m22
!{'M'}:e1 j20 'C':r1 o22
'M':t1 u20 !{'C'}:q1 v22
!{'M'}:w1 x20 !{'C'}:s1 y22
'M':d1 h20 'C':p1 m22
!{'M'}:e1 j20 'C':r1 o22
'M':t1 u20 !{'C'}:q1 v22
!{'M'}:wx x20 !{'C'}:s1 y22
/2/ IF LOCATION-TYPE = 'M'


CARD-TRANSACTION-REC
'M':d1
h20
'C':p1
m22
ATM-DETAILS
MERC-DETAILS
CARD-NUM
CASHBACK-RATE
CASH BACK
!{'M'}:e1
j20
'C':r1
o22
'M':t1
u20
!{'C'}:q1
v22
!{'M'}:w1
x20
!{'C'}:s1
y22
/2/ IF LOCATION-TYPE = 'M'
/3/ MOVE LOCATION-DETAILS TO MERC-DETAILS
/4/ ELSE
/5/ MOVE LOCATION-DETAILS TO ATM-DETAILS
/7/ IF CARD-INFO[1:1] = 'C'
/8/ MOVE CARD-INFO[2:3] TO CASHBACK-RATE
/9/ MOVE AMT*CASHBACK-RATE/100 TO CASHBACK
/10/ MOVE CARD-INFO[4:19] TO CARD-NUM
/11/ WRITE CARD-NUM, CASHBACK TO CASHBACK-FILE
/12 ELSE
/13/ MOVE CARD-INFO[2:17] TO CARD-NUM
/15/ IF LOCATION-TYPE = 'M'
/16/ WRITE MERCHANT-ID, AMT, CARD-NUM TO M-FILE
/17/ ELSE
/18/ WRITE ATM-ID, ATM-OWNER-ID, AMT,CARD-NUM TO A-FILE.
'M':d1
h1 8
h2 12
'C':p1
m1 2
m2 16
m3 4
h1 8
h2 12
m2 16
m1 2
z
!{'M'}:e1
j1 5
j2 12
j3 3
'C':r1
o1 2
o2 16
o3 4
j1 5
j2 12
j3 3
o2 16
o1 2
z
'M':t1
u1 8
u2 12
!{'C'}:q1
v1 16
v2 2
v3 4
u1 8
u2 12
v1 16
!{'M'}:w1
x1 5
x2 12
x3 3
!{'C'}:s1
y1 16
y2 2
y3 4
x1 5
x2 12
x3 3
y1 16

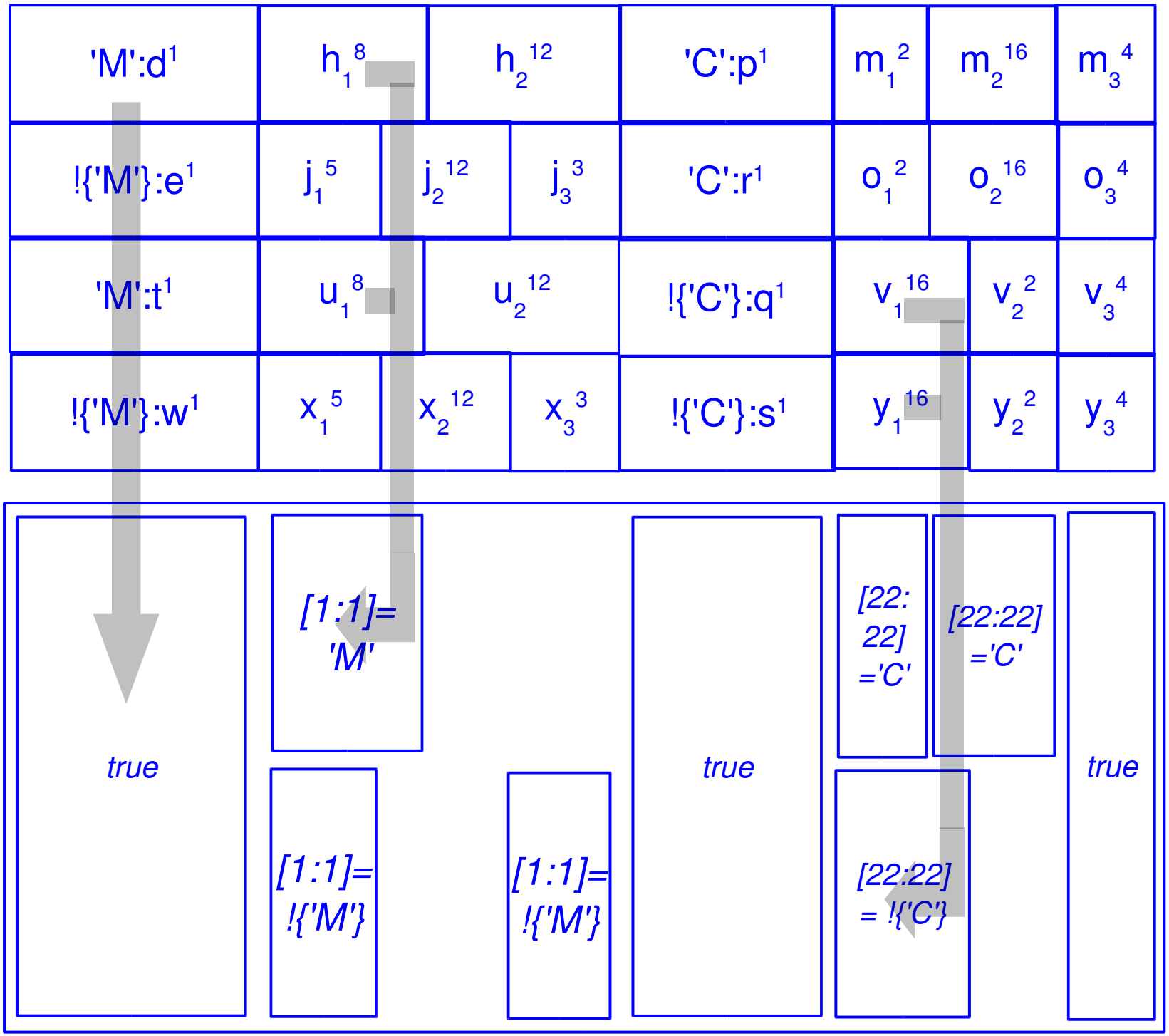
Correctness characterization



Characteristics of the solution

- Fow and path sensitive:
  - Each occurrence of a variable is assigned a type
  - Uses guards to ignore certain infeasible paths
- Determines variables of the same type, reveals record structure within variables, as well as disjoint unions
- Shortcomings:
  - Dataflow facts are “unfactored”, potentially of exponential size

/1/ READ CARD-TRANSACTION-REC.



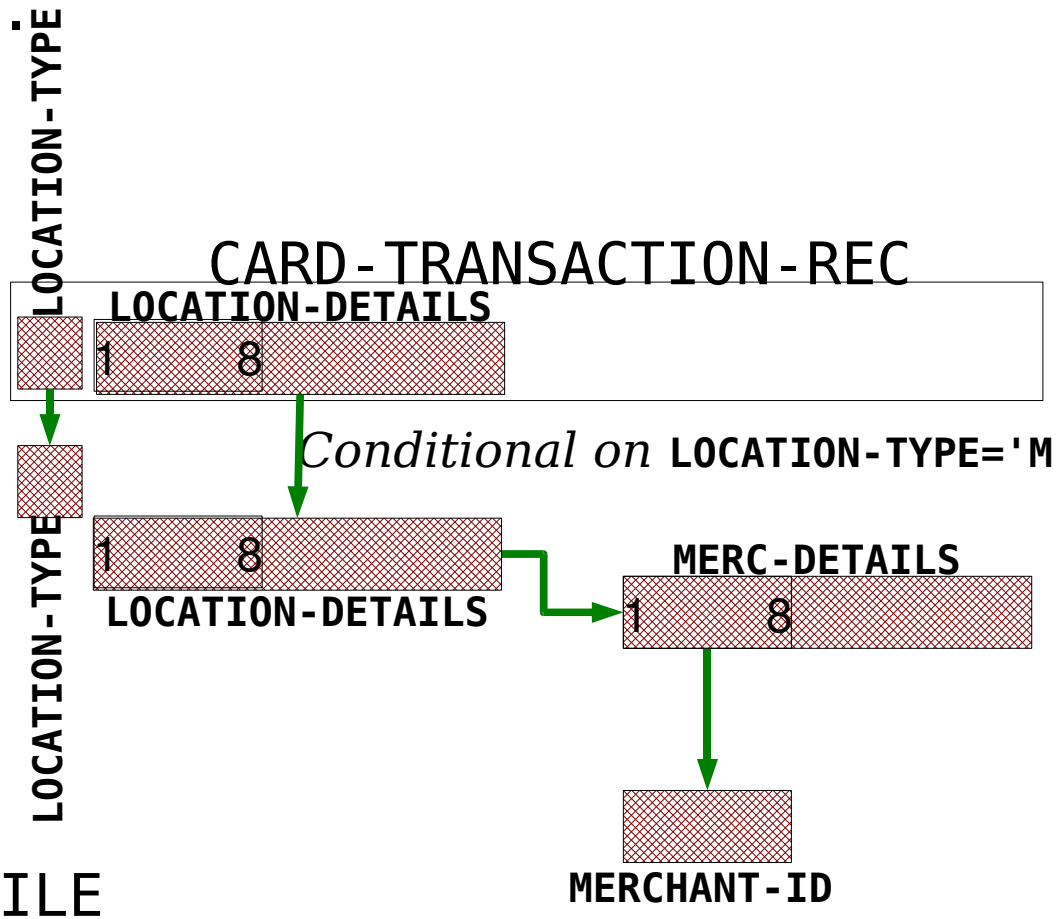
Algorithm 2 [ICSE '06, WCRE '07]

1. Compute guarded dependences
2. Compute *cuts* at each data-source statement (i.e., READ statement).
3. Organize the cuts as a cut-structure tree
   - *It is possible, but not desirable, to translate cut-structure tree directly into a class hierarchy*
4. *Factor* the cut-structure tree to capture better the grouping/structure of sibling cuts
5. Translate cut-structure tree into a class hierarchy

Step 1. Compute guarded dependences

```
01 CARD-TRANSACTION-REC.
 05 LOCATION-TYPE PIC X.
 05 LOCATION-DETAILS PIC X(20).
 ... 23 more bytes ...
01 MERC-DETAILS.
 05 MERCHANT-ID PIC X(8).
 ...
/1/  READ CARD-TRANSACTION-REC

/2/  IF LOCATION-TYPE = 'M'
/3/   MOVE LOCATION-DETAILS
        TO MERC-DETAILS
/4/  ELSE ...
/15/ IF LOCATION-TYPE = 'M'
/16/   WRITE MERCHANT-ID,
        AMOUNT, CARD-NUM TO M-FILE
/17/ ELSE ...
```

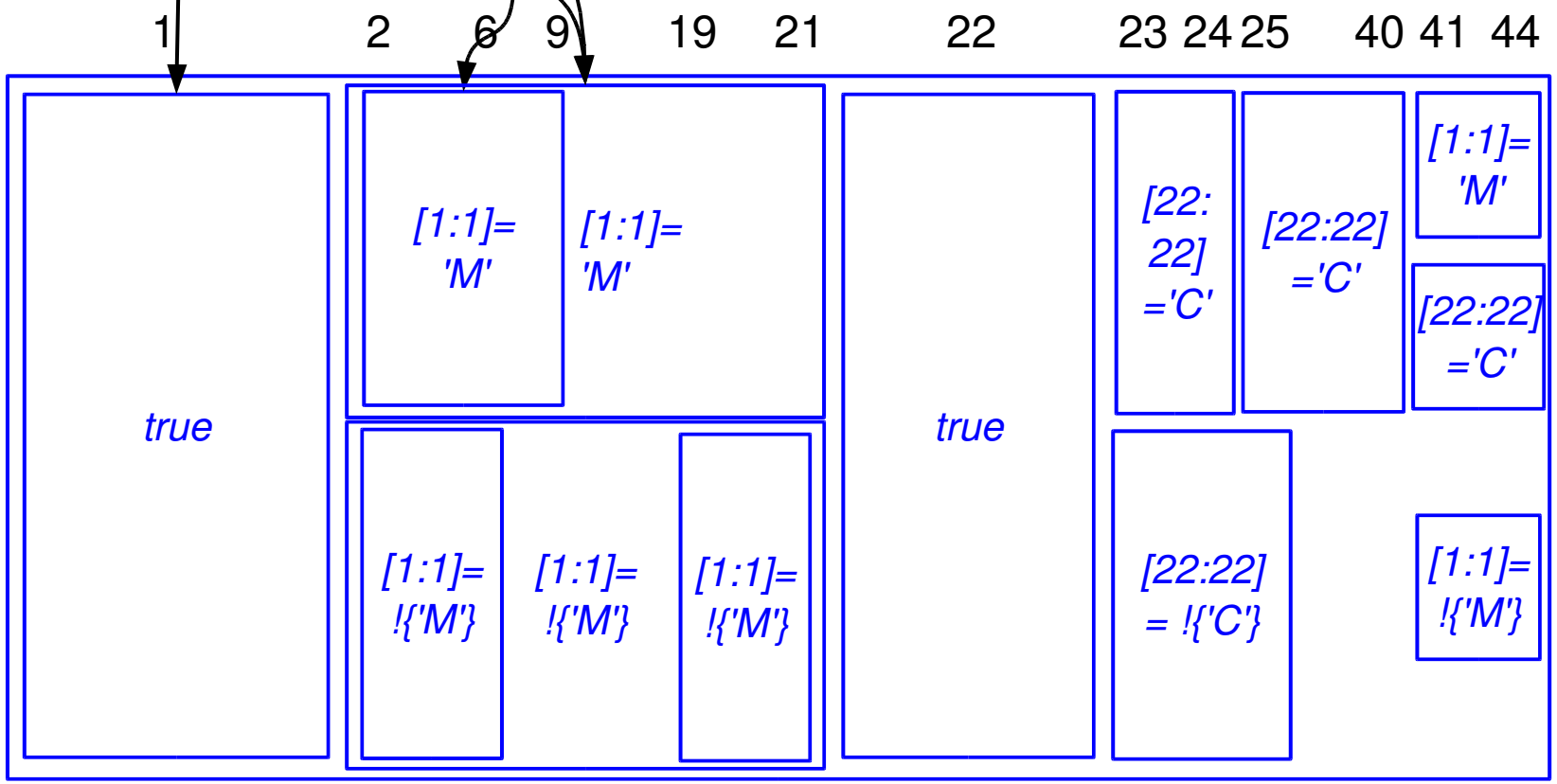
LOCATION-TYPE='M' ▶ LOCATION-DETAILS[1:8]@/1/ → **MERCHANT-ID**@/16/



Step 2: Compute cuts at each data source

CARD-TRANSACTION-REC

1. true ▶ **LOCATION-TYPE**@/1/ → **LOCATION-TYPE**@/2/
2. (**LOCATION-TYPE='M'**) ▶ **LOCATION-DETAILS**@/1/ → **LOCATION-DETAILS**@/3/
1. (**LOCATION-TYPE='M'**) ▶ **LOCATION-DETAILS**@/1/ → **MERC-DETAILS**@/3/
2. (**LOCATION-TYPE='M'**) ▶ **LOCATION-DETAILS**[1:8]@/1/ → **MERCHANT-ID**@/16/



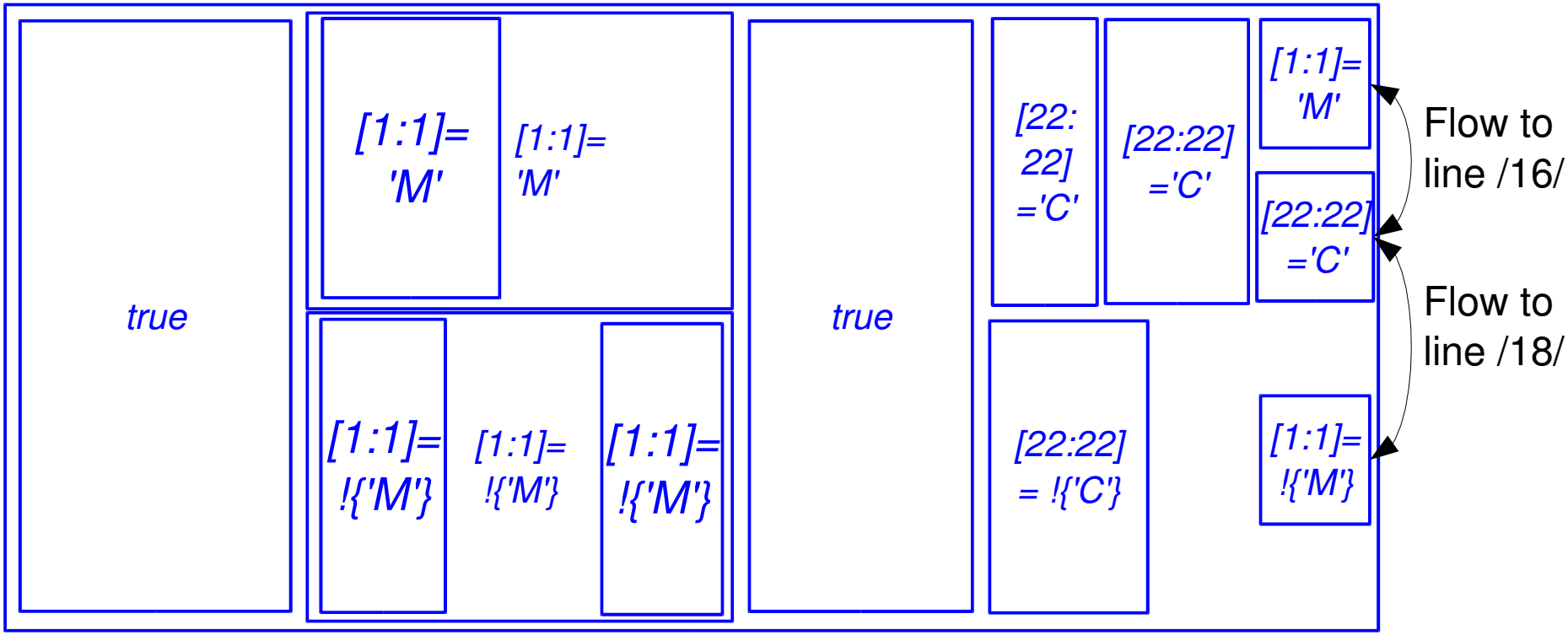
Step 3: Organize cuts as tree

- There is intuitively a containment relation among cuts. Formalization:
  - c_i's range "wider" than c_j's range *and* c_i's predicate "broader" than c_j's predicate ⇒ c_i contains c_j
- We broaden predicates of certain cuts such that

  1) Containment imposes a *tree* structure (not a DAG)
     - *Allows generation of a single-inheritance class hierarchy*
  2) Between any two siblings c_j and c_k there is no overlap;

  i.e.:
     - *Either* their ranges are non-overlapping, *or* their predicates are non-overlapping

Illustration of Step 3

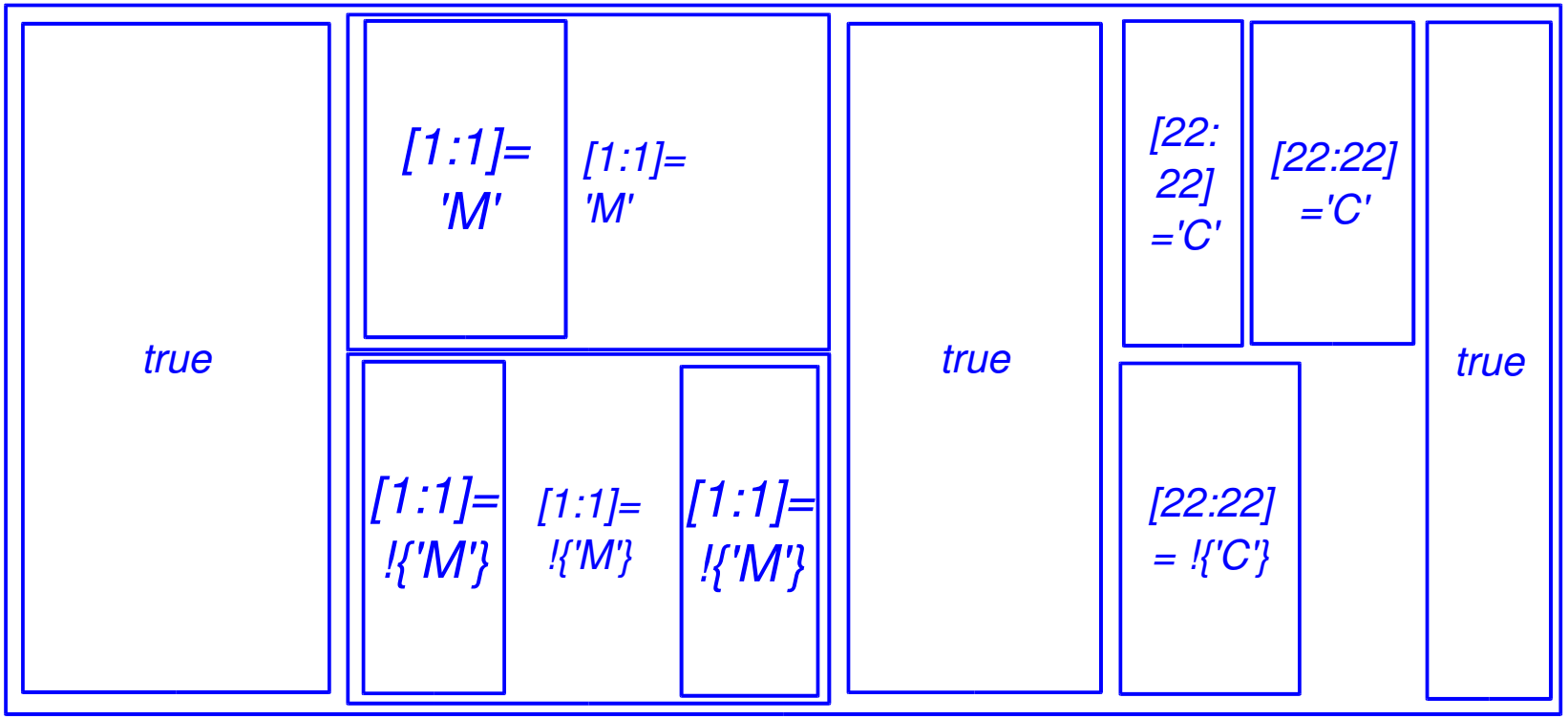


1) Cuts already form a tree structure. Good.

2) However, we have overlap problem!

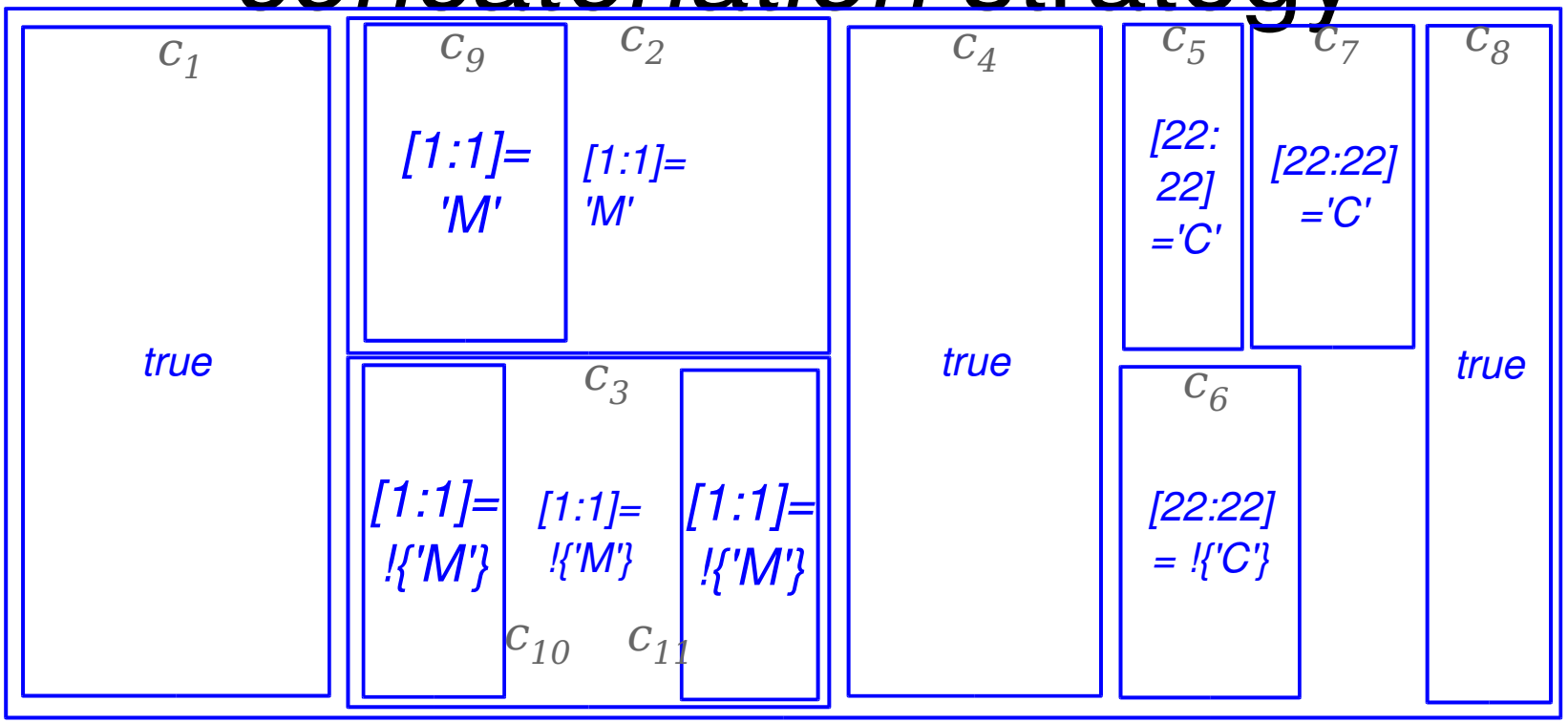
- Intuitively, two overlapping cuts ⇒ both flow into some variable reference;
- We would like a unique cut to flow into each variable ref.

Illustration of Step 3



Merge the three cuts. That is, take logical disjunction of their predicates.

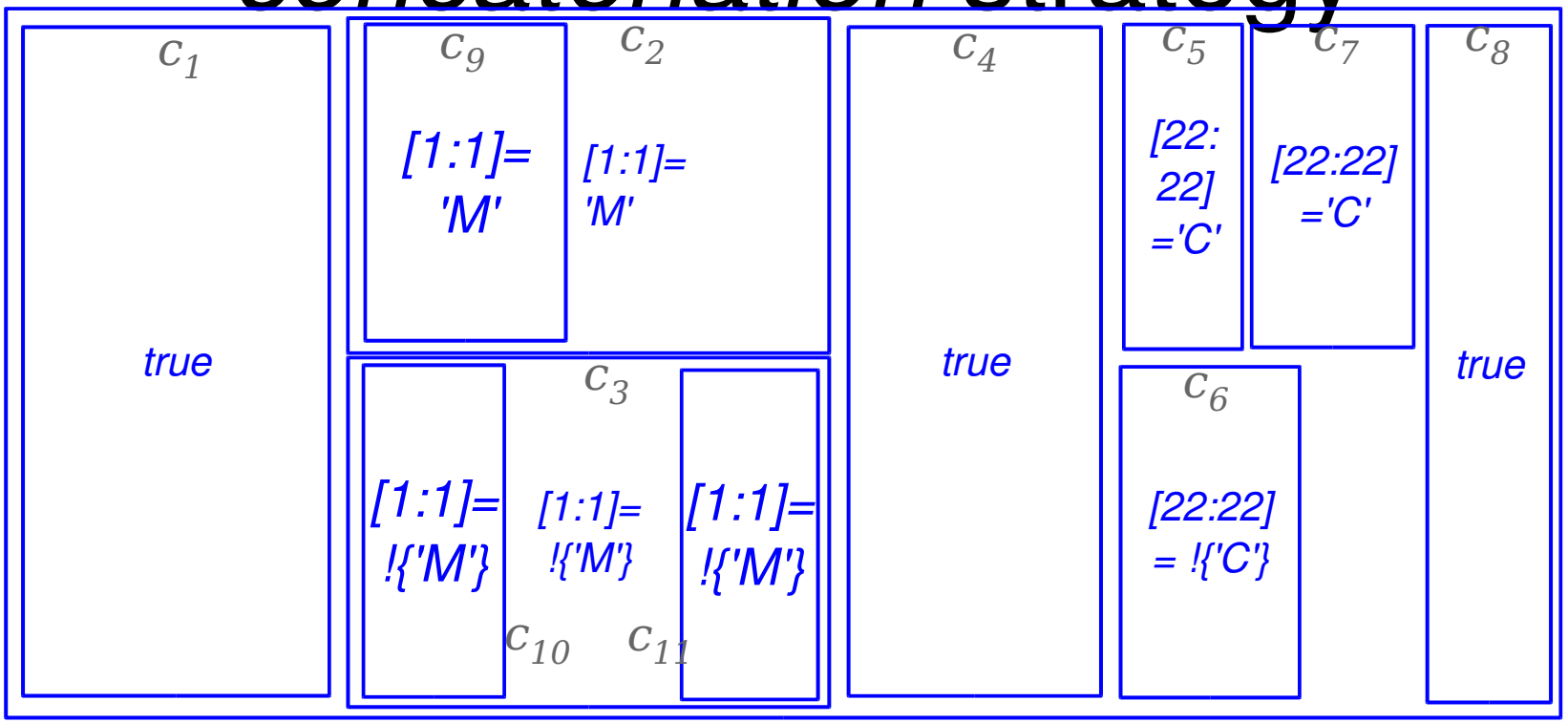
Generating a class hierarchy: *concatenation* strategy



Approach: Turn each cut into a class, and each edge into a *field-of* relation.

- Class c_0 {f1: c_1, f2: c_2, f3: c_3, ..., f8: c_8}, Class c_1{},..., Class c_8{}

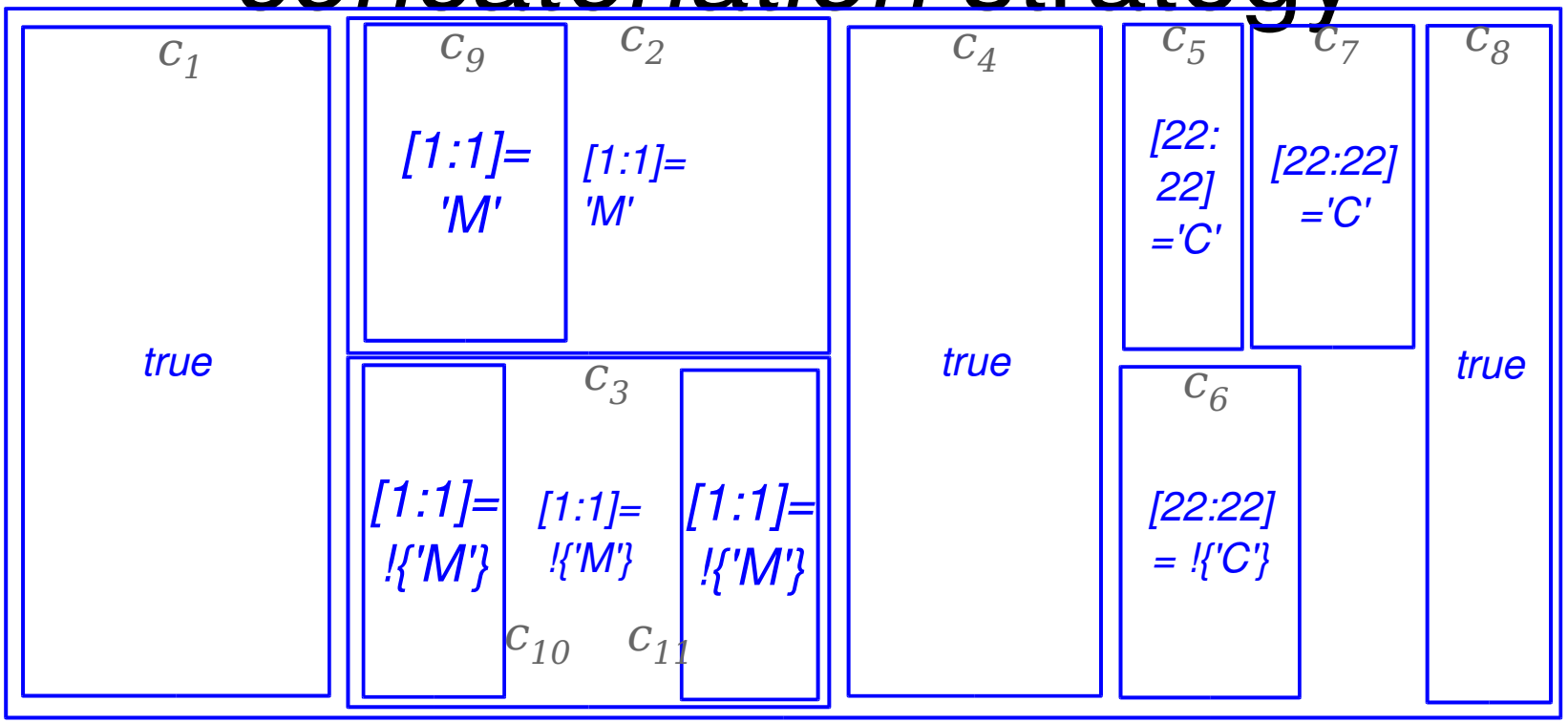
Generating a class hierarchy: *concatenation* strategy



Approach: Turn each cut into a class, and each edge into a *field-of* relation.

- Class c_0 {f1: c_1, f2: c_2, f3: c_3, ..., f8: c_8}, Class c_1{},..., Class c_8{}
- However, predicates are *lost in translation*, hence loss of precision: fields f2 and f3 ought not to co-exist!

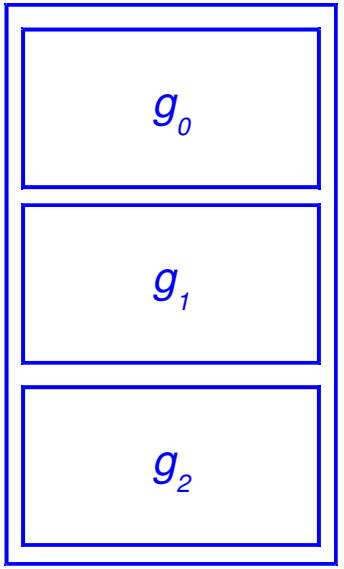
Generating a class hierarchy: *concatenation* strategy



Approach: Turn each cut into a class, and each edge into a *field-of* relation.

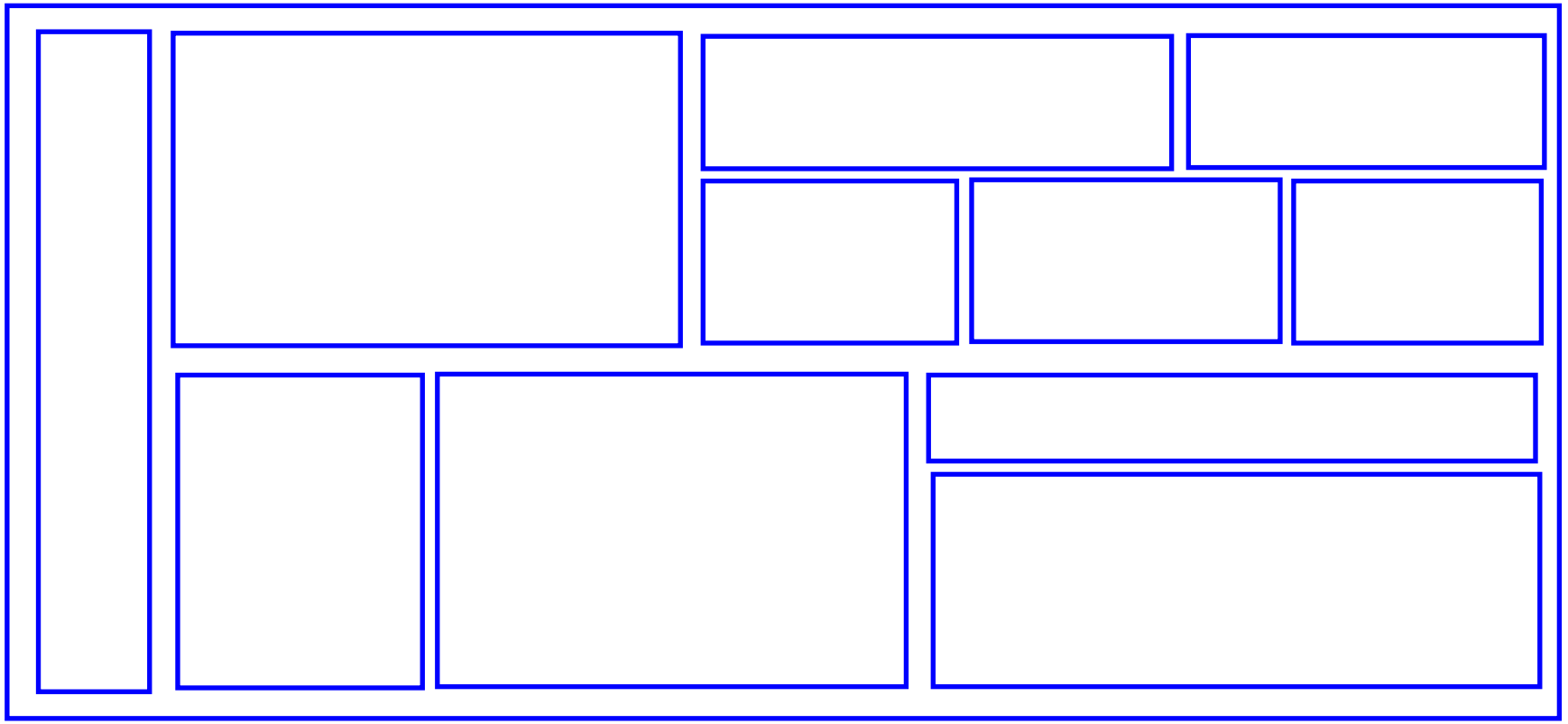
- Class c_0 {f1: c_1, f2: c_2, f3: c_3, ..., f8: c_8}, Class c_1{},..., Class c_8{}
- However, predicates are *lost in translation*, hence loss of precision: fields f2 and f3 ought not to co-exist!
- No loss of precision when when all children have the same guard

Vertical partitioning



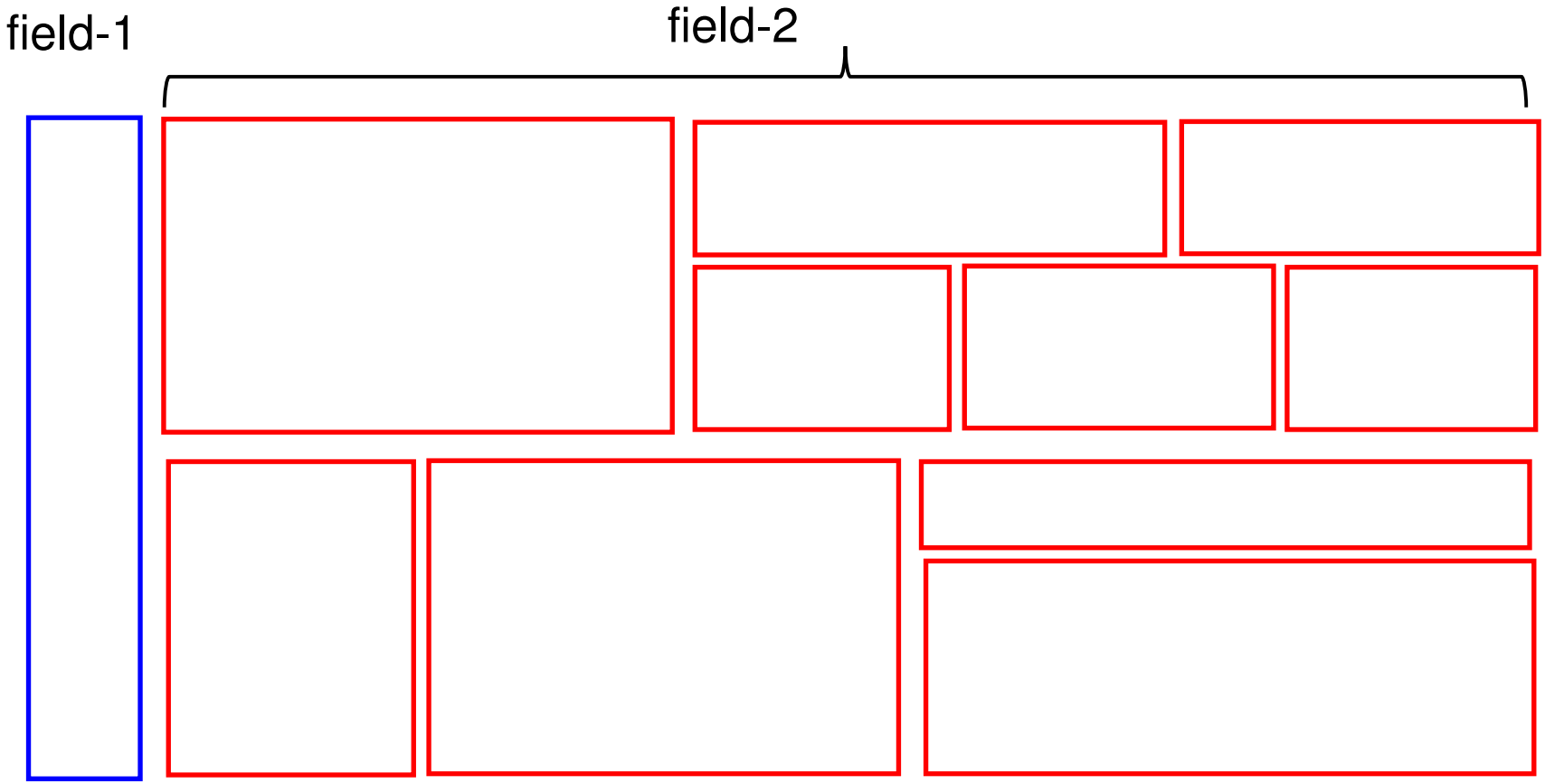
- Applicable *only when* all children have mutually disjoint predicates
  - parent corresponds to a base class
  - children correspond to derived classes

Step 4: Factoring cut-structure tree



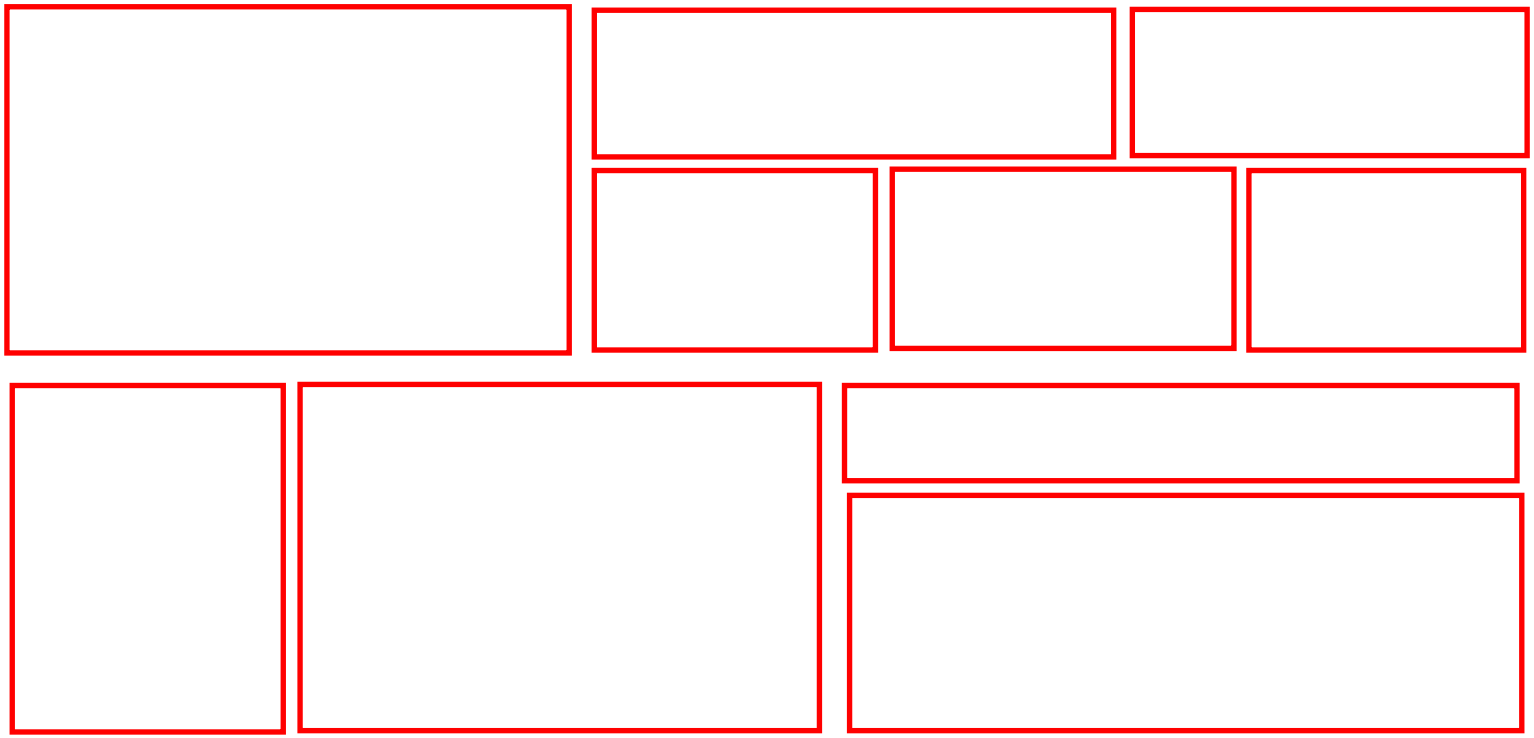
Generalized Horizontal Partitioning
•Add edges between boxes with disjoint guards
• each connected component == a field

Step 4



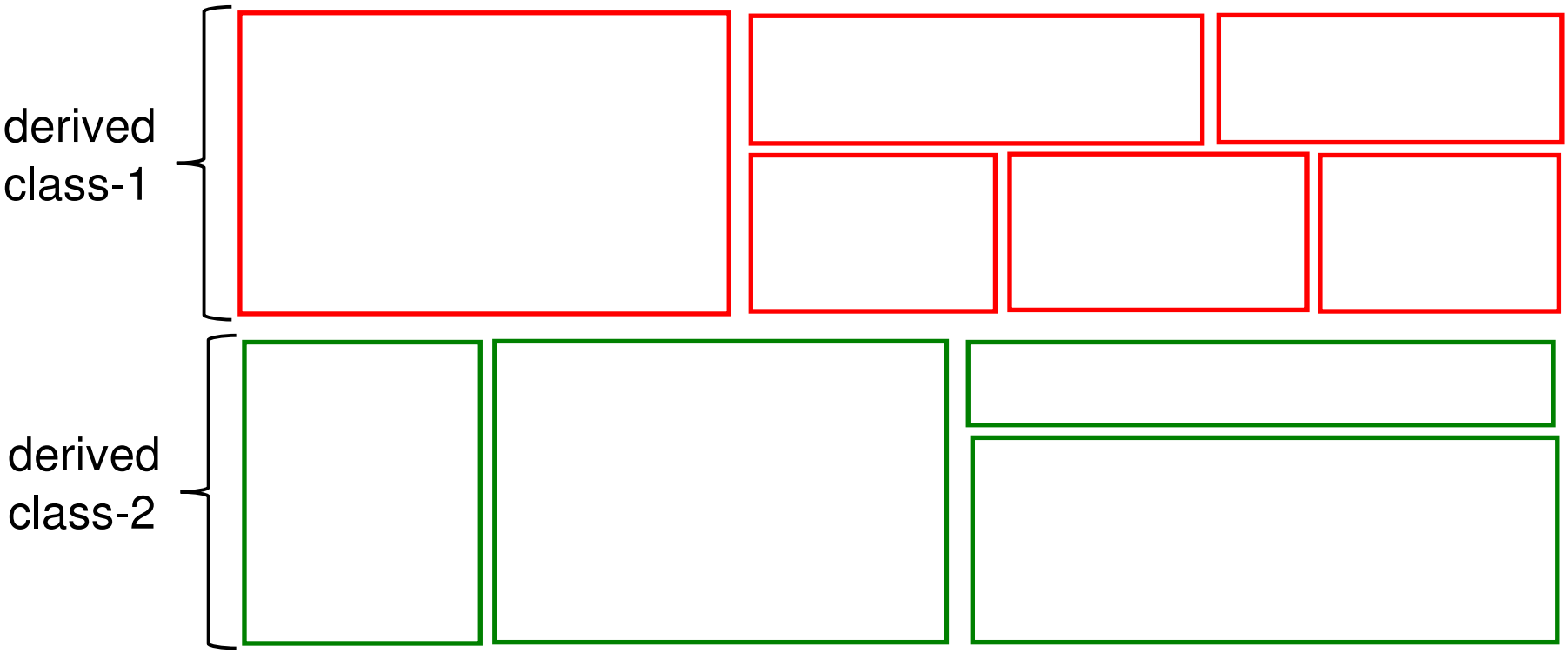
Generalized Horizontal Partitioning
•Add edges between boxes with disjoint guards
• each connected component == a field

Step 4



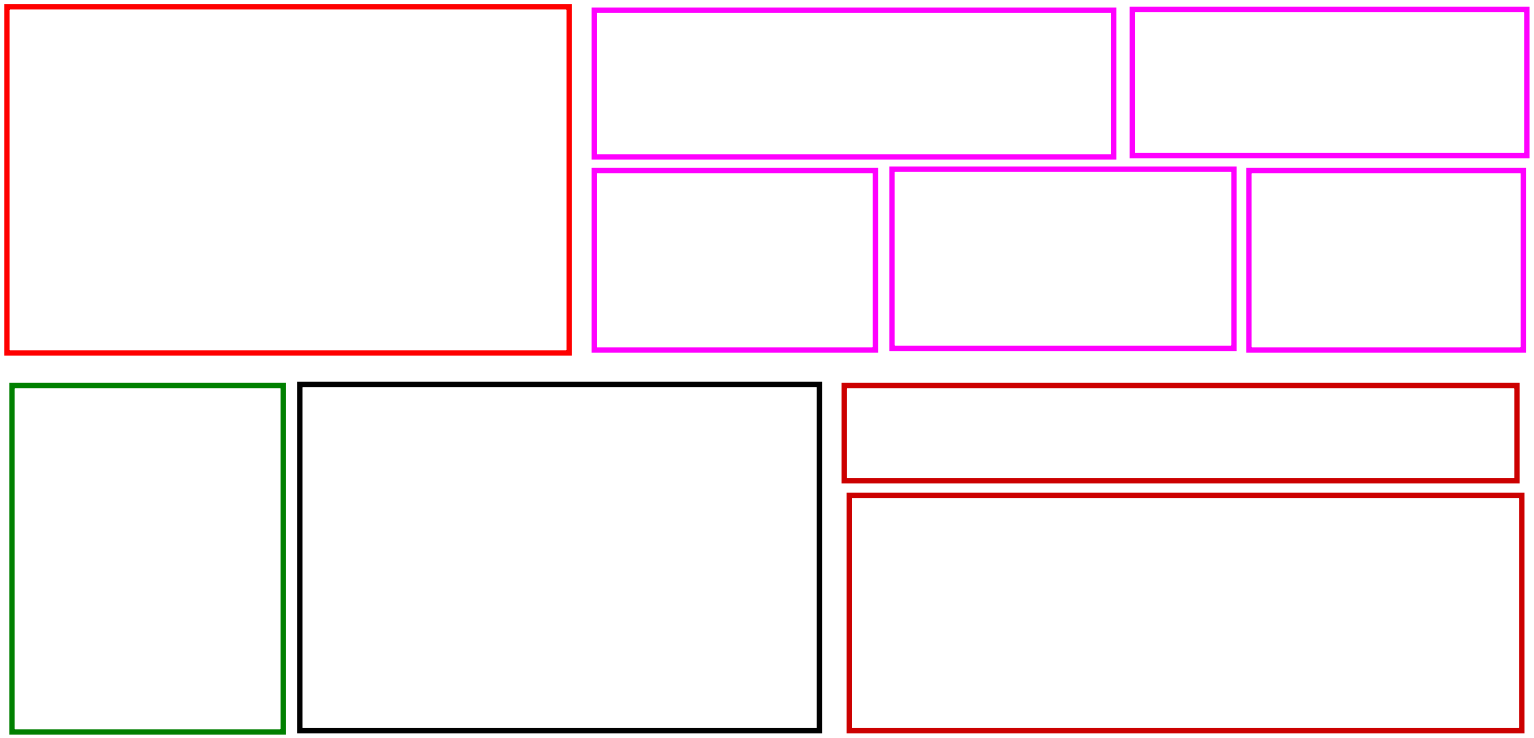
Generalized Vertical Partitioning
•Add edges between boxes with overlapping guards
• each connected component == a derived class

Step 4



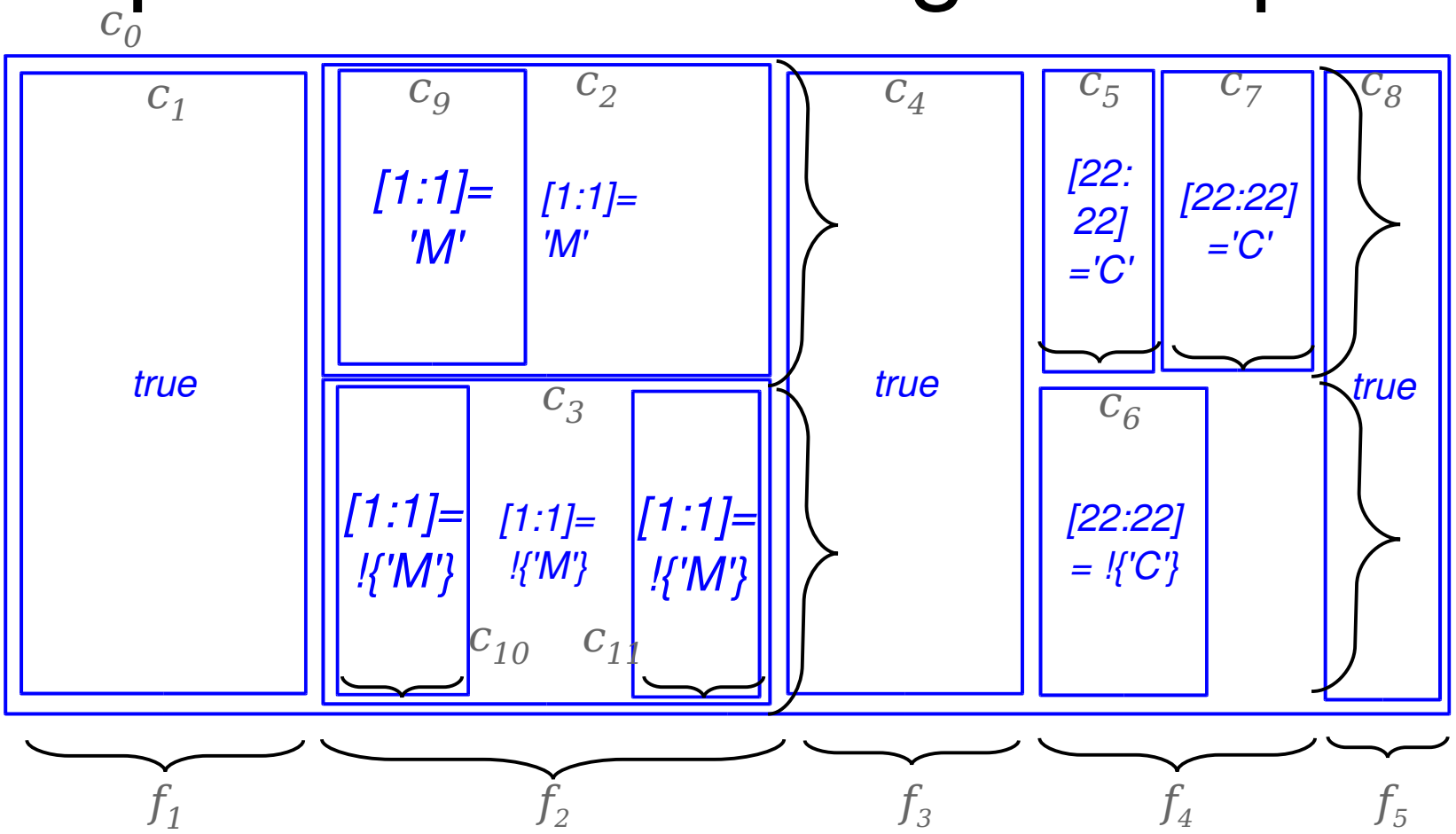
Generalized Vertical Partitioning
•Add edges between boxes with overlapping guards
• each connected component == a derived class

Step 4



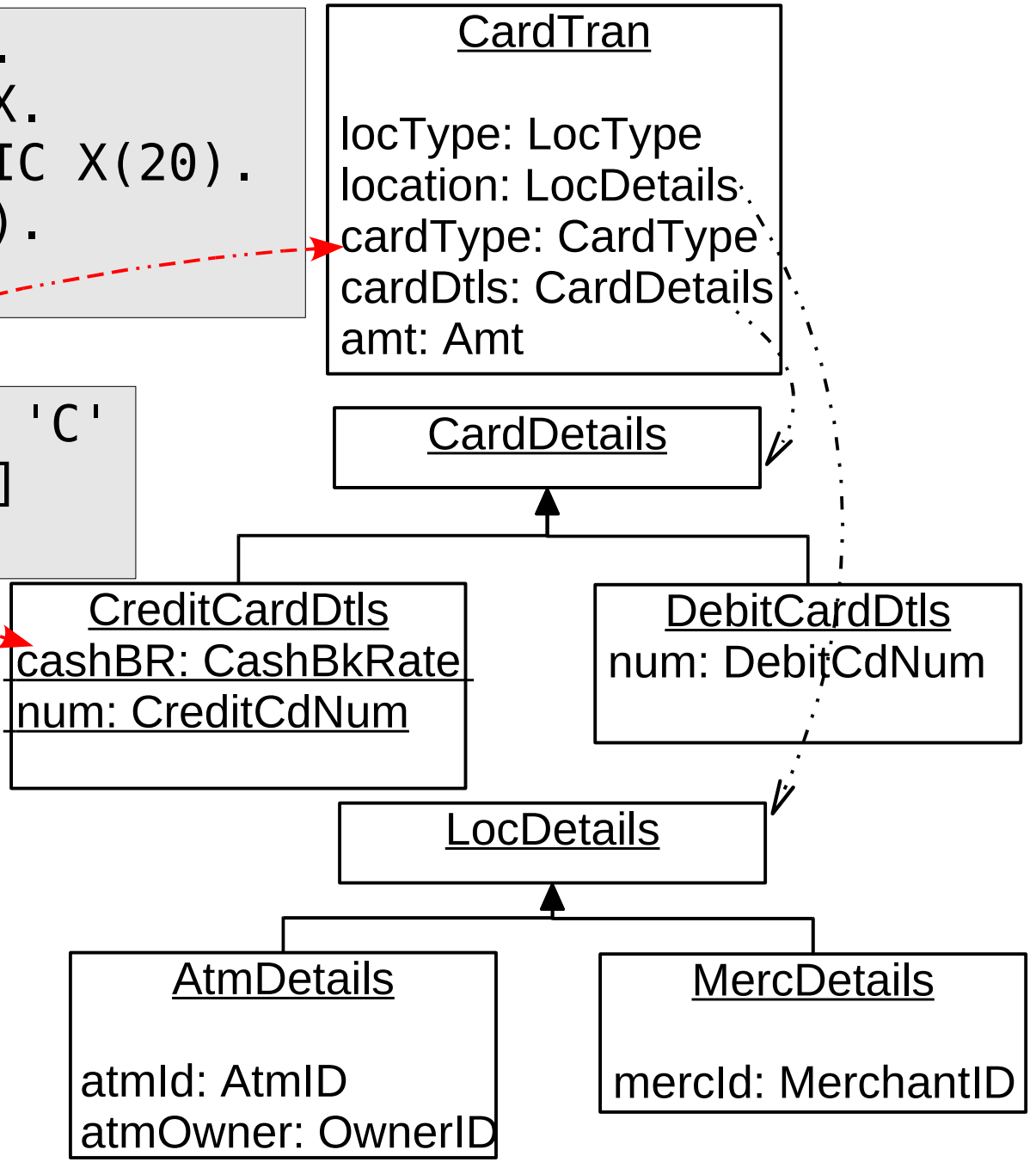
Generalized Horizontal Partitioning
•Add edges between boxes with disjoint guards
• each connected component == a field

Step 4 on the running example ...



```
01 CARD-TRANSACTION-REC.
   05 LOCATION-TYPE PIC X.
   05 LOCATION-DETAILS PIC X(20).
   05 CARD-INFO PIC X(19).
   05 AMT PIC X(4).
```

```
/7/ IF CARD-INFO[1:1] = 'C'
/8/   MOVE CARD-INFO[2:3]
        TO CASHBACK-RATE
```



- How can we say if a given OO model is correct for a given program?
- Executing the program using an altered data representation as suggested by the OOM does not affect the observable behavior of the program.
  - See [ICSE '06] for details

Details of Step 1: Computing guarded dependences

- guard ▶source →target
  - source is a pair memory range @ program-pt
  - target is similar (however, we restrict ourself to variable dereference sites)
  - guard is a predicate on the state at source program-point.
- when guard is true, value at source may reach target (via some sequence of copies)

Guarded dependence analysis

- Guarded dependences
  - capture transitive data-dependences
  - capture conditions under which dependence is manifested
- Parametric guarded dependence computation
  - parameterized by abstraction for guards
  - can be computed in polynomial time for simple (common type of) guards

Transfer functions (without guards)

Statement S	$\alpha_{pi}[S] : 2^{\mathcal{D}_{pi}} \rightarrow 2^{\mathcal{D}_{pi}}$
`WRITE Y`^{d}	λOut. Out $\cup \{ Y \rightsquigarrow d \}$
`READ Y`^{d}	λOut. $\{ Y \rightsquigarrow d \} \cup$ $\{ r \rightsquigarrow t \mid r \rightsquigarrow t \in$ Out and $r \not\subseteq Y \}$
`MOVE X`^{dX} `TO Y`^{dY} where `Y` $= [y_1 : y_2]$ and `X` $= [x_1 : x_2]$	λOut. $\{ X \rightsquigarrow dX, Y \rightsquigarrow dY \} \cup$ $\{ r \rightsquigarrow t \mid r \rightsquigarrow t \in \text{Out} \wedge r \not\subseteq Y \} \cup$ $\{ r' \rightsquigarrow t \mid r \rightsquigarrow t \in \text{Out}, r \subset Y,$ and $r' = r - y_1 + x_1 \}$
`ASSUME pred`	λOut. Out $\cup \{ r \rightsquigarrow d \mid$ pred contains a data-reference d to a range $r \}$

Backward dataflow analysis. Dataflow fact is: *set of* memory-range X variable-reference-site. Meet operation: set union.

Transfer functions (with guards)

$$\alpha_{eps}[t](\mathbf{Out}) = \{true \rhd r' \rightsquigarrow d' \mid r' \rightsquigarrow d' \in \alpha_{pi}[t]\{\}\} \bigcup \{\alpha_g[t](g) \rhd r' \rightsquigarrow d' \mid g \rhd r \rightsquigarrow d \in \mathbf{Out}, r' \rightsquigarrow d' \in \alpha_{pi}[t]\{r \rightsquigarrow d\}\}$$

$\alpha_g[t](g)$ = weakest pre-condition semantics; i.e.,

broadest condition *before* statement *t* that implies *g* after statement *t*.

Ensuring polynomial-time analysis

- Atomic predicates
  - variable = constant | variable $\notin$ set-of-constants
  - $x = 1, \quad y \neq 2, \quad z \notin \{1,2\}$
- Each guard is
  - a conjunction of atomic predicates
  - (at most one per variable)
- Use Map(memory-range X variable-reference-site, guard) as dataflow fact domain, instead of memory-range X variable-reference-site X guard.

Algorithm 2 : Contributions

- An efficient approach to infer OO data models from weakly-typed programs
- Inferred models are provably compact and *correct*
- Prototype implementation, and manual examination of results
- Therefore, is a sound basis for program understanding, migration, and transformation

Related work

- Canfora et al. [SEKE 96]
- O'Callahan and Jackson [ICSE 97]
- van Deursen and Moonen [WCRE 98, ...]
- Eidorff et al. [POPL 99]
- Ramalingam, Field, and Tip [POPL 99]
- Balakrishnan and Reps [CC 04]

- Distinguishing attributes of our work:
  - path-sensitive analysis
  - semantic correctness criterion of inferred OO model