Patti

Compiling Unification-Based Finite-State Automata
into Machine Instructions for PowerPC

Sascha Brawer

http://www.coli.uni-sb.de/~brawer
brawer@acm.org

Patti

Overview

- Past Development, Applications, Patti's place in a larger context
- Formalism
- Runtime Execution
- Optimizations
- Evaluation

More information

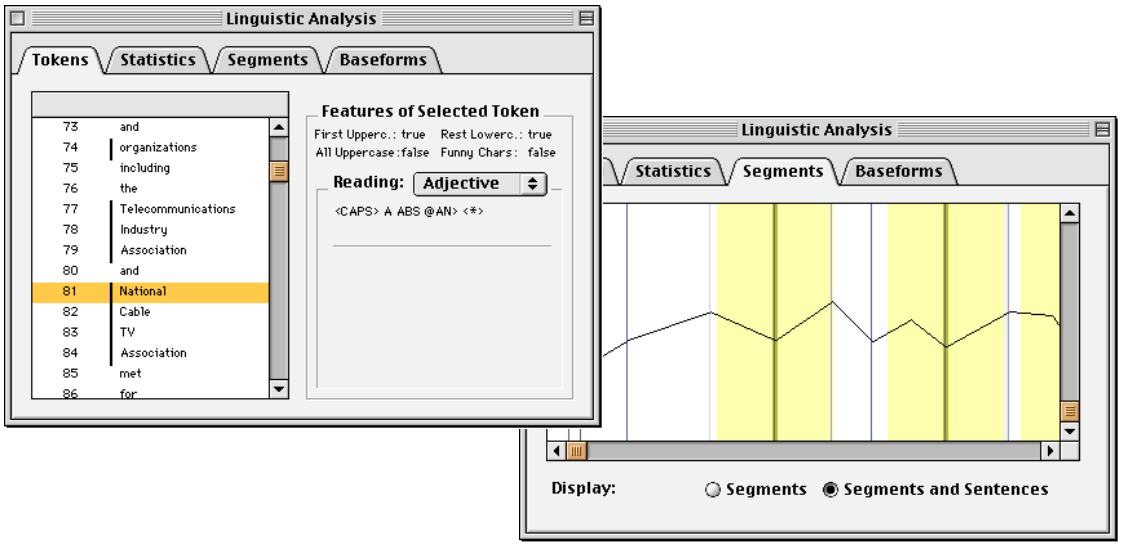
- http://www.coli.uni-sb.de/~brawer/articles/patti

Past Development

Past development of the "Patti" project

- 1995/96: Comp. Linguistics, University of the Saarland, Germany
  - Severely restricted formalism → data structures processed by C unifier
- 1997: Advanced Technology Group, Apple Computer, USA
  - Still quite restricted formalism → PowerPC assembly code for matching/unification
  - September 1997: Apple discontinues ATG
- Future
  - More general formalism → C/assembly code for efficient and *robust* parsing

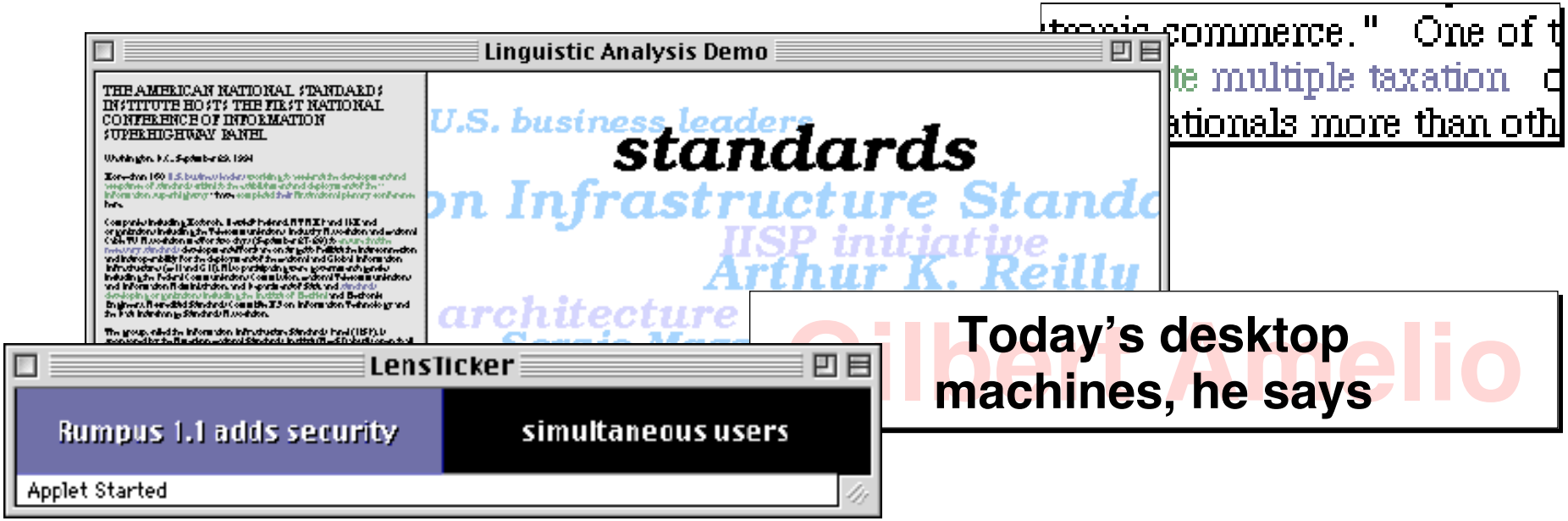
Applications



Applications for Linguistic Analysis/"Patti" at Apple ATG

- Compiler output embedded into Shared Library
  - C and Lisp API
  - Intention: general-purpose linguistic analysis → part of MacOS

Applications



Applications for Linguistic Analysis/"Patti" at Apple ATG

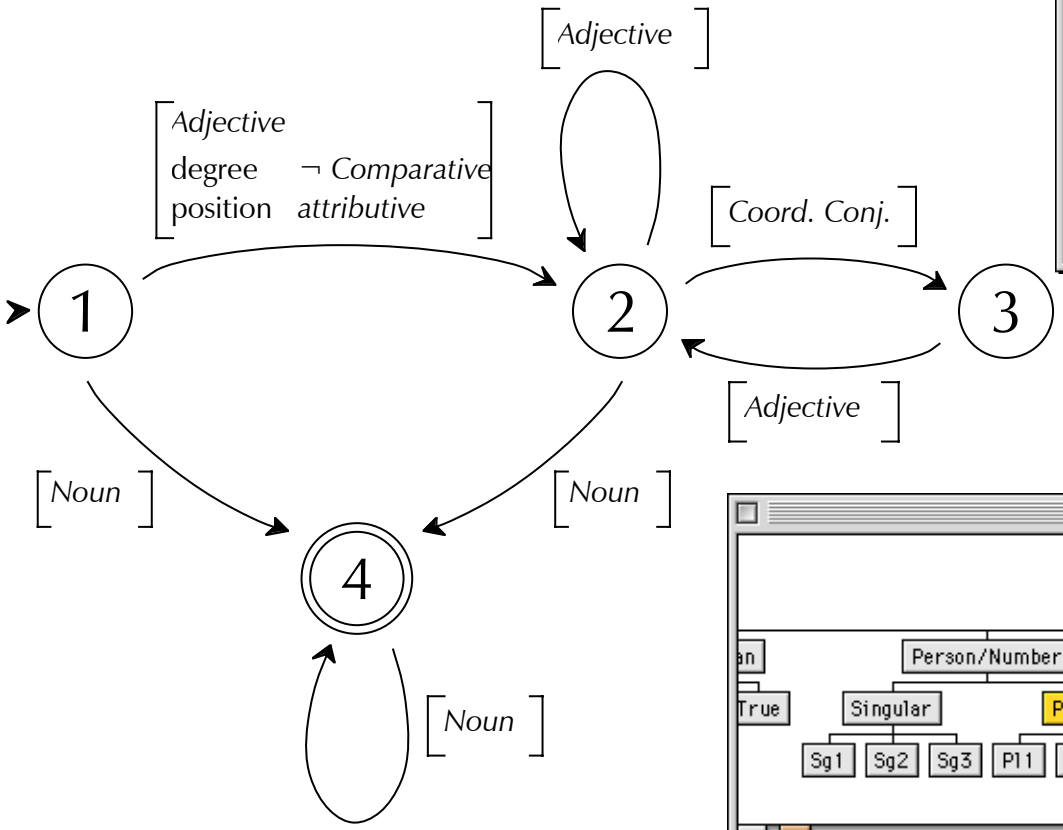
- Several viewers for dynamic display of document content
  - Interactive browsing + background ticker for peripheral vision
  - SIGCHI Bulletin Vol. 30, Number 2, April 1988

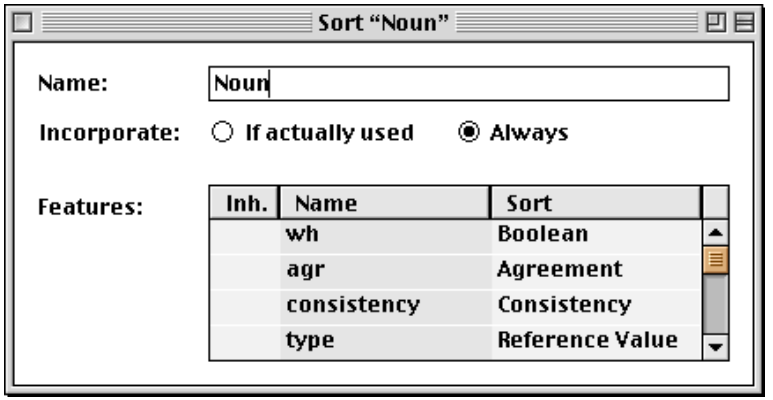
Patti's Place in a Larger Context

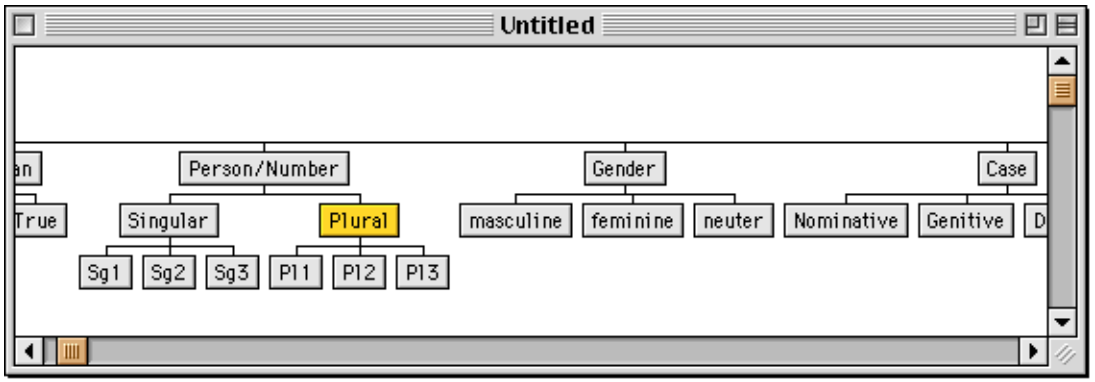
"Patti" was just one component in a larger system

- Part-of-Speech tagging
  - English Constraint Grammar by Lingsoft (Voutilainen et al.)
- Text segmentation
- **Pattern matching — "Patti"**
- Detection of structural elements (itemized lists, titles)
- Identification of technical terminology
- Anaphora resolution
- Determination of most topical noun phrases
  - Quantitative salience measure
- Identification of "context" of topical NPs (for display to users)

Formalism


Adjective
Adjective
degree
position
¬ Comparative
attributive
Coord. Conj.
1
2
3
Adjective
Noun
Noun
4
Noun


Sort "Noun"
Name:
Noun
Incorporate:
If actually used
Always
Features:
Inh.
Name
Sort
wh
Boolean
agr
Agreement
consistency
Consistency
type
Reference Value


Untitled
Person/Number
Gender
Case
True
Singular
Plural
masculine
feminine
neuter
Nominative
Genitive
Sg1
Sg2
Sg3
Pl1
Pl2
Pl3

Formalism

Formalism: Non-Deterministic Finite-State Automata

- Non-standard input "alphabet" and transition labels
  - Typed Feature Structures that conform to a restricted type logic
- Transition can be taken iff. its label is unifiable with one of the readings of the current token
- Two sources of non-determinism
  - Ambiguous input — one token has several readings
  - Ambiguous grammar — multiple transition labels match the same reading
- Output: longest non-overlapping ranges of matching tokens
  - "Greedy Match"
  - Currently, no output *data* is constructed. Non-trivial, but feasible extension with some potential impact on efficiency

Formalism

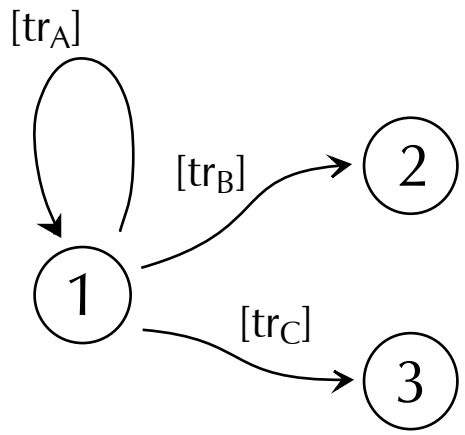
Restrictions

- Simple-inheritance type hierarchy
- Disjunction, negation, conjunction only for atoms
- No lists or sets
- No structure sharing/co-indexing
- Certain other constraints (mainly on Appropriateness Conditions)

Why so rigid?

- "The need for speed"
  - AVM can be represented as one bit-vector, no pointers
  - Some restrictions could probably be weakened while retaining efficiency
    - Further research

Runtime Execution: States



Find 1st Match

```
state_1:
  stateEntry
  if performChecks (trA)
    call n_B
    jump state_1
  if performChecks (trB)
    call n_C
    jump state_2
  if performChecks (trC)
    jump state_3
  pop
```

Find Other Matches

```
n_B:
  if performChecks (trB)
    push <state_2, …>
  /* fall through */
n_C:
  if performChecks (trC)
    push <state_3, …>
  return
```

Runtime Execution: States

Code for each state

- Initialization
- Find the first transition that can be taken
- If there is such a transition:
  - Find other transitions that could be taken as well
  - Push information onto stack that will be needed to establish the FSA configurations that would result from taking those “other” transitions
  - Jump to the code for the new state
- If no transition can be taken:
  - Pop FSA configuration from stack
  - ~ Backtracking

Runtime Execution: States

Some remarks on efficiency

- Per-state initialization: Load register, increment pointer
  - PowerPC: 1 machine cycle (in case of data cache hit)
- FSA jumps to another state = CPU jumps to another code segment
  - no explicit representation of current state besides CPU's program counter
  - PowerPC: zero machine cycles (under certain conditions)
- Fast check for most unification *mismatches*
  - "Unification Oracle", presented later in this talk
  - PowerPC: 1 machine cycle in most cases where transition can not be taken
- Very few memory accesses
  - in most cases, one single machine word is loaded per state
  - FSA configuration stack only accessed in case of non-determinism

Runtime Execution: Unification

Each transition label AVM is compiled to code

- Checks whether the unification with one of the readings of the current token would succeed
- It is an old idea to compile feature-logic grammars to code
  - ALE (Carpenter/Penn, 1995): AVMs → Prolog
  - AMALIA (Wintner/Francez, 1994): AVMs → instructions for Abstract Machine
- So, what is new about this?
  - AVMs → instructions for a *concrete* machine
  - compilation result is directly executed by the CPU (instead of WAM or AMALIA)
  - better heuristics than a general-purpose {Prolog, Lisp, C, …} compiler would use (due to knowledge about the task)

Runtime Execution: Unification

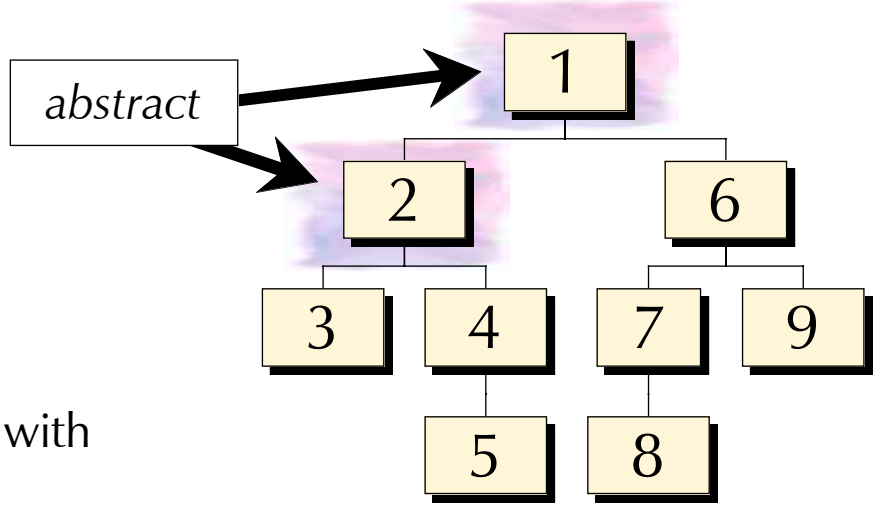
To check whether the label AVM is unifiable with input

- Execute unification oracle (explained later)
- For each reading of the current token
  - Type unification
  - For each accessed word in feature bit-vector (this depends on the encountered type)
    - Load contents into register
    - Execute checks on that word (boolean operations)
    - In case of unification failure: try next reading
  - If all checks on all accessed words succeed: unification success
- After having tried the last reading: unification failure

Runtime Execution: Unification

Type Unification

- Compiler determines type IDs
  - pre-order, depth-first traversal
- Question at runtime
  - Can current reading's type *R* be unified with transition label's type *L*?
  - $FirstUnif(L) \leq R \leq LastUnif(L)$
- *FirstUnif*(*L*) = most general ancestor of *L* that can be instantiated
  - compiler knows about abstract types
- *LastUnif*(*L*) = last descendant of *L*

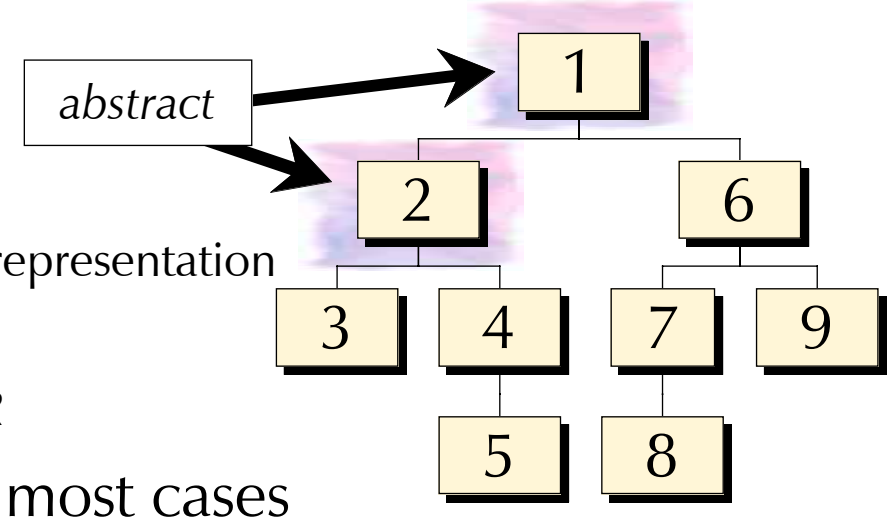


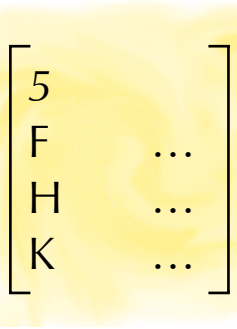
L	*FirstUnif*(*L*)	*LastUnif*(*L*)
3	3	3
4	4	5
5	4	5
6	6	9
7	6	8
8	6	8
9	6	9

Runtime Execution: Unification

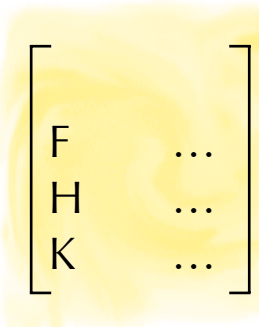
Type Dispatch

- if *R* is more general than *L*
  - some label features are not part of *R*'s representation
  - these checks succeed by definition
  - execute different code depending on *R*
- Type dispatch can be omitted in most cases
  - typically, only leaf types are instantiable

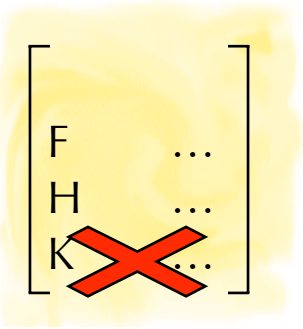




Transition Label
L = 5



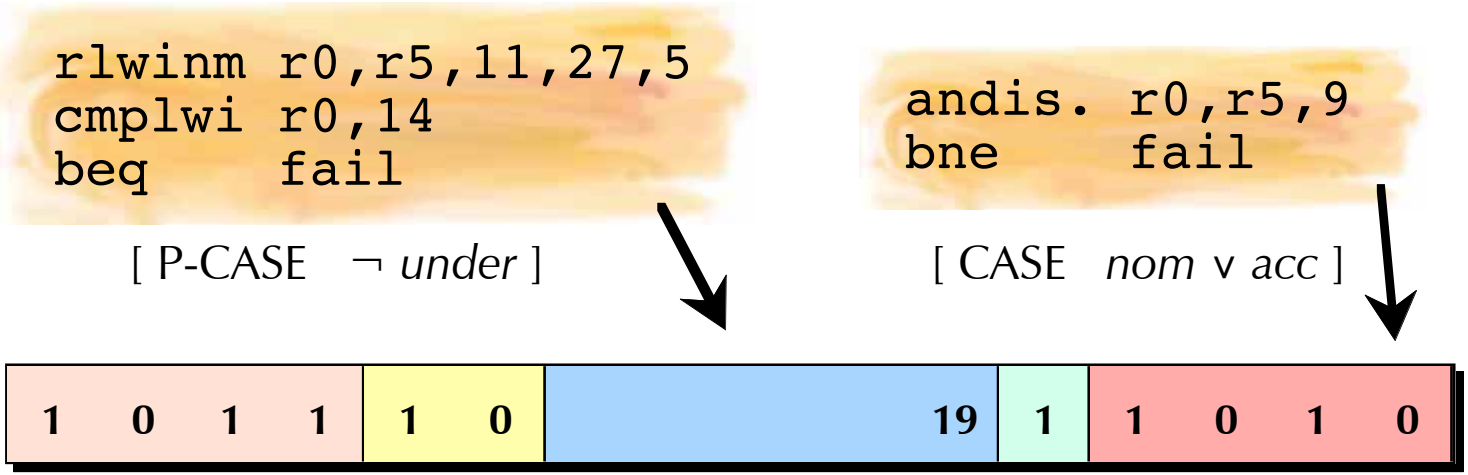
Checks for R = 5



Checks for R = 4

Type	*Introduced Features*
1	F, G
2	H
4	I, J
5	K

Runtime Execution: Unification



Feature Unification

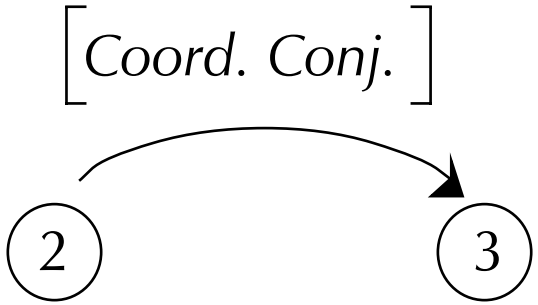
- Load parts of bit-vector (unit: one machine word)
- Execute checks on that part
  - Integer Instructions (Logical, Compare, PowerPC Rotate & Extract)
  - Depending on PoS tagger, a single reading can have multiple values for a feature
    - Examples: *Person, Number*

Optimizations: Unification Oracle

Unification Failure Oracle

- Predicts reliably that no reading will unify with a transition label AVM of type *L*
  - … without actually looking at the individual readings
- Part of *per-token* runtime representation:
  - Bit-OR [(type ID of reading's AVM) mod 32]
- Action at runtime for a transition label AVM of type *L*, before looping over the readings:
  - Check whether bit #(*L* mod 32) is set
  - If not:
    → none of the readings can unify
    → no need to enter the loop
    → save CPU cycles and memory accesses

Optimizations: Unification Oracle

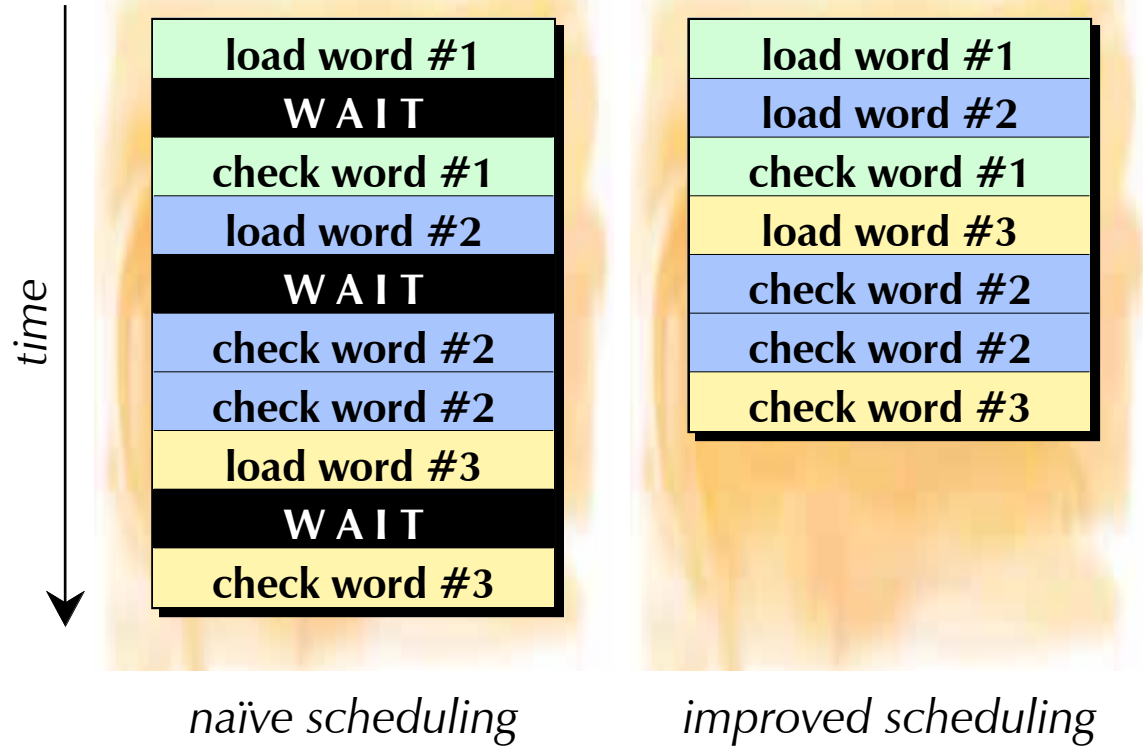



```
andi. r0, r9, 4
bne-  state_3
```

Unification Success Oracle

- Under certain conditions …
  - transition label is empty besides type *L*
  - bit *#L* of oracle word is only set if the current token has a reading of type *L*
- … the Unification Oracle reliably predicts unification success
  - … without actually looking at the individual readings

Optimizations: Instruction Scheduling



Instruction Scheduling increases performance

- no delay after load; better use of superscalarity

Optimizations

Data Cache Control

- inform CPU that a specific memory location will be accessed in the near future
  - memory subsystem can transfer contents of that address into the cache while the normal execution continues
  - used by very few C compilers (e.g. not by Metrowerks CodeWarrior Rel. 2)
  - → 30% faster (caches are important!)

Static Branch Prediction

- Patti knows that certain branches are likely/unlikely to be taken
  - Example: Non-determinism stack will almost never overflow
- heuristics based on task knowledge are better than general ones

Evaluation

Evaluation

- find the range of simple NPs
- three real-world texts
  - magazine article — 4092 tokens, 206 sentences
  - “readme” file — 2894 tokens, 224 sentences
  - newswire text — 681 tokens, 22 sentences
- three machines
  - PPC 601, 60 MHz — Power Macintosh 6100/60av, 40 MB RAM, MacOS 8.0
  - PPC 604, 150 MHz — Power Macintosh 9500/150, 88 MB RAM, MacOS 8.0
  - PPC G3, 334 MHz — PowerMacintosh G3, 192 MB RAM, MacOS 8.1

Evaluation

	601/66 MHz		604/150 MHz		G3/333 MHz	
	Time	Tok/s	Time	Tok/s	Time	Tok/s
Magazine Article	2182 μs	1.9 Mio	490 μs	8.4 Mio	192 μs	21.3 Mio
Readme File	1474 μs	2.0 Mio	366 μs	7.9 Mio	152 μs	19.0 Mio
Newswire Text	332 μs	2.1 Mio	101 μs	6.7 Mio	39 μs	17.5 Mio

How long does it take to find the NPs?

- total execution time for per-sentence matching code
  - Profiler of Metrowerks CodeWarrior Professional Release 2
  - uses PPC Timing Facility (clock granularity: 128 ns)

Evaluation: Common Objections

Common objections to the evaluation

- *"Evaluating Caches, not Algorithms"*
  - *"You are measuring CPU and cache, not the efficiency of your algorithms."*
  - ▸ Agreed. But that's what compiling is about—making good use of the means offered by modern computer architecture.
- *"PoS Tagging, Disk Access not Counted"*
  - *"Your results are not realistic. Factors of utmost influence upon efficiency, such as disk access time or the speed of the Part-of-Speech Tagger, have been neglected in the evaluation."*
  - ▸ These factors are relevant for evaluating a system as a whole, but not for evaluating Patti.

Evaluation: Common Objections

Common objections to the evaluation

- *"Virtual Memory not Counted"*
  - *"Your results are not realistic. If 21 million tokens were to be processed, they would not fit into the working set, and Virtual Memory (paging) would terribly reduce the speed."*
  - ▸ Agreed. However, with a reasonable architecture, neither tagger nor partial parser would work sequentially on the entire text. Instead, much smaller entities (e.g. paragraphs) would be processed incrementally; only very little information needs to be preserved. → Keep the working set small!
- *"Conversion into Bit-Vectors not Counted"*
  - *"You did not measure how long it takes to convert the output of the PoS tagger into Patti's bit-vectors."*
  - ▸ A textual representation of the tagger's findings is not really needed. The tagger could generate in-memory structures as required by the subsequent components.

Evaluation: Common Objections

Common objections to the evaluation

- *"Scalability?"*
  - *"Your automaton is a toy example. What about larger grammars?"*
  - **More states** — Unlikely to affect performance. The time needed to traverse a FSA does not depend on the number of its states. However, more machine code makes Instruction Cache less effective → (rather small) degradation possible.
  - **More types** — Some degradation likely. The Unification Oracle will be less effective.
  - **More features** — No degradation at all. Larger bit-vectors have no effect due to data cache control instructions.
  - **Larger transition labels** — Some degradation likely. More checks → more machine code → more Instruction Cache misses. Alleviated by Branch Prediction and Unification Oracle.
  - **More automata** — Degradation. Linear growth of execution time with number of automata.