Declarative Programming

Towards a language that fundamentally abstracts away from time
Overview

• Problem & Solution
• Comparing Paradigms
• Declare
• Demo
• Conclusions
CPU Power

MIPS/CPU clock speed

MIPS per die
Problem & Solution

• Problem
  – poor maintainability of software
  – last 6 years CPU power only increased through parallelism

• Solution: declarative programming
  – abstract away from time
  – can therefore be executed in parallel
Comparing Paradigms

• Imperative
• Functional
• Declare
Imperative (e.g. Java)

- **Assignments**: changeable fields and variables
- Program **changes** the **state** of the system while running
- Strict sequential definition puts a **straitjacket** on the execution
- **Monolithic** process
- Side-effects make it difficult to reason about the **correctness** of the system
Functional (e.g. Haskell)

- No side-effects, only **constants**
- Lazy and partial **parallel** execution
- High local correctness **provability**
- The main function has the **whole** system state as parameter and output
- **Monolithic** state transition
- Not a natural fit to express '**dynamic**' systems (e.g. GUI's)
• Origin
• Language
• Examples
• Demo
Conception of *Declare*

- **Functional**
  - Functions
  - Lambda

- **Object Oriented**
  - Inheritance
  - Generics

- **OO Modeling**
  - Opposites
  - Composition

- **Spreadsheet**
  - Cells (slots)
  - Declarative Invariants
Language Ideas

- Regular full-fledged OO constructs
- Strong abstraction mechanisms to make any aspect reusable
- Functions as in functional programming languages
- Easily declare enforceable invariants, as in spreadsheets (up to 90% of the functionality)
- Explicitly declare temporal behavior, based on history of state
Language Constructs

- Classes, Inheritance, Generics
- Attributes, References, Composition
  Opposite References, Multiplicity
- Functions (named/anonymous/generic)
  - No side-effects
- Constraints (named/anonymous/generic)
  - Enforced invariants (as in spreadsheets)
  - Enforced temporal constraints
- Types for Functions, Constraints, Tuples, Properties
Runtime Characteristics

• The runtime is a Java library
• Each constraint is running on an arbitrary thread of a configurable thread-pool
• Invariant constraints are run:
  – once for initialization; when a slot is changed that is accessed in a previous run
  – State only changes when the invariant is broken
• Temporal constraints run ones for each time that a change has occurred
Demo time!
Execution Environment

Your System

Declare libraries

Constraints

Slots

Transactional Memory

Java
Transactional Memory

- Double isolation (read-consistency)
  - Changing state not visible outside transaction
  - State outside transaction appears frozen
  - Guaranties functional semantics

- Implied Nested Transaction
  - Based on class and constraint composition
  - Atomic constraints are leaf-transactions running in multiple threads
  - Feature state not visible outside container if not all container-constraints are met
Almost the end

- *Declare* in the MDD world
- Future plans
- Conclusions
- Q & A
Declare in the MDD world

- Building Language Workbenches
  - Graphical engines
  - Transformations
  - Model / Type checking / ...

- DSL Execution
  - Helps separating the execution concern from the domain concern
  - Internal DSL (using generics and lambdas)
  - External DSL (target of generated code)
Future Plans

• Implement Language Parser/Compiler
• Improve runtime library:
  – Transactional Memory
  – Use Java lambda's (JDK8)
• Evangelize paradigm shift
• Spreadsheet-like graphical IDE/Runtime
• Hands-on for CG2013!
Conclusions

• We need to change the way we conceptualize software
  – This paradigm shift is the biggest challenge

• Parallelism needed on all levels
  – Declarative programs can be executed efficiently
  – Transactional memory is key to a multi-threaded runtime environment

• We have proven that the Declare concept works!