JavaParty - a domain-specific language for cluster computing

Dagstuhl Seminar 23.03.-28.03.2003

“Domain-Specific Program Generation”

UNIVERSITÄT KARLSRUHE (TH)
Fakultät für Informatik

Bernhard Haumacher
Institut für Programmstrukturen
und Datenverarbeitung · Prof. Tichy

Classification of the topic

- What is *the* domain?
  - "a set of people"
    - High-performance computing "in Java"
  - "a set of techniques"
    - cluster computing
    - easy to learn programming environment
    - efficient remote method invocation
    - efficient object marshaling
  - "a set of programs"
    - parallel data mining
    - parallel image analysis

JavaParty: Why?, What?, How? (1)

- Why?
  - Java has built-in support for parallelism
    - but only little support for distribution.
    - ⇒ an easy to use parallel distributed environment required.
- What?
  - JavaParty provides
    - Transparent remote classes and objects with
      - transparent distributed threads
      - object migration
      - single system view of a cluster like a single virtual machine

JavaParty: Why?, What?, How?

- How?
  - Source-to-source program transformation
  - Library support, many "Abstract machines" for
    - low-latency remote method invocation
    - high-performance object marshaling
    - cluster-wide transparent thread semantics

A closer look at the “what?”

- Transformation of an annotated parallel Java Program into a parallel distributed one.
  - Classes may be declared "remote".
  - Remote classes and their instances can be accessed from everywhere within the distributed environment.
  - Remote classes and their instances are the first-class objects in the distributed environment.
  - Remote objects can move within the distributed environment after creation.

```
public remote class R {
   ...
}
```

A closer look at the “how?”

- The JavaParty transformation maps JavaParty code to pure Java plus remote method invocation.
  a) Regular RMI shipped with each JDK, not well suited for HPC
  b) High performance version of RMI called KaRMI
     - adapted to high-performance network hardware
     - advanced features:
       - fast object marshaling "uka.transport"
       - transparent distributed threads
       - remote monitor acquisition.

  domain-specific optimization

  ⇒ domain-specific optimization uses transformation

  ⇒ slightly different transformations for different target libraries.

The JavaParty transformation

- There is more than one...
  1) Remote class transformation ("server-side")
     - make remote objects accessible via RMI
  2) Transformation of method bodies ("client-side")
     - Rewrite remote object accesses where neccessary
  3) Generation of glue code to transparently access remote objects like regular Java objects.
  4) Generation of fast marshaling routines
  5) Stub/skeleton generation for KaRMI
     - What rmic does for regular RMI

Remote class transformation

- Operates on the interface of a remote class
  - make a remote class and its instances accessible via RMI
    - separate a class implementation into its static and non-static parts
      - static and non-static parts are represented as different objects at runtime:
        - one representing the class within the distributed environment
        - one for each instance of a remote class
    - generate getters and setters
    - generate array accessors

Method body transformation

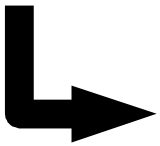
- Rewrites are extremely context sensitive:
  - Rewrite direct member access with calls to generated getters and setters (only “remote” accesses)
    - Rewrite array accesses
  - Rewrite method invocations to direct them to the appropriate runtime object (class or instance)
  - Rewrite special constructs
    - synchronized (x) {...}
    - Weave in user object distribution annotations
  - Rewrite type references to reflect the split between static and non-static parts

Generation of fast marshaling routines

- The magic, the "regular JDK object serialization" does this via (slow) dynamic introspection
  - fast marshaling requires marshal/unmarshal methods/constructors for transportable classes.
    - tedious and error prone to construct by hand
      - like the XDR problem/solution
    - closely interact with the "abstract marshaling machine"
      - represented by the marshaling stream classes.
      - may require changes, whenever the API of the marshaling machine evolves.
  - ⇒ generate marshal methods from the class signature
    - the (Java) class is the specification (in contrast to XDR)

Example: Remote class transformation

```
remote class R {
    ...
    T[]...[] m;
    ...
}
```



```
interface R_instance_interface
    extends RemoteObject_instance_interface
{
    ...
    public T getM(int d0, ..., int dn)
        throws RemoteException;
    ...
}

class R_instance_implementation
    extends RemoteObject_instance_implementation
    implements R_instance_interface
{
    T[]...[] m;
    ...
    public T getM(int d0, ..., int dn) {
        return this.m[d0]...[dn];
    }
}
```

Properties and requirements

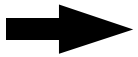
- Large amounts of "plain" code are generated
  - specifying the transformation result in terms of abstract syntax tree node constructors is unreadable and unmaintainable (the first approach, JavaParty was implemented with)
  - The transformation language requires code fragment literals
  - ⇒ use JTS/Jak, JSE, Maya

Example: Method body transformation

- Invocation of a static method from within a static context of the same class using a non-static reference

```
remote class R {
   void foo() {
      ...
      bar();
      ...
   }

   static void bar() {
      ...
   }
}
```



```
class R_instance_implementation {
   void foo() {
      ...
      R.bar();
      ...
   }
}
class R_class_implementation {
   void bar() {
      ...
   }
}
```

Properties and requirements

- The transformation is guided by the program meta model, not the pure (abstract) syntax tree.
  - semantic analysis of a regular (Java-) compiler is required.
    - Embed the transformation into an existing Java compiler
      - espresso, pizza, gj
    - use a meta programming framework with an elaborated view of the language's type system and meta model:
      - RECODER, OpenJava

Example: Language extension

```
remote class R {
   public static final const int[] x =
      new int[] {7, 13, 42};
   void foo() {
      /** @at n */
      new R();
   }
}
```

- JavaParty is a (small) *extension* of Java.

⇒ base the transformation on an *extensible* Java compiler:

  - JaCo, Polyglot

Last but not least

- The transformation order matters
- Matches decide about applicability of other rules
  - specify the rule set separate from rule application order
- ⇒ Use tool allowing constrained pattern substitution
  - Stratego
  - James Gosling's proposal for a refactoring tool based on guarded pattern matching rules (OOP'2003).
    - “pattern => replacement :: conditions;”
    - a.show() => a.setVisible(true) :: a instanceof Component;
  - XSLT-like combination of pattern matching templates and callable named templates

Method body example revisited

```
match
      callStat := fun(args);
if
      fun.isStatic() &&
      fun.getOwner() instanceof RemoteObject &&
      (! context(callStat).isStatic()) &&
      (context(callStat).getOwner() instanceof
         fun.getOwner()) &&
=>
      createReference(fun.getOwner()).fun(args)
```

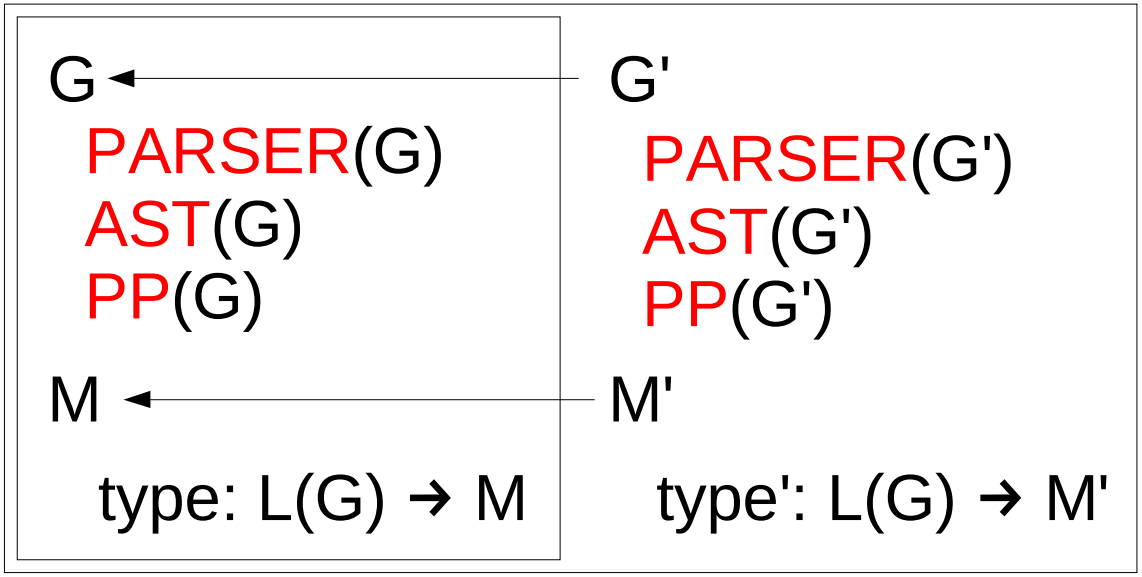
JavaParty current state

- Implemented inside an existing Java compiler
  - a derivate of the generic Java compiler (gj)
- What's good:
  - full access to the program meta model
  - direct byte code generation (if desired)
- What's bad:
  - not extensible: original compiler needs modifications
  - transformation rules are interwoven with their application strategy
  - transformation result is expressed in concrete syntax

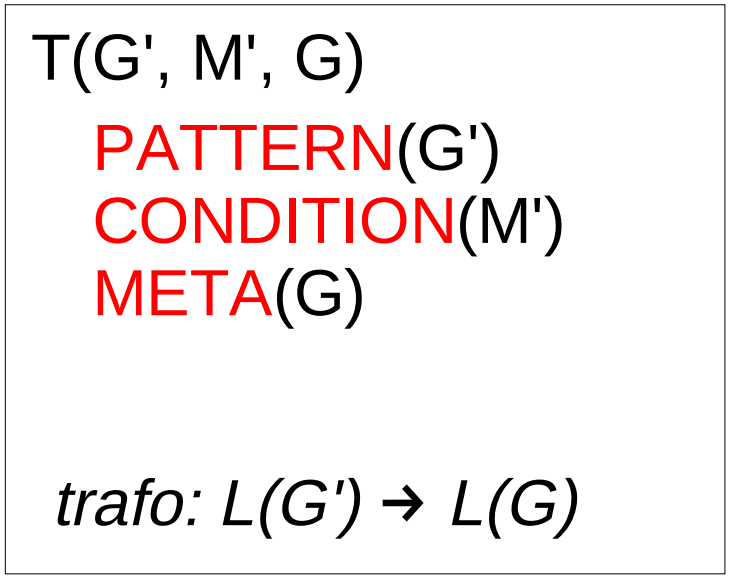
Vision

Extensible framework for DSL implementation

Extensible meta-programming framework



Extensible transformation framework



*Generators