Improving the Reuse of Language Infrastructures

Karl Trygve Kalleberg
University of Bergen
Motivation

• People build language infrastructures all the time
  – compilers, language-specific transformation systems, code analyzers and generators
• Reusing these is surprisingly difficult
  – very few compilers/analyzers are open and extensible
  – limited plug-in capabilities
• Consequences
  – language processing is difficult for most developers
  – light-weight, text-based scripts are preferred
  – new infrastructures mostly built from scratch
  – these systems are frequently brittle and incomplete
  – *programmable transformation systems* seldom used
Language Infrastructure

- parser
- type analyzer
- style checker
- editor
- doc generator
- interpreter

Current clients

- x86 optimizers
- x86 code generator
- x86 exec
- ppc optimizers
- ppc code generator
- ppc exec

- backends
- frontend
- compiler
Language Infrastructure

- **Parser**
- **Type Analyzer**
- **Style Checker**
- **Doc Generator**
- **Interpreter**

**Current Clients**

**Transformation System**

- **Compiler**
- **Backend**
- **Frontend**

**Reuse**

- Grammar
- Typing Logic
- Typing Rules
- Style Logic
- Style Rules
- Analysis Logic
- Machine Format
- IRs (SSA, CFG, ..)

**Optimizers**

- X86 Optimizers
- PPC Optimizers

**Code Generators**

- X86 Code Generator
- PPC Code Generator

**Executors**

- X86 Exec
- PPC Exec
Us and Them and The Other Guys

- **Us** = the software transformation community
  - rewriting of source code and other software artifacts
  - language infrastructure is a means to an end
  - obvious reusers of mainstream language infrastructure
  - some degree of community interoperability and reuse

- **Them** = language providers
  - providers (and maintainers!) of language infrastructure
  - mostly closed solutions; no extensibility
  - design goals do not include code rewriting

- **The Other Guys** = library/framework developers
  - potential users of programmable transformation systems
Why is Reuse so Poor?

• Technical barriers
  – not designed for reuse (no documentation, no libraries)
  – no de facto standards for interoperability
  – poorly compatible implementation languages
  – (incompatible licenses)

• Sociological barriers
  – lack of awareness
  – no project support infrastructure (issue tracker, forums)
  – misconception that “parsing is enough”
  – “not invented here”-syndrome
Some Suggestions

- Technical
  - data integration
    - serialize ASTs (UPTR)
    - experiment with more general interchange formats
  - functional integration
    - co-develop sensible compiler rewriting APIs
- Sociological
  - promote existing language infrastructures
    - place prominently on pt.org
  - combat “not invented here syndrome” – collaborate!
  - point to, and document, success stories
  - promote killer feature: adaptable domain-support
Conclusion

• Current status: “Have solution, need problem”
  – at least, “have product, want clients”
• Promotion and advocacy is necessary
  – examples, documentation, hyperiding
• Open-sourcing improves code reuse
  – potentially high maintenance cost
• Complete openness not required
  – exposing a stable AST interface is already useful
• Tendency towards opening mainstream compilers
  – ECJ DOM, JSR 269 (APT), JSR 199 (compiler API)