Logical Foundations for

Automated Code Generation

5th Estonian Summer School in Computer and Systems Science (ESSCaSS’06)

Bernd Fischer
School of Electronics and Computer Science
University of Southampton

b.fischer@ecs.soton.ac.uk
First Things First: Thanks to Ewen!

Mini-course on

Automated Code Generation

Brazilian Symposium on Formal Methods (SBMF ‘05)

Ewen Denney     Bernd Fischer
Robust Software Engineering Group
NASA Ames Research Center
{edenney,fisch}@email.arc.nasa.gov
What this course is...

• An introduction to
  – *general concepts*
  – *common architectures*
  for automated code generation
• A little history and current trends
• An overview that is
  – high-level
  – highly selective
  – highly biased
What this course is *not*...

- Going to teach you how to use
  - specific tools
  - specific languages
- Technical
- Comprehensive
- A Grand Unified Theory (unfortunately)
Overview

• Introduction to automated code generation

• Themes (= underlying concepts)
  – knowledge, information, and meta-information
  – transformation
  – search and control

• Paradigms (= existing approaches)
  – code-based or generative (incl. template-based)
  – model-based or transformative (incl. MDA)
  – proof-based or deductive (incl. algebraic)

• Certifiable Code Generation
  – combining deductive and generative approaches

• Summary
  – applications
  – research issues
What is automated code generation?

• *Techniques and tools for automatically constructing (low-level) software from (high-level) problem specifications:*
  – domain-specific languages
  – specialized domain modeling (e.g., feature models)
  – transformations and symbolic computations
  – search control
  – domain-specific target platforms (e.g., CLARAty)
What is automated code generation?

• *Techniques and tools for automatically constructing (low-level) software from (high-level) problem specifications:*
  – domain-specific languages
  – specialized domain modeling (e.g., feature models)
  – transformations and symbolic computations
  – search control
  – domain-specific target platforms (e.g., CLARAty)

• *automated code generation = knowledge representation + meta-programming*
What is automated code generation?

• *Techniques and tools for automatically constructing (low-level) software from (high-level) problem specifications:*
  – domain-specific languages
  – specialized domain modeling (e.g., feature models)
  – transformations and symbolic computations
  – search control
  – domain-specific target platforms (e.g., CLARAty)

• *automated code generation =*
  
  *knowledge representation + meta-programming*

• but not…
  – compilation (not source-to-source)
  – partial evaluation (no high-level model)
  – generic programming (no high-level model)
  – aspect-oriented programming (use as transformation technology)
What is automated code generation?

• Techniques and tools for automatically constructing (low-level) software from (high-level) problem specifications:
  – domain-specific languages
  – specialized domain modeling (e.g., feature models)
  – transformations and symbolic computations
  – search control
  – domain-specific target platforms (e.g., CLARAty)

• automated code generation = knowledge representation + meta-programming

• but not…
  – compilation
  – partial evaluation
  – generic programming
  – aspect-oriented programming

“conventional compiler stuff”
What is automated code generation?

• *Techniques and tools for automatically constructing (low-level) software from (high-level) problem specifications:*
  – domain-specific languages
  – specialized domain modeling (e.g., feature models)
  – transformations and symbolic computations
  – search control
  – domain-specific target platforms (e.g., CLARAty)

• *automated code generation = knowledge representation + meta-programming*

• *automated code generation = “optimizations which are beyond the then current state of the compiler art”* [Bal85]
What is a problem specification?

- (Formal) requirements
  - SpecTRM, RSML, SCR
  - UML state charts
- (Formal) specifications
  - algebraic specification
  - logical conjecture
- Models
  - mathematical model (e.g., differential equations)
  - Simulink model
  - UML class diagram (+ OCL)
- Code (+ mark-up)
- Any combination of these
- Anything you want it to be… (almost)
How does a generator work?

Code generation steps:
• read input specification (if necessary)
• check validity of specification
• assemble code fragments ("compile specification")
• optimize (if necessary)
• generate implementation in target language
How does a generator work?

Code generation steps:
• read input specification (if necessary)
• check validity of specification
• assemble code fragments (“compile specification”)
• optimize (if necessary)
• generate implementation in target language

What’s the difference from a compiler then???
• semantic gap is bigger:
  “|model – source code|  >>  |AST – object code|”
• steps may be interleaved and/or repeated (search & control)
→ code generators are more knowledge-intensive
Advantages of Generators

• Increase *productivity*
  – take high-level specification directly to system implementation
    spec → ??? → code
  – fast turn-around

• Increase *reliability*
  – no coding errors

• Raise the level of *intentionality*
  – represent algorithms in domain-specific concepts
    (instead of low-level code based on efficient implementations)

• Avoid the *library scaling* problem
  – automate combination of *n* components
    (instead of having *n!* combinations)
  – increase flexibility over *generic programming*
Simple Generators

- **lex** and **yacc**
  - generate table and driver function, splice in action code
- **cpp** and other macro- / pre-processors
- Template-based meta-programming
  - generate single classes and procedures
- CASE tools
  - GUI represents component configuration,
    “generator” traverses system structure
  - Real-Time Workshop (for Matlab/Simulink)
  - Java stub generators, EJB
- Webpage generators
“Code-Generation-by-Macros”

- Macro-processors are low-cost code generators:

  ```cpp
  #define swap(v,w) {int tmp=v; v=w; w=tmp;}
  ...
  int cur = thermometer();
  if (cur > max) swap(cur, max);

  So, `cpp` yields:

  ```cpp
  int cur = thermometer();
  if (cur > max) {int tmp=cur; cur=max; max=tmp;};
  ```
“Code-Generation-by-Macros”

• Macro-processors are low-cost unreliable code generators:

```cpp
#define swap(v,w) {int tmp=v; v=w; w=tmp;}
...
int tmp = thermometer();
if (tmp > max) swap(tmp, max);
```

So, `cpp` yields:

```cpp
int tmp = thermometer();
if (tmp > max) {int tmp=tmp; tmp=tmp; tmp=tmp;};
```

• Solution: automatic alpha-conversion (*hygienic macros*)
  – only convert variables in binding positions
  – allow controlled “break of the hygiene”
Fancy Generators (I)

- **KIDS (Kestrel Institute, D. Smith)**
  - Generates schedulers, planners
- **PlanWare / SpecWare (Kestrel Institute, D. Smith)**
  - Successor to KIDS: based on category theory
  - Generates operations plans for US Navy from logistics specification
- **Amphion (NASA Ames)**
  - Solves astronomical problems
- **McIlraith (Stanford Knowledge Systems Lab)**
  - Automated web service composition
- **Waldinger (SRI)**
  - Solves geographical queries
- **Automated Interaction Designer (Iterativity, R. Penner)**
  - Generates customized cockpit interface from domain model
SciFinance (SciComp, E. Kant)
  - generates financial programs
    > from differential equations
  - commercially marketed
  - implemented in Mathematica

Ellman’s System (Vassar)
  - generates animation programs
    > from differential equations
  - implemented in Mathematica

AutoFilter / AutoBayes (NASA Ames)
  - generate state estimation code and data analysis programs
    > from differential and statistical equations
    > from generative statistical models
  - implemented in Prolog
What makes a generator nice?

• Generation of auxiliary artifacts
  – documentation
    > description of input/output, man pages, manual
    > installation scripts
    > description of design knowledge, code derivation, and justification
  – test data, simulation runs, and scripts
• Automated proofs of safety, effectiveness, correctness
• Multiple implementations from single specification
• Tracing between specification and code
  – round-trip engineering
  – incremental consistency
• Extensibility, adaptability
• Any blend of automated and manual development
Early History

• Heuristic compiler (Simon, 1963)
  – program ≡ problem to be solved
  – used General Problem Solver (GPS)
  – apply heuristic problem-solving techniques: goals, operators, ...

• Deductive synthesis (Green, 1969 / Waldinger, 1969)
  – program ≡ logical theorem to be proven: ∃ x • R(a,x)
  – full proof too difficult, so give ATP hints (i.e. lemmas)
  – can represent transformations as inference rules
    > algebraic transformational systems, SpecWare

• Inductive synthesis of logic programs (Plotkin, 1971)
  – (logic) program ≡ set of clauses to be learned
  – inductively infers general case from examples
    > learning-by-example, end user development (EUD)
Current Trends

• Generic programming has become mainstream
  – templates (C++, Java, Haskell)
  – polymorphism (ML, Haskell)

• Generative programming is emerging
  – quote/unquote mechanisms for mainstream languages
  – template meta-programming
  – aspect-oriented programming
  – feature-oriented programming

• Model-based software development looks like Big Money!
  – object-oriented modeling
    > model-driven architecture (MDA)
  – scientific modeling
    > model-integrated computing (MIC)
Themes

Code generation is knowledge-intensive:

• domain knowledge
  – domain engineering

• generation knowledge (or meta-information)
  – transformations
  – search and control

→ Different generation paradigms adopt different approaches to represent each of these kinds of knowledge.
Domain Engineering

Goal: analyze a domain to find concepts and reusable solutions

• originated in software reuse (Draco [Nei84])
• common in OO analysis and design techniques
• applicable to design and implementation of
  – component libraries
  – domain-specific languages (DSLs)
  – architectures, system families, product lines
  – code generators
• domain engineering = domain analysis + domain design
Domain Engineering

Goal: analyze a domain to find concepts and reusable solutions

- originated in software reuse (Draco [Nei84])
- common in OO analysis and design techniques
- applicable to design and implementation of:
  - component libraries
  - domain-specific languages (DSLs)
  - architectures, system families, product lines
  - code generators
- domain engineering = domain analysis + domain design

yields
domain model
Domain Model

Goals:

- establish and define
  - vocabulary
  - concepts and roles
  - features ("options")
- define common and variable properties of systems
  - semantics
  - dependencies
- define common architecture(s)

→ Good domains (for automatic code generation) have Good domain models!
**Domain Model for Kalman Filters (I)**

**Process model**

\[ x_{k+1} = F_k x_k + G_k u_k \]

- State variables
- Process transition matrix \((n \times n)\)
- Process noise \((n \times 1)\)
- Coupling matrix \((n \times 1)\)

\[ u_k \sim N(0, Q_k) \]

\[ E[u_k u_i^T] = \begin{cases} Q_k, & i = k \\ 0, & i \neq k \end{cases} \]

**Measurement model**

\[ z_k = H_k x_k + v_k \]

- Measurement variables \((m \times 1)\)
- Measurement sensitivity matrix \((m \times n)\)

\[ v_k \sim N(0, R_k) \]

\[ E[v_k v_i^T] = \begin{cases} R_k, & i = k \\ 0, & i \neq k \end{cases} \]

\[ E[v_k u_i^T] = 0 \]
Domain Model for Kalman Filters (I)

**Process model**

\[ x_{k+1} = F_k x_k + G_k u_k \]

- State variables
- Process transition matrix
- Process noise variables
- Coupling matrix

\[ u_k \sim N(0, Q_k) \]
\[ E[u_k u_i^T] = \begin{cases} Q_k, & i = k \\ 0, & i \neq k \end{cases} \]

**Measurement model**

\[ z_k = H_k x_k + v_k \]

- Measurement variables
- Measurement sensitivity matrix
- Measurement noise variables

\[ v_k \sim N(0, R_k) \]
\[ E[v_k v_i^T] = \begin{cases} R_k, & i = k \\ 0, & i \neq k \end{cases} \]
\[ E[v_k u_i^T] = 0 \]
Use formal modeling language:

- **UML** most common
  - classes as concepts
  - relations as roles
  - OCL to restrict combinations

- description logics, ...

- tool support
  - graphical editors
  - ...

Domain Model for Kalman Filters (II)
Generative Domain Modeling

• “Plain old” domain model too problem-centric
• Generative domain model adds solution concepts:
  – code fragments and components
  – system configuration knowledge
  – relation between instances (“family members”)
    → model of the implementation domain
• Generative domain model = model of system family with
  – means of specifying family members
  – implementation components
  – configuration knowledge
Generative Domain Modeling

• “Plain old” domain model too problem-centric
• Generative domain model adds *solution concepts*:
  – code fragments and components
  – system configuration knowledge
  – relation between instances (“family members”)
    → model of the implementation domain
• Generative domain model = model of system family with
  – means of specifying family members (in problem space)
  – implementation components (in solution space)
  – configuration knowledge (mapping between spaces)
• Generative domain model = problem space + solution space
Generative Domain Modeling

**Problem space**
- application domain-specific:
  - high-level concepts
  - features
  - functional model
  - non-functional QoS
  maps directly or indirectly to components

**Generic configuration knowledge:**
- dependencies: defaults, mandatory, illegal

**Implementation knowledge:**
- constructions, optimizations
- express transformations using meta-information

**Specific requests:**
- optimize space / time
- use specific algorithms
- generate additional artifacts
- target languages

**Solution space**
- implementation components
  - procedures
  - functions
  - fragments
  - templates
  minimize redundancy
  maximize reuse

But not always a clear separation …
Generative Model for Kalman Filters

- uses domain model
  - concepts
  - constants
- adds solution concepts
  - code fragments
  - individual variables
- limited relations
  - variable types
  - fragment structure
- difficult to express
  - order
  - dependencies
Feature Models

- Customized notation to model variability and configurations
  - UML class diagrams (+ OCL constraints) can represent some variability but become clumsy

- Feature model =
  feature diagram + semantic descriptions of features
  - rationales
  - stakeholders
  - constraints
  - default dependencies

- Configuration language (DSL)
  - represents concrete features and model
  - used to select members of system family (design decisions, instantiations)
  - wizards

[Czarnecki02]
Transformations

• Everything is a transformation!
  – Turing-complete computation model
• Coarse-grained program transformations
  – program refinement and decomposition
  – optimization
    > loop fusion, common sub-expression elimination, ...
  – restructuring
    > parametrizing a class
  – data refinement
    > implementing sets with lists
• Fine-grained program transformations
  – arithmetic simplification
  – fold/unfold
• Model transformations
Software Transformations

- Horizontal transformations
  - evolution: evolve specification
  - refactoring: evolve architecture
- Vertical transformations
  - refinement: implement/refine to code

- Big-step vs. small-step
Automating Transformations

• Implementation alternatives
  – dedicated implementation ("one-off")
    > usually big-step transformations, e.g., refactoring tools
  – transformation system

• Application modes
  – manual: command-line control
    > refactoring tools
  – interactive: select location and transformation
    > fold/unfold systems
  – semi-automatic: split transformations in interactive / automatic
    > interactive are creative (big-step) choices
    > automatic are routine transformations (simplifications)
    > interactive theorem provers
  – automatic: exhaustive application of all transformations
    > (programmable) strategies
Transformation Systems (I)

- **Tools** to build automatic code generators
  - “proto-generators”
  - no support for domain engineering

- Typical architecture (source-to-source transformation system)
Transformation Systems (II)

- “Transformation-friendly” programming languages
  - Prolog, Haskell, ML, …

- Language definition environments
  - Refine (Reasoning Systems, 1990)
  - ASF+SDF (CWI, [BHKO02])
  - TXL (Tree Transformation Language, [CDMS02])
  - DMS (Design Maintenance System, [Bax92])

- Term and graph rewrite engines
  - ELAN (INRIA, [BKK+04])
  - Maude (SRI/UIUC, [CDE+99])
  - Stratego (Utrecht, [Vis01])

- XML-based systems
  - XSLT
  - XVCL (XML-based Variant Configuration Language, [JBZZ03])
Transformations - Rewrite Rules

- Usually used for fine-grained transformations
  - arithmetic simplification
    \[ X + 0 \rightarrow X \]
    \[ d(F, X) \rightarrow 0 \text{ if } X \text{ in } F \rightarrow \text{false} \]
    \[ M \models X / X \rightarrow 1 \text{ if } M \models X = 0 \rightarrow \text{false} \]
  - peephole optimization
    \[ [[ \text{for } I=E; I=<E; I++ \text{ do } S \text{ end }] ] \rightarrow [[ I=E; S ]] \]
    \[ \text{if } I = E \rightarrow \text{false} \]

- Conditional rules (if ... \rightarrow ...)
- Contextual rules (M \models \text{...})
- Concrete syntax ([[ ... ]])
- Programmable strategies
Transformations - Templates

• Templates can be parameterized by values and code
• Compiler chooses most specific version of parameters
• Use `static` or `const` to force evaluation by compiler

```cpp
template <int n, class B> struct UNROLL {
    static void loop() {
        B::body(i);
        UNROLL<n-1, B>::loop();
    }
};

template <class B> struct UNROLL<0, B> {
    static void iteration(int i) {
    }
};

UNROLL<5, MyClass>::loop()

→ Template meta-programming
Transformations - Templates

- Templates can be parameterized by values and code
- Compiler chooses most specific version of parameters
- Use `static` or `const` to force evaluation by compiler

```cpp
template <int n, class B> struct UNROLL {
    static void loop() {
        B::body(i);
        UNROLL<n-1, B>::loop();
    }
};

template <class B> struct UNROLL<0, B> {
    static void iteration(int i) {
    }
};

UNROLL<5, MyClass>::loop()
```

→ Template meta-programming
Transformations - Schemas

- **Big-step transformations**
  - horizontal (model decompositions / transformations)
  - vertical (domain-specific algorithms)

- Implemented as combination of techniques
  - meta-program (check conditions)
  - graph rewriting (transform model)
  - templates (represent code fragments)
Transformations - Schemas

• Big-step transformations
  – horizontal (model decompositions / transformations)
  – vertical (domain-specific algorithms)

• Implemented as combination of techniques
  – meta-program (check conditions)
  – graph rewriting (transform model)
  – templates (represent code fragments)

“mega rewrite rules”
Meta-Programming

• Meta-program ≡ program that represents and manipulates programs as data objects
  – parser, compiler, …
  – partial evaluator, profiler, …
  – code generator

• Reflection: programs represent and manipulate themselves
  – Prolog: \texttt{clause/3}, \texttt{assert/1}, \texttt{retract/1}
  – Java: Reflection API

• Pervasive in code generators
  – implement domain predicates and model operations
  – control application of transformations
  – generate non-templatized code fragments
Meta-Level Architecture

• Split system into two parts:
  – base level: dynamically executed (i.e., at run time)
  – meta level: statically executed    (i.e., at compile time)

• Two-level languages combine base code and meta-code
  – UniFrame
  – C++, …
  – MetaML, MetaOCaml, MetaAspectJ, …

• Template ≡ generative code fragment in two-level language

• Specification is either…
  – implicitly given as two-level program
    > “compile out” meta-code to pure code
  – explicitly given as parameters to templates
    > typically configuration parameters that guide reduction process
Meta-Information (I)

- **Meta-information** ≡ information about base-level components
  - inputs / outputs
  - comments
  - variable dependencies
  - design justifications

- Represents properties difficult to recover from fragments
- Meta-information environment
  - decentralized: components record and advertise properties
    - attributes
    - intentional programming
  - centralized: configuration repository for all components
Meta-Information (II)

• *Configuration knowledge* ≡ meta-information to select and generate particular instance of component family
  – static configuration
    > customize components
  – code generation
    > customize, extend, and assemble components

• *Components under influence*: properties can be determined by the meta-information environment
Code Generation using *quote / unquote*

- Two-level languages use *quote / unquote* to switch levels:
  - *quote*: turn program text into data object
  - *unquote*: splice value of expression into quoted text
- Additional operators:
  - *emit*: turn data object into source code (i.e., program text)
  - *eval*: turn data object into object code
  - *run*: execute data object (runtime code generation)
- ≈ programmable macros
- Invented in Lisp
  - *quote*: ` (back-tick symbol, quasi-quote)
  - *unquote*: , (comma symbol)
  - no parsing problems: Lisp’s simple syntax (s-expressions)
  - no typing problems: Lisp’s simple type system (untyped)
Using *quote* / *unquote* in Lisp (I)

• Start with a function definition in Lisp:

\[
\text{(defun power (n x) "n-th power of x"
  (if (= n 0)
    1
    (* x (power (- n 1) x))))}
\]

• Check that it works:

LISP-eval: (power 3 3)
27
LISP-eval: (power 3)
(l lambda (x) (if (= 3 0) 1 (* x (power (- 3 1) x)))
LISP-eval: (lambda (x) (power 3 x))
(l lambda (x) (* x (power 2 x)))
Using *quote* / *unquote* in Lisp (II)

• Turn function into meta-function:

```lisp
(defun power_meta (n x) "n-th power of `x"
  (if (= n 0)
    `1
    `(* ,x ,(power_meta (- n 1) x))))
```
Using *quote* / *unquote* in Lisp (II)

- Turn function into meta-function:
  
  ```lisp
  (defun power-meta (n x) "n-th power of `x"
    (if (= n 0)
      `1
      (* ,x (power-meta (- n 1) x)))
  )
  ```

  Base level – “normal” Lisp (evaluated at runtime of `power-meta`)
  
  Base level inside meta level - unquoted meta-expressions, yield code objects at runtime of `power-meta`

  Meta level – code object (evaluated at compile-time of `power-meta`)
Using *quote / unquote* in Lisp (II)

- Turn function into meta-function:

```lisp
(defun power_meta (n x) "n-th power of `x"
  (if (= n 0)
    `1
    `((* ,x , (power_meta (- n 1) x))))
)
```

- Check that it works:

```
LISP-eval: (power_meta 3 `3)
(* 3 (* 3 (* 3 1)))
LISP-eval: (eval (power_meta 3 `3))
27
LISP-eval: (power_meta 3 `x)
(* x (* x (* x 1)))
```

Remember: LISP-evaluator does an implicit eval on result.
Using *quote* /*unquote* in Lisp (III)

• Turn function into meta-function:

```
(defun power_meta (n x) "n-th power of `x"
  (if (= n 0)
    `1
    `(* ,x ,(power_meta (- n 1) x)))
```

• Partially evaluate meta-function:

```
(defun power_fn (n) "(fixed) n-th power of `x"
  `(lambda (x) ,((power_meta n `(x)))))

(define power3 (power_fn 3))
```

• Check that it works:

```
LISP-eval: (power3)
(lambda (x) (* x (* x (* x 1))))
LISP-eval: (power3 3)
27
```
Using *quote/unquote* in Lisp (IV)

- Turn function into meta-function:
  
  ```lisp
  (defun power_meta (n x) "n-th power of `x"
    (if (= n 0)
        `1
        `(* ,x ,(power_meta (- n 1) x))))
  ```

- Recipe for (runtime) code generation using *quote/unquote*:
  - constant code is quoted
  - (unknown) arguments are back-quoted
  - (known) arguments compiled out at runtime of meta-function (i.e., generation time)

- *quote/unquote* are staging annotations
  - can be generalized to multi-stage programming
Using *quote* / *unquote* in Java / AspectJ

- Java and AspectJ are a lot more complicated than Lisp:
  - complicated and rich syntax
  - rich type structure incl. inheritance
- Syntax tree must be reified as datatype: `Dcl v = ...`
- *quote* (`` [...``]) must parse program text
  - multiple-entry parser for different syntactic categories
  - different quote operators for different syntactic categories:
    ```java
    Dcl v = `(Dcl)[ int x=0; ];
    ```
- *unquote* (#) must ensure well-formed syntax-trees

```java
Stm s = `(Stm)[ if (#v) x++; ]; // BAD!
Dcl c = `(Dcl)[ class C { #v } ]; // Ok - or?
```
Using *quote / unquote* in Java / AspectJ

- Java and AspectJ are a lot more complicated than Lisp:
  - complicated and rich syntax
  - rich type structure incl. inheritance
- Syntax tree must be reified as datatype: \texttt{Dcl v = ...}
- *quote* (``[...]``) must parse program text
  - multiple-entry parser for different syntactic categories
  - different quote operators for different syntactic categories:
    \begin{verbatim}
    Dcl v = `(Dcl)[ int x=0; ];
    \end{verbatim}
- *unquote* (#) must ensure well-formed syntax-trees
  - syntactic categories of unquoted meta-variables must be known
    \begin{verbatim}
    Stm s = `(Stm)[ if (#(Dcl)v) x++; ];  // BAD!
    Dcl c = `(Dcl)[ class C { #(Dcl)v } ];  // Ok!
    \end{verbatim}
Using quote / unquote in Java / AspectJ

• Java and AspectJ are a lot more complicated than Lisp:
  – complicated and rich syntax
  – rich type structure incl. inheritance
• Syntax tree must be reified as datatype: Dcl v = …
• quote (‘[…]) must parse program text
  – multiple-entry parser for different syntactic categories
  – different quote operators for different syntactic categories:
    infer v = ‘[ int x=0; ];
• unquote (#) must ensure well-formed syntax-trees
  – syntactic categories of unquoted meta-variables must be known
    infer s = ‘[ if (#v) x++; ]; // BAD!
    infer c = ‘[ class C { #v } ]; // Ok!
• Type inference possible (Meta-AspectJ)
  – parsing becomes even more difficult…
Paradigms

Code generators can be classified in several dimensions:

• specific vs. generic
  – customized domain-specific implementations ("one-off")
  – generic frameworks for building generators

• compositional vs. holistic
  – directly mapping structure ("boxes and arrows") to code
  – gradually applying arbitrary transformations

• generative vs. transformative vs. deductive
  – assembling code from fragments
  – refining model into code
  – logically deducing code from specification

→ Distinction is gradual
→ Systems are often mixed paradigm
Generator Frameworks

• Manual generator implementations …
  + … can be highly customized
  + … can be highly optimized
  – … are typically standalone
  – … are typically non-interoperable
  → Writing a generator from scratch is lots of work!

• Generic frameworks provide dedicated generator infrastructure
  – meta-modeling tools (MetaEdit+, GME, …)
  – meta-programming (C++ templates, …)
  – transformation and search (Stratego, …)

• Different flavors
  – two-level (C++ templates, IP, Refine, …)
  – algebraic (SpecWare, Meta-Amphion, …)
Bootstrapping

- **Bootstrapping** ≡ use system for own development
  - compilation: write compiler in source language
  - code generation: use generator to generate itself
- Requires metamodel of system
  - UML defined within MOF
- Supports customization of generator
  - modify specification language
  - retarget implementation language
  - extend transformation base
- Bootstrapped systems:
  - Refine
    - rewrite engine via rewriting
  - Intentional Programming
    - IP compiler via IP
  - MetaEdit+, GME, XMF, ...
    - transformations via core
  - “Coq-in-Coq”
    - impredicative (reflective) logic
Compositional vs. Holistic

- **Compositional** generators apply only vertical refinements
  - introduced by Batory [Bat96]
  - typical for CASE tools
    > GUI-based model builders
    > code generation via simple final traversal
  - structure-preserving
    > generated code traced back to initial specification
  - specifications often executable (at each level)

- **Holistic** generators apply “whole-system” transformations
  - optimization
  - refactoring
  - weaving

[CE00]
Program Synthesis

• Program synthesis ≡ code generation using AI techniques
  – search
  – theorem proving
  – symbolic reasoning

• Goals
  – highest level of reliability
    > “correct-by-construction”
  – highest level of automation
    > “automatic programming”
  – highest leverage
    > minimize input, maximize discovery
Program Synthesis

• Program synthesis ≡ code generation using AI techniques
  – search
  – theorem proving
  – symbolic reasoning

• Goals
  – highest level of reliability
    > “correct-by-construction”
  – highest level of automation
    > “automatic programming”
  – highest leverage
    > minimize input, maximize discovery

L. S. van Bentham Jutting. Checking Landau’s "Grundlagen" in the AUTOMATH system.

• “Hidden agenda:” automate Math and CS textbooks
Templates

• Templates have been used…
  – for static parametrization and configuration
    > original purpose
    > generic programming
  – to represent design patterns
    > more recently
  – for code generation
    > more by accident

• Template languages exist for variety of base languages
  – C++
    > part of C++ ISO-standard
  – Haskell
    > TemplateHaskell
  – Java
    > Velocity, JET, TL
  – ML / OCaml
    > MetaML / MetaOCaml
• Templates have been used…
  – for static parametrization and configuration
    > original purpose
    > generic programming
  – to represent design patterns
    > more recently
  – for code generation
    > more by accident

• Template languages exist for variety of base languages
  – C++
    > part of C++ ISO-standard
  – Java
    > Velocity, JET, TL
  – Haskell
    > TemplateHaskell
  – ML / OCaml
    > MetaML / MetaOCaml

most common, focus here
C++ Templates: Parametrization

- Integral value (int, short, char) and type / class parameters
  template <class T, int x> class C {...}

- Parametrized inheritance
  // object code depends on argument class
  template <class T> class C : public T { ... }

- Function parameters
  template<double Tfunc(double)> class C {...}

- Parametrization over meta-function
  // takes a meta-function which takes an int
  template <template<int> class T> class C {...}

- Partial template instantiation implements specialization:
  // if no argument is given, set n to 0
  template <int n> class C {...};
  template <> class C<0>;
C++ Templates: Meta-Programming

• Templates can be recursively combined to form a static functional language:
  – parametrize over integers and types to define meta-functions
  – encode compile-time control structures statically
  – template recursion instead of loops
  – nested templates represent abstract syntax for complex data-types

• Representing meta-information:
  – member traits
    > properties of a type, e.g. range of values
  – member types
  – member constants
C++ Templates: Code Generation

- Static-level control structures can be used to generate code
  - `WHILE`, `DO`, `FOR`, `IF`, `SWITCH`
  - templates composed using meta-function wrappers
  > e.g. `IF<b, T1<a>, T2<a>>`
  - code selection based on static data
    > static types, integer comparisons, etc.

- Example:

  template T<x,y,n>
  
  {IF< x=y, 
    FOR i = 1 TO n {stmt[i];} , 
    {stmt(x);stmt(y)} > ; 
  stmt(4)}

  - `T<1,1,3>` reduces to `stmt(1);stmt(2);stmt(3);stmt(4)`
  - `T<1,2,3>` reduces to `stmt(1);stmt(2);stmt(4)`
C++ Templates: Code Generation

• Expression templates:
  – (re-) define operators / functions as template meta-functions
    > templates construct expression objects
    > static type encodes expression structure (i.e., the parse tree)
  – extend concrete syntax with specification operators

• Example: vector operations

  ```cpp
  Vector v, a, b, c;
  v = a + b + c;
  ```

  should generate

  ```cpp
  for (uint i=0 ; i < v.size(); i++)
    v[i] = a[i] + b[i] + c[i];
  ```

  – no temporary variables
  – single pass through memory (to preserve cache locality)
C++ Templates: Code Generation

//-- operator tags; similar for minus, times, ...
class plus {
    static float apply(float a, float b)
    { return (a + b); } 
};

//-- General expression template
template <class L, class BinOp, class R>
class Expr {
    Expr(L &l, R &r) : l_(l), r_(r) {};
    float operator[](uint i)
    { return BinOp::apply(l_[i], r_[i]) }; 
    L &l_; R &r_;
};

//-- Addition operator; similar for minus, times, ...
template <class L, class R>
Expr<L, plus, R> operator+(L &l, R &r)
{ return Expr<L, plus, R>(l, r); }
C++ Templates: Code Generation

//-- operator tags; similar for minus, times, ...
class plus {
    static float apply(float a, float b)
    { return (a + b); }
};

//-- General expression template
template <class L, class BinOp, class R>
class Expr {
    Expr(L &l, R &r) : l_(l), r_(r) {}
    float operator[](uint i)
    { return BinOp::apply(l_[i], r_[i]); }
    L &l_; R &r_;
};

//-- Addition operator; similar for minus, times, ...
template <class L, class R>
Expr<L, plus, R> operator+(L &l, R &r)
{ return Expr<L, plus, R>(l, r); }
C++ Templates: Code Generation

//-- operator tags; similar for minus, times, ...
class plus {
    static float apply(float a, float b)
    { return (a + b); }
};

//-- General expression template
template <class L, class BinOp, class R>
class Expr {
    Expr(L &l, R &r) : l_(l), r_(r) {}
    float operator[](uint i)
    { return BinOp::apply(l_[i], r_[i]) ; }
    L &l_; R &r_;
};

//-- Addition operator; similar for minus, times, ...
template <class L, class R>
Expr<L, plus, R> operator+(L &l, R &r)
{ return Expr<L, plus, R>(l, r); }

builds expression object w/ constructor
refines general template
syntactic sugar
C++ Templates: Code Generation

//-- operator tags; similar for minus, times, ...
class plus {
    static float apply(float a, float b)
    { return (a + b); }
};

//-- General expression template
template <class L, class BinOp, class R>
class Expr {
    Expr(L &l, R &r) : l_(l), r_(r) {}
    float operator[](uint i)
    { return BinOp::apply(l_[i], r_[i]); }
    L &l_; R &r_;
};

//-- Addition operator; similar for minus, times, ...
template <class L, class R>
Expr<L, plus, R> operator+(L &l, R &r)
{ return Expr<L, plus, R>(l, r); }

C++ Templates: Code Generation

• Expression templates:
  – (re-) define operators / functions as template meta-functions
    > templates construct expression objects
    > **static type encodes expression structure (i.e., the parse tree)**
  – extend concrete syntax with specification operators
• Example: vector operations
  \[ a + b + c == ((a + b) + c) \]
  has type

\[
\text{Expr<Expr<Vector, Plus, Vector>, Plus, Vector>}
\]

• So:
  \[ (a + b + c)[i] == ((a + b) + c)[i] \]
  \[ == \text{plus::apply}((a + b)[i], c[i]) \]
  \[ == \text{plus::apply}(\text{plus::apply}(a[i], b[i]), c[i]) \]
  \[ == \text{plus::apply}(a[i] + b[i]), c[i]) \]
  \[ == a[i] + b[i] + c[i] \]
C++ Templates: Code Generation

• Expression templates:
  – (re-) define operators / functions as template meta-functions
    > templates construct expression objects
    > static type encodes expression structure (i.e., the parse tree)
  – extend concrete syntax with specification operators

• Example: vector operations
  \[ v = a + b + c; \]

• Define missing \(-\)-operator:

  template <class Expr>
  Vector& Vector::operator=(Expr &x) {
    for (uint i=0 ; i < this->size(); i++)
      (*this)[i] = x[i];  // force evaluation
    return *this
  }
C++ Templates: Code Generation

- **Expression templates:**
  - (re-) define operators / functions as template meta-functions
    - templates construct expression objects
    - static type encodes expression structure (i.e., the parse tree)
  - extend concrete syntax with specification operators

- **Drawbacks:**
  - clumsy template syntax
  - limited applicability
    - algebraic libraries
  - testing difficult:
    - generate-compile-run cycle
    - cryptic error messages
Java Emitter Templates (JET)

• Template-based technique for Eclipse Modeling Framework
  – based on org.eclipse.emf.codegen package
  – “Eclipse-wizardry”
  – derived from JSP (Java Server Pages, web technology)

• Two-(and-a-half)-stage process
  – compile template into Java class using JETBuilder
    > template instantiation class provides generate-method
    > generator can be customized via meta-model: Java, XML, SQL, …
    > generator ≠ compiler: JET is not an extension of Java
  – invoke generate-method from Java
    > can pass (model) objects to templates: argument meta-variable
  – (compile output using javacc)

• Easy to use but not very powerful
  – … nor very safe!
Java Emitter Templates (JET)

- Template syntax:
  
  source text  /* Auto-generated by JET */
  > directly copied into output

  + directives:  <%@ class="JETexample" %>
  > set/access meta-information

  + scriptlets:  <%  Model mdl=(Model)argument;%>
  > meta-statements, executed at template compilation time
  > (syntactically) valid Java-statement

  + expressions:  package <%= mdl.getName()%>;
  > meta-expressions, evaluated at template invocation time
  > result is appended to output buffer
  > (syntactically) valid Java-expression

- String-oriented processing model
  - “token collision”, no type checks, …
  - everything is quoted, `<%...%>` is unquote
Generic vs. Generative Programming

• Both (ab-) use the same mechanism (i.e., templates)
  – templates and generic programming co-designed
  – generative programming discovered by chance
    > Erwin Unruh’s prime number hack

• Generic programming =
  parametric polymorphism for the masses
  – focus is on algorithm representation
  – code expansion is used to cope with run-time overheads induced by C++

• Generative programming =
  code generation for the daredevils
  – focus on meta-programming
  – quirks in the template expansion algorithm are exploited to control code generation
Model-Based Software Development

Industry-wide trend towards model-based development:

• engineering:
  – Matlab / Simulink
    > many industry-specific extensions
  – model-integrated computing (MIC)

• software engineering:
  – OO / UML
    > common standards (OMG)
  – model-driven architecture (MDA)
    > development methodology based on formalizing models
    > automate coding as model transformation
    > retain the model-code link during the software lifecycle
    > shift focus from code to models
MDA Overview (I)

• MDA models software at different levels of abstraction
  – *platform* can be an OS, language, framework, etc.

• Platform-independent model (PIM)
  – represents abstract “semantic” concepts
  – high-level description, independent of implementation
  – typically in UML

• Platform-specific model (PSM)
  – represents specific implementation technology
    > J2EE
    > JSP
    > SQL
  – not (necessarily) code
  – alternative or intermediate PSMs
MDA Overview (II)

• Coding is automated as a series of *model transformations*:

  ![Diagram showing PIM, PSM, and Code with transformation arrows]

  - Explicit transformation definitions
    - transformations are “big-step”
    - single transformation comprises several rules
  - Transformation tools
    - apply transformation definitions
    - QVT languages (“queries views transformations”)
      - ATL, GreAT, OptimalJ, QVTP, ...
      - currently being standardized
Modeling

• Models must be formally defined to support automation
  – syntactic or graphical definitions
  – definition uses meta-model

• Meta-model also formally defined
  – definition uses “meta-meta-model” or meta-object facility (MOF)
    instance : model : meta-model : MOF

• Any model defined (ultimately) by MOF
  – MOF defined by MOF
  – can be annotated with OCL
  – can be interchanged via XMI

• Specific support for PSMs via UML profiles
  – Profile = UML + stereotypes + constraints + tagged values

• Avoid model pollution
Model transformations

• OO style:
  – UML Class diagram to ER diagram
  – ER diagram to SQL script
  – EJB model to web component model
  – web diagram to JSP

• CASE tool code generation:
  – (black-box) generator takes predefined PSM's to code

• Design / re-engineering transformations:
  – normalization of ER diagram
  – refactoring of code

• Documentation generation

• Mathematical transformations:
  – linearization of filter process/measurement model
  – decomposition of Bayesian network
Transformations: Features

• Customizability
  – allow manual control over rules applied during transformation
  – allow rules to be modified by
    > adding conditions
    > parameterizing

• Traceability
  – warn user when modifying generated code
  – rename operations in PIM
  – give impact analysis on PIM

• Incremental Consistency
  – maintain changes in PSM when regenerating from PIM
    > important for iterative development
Transformations: Implementation

• Customizability needs transformation parameters:
  – add parameters (with defaults) to rules
  – actual values should be persistent
    > can tag PIM (but clutters)
    > can tag PSM (after generation, but clutters)
    > create intermediate transformation object with state

• Traceability needs persistent source-target relationship:
  – rather than polluting models, have separate objects
  – treat each transformation application as an object, containing state, structure, and links
Transformations: Objects

Conceptual Model

Customer
+customer

Order
* +orders

Transformation Instance

Association-end-attribute

Design Model

Customer
-orders :Set
+getOrders() :Set
+setOrders(o :Set)

Order
-customer :Customer
+getCustomer() :Customer
+setCustomer(c :Customer)

[KWB03]
Transformations: Details

• Transformations often given declaratively
  – typically in report generation language (template pattern language)
  – applied opportunistically to all applicable elements in source model
    (if none apply, do nothing)

• Model transformation consists of
  – several transformation rules, which consist of
    – several mapping rules

• Rules implicitly refer to other rules
  – transformations are not explicitly named
  – assumed to apply uniquely
  – based on primitive rules for datatypes

• Transformations can be composed (iteration is implicit)
• Transformations manipulate elements specified in meta-model
Transformations: Meta-modeling

- Meta-language
  - Extends
  - Written-in
  - Language
    - Transformation definition
      - Written-in
      - Language
        - Written-in
          - Transformation definition
            - Written-in
            - Transformation tool
              - Used-by
              - PIM
                - Written-in
              - PSM
                - Written-in

[KWB03]
Transformations: Syntax

Typical style for transformation rules (from [KWB03]):

Transformation <name> (<src lang>, <tgt lang>) {
   params
      <name> : <type> = <default>
   source
      <variable> : <lang> : <type>
   target
      <variable> : <lang> : <type>
   <direction indicator>
   source condition
      <OCL constraint>
   target condition
      <OCL constraint>
   mapping
      <OCL query> <~> <OCL expression>
}
Transformations: Parameters

- **Parameters** specify how a transformation can be modified and give defaults

**Example**: “allow the state transition and measurement matrix identifiers in the generated code to be modified from their defaults, ‘phi’ and ‘h’.”

```
params
    state_trans_mx : string = 'phi';
    measurement_mx : string = 'h';
```
Transformations: Conditions

• **Source conditions:** applicability of rules to source elements
• **Target conditions:** what must hold in generated elements
• Separate the conditions for source and target model
  – link is expressed with mapping

**Example:** “If source is public set target to be private.”

```plaintext
source condition
  sourceAttr.visibility = VisibilityKind::public;
target condition
  targetAttr.visibility = VisibilityKind::public;
```
Transformations: Mappings

- **Mappings** are defined declaratively:
  - element \(<~>\) element
  - set \(<~>\) set
    > interpreted as “\(\text{l} \cdot \text{LHS} \cdot \text{r} \cdot \text{RHS} \cdot \text{l} \,<~>\, \text{r}\)”

- Cannot be conditional
  - separate rules for each condition

**Example**: “Target names are defined from source names by …”

**mapping**

```plaintext
sourceAttr.name \,<~>\, targetAttr.name;
class.name + 'Key' \,<~>\, keyClass.name;
id1.name \,<~>\,
  assocClass.end->first().type.name + 'ID';
rootDataClass.feature \,<~>\,
  class.feature->
    select(oclKindOf(UML::Attribute) or
    oclKindOf(UML::AssociationEnd))
```
Transformations: Model Queries

- Mappings use OCL operators:
  - first-order, no side-effects
  - quantification over finite collections, set operations, iteration
    > extensible by defining query and navigation operations over model
  - can be constructive
    > express comprehensions using queries and expressions

Example: “getAllContained(c) is the set of classes associated directly or indirectly with c”

```java
class Context

context Class def:
getAllCont(contained : Set(Class)) : Set(Class) =
  if contained->includes(self)
  then result = contained
  else
    iterate(contClass : Class; acc : Set(Class) = result |
             acc->union(containedClass.getAllCont(allCont)));
```

```java
```
Program Extraction

- Constructive logic
  - mathematical entities only exist if they can be constructed
  - constructive proofs “correspond” to (typed) lambda-terms
  - Curry-Howard isomorphism
    > aka “formulas as types” or “proofs as programs”

- lambda terms = functional programs
  ⇒ theorem prover + extraction mechanism = code generator
  - code generator is as automated as the prover

- Main advantage: “correctness by construction”
  - if program p is extracted from a constructive proof M of specification S, then p is guaranteed to satisfy S
  - formally based on theory of realizability:
    > [[…]] removes computationally irrelevant parts from the proof
    > p = [[M]] “realizes” S (in symbols p = [[M]] : type(S))
    > if <t,p> : (∃ x:T • S(x)), then [[<t,p>]] = [[t]]
Proofs as Programs

- Logical *introduction* rules correspond to term *construction* rules:

  \[
  \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \sim \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \rightarrow B}
  \]

  \[
  \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \land B} \quad \sim \quad \frac{\Gamma \vdash u : A \quad \Gamma \vdash t : B}{\Gamma \vdash \langle u, t \rangle : A \times B}
  \]

- Similar correspondences for elimination and reduction rules

- Classical proof steps correspond to non-local control (i.e. jumps).
Extraction Methodology

1. Formulate domain knowledge:
   - Assumptions as definitions & axioms
   - Transformations as inference rules
   - Specifications as theorems

2. Prove theorem:
   - Guide prover to discharge all subgoals
     - manually
     - customized tactics
   - Typically top-down (decompose into subgoals)

3. Extract program:
   - Proof assistant converts proof term to functional code:
     - theorems $\rightarrow$ types
     - proof terms $\rightarrow$ programs
     - sub-proofs (lemmas) $\rightarrow$ subroutines
Several proof assistants support extraction:

- **Minlog** (Schwichtenberg, Berger)
  - based on first-order minimal logic
  - extracts to lambda-terms
- **Coq** (Paulin-Mohring, Werner, Letouzey)
  - based on type theory (CIC)
  - extracts to OCaml or Haskell
- **Isabelle** (Berghofer)
  - based on higher-order logic
  - extracts to SML
- **SNARK, Agda/Alfa, NuPrl, …**
- **Tools provide optimizations to extract reasonable programs**
  - limited control possible
  - often need to give computational hints at the logical level
Amphion

- Astronomical problems
  - library axiomatization
  - SNARK prover
- Component assembly
- Proof term translation
  - FORTRAN77
  - animation subroutines
- Graphical specification
  - geometric / kinematic relationships
  - input / output
- Developed at NASA Ames (1990’s)
Amphion

Ansa phase angles (deg): 51.80 55.85

2004 Nov 23 12:00:00
Extraction in Practice

• Applications:
  – still limited (e.g. [Den01]) due to large investment needed in time, theories, expertise
  – tend to be one-off: the “heroic proof”
  – often from proofs of meta-theoretic results
  – little domain engineering or reuse

• As a code generation methodology:
  – heavy duty: must prove everything (although subgoals can be deferred)
  – no integration with modeling tools; user-defined syntax gives limited support for DSLs
  – not good for representing domain and generative knowledge (meta-information)
  – can't generate auxiliary artifacts
  – optimizations in current provers are limited

• Suggested modes of use:
  – extract safety-critical algorithms for trusted component base (e.g. VCG, type checker, decision procedure)
  – combine with other techniques
Generator Assurance

Should you trust a code generator?

- Correctness of generated code depends on correctness of generator
- Correctness of generator is difficult to show
  - very large
  - very complicated
  - very dynamic

```c
... model map an 'Mixture of Gaussians'.
... % Class probabilities
double rho(1..c).
where 1 = sum(1..c, rho(i)).
% Class parameters
double mu(1..c).
double sigma(1..c).
where 0 < sigma(i).
% wider variable
int z(1..n) = discrete(rho).
% num
data double x(1..n); %Gauss(0(1..n),sigma(1..n)).
% goal
max pr(x|{rho,mu,sigma})
for (rho,mu,sigma).
... // Initialization
... for (v44 = 0; v44 <= n - 1; v44++)
for (v45 = 0; v45 <= c - 1; v45++)
q(v44,v45) = 0;
... for (v46 = 0; v46 <= n - 1; v46++)
q(v46,z(v46)) = 1;
...
... for (v12 = 0; v12 <= n - 1; v12++)
for (v13 = 0; v13 <= c - 1; v13++) {
  pv68 = 0;
  for (v14 = 0; v14 <= c - 1; v14++)
    pv68 += exp((x(v12) - mu(v14)) * (x(v12) - mu(v14)) / (double)(-2));
... }
...
```

High-Level Model | Code Generator | Efficient Code
Generator Assurance

Should you trust a code generator?
- Correctness of generated code depends on correctness of generator
- Correctness of generator is difficult to show
  - very large
  - very complicated
  - very dynamic

Code Generation Dilemma:
- Deductive approaches
  - formally correct
  - difficult to scale
- Generative approaches
  - powerful generation
  - requires trust in generation process
Generator Assurance Approaches

Correctness-by-construction:
Generator is based on logical framework; code is derived by correctness-preserving transformations

Techniques:
• Deductive program synthesis
• Refinement and transformation systems
• Translation verification

Advantages:
• Highest degree of confidence (“proofs as programs”)

Disadvantages:
• Expensive – systems difficult to build and maintain
• Opaque – correctness argument convoluted (and buried in generator)

Trust me – I’m a doctor
Generator Assurance Approaches

Generator Qualification:
Generator is tested to same level of criticality as generated code

Advantages:
• Currently only technique accepted by FAA
• State of the practice

Disadvantages:
• Expensive – testing efforts very high
• Expensive – re-qualification required after changes
• Limited – only partial assurance
• Opaque – no explicit correctness argument

Trust me – I’m an engineer
Certifiable Code Generation:

Generator is extended to support independent post-generation verification

- certify generated programs, not the generator
- product-oriented approach, not process-oriented

Advantages

- No need to re-certify generator
- Minimizes trusted component base
Certifiable Code Generation

Extend generator to support certification:

• Generate code with “mark-up” to support an independent assurance demonstration (after generation)
  – **IMPORTANT:** keep certification independent of generation

• Use Floyd-Hoare program verification techniques
  – mark-up: pre-/postconditions, loop invariants

• Proofs are independently verifiable evidence (certificates)

• Focus on specific safety properties
  – Array bounds, partial operators
  – Variable initialization, def-use
  – Physical units, frames
  – Volatile memory restrictions
  – Vector norms, matrix symmetry
  – …
Certification Architecture

Wrap generator with post-generation certification mechanism:

- Minimize trusted code base
- “Large” components untrusted
- Trusted components (more) deterministic

Approach:

- Generate safety obligations (i.e. VCG applies safety policy to code)
- Simplify, prove, & check
Roadmap: Hoare Logic and Safety

Basic underlying logic (without safety):
- Hoare logic rules
- Operational semantics
- Semantics of rules

Define safety property:
- Semantic safety
- Safety predicate

Extend the logic with safety:
- Operational semantics with safety
- Extended rules
- Semantics of extended rules
- Extend generator to produce annotations using safety predicate
Hoare Rules

- Forwards formulation of inference rules:
  - “If precondition $P$ holds, then after executing statement $c$, postcondition $Q$ will hold.”

- Backwards formulation of inference rules:
  - “In order to establish postcondition $Q$ after executing $c$, precondition $P$ must hold.”
  - $P$ must be weakest precondition: $\text{wpc}(c, Q)$
  - standard interpretation for automation
Hoare Logic

\[(\text{decl})\quad Q \{\text{var } x\} \quad Q\]

\[(\text{adecl})\quad Q[n/x_{\text{hi}}] \{\text{var } x[n]\} \quad Q\]

\[(\text{skip})\quad Q \{\text{skip}\} \quad Q\]

\[(\text{assign})\quad Q[e/x] \{x := e\} \quad Q\]

\[(\text{update})\quad Q[\text{upd}(x, e_1, e_2)/x] \{x[e_1] := e_2\} \quad Q\]

\[(\text{if})\quad P_1 \{c_1\} \quad P_2 \{c_2\} \quad Q\]

\quad \frac{(b \Rightarrow P_1) \land (-b \Rightarrow P_2) \quad \{\text{if } b \text{ then } c_1 \text{ else } c_2\} \quad Q}{P \{c\} \quad I \land b \Rightarrow P \quad I \land -b \Rightarrow Q \quad I \quad \{\text{while } b \text{ inv } I \text{ do } c\} \quad Q}\]

\[(\text{while})\quad P \{c\} \quad R \quad \{c_2\} \quad Q\]

\quad \frac{R \quad P \{c_1 \ ; \ c_2\} \quad Q}{P \{c_1\} \quad R \quad P \{c\} \quad Q' \quad Q' \Rightarrow Q \quad P \{\text{pre } P' \text{ c post } Q'\} \quad Q}\]

\[(\text{assert})\quad P \Rightarrow P' \quad P \{c\} \quad Q' \quad Q' \Rightarrow Q\]
Safety Framework

• Safety property: operational characterization of intuitively safe programs
  – introduce shadow variables to record safety information
  – safety predicate $safe_{init}(e)$ corresponds to semantic safety conditions
    $> safe_{init}(x) \equiv x_{init} = \text{init}$
    $> safe_{init}(x[e]) \equiv x_{init}[e] = \text{init} \land safe_{init}(e)$

• Safety policy: proof rules to show that safety property holds for program

• Extend logical framework to account for shadow variables
  – Operational semantics
  – Semantic safety
  – Hoare rules

• See [DF03] for more details
Extended Hoare Logic: Semantics

- Shadow environments bind shadow variables

\[ \eta : \text{Var} \rightarrow \text{Val} \]
\[ \bar{\eta} : \text{Var} \rightarrow \overline{\text{Val}} \]
\[ \eta, \bar{\eta} \models c \text{ safe}_{sp} \]

- Use to give extended semantics of Hoare triples [DF03]
Initialization Safety Semantics

Semantic definition of statement safety:

\[ \eta, \bar{\eta} \models x \text{ safe}_{\text{init}} \quad \iff \quad x_{\text{init}} = \text{INIT} \]
\[ \eta, \bar{\eta} \models x[e] \text{ safe}_{\text{init}} \quad \iff \quad \bar{\eta}(x_{\text{init}})[e]_{\eta, \bar{\eta}} = \text{INIT} \quad \text{and} \quad \eta, \bar{\eta} \models e \text{ safe}_{\text{init}} \]
\[ \ldots \]
\[ \eta, \bar{\eta} \models x[e_1] := e_2 \text{ safe}_{\text{init}} \quad \iff \quad \eta, \bar{\eta} \models e_1 \text{ safe}_{\text{init}} \quad \text{and} \quad \eta, \bar{\eta} \models e_2 \text{ safe}_{\text{init}} \]

Extension of operational semantics with safety:

\[ \langle x := e, \eta, \bar{\eta} \rangle \quad \Rightarrow \quad \langle \text{skip}, \eta \oplus \{ x \mapsto [e]_{\eta} \}, \bar{\eta} \oplus \{ x_{\text{init}} \mapsto \text{INIT} \} \rangle \]
\[ \langle x[e_1] := e_2, \eta, \bar{\eta} \rangle \quad \Rightarrow \quad \langle \text{skip}, \eta \oplus \{ x \mapsto (x \oplus \{ [e_1]_{\eta} \mapsto [e_2]_{\eta} \}) \}, \]
\[ \quad \bar{\eta} \oplus \{ x_{\text{init}} \mapsto (x_{\text{init}} \oplus \{ [e_1]_{\eta} \mapsto \text{INIT} \}) \} \rangle \]
Initialization Safety Policy

(decl)
\[ Q \{ \text{var } x \} \quad Q \]

(adcl)
\[ Q \{ \text{var } x \ [n] \} \quad Q \]

(skip)
\[ Q \{ \text{skip} \} \quad Q \]

(assign)
\[ Q[e/x, \text{INIT}/x_{\text{init}}] \land \text{safe}\_\text{init}(e) \{ x := e \} \quad Q \]

(update)
\[ Q[\text{upd}(x, e_1, e_2)/x, \text{upd}(x_{\text{init}}, e_1, \text{INIT})/x_{\text{init}}] \land \text{safe}\_\text{init}(e_1) \land \text{safe}\_\text{init}(e_2) \{ x[e_1] := e_2 \} \quad Q \]

(if)
\[ (b \Rightarrow P_1) \land (\neg b \Rightarrow P_2) \land \text{safe}\_\text{init}(b) \{ \text{if } b \text{ then } c_1 \text{ else } c_2 \} \quad Q \]

(while)
\[ P \{ c \} \quad I \quad I \land b \Rightarrow P \quad I \land \neg b \Rightarrow Q \quad I \land \text{safe}\_\text{init}(b) \{ \text{while } b \text{ inv } I \text{ do } c \} \quad Q \]

(comp)
\[ P \{ c_1 \} \quad R \quad R \{ c_2 \} \quad Q \quad P \{ c_1 ; c_2 \} \quad Q \]

(assert)
\[ P \Rightarrow P' \quad P \{ c \} \quad Q' \quad Q' \Rightarrow Q \quad P \{ \text{pre } P' \text{ c post } Q' \} \quad Q \]
Initialization Safety Policy

\[(\text{decl})\quad Q \{ \text{var } x \} \quad Q\]

\[(\text{adect})\quad Q \{ \text{var } x [n] \} \quad Q\]

\[(\text{skip})\quad Q \{ \text{skip} \} \quad Q\]

\[(\text{assign})\quad Q[e/x, \text{INIT}/x_{\text{init}}] \land \text{safe}_{\text{init}}(e) \land x := e \quad Q\]

\[(\text{update})\quad Q[\text{upd}(x, e_1, e_2)/x, \text{upd}(x_{\text{init}}, e_1, \text{INIT})/x_{\text{init}}] \land \text{safe}_{\text{init}}(e_1) \land \text{safe}_{\text{init}}(e_2) \quad \{ x[e_1] := e_2 \} \quad Q\]

\[(\text{if})\quad (b \Rightarrow P_1) \land (\neg b \Rightarrow P_2) \land \text{safe}_{\text{init}}(b) \quad \{ \text{if } b \text{ then } c_1 \text{ else } c_2 \} \quad Q\]

\[(\text{while})\quad P \{ c \} \quad I \land b \Rightarrow P \quad I \land \neg b \Rightarrow Q \quad I \land \text{safe}_{\text{init}}(b) \quad \{ \text{while } b \text{ inv } I \text{ do } c \} \quad Q\]

\[(\text{comp})\quad P \{ c_1 \} \quad R \quad R \{ c_2 \} \quad Q \quad P \{ c_1 ; c_2 \} \quad Q\]

\[(\text{assert})\quad P \Rightarrow P' \quad P \{ c \} \quad Q' \quad Q' \Rightarrow Q \quad P \{ \text{pre } P' \text{ c post } Q' \} \quad Q\]

\text{safety condition}

\text{safety substitutions}
Array-bounds Safety Policy

\( (\text{decl}) \quad Q \{ \text{var} \ x \} \ Q \)

\( (\text{a decl}) \quad Q[n/x_{\text{hi}}] \{ \text{var} \ x [n] \} \ Q \)

\( (\text{skip}) \quad Q \{ \text{skip} \} \ Q \)

\( (\text{assign}) \quad Q[e/x] \land \text{safe}_{\text{mem}} (e) \{ x := e \} \ Q \)

\( (\text{update}) \quad Q[\text{upd}(x, e_1, e_2)/x] \land \text{safe}_{\text{mem}} (x[e_1]) \land \text{safe}_{\text{mem}} (e_2) \{ x[e_1] := e_2 \} \ Q \)

\( (\text{if}) \quad \frac{P_1 \{c_1\} \ Q \quad P_2 \{c_2\} \ Q}{(b \Rightarrow P_1) \land (\neg b \Rightarrow P_2) \land \text{safe}_{\text{mem}} (b) \{ \text{if} \ b \ \text{then} \ c_1 \ \text{else} \ c_2 \} \ Q} \)

\( (\text{while}) \quad \frac{P \{ c \} \ I \ I \land b \Rightarrow P \quad I \land \neg b \Rightarrow Q}{I \land \text{safe}_{\text{mem}} (b) \{ \text{while} \ b \ \text{inv} \ I \ \text{do} \ c \} \ Q} \)

\( (\text{comp}) \quad \frac{P \{ c_1 \} \ R \quad R \{ c_2 \} \ Q}{P \{ c_1 \ ; \ c_2 \} \ Q} \)

\( (\text{assert}) \quad \frac{P \Rightarrow P' \quad P \{ c \} \ Q' \quad Q' \Rightarrow Q}{P \{ \text{pre} \ P' \ c \ \text{post} \ Q' \} \ Q} \)
Annotation Generation

Example: annotations for array-safety:

begin
  var c[C], w[N,C];
  ...
  for i:= 1 to N do
    c[i] := rnd(C);
    ...
    for i := 1 to N do
      for j := 1 to C do w[i,j] := 0.0;
      w[i,c[i]] := 1.0;
    ...
  end
Example: annotations for array-safety:

```
begin
  var c[C], w[N,C];
  ...
  for i:= 1 to N do
    c[i] := rnd(C);

  ...
  for i := 1 to N do
    // inv : 1 ≤ c[i] ≤ C
    for j := 1 to C do w[i,j] := 0.0;
    w[i,c[i]] := 1.0;
  ...
end
```
Example: annotations for *array*-safety:

```plaintext
begin
    var c[C], w[N,C];
    ...
    for i := 1 to N do
        c[i] := rnd(C);
        // post: forall j . 1 ≤ j ≤ N => 1 ≤ c[j] ≤ C
        ...
    for i := 1 to N do
        // inv : 1 ≤ c[i] ≤ C
        for j := 1 to C do w[i,j] := 0.0;
        w[i,c[i]] := 1.0;
        ...
end
```
Example: annotations for array-safety:

begin
    var c[C], w[N,C];
    ... 
    for i := 1 to N do 
        // inv : forall j . 1 ≤ j < i ⇒ 1 ≤ c[j] ≤ C
        c[i] := rnd(C);
        // post: forall j . 1 ≤ j ≤ N ⇒ 1 ≤ c[j] ≤ C
        ... 
    for i := 1 to N do 
        // inv : 1 ≤ c[i] ≤ C
        for j := 1 to C do w[i,j] := 0.0; 
        w[i,c[i]] := 1.0;
        ... 
end
Related Approaches

• Subject-oriented programming (SOP)
  – subject = set of mixins (class fragment representing a view)
  – can be implemented by inheritance
  – correspondence rules: relationships between parts of different subjects
  – combination rules: how to combine subjects

• Adaptive programming (AP)
  – write code as algorithm + traversal over graph in object diagram
  – automatically generate code for information passing
  – when design is modified, code adapts

• Others:
  – AOP, feature-oriented programming (e.g., GenVoca)
  – active libraries (e.g., Blitz ++), extensible programming environments (e.g., IP, Polyglot)
  – product-line engineering: implement family of products based on common features rather than common implementation assets
Summary: Building a Generator

- Domain engineering
  - concepts and roles, reusable solutions
  - features, configuration knowledge
- Domain-specific notations
  - specification language, expression templates, intentions
- Generative knowledge
  - algorithm families, datatypes, optimizations
  - model transformations, constructive proof tactics, templates
- Certification system (if required)
  - language semantics, program logic, safety policies
  - report generator
Applications

• Reuse and re-engineering
  – capture complex domain knowledge
  – raise level of abstraction
• Reconfiguration
  – allow (static or dynamic) adaptations based on changing environment
• Platform adaptation
  – separate algorithm development from platform deployment
• Prototyping and simulation
  – generate executable for prototype design
  – manually add production/mission-critical features later
  – automatically add logging, debugging code
• Component/service assembly
  – generate glue code
  – solve queries
• End-user development (EUD)
  – give novice users advanced capabilities for customizing applications
Suggestions
(for the practically inclined)

• Best approach depends on legacy assets:
  – code → code-based
  – models → model-based
  – proofs → proof-based

• Also depends on
  – target domain (science, engineering, business, …)
  – application (simulation, prototyping, production, …)
  – level of formality/correctness required

• Textbook mining
  – Automating “the classics” (e.g. Knuth, Numerical Recipes, …)

• Proof mining
  – extract numerical algorithms from analytic proofs

• Can we have generator-independent generation knowledge?
  – Knowledge representation is harder challenge than automation!
Suggestions
(for the theoretically inclined)

• Many concepts in need of semantics and (meta-)logic
  – 2-level languages, templates, schemas, …
• Exploit links to AI
• Exploit links to linguistics
  – explanations for derivations and verifications
  – intelligent summaries
• Develop theory of traceability
  – trace between model, code, VCs, etc.
• Raise level of abstraction for certifiable generation
  – generate plans rather than annotations
• Merge the paradigms
  – correct MDA
  – extend templates with symbolic reasoners
  – extend extraction with generative tactics, modeling support
More Thanks:

- Krzysztof Czarnecki
- Yannis Smaragdakis
- Todd Veldhuizen
- Juhan Ernits
- Maia-Liisa Anton
- ... you guys

... for slides and papers!

... for organization!

... for staying awake!