TECHNISCHE UNIVERSITEIT DELFT
Faculteit Elektrotechniek, Wiskunde en Informatica

TU Delft

Midterm Exam
IN4313 Model-Driven Software Development

March 31, 2009

- This is an examination with 7 questions worth of 70 points in total.
- Use of books, readers, or notes is not allowed
- Write clearly and avoid verbose explanations
- Specify your name, student number and degree programme
- Please use a separate sheet for each question
- Check whether you have received all pages and exercises
- Answer questions in correct English or Dutch
- Total number of pages: 5.

1. (10 points) Give a definition of the concept 'domain-specific language' and illustrate your definition with an example.

2. (10 points) Explain how type checking can be composed from rules for name resolution, type analysis, and type constraints. Use examples to illustrate your explanation.

3. Consider the following modules defining the signature of context-free grammars, the signature of signatures, and the transformation of context-free grammars to signatures, and answer the following questions:

   (a) (5 points) The transformation `cfg2sig` is not a bijection, that is, there is no transformation `sig2cfg` such that

   `<cfg2sig; sig2cfg> cfg == cfg`

   for any context-free grammar `cfg`. Explain why this is the case and provide a counter example. Point out the part of the transformation that causes this.

   (b) (5 points) While cfg2sig is not a bijection, there are transformations sig2cfg such that

   `<sig2cfg; cfg2sig> sig == sig`

   for any signature `sig`. Define such a transformation.

```
module CFG
signature
  constructors
    Grammar     : List(Sort) * List(Production) -> Grammar
    Anno        : Term -> Annotation
    Prod        : List(Symbol) * Symbol * List(Annotation) -> Production
    ListPlusSep : Symbol * Symbol -> Symbol
    ListStarSep : Symbol * Symbol -> Symbol
    ListPlus    : Symbol -> Symbol
    ListStar    : Symbol -> Symbol
    Option      : Symbol -> Symbol
                : Sort -> Symbol
    Lit         : String -> Symbol
    Sort        : Id -> Sort
    Tuple       : List(Term) -> Term
    List        : List(Term) -> Term
    Application : Constructor * List(Term) -> Term
    Constant    : Constructor -> Term
                : String -> Constructor
                : String -> String
                : String -> Id

module Signature
signature
  constructors
    ListType   : Sort -> Type
    SimpleType : Sort -> Type
    ConsDecl   : Id * List(Type) * Sort -> ConsDecl
```

```
    Signature   : List(Sort) * List(ConsDecl) -> Signature
    Sort        : Id -> Sort
                : String -> Id

module cfg2sig
  imports stratego-lib CFG Signature

strategies
  main = io-wrap(cfg2sig)

rules
  cfg2sig :
    Grammar(ss, ps) -> Signature(ss, cds)
    with cds := <map(production-to-consdecl)> ps

  production-to-consdecl :
    Prod(symbols, sym, annos) -> ConsDecl(x, types, sort)
    with x := <annos-to-constructor> annos
       ; types := <filter(symbol-to-type)> symbols
       ; sort := sym

  annos-to-constructor :
    [Anno(Application("cons", [Constant(c)]))] -> constr
    with constr := <unescape; un-double-quote> c

  symbol-to-type : Sort(x) -> SimpleType(Sort(x))
  symbol-to-type : ListStar(Sort(x)) -> ListType(Sort(x))
  symbol-to-type : ListPlus(Sort(x)) -> ListType(Sort(x))
  symbol-to-type : ListPlusSep(Sort(x), sym) -> ListType(Sort(x))
  symbol-to-type : ListStarSep(Sort(x), sym) -> ListType(Sort(x))
```

4. Consider the signature and rewrite rules for Prop terms. Provide a strategy and a selection of rules that normalize Prop terms to the following normal forms. Use the simplest strategy possible. Include the definition of the strategy.

   (a) (5 points) Conjunctive normal form

   (b) (5 points) NATA normal form (using only Not, And, True, and Atoms)

```
signature
  sorts Prop
    constructors
      False : Prop
      True  : Prop
      Atom  : String -> Prop
      Not   : Prop -> Prop
      And   : Prop * Prop -> Prop
      Or    : Prop * Prop -> Prop
      Eq    : Prop * Prop -> Prop
      Impl  : Prop * Prop -> Prop
  rules
    DAOL  : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
    DAOR  : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
    DOAL  : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
    DOAR  : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
    DN    : Not(Not(x)) -> x
    DMA   : Not(And(x, y)) -> Or(Not(x), Not(y))
    DMO   : Not(Or(x, y)) -> And(Not(x), Not(y))
    DefT  : True() -> Not(False())
    DefF  : False() -> Not(True())
    DefN  : Not(x) -> Impl(x, False)
    DefI  : Impl(x, y) -> Or(Not(x), y)
    DefE  : Eq(x, y) -> And(Impl(x, y), Impl(y, x))
    DefO1 : Or(x, y) -> Impl(Not(x), y)
    DefO2 : Or(x, y) -> Not(And(Not(x), Not(y)))
    DefA1 : And(x, y) -> Not(Or(Not(x), Not(y)))
    DefA2 : And(x, y) -> Not(Impl(x, Not(y)))
    IDefI : Or(Not(x), y) -> Impl(x, y)
    IDefE : And(Impl(x, y), Impl(y, x)) -> Eq(x, y)
```

5. Consider the following syntax definition of the language TRS that consists of rewrite rules over terms, the language EXP of expressions, and the embedding of the concrete syntax for the language EXP in TRS.

```
module TRS
  exports
    context-free syntax
      Id                      -> Term {cons("Var")}
      Id "(" {Term ","}* ")" -> Term {cons("Op")}
      Term "->" Term          -> Rule {cons("Rule")}
      Rule*                   -> TRS  {cons("TRS")}
module Exp
  exports
    sorts Exp
     context-free syntax
       Id                     -> Exp {cons("Var")}
       Id "(" {Exp ","}* ")" -> Exp {cons("Call")}
       Exp "+" Exp            -> Exp {cons("Plus"),left}
module TRS-Exp
  imports TRS Exp
  exports
    context-free syntax
      "|[" Exp "]|" -> Term {cons("ToTerm"),prefer}
      "~" Term      -> Exp  {cons("FromTerm"),prefer}
    variables
      "e"[0-9]* -> Exp {prefer}
```

(a) (6 points) Define rewrite rules and a strategy that transform a TRS-EXP TRS to a plain TRS.

(b) (2 points) Give an example of the application of the transformation.

(c) (2 points) Explain the role of the `ToTerm` and `FromTerm` productions.

6. Consider the following abstract definition of access control rule weaving in WebDSL:

`define page` p($\vec{x}$) { *elem** }

Page: ⇓ `rule page` p($\vec{x}$) { e } ⇓

`define page` p($\vec{x}$) {
  `init{ if(!`e`) { redirect accessDenied(); } }`
  *elem**
}

(a) (3 points) Explain the transformation.

(b) (7 points) Give an implementation of this transformation using dynamic rules.

7. (10 points) Consider the following abstract syntax definition describing a simplified version of template definitions and template calls in WebDSL.

```
module TEMPL
signature
  sorts Elem Exp App
  constructors
    App  : List(Elem)                       -> App  // application
    Def  : Id * List(Id)  * List(Elem) -> Elem // template definition
    Call : Id * List(Exp) * List(Elem) -> Elem // template call
    Var  : Id                               -> Exp  // variable
    Fld  : Exp * Id                         -> Exp  // field access
```

The following term is an example of the use of this language, defining a `blog` template with a local definition of the `body` template, and calling the `main` template.

```
App([
  Def("main", [], [ Call("top",[],[]), Call("body",[],[])])
  Def("top",  [], [ Call("menubar",[],[ ... ]) ])
  Def("body", [], [])
  Def("entry", ["e"], [ Call("header",[Fld(Var("e"), "title")],[]) ])
  Def("blog", ["b"], [
    Def("body",[],[ Call("entry",[Fld(Fld(Var("b"), "entries"), "first")],[]) ])
    Call("main",[],[])
  ])
])
```

One implementation of such template definitions is inlining of template definitions. Using inlining, the definition of the `blog` template is transformed to:

```
Def("blog", ["b"], [
  Call("menubar",[], [ ... ])
  Call("header",[Fld(Fld(Fld(Var("b"), "entries"), "first"), "title")])
])
```

Note that actual expression parameters are substituted for formal parameters. Give rewrite rules and a strategy implementing template inlining for TEMPL.

End of exam