Term Annotation in Stratego

Martin Bravenboer, Eelco Visser

mbravenb@cs.uu.nl

Institute of Information and Computing Sciences
University Utrecht
The Netherlands

Introduction

What is an annotation?

term attached to a term without being part of the structure

Rules and strategies can be applied to a structure without knowing about the annotation.

general usage: store terms that

- don’t fit into a signature
- you don’t want in a signature

Example usage (1)

maintaining semantic information on terms during a transformation

- type (for example in type-checker)
- scope of variables and functions
- escape information of variables
- mode of a tile in instruction selection

Example usage (2)

cooperation with other datatypes

- XML (limited attributes)
- transformation to XHTML

data-oriented applications:

- data not being part of the structure

Syntax and basic strategies

Annotations in match and build

match:

```
?BinOp(op, e1{Int}, e2{Int})   ?e1{Int}  ?_{_}
```

build:

```
!BinOp(PLUS, e1, e2){Int}
```

Annotations in rules

rule example:

```
TcExp:
  BinOp(op, e1{Int}, e2{Int})  ->
  BinOp(op, e1, e2){Int}
```

before annotations:

```
TcExp:
  BinOp(op, Typed(e1, Int), Typed(e2, Int)) ->
  Typed(BinOp(op, e1, e2), Int)
```

What is an annotation?

options:

- Term has a list of annotations (ATerm).
- Term has one annotation, which is one term, which might be a list (Stratego).

list-approach:

- list-matching
- `!Var("a"){[Int, Float]}` should become `!Var("a"){Int, Float}`

Basic strategies for annotation

Annotations module provides some basic strategies:

```
get-annotations = ?t;       prim(...)
set-annotations = ?(t, a); prim(...)
rm-annotations  = ?t;       prim(...)
has-annos = ?_{_}
strip-annos = bottomup(rm-annotations)
```

How do annotations fit into existing Stratego constructs?

Annotation construction in overlays

An annotation can be attached in an overlay.

```
overlays
   IntBinOp(op, x, y) = BinOp(op, x, y){Int}
```

Congruences and annotations

Annotations are preserved on the application of a congruence.

```
<Call(Var(is-string), list(exp))>
    Call(Var("f"), []){Scope(Var("g"))}
 => Call(Var("f"), []){Scope(Var("g"))}
```

Congruences can apply strategies to annotations.

```
Call(Var(is-string), list(exp))
    {Scope(Var(is-string))}
```

Deconstruction with annotation

pattern:

```
?p1#(p2){anno}
```

example:

```
deconstruct:
  p1#(p2){anno} -> (p1, p2, anno)

<deconstruct> Plus(e1, e2){Int}
  => ("Plus", [e1, e2], Int)
```

all, one, some preserve annotations

```
test28 =
  apply-test(!"test28"
, all(id); get-annotations
, !Var("a"){Int}
, Int
)
```

→ simple-traversals preserve annotations

Don’t loose your annotation

Transparency of annotation

Annotation is not part of the structure of a term.

```
!Plus(e1, e2){Int} => Plus(e1, e2)
```

```
Desugar: Plus(e1, e2) -> BinOp(PLUS, e1, e2)
```

```
<Desugar> Plus(e1, e2){Int}
  => BinOp(PLUS, e1, e2)
```

Problem

Annotations are *not* preserved over the application of a classic rule.

```
Desugar:
  Plus(e1, e2) -> BinOp(PLUS, e1, e2)
```

application on term with annotation:

```
<Desugar> Plus(e1, e2){Int}
  => BinOp(PLUS, e1, e2){}
```

Preserving annotations

```
preserve-anno(s)
```

preserves the annotation of the current term over the application of a strategy

```
<preserve-anno(Desugar)>
     Plus(e1, e2){Int}
  => BinOp(PLUS, e1, e2){Int}
```

future: attributes for rules and strategies?

More annotations

Properties

anno properties: `[(key, value)]`.

currently no special syntax, just strategies

- `has-prop(k) has-prop(k, c) get-prop(k)`
- `apply-prop(k, s)`
- `replace-prop(k, v) add-prop(k, v) set-prop(k, v)`

More and more annotations

If annotations are passed between applications, namespaces for properties can be useful.

```
signature
  constructors
    Tiger: Namespace     Type: Property
overlays
  TigerType = (Tiger, Type)
strategies
  get-type    = get-prop(!TigerType)
  set-type(t) = set-prop(!TigerType, t)
```

Discussion and status

Drawbacks

- Annotations easily get lost.

Drawbacks

- Annotations easily get lost.

```
main = !(1,2){Int}; Swap; ?(2, 1){Int}
```

→ Although annotations are transparent, some generic strategies must handle annotations:

- standard library
- build-in primitives like all, one

Drawbacks

- Annotations easily get lost.
- Annotations with semantic information must be kept up-to-date.
- danger of variants

Current implementation problems

Implementation must be considered alpha.

- list versus single term
- Anno in overlay is not allowed.
- Anno gets lost in congruence `Term(...)`.
- `?_{_}` matches term without anno (`has-annos`).
- `?_{Term}` results in seg-fault on term without anno.
- most likely a lot of library problems

Questions or remarks?