Software Transformation

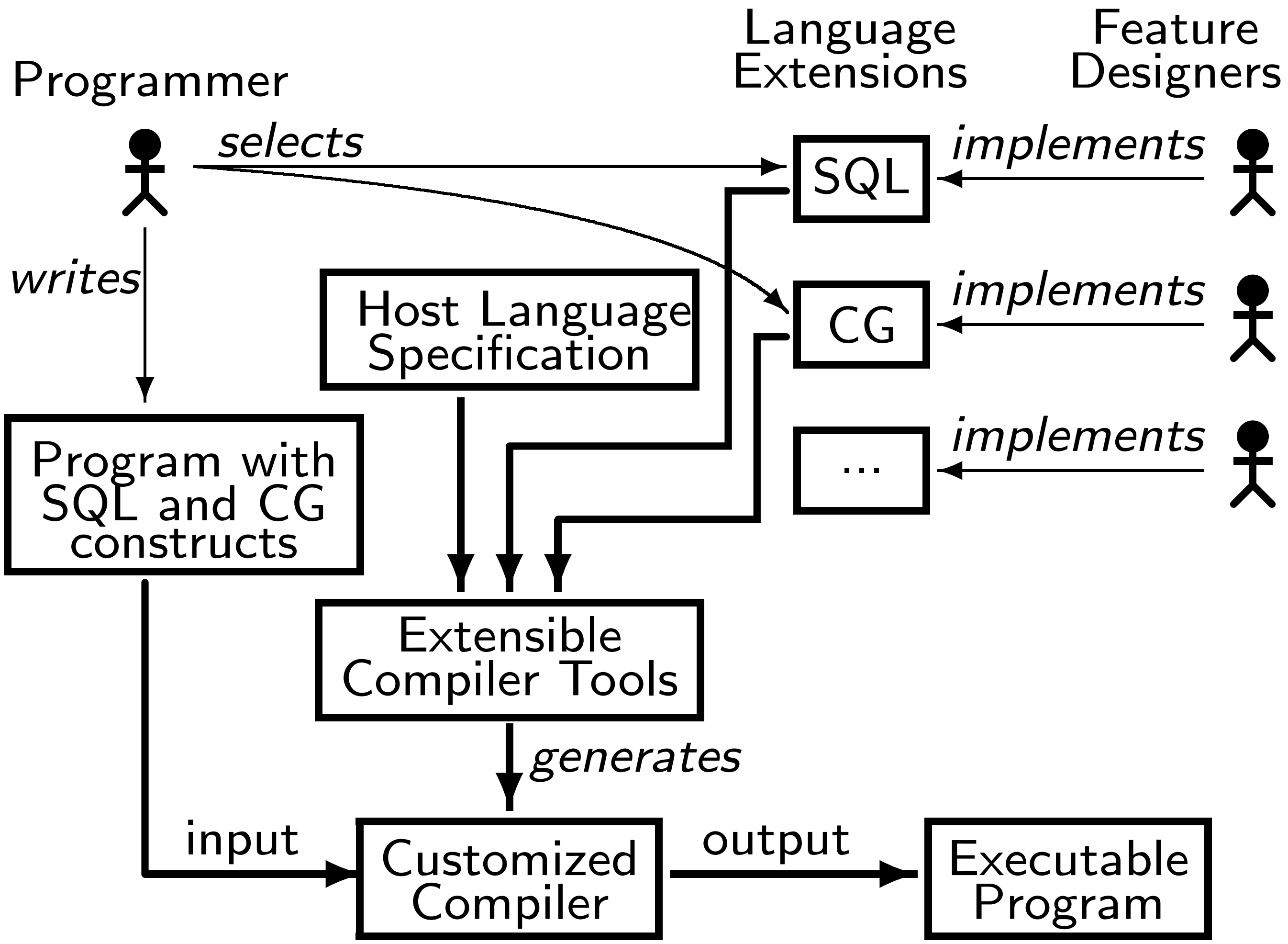
2 questions in transformation:

1. what constructs are to be transformed?
2. what are they to be transformed into?

Our Position: Transformations can be made more expressive and useful when they are informed by semantic information of the source.

- This can be general-purpose information like a constructs type
- or domain-specific information like the space required for unbounded integers in a computational geometry application.

Context: Extensible Languages



Attribute Grammars with Forwarding

```
foreach Cow c in herd do c.milk();
```

⇒

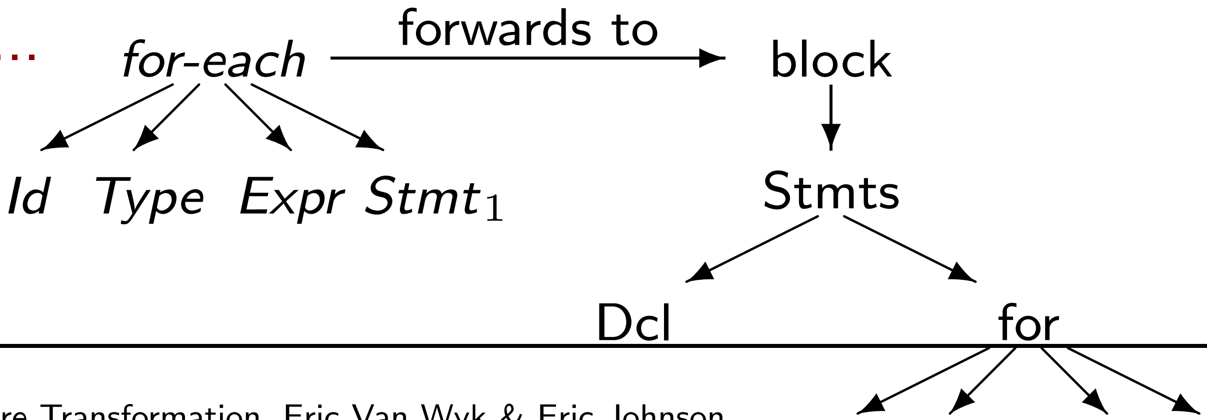
```
{ Cow c ;
  for ( Iterator iter = herd.iterator(); iter.hasnext(); )
      { c = ( Cow ) iter.next(); c.milk();  } }
```

foreach: $\langle St \rangle ::=$ “foreach” $\langle Type \rangle$ $\langle Id \rangle$ “in” $\langle E \rangle$ “do” $\langle St \rangle$

$St.errors = if\ \ not\ \ Type.implements(Collection)\ \ then\ \ldots$

forwardsTo “ ... specification of above for loop ... ”

Graphically...



For example: Computational Geometry

- Algorithms based on primitives that make qualitative decision: *e.g.* is "a point x to the left or right of line l"
- Many algorithms are simplified if intermediate values can have unbounded precision.
- We can statically compute their size in bits.
- unbounded_add: $\langle E \rangle ::= \langle E \rangle +_u \langle E \rangle$

  $$E_0.size = max(E_1.size, E_2.size) + 1$$

  unbounded_var: $\langle E \rangle ::= \langle Id \rangle$

  $$E.size = 53$$

- Generate "unrolled loops" that perform these operations

Question(s):

Can we unify semantic analysis and efficient rewriting mechanisms?

- Can we do more than combine incremental attribute grammar evaluators and conditional rewrite rules?
- For example, do some transformations preserve semantic values (such as type) so that attribute re-computation is not necessary?
- Can rewrite rules also define semantics for the rewritten term?