HACS—A Formalism for Compiler Development

Kristoffer H. Rose
Two Sigma *Labs* & NYU

Tuesday, May 27, 2014
DIKU “COPLAS”

Outline

Compilers and translators

- Programming Culture

Compilers and translators

- Programming Culture bordering on Art.

Compilers and translators

- Programming Culture bordering on Art.
- We teach semantic techniques

Compilers and translators

- Programming Culture bordering on Art.
- We teach semantic techniques but then abandon them to hand-written algorithmic code.

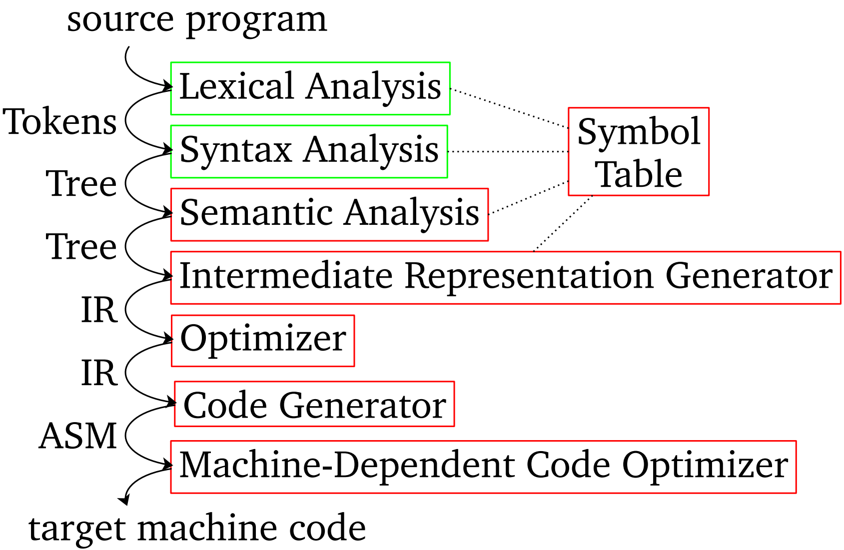
Compilers and translators

- Programming Culture bordering on Art.
- We teach semantic techniques but then abandon them to hand-written algorithmic code.
- Little compiler programmer support after parser generation.

Compilers and translators

- Programming Culture bordering on Art.
- We teach semantic techniques but then abandon them to hand-written algorithmic code.
- Little compiler programmer support after parser generation.
- HACS aims to change this, and provide high level support for full compiler programming experience.

Outline


source program
Lexical Analysis
Tokens
Syntax Analysis
Tree
Semantic Analysis
Tree
Intermediate Representation Generator
IR
Optimizer
IR
Code Generator
ASM
Machine-Dependent Code Optimizer
target machine code
Symbol Table

What formalizations are we using?

Lexical Analysis. Regular Expressions.

What formalizations are we using?

Lexical Analysis. Regular Expressions.

Syntax Analysis. LL/LALR Parser Generators.

What formalizations are we using?

Lexical Analysis. Regular Expressions.

Syntax Analysis. LL/LALR Parser Generators.

Semantic Analysis. Attribute Grammars.

What formalizations are we using?

Lexical Analysis. Regular Expressions.

Syntax Analysis. LL/LALR Parser Generators.

Semantic Analysis. Attribute Grammars.

IR Generator. Translation Schemes.

What formalizations are we using?

Lexical Analysis. Regular Expressions.

Syntax Analysis. LL/LALR Parser Generators.

Semantic Analysis. Attribute Grammars.

IR Generator. Translation Schemes.

Optimizer. Attribute Grammars, Translation Schemes, Custom Algorithms, . . .

What formalizations are we using?

Lexical Analysis. Regular Expressions.

Syntax Analysis. LL/LALR Parser Generators.

Semantic Analysis. Attribute Grammars.

IR Generator. Translation Schemes.

Optimizer. Attribute Grammars, Translation Schemes, Custom Algorithms, . . .

Code Generator. Translation Schemes.

What formalizations are we using?

Lexical Analysis. Regular Expressions.

Syntax Analysis. LL/LALR Parser Generators.

Semantic Analysis. Attribute Grammars.

IR Generator. Translation Schemes.

Optimizer. Attribute Grammars, Translation Schemes, Custom Algorithms, . . .

Code Generator. Translation Schemes.

Peep-hole optimizer. All bets are off. . .

What formalizations are we using?

Lexical Analysis. Regular Expressions.

Syntax Analysis. LL/LALR Parser Generators.

Semantic Analysis. Attribute Grammars.

IR Generator. Translation Schemes.

Optimizer. Attribute Grammars, Translation Schemes, Custom Algorithms, . . .

Code Generator. Translation Schemes.

Peep-hole optimizer. All bets are off. . .

Symbol Table. Side effects and explicit scope structures.

Compilers and translators

- Programming Culture

Compilers and translators

- Programming Culture bordering on Art.

Compilers and translators

- Programming Culture bordering on Art.
- We teach semantic techniques

Compilers and translators

- Programming Culture bordering on Art.
- We teach semantic techniques but then abandon them to hand-written algorithmic code.

Compilers and translators

- Programming Culture bordering on Art.
- We teach semantic techniques but then abandon them to hand-written algorithmic code.
- Little compiler programmer support after parser generation.

Compilers and translators

- Programming Culture bordering on Art.
- We teach semantic techniques but then abandon them to hand-written algorithmic code.
- Little compiler programmer support after parser generation.
- HACS aims to change this, and provide high level support for full compiler programming experience.

HACS—A New *Formally Founded* Compiler Generator

1. Designed to directly support existing formal notations.
2. Formally defined in terms of a “core” language, which is in fact CRSX.
3. Integrated with special purpose compiler support algorithms.

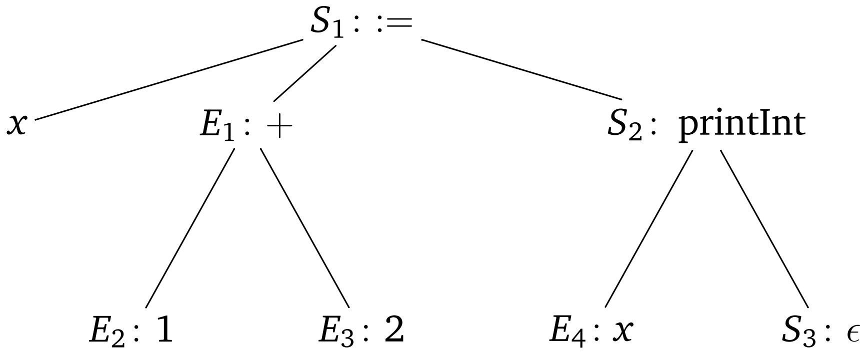
On Attribute Grammars

- Based on Knuth's seminal 1968 "Semantics of Context-Free Languages"
- Dominate teaching in compiler construction
- Can be adapted to imperative programming ($ in yacc)
- Compatible with rewriting, can be implemented with rewriting (ASF+SDF)

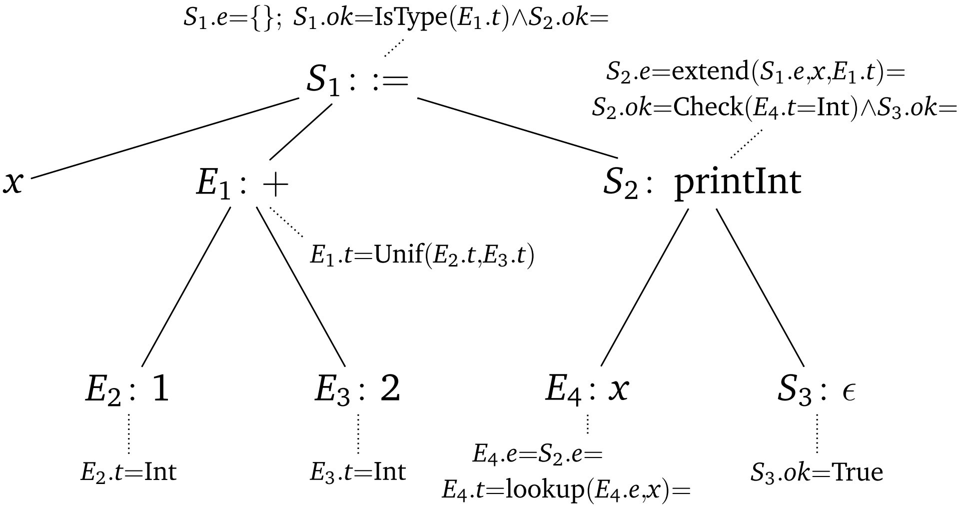
Attribute Grammar = Syntax-Directed Definition

PRODUCTION	SEMANTIC RULES	
$S \rightarrow \textbf{name} := E_1; S_2$	$E_1.e = S.e; S_2.e = \text{extend}(S.e, \textbf{name}.sym, E_1.t);$ $S.ok = \text{IsType}(E_1.t) \wedge S_2.ok$	$(S1)$
$\mid \{\ S_1\ \}\ S_2$	$S_1.e = S.e; S_2.e = S.e; S.ok = S_1.ok \wedge S_2.ok$	$(S2)$
$\mid \text{printInt}\ \ E_1; S_2$	$E_1.e = S.e; S.ok = \text{Check}(E_1.t = \text{Int}) \wedge S_2.ok$	$(S3)$
$\mid \text{printFloat}\ \ E_1; S_2$	$E_1.e = S.e; S.ok = \text{Check}(E_1.t = \text{Float}) \wedge S_2.ok$	$(S4)$
$\mid \epsilon$	$S.ok = \text{True}$	$(S5)$
$E \rightarrow E_1 + E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \text{Unif}(E_1.t, E_2.t)$	$(E1)$
$\mid E_1 * E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \text{Unif}(E_1.t, E_2.t)$	$(E2)$
$\mid \textbf{int}$	$E.t = \text{Int}$	$(E3)$
$\mid \textbf{float}$	$E.t = \text{Float}$	$(E4)$
$\mid \textbf{name}$	$E.t = \text{if defined}(E.e, \textbf{name}.sym)$ $\text{then lookup}(E.e, \textbf{name}.sym)$ else TypeErr	$(E5)$

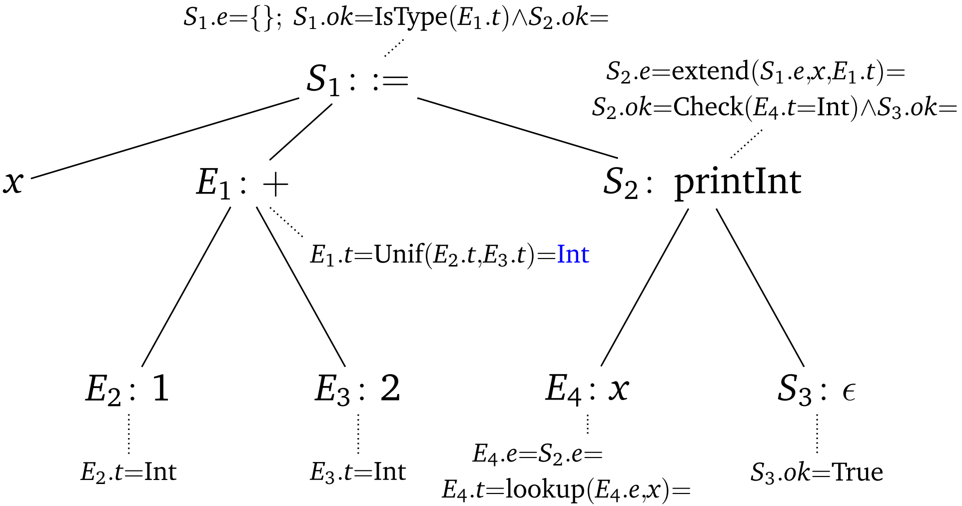
Example: “`x:=1+2; printInt x`”



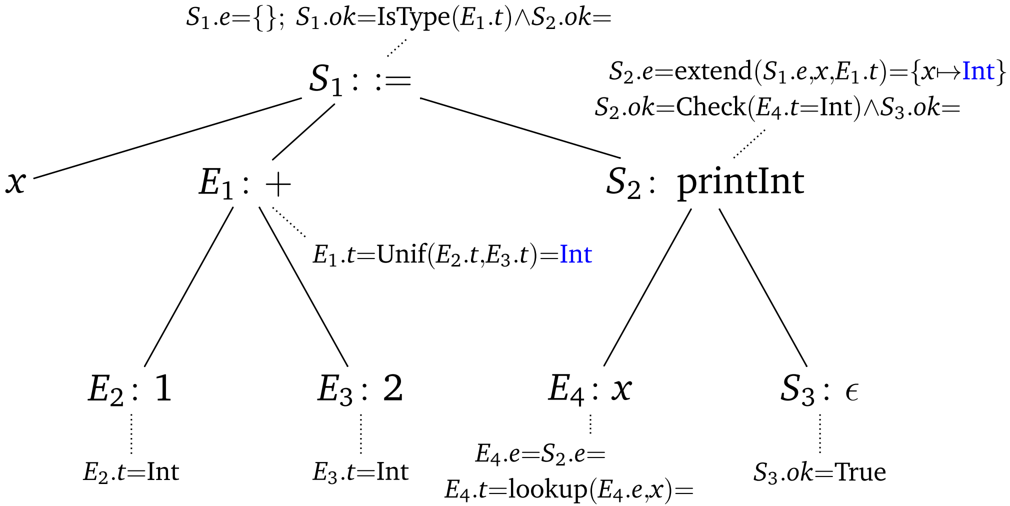
Example: "`x:=1+2; printInt x`" with Attributes



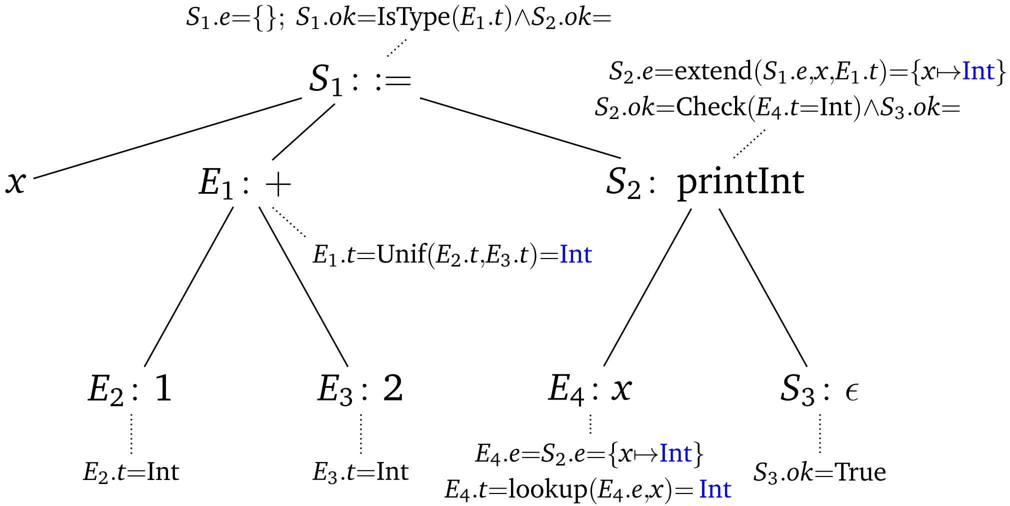
Example: "`x:=1+2; printInt x`" with Attributes



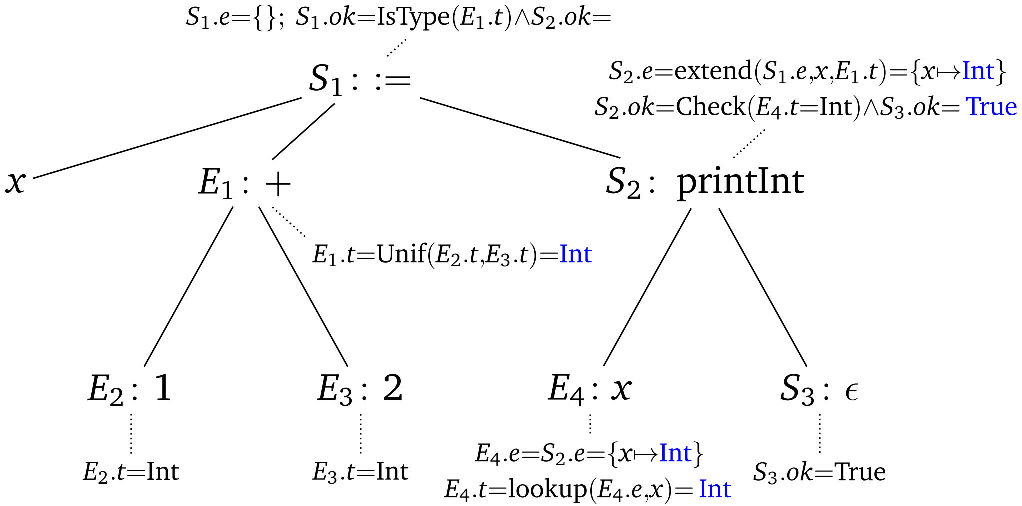
Example: "`x:=1+2; printInt x`" with Attributes



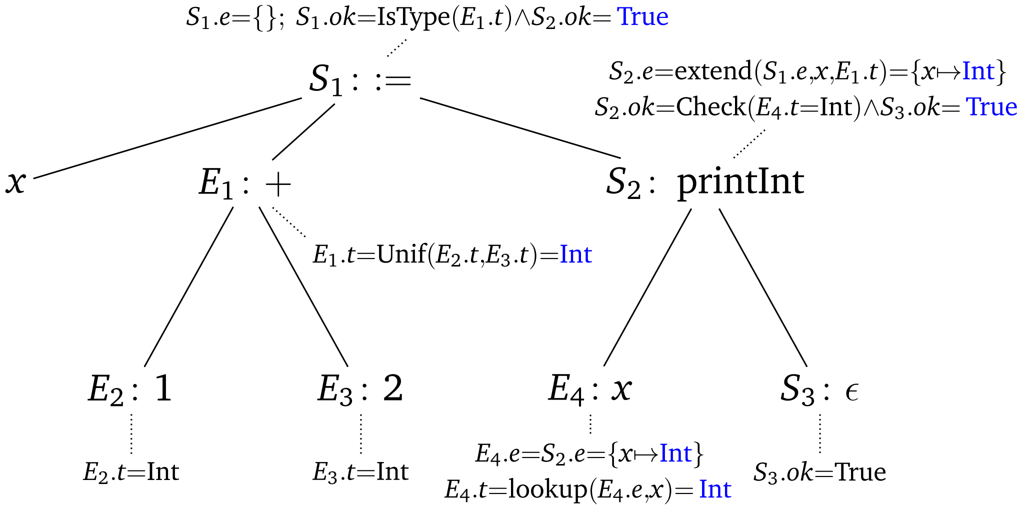
Example: “`x:=1+2; printInt x`” with Attributes



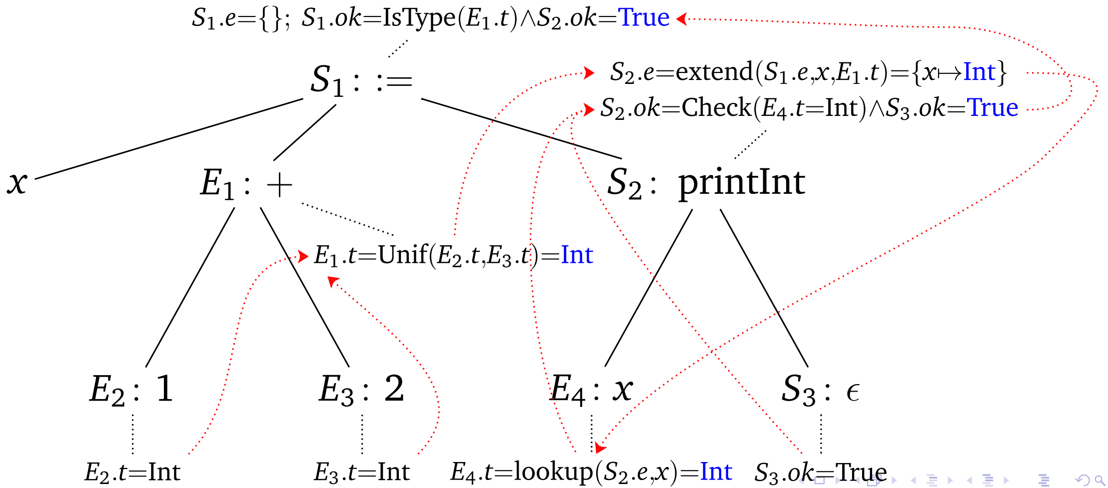
Example: “`x:=1+2; printInt x`” with Attributes



Example: “`x:=1+2; printInt x`” with Attributes



Example: "`x:=1+2; printInt x`" with Attributes



Attribute Propagation Patterns

- Synthesized attributes mostly depend only on synthesis. (An AG is "S-attributed" if this is exclusively the case.)
- Cross-synthesized-inherited attributes depend left to right. (It is "L-attributed," like inference systems.)

HACS Features

- Parser generation notation.
  Supports Higher-Order Abstract Syntax.
- Rules for synthesis.
- Rules for recursive compilation schemes.
- Polymorphic sort system.
- All rules and sorts use native user syntax.

Words, words, words. . .

Let’s code.

HACS Program (I)

```
module "edu.nyu.cims.cc.Check" {

// PARSER.

space [ \t\n] ;
token INT    | [0−9]+ ;
token FLOAT  | [0−9]+ [.] [0−9]∗ ([Ee] [−+]? [0−9]+)? ;
token ID     | [a−z] [a−zA−Z0−9_]∗ ;
sort Name | symbol ⟦⟨ID⟩⟧ ;

sort S  | ⟦ ⟨[x:Name]⟩ := ⟨E⟩; ⟨S[x:Name]⟩ ⟧ | ⟦ { ⟨S⟩ } ⟨S⟩ ⟧
        | ⟦ printInt ⟨E⟩; ⟨S⟩ ⟧ | ⟦ printFloat ⟨E⟩; ⟨S⟩ ⟧
        | ⟦⟧ ;

sort E  | ⟦ ⟨E⟩ + ⟨E@1⟩ ⟧| ⟦ ⟨E@1⟩ ∗ ⟨E@2⟩ ⟧@1
        | ⟦ ⟨INT⟩ ⟧@2 | ⟦ ⟨FLOAT⟩ ⟧@2 | ⟦⟨Name⟩ ⟧@2
        | sugar ⟦ ( ⟨E#1⟩ ) ⟧@2 →E#1 ;
```

HACS Program (II)

// SEMANTIC SORTS & HELPERS.

```
sort Bool | ⟦T⟧ | ⟦F⟧;
| scheme ⟦⟨Bool⟩∧⟨Bool⟩⟧;  ⟦T∧T⟧ → ⟦T⟧;  default ⟦⟨Bool#1⟩∧⟨Bool#2⟩⟧ →⟦F⟧;

sort Type | Int | Float | TypeErr;
| scheme Unif(Type,Type);
Unif(Int, Int) → Int;          Unif(Int, Float) → Float;
Unif(Float, Int) → Float;   Unif(Float, Float) → Float;
default Unif(#1, #2) →TypeErr ;

sort Bool;
| scheme IsType(Type); IsType(Int) →⟦T⟧;  IsType(Float) → ⟦T⟧;  IsType(TypeErr) →⟦F⟧;
| scheme CheckInt(Type); CheckInt(Int) →⟦T⟧;  default CheckInt(#) →⟦F⟧;
| scheme CheckFloat(Type); CheckFloat(Float) →⟦T⟧; default CheckFloat(#) →⟦F⟧;
```

HACS Program (III)

```
// TYPE CHECKING.

sort CheckResult | ⟦Yes⟧;
| scheme Check(S) ;
Check(#s↑ok(⟦T⟧)) →⟦Yes⟧ ;
Check(#s↑ok(⟦F⟧)) →error⟦Type Error.⟧ ;

// Attributes.
attribute ↑ok(Bool);
attribute ↑t(Type);
attribute ↓e{Name:Type};

// Rules for S sort:
// – Synthesizes attribute ok.
// – Inherited attribute S.e is distributed by Te scheme (and helper Te2).
sort S | ↑ok | scheme ⟦Te ⟨S⟩⟧ ↓e | scheme ⟦Te2 ⟨S⟩⟧ ↓e ;
```

HACS Program (IV)

// (S1) $S \to \textbf{name} := E_1; S_2$ has three stages:
// 1. $E_1.e = S.e$ and recurse over E_1 (to get automatic propagation).
$[\![$ Te x := $\langle$E#1$\rangle$; $\langle$S#2[x]$\rangle$ $]\!]$ $\to$ $[\![$ Te2 x := Te$\langle$E#1$\rangle$; $\langle$S#2[x]$\rangle$ $]\!]$;
// 2. When $E_1.t$ is available then $S_2.e = \text{extend}(S.e, \textbf{name}.sym, E_1.t)$ and recurse over S_2.
$[\![$ Te2 x := $\langle$E#1$\uparrow$t(#t1)$\rangle$; $\langle$S#2[x]$\rangle$ $]\!]$ $\to$ $[\![$ x := $\langle$E#1$\rangle$; Te$\langle$S#2[x]$\downarrow$e{x:#t1}$\rangle$ $]\!]$;
// 3. When $S_2.ok$ is (also) available then we can synthesize $S.ok = \text{IsType}(E_1.t) \wedge S_2.ok$.
$[\![$ x := $\langle$E#1$\uparrow$t(#t1)$\rangle$; $\langle$S#2[x]$\uparrow$ok(#ok2)$\rangle$ $]\!]$$\uparrow$ok($[\![$$\langle$Bool IsType(#t1)$\rangle$$\wedge$$\langle$Bool#ok2$\rangle$$]\!]$) ;

// (S2) $S \to \{S_1\}\, S_2$:
// 1. $S_1.e = S.e$ and $S_2.e = S.e$ are propagated and recursed over.
$[\![$ Te { $\langle$S#1$\rangle$ } $\langle$S#2$\rangle$ $]\!]$ $\to$ $[\![$ { Te$\langle$S#1$\rangle$ } Te$\langle$S#2$\rangle$ $]\!]$;
// 2. When $S_1.ok$ and $S_2.ok$ are available then synthesize $S.ok = S_1.ok \wedge S_2.ok$.
$[\![$ { $\langle$S#1$\uparrow$ok(#ok1)$\rangle$ } $\langle$S#2$\uparrow$ok(#ok2)$\rangle$ $]\!]$$\uparrow$ok($[\![$$\langle$Bool#ok1$\rangle$ $\wedge$$\langle$Bool#ok2$\rangle$$]\!]$) ;

HACS Program (V)

// $(S3) \quad S \to \text{printInt } E_1; S_2$:
// 1. $E_1.e = S.e$ and $S_2.e = S.e$ are propagated and recursed over.
$[\![$ Te printInt $\langle$E#1$\rangle$; $\langle$S#2$\rangle$ $]\!] \to$ $[\![$ printInt Te$\langle$E#1$\rangle$; Te$\langle$S#2$\rangle$ $]\!]$;
// 2. When $S_1.t$ and $S_2.ok$ are available then synthesize $S.ok = \text{Check}(\text{Int} = E_1.t) \wedge S_2.ok$.
$[\![$ printInt $\langle$E#1↑t(#t1)$\rangle$; $\langle$S#2↑ok(#ok2)$\rangle$ $]\!]$↑ok($[\![$ $\langle$Bool CheckInt(#t1)$\rangle$ $\wedge\langle$Bool#ok2$\rangle$ $]\!]$) ;

// $(S4) \quad S \to \text{printFloat } E_1; S_2$:
// 1. $E_1.e = S.e$ and $S_2.e = S.e$ are propagated and recursed over.
$[\![$ Te printFloat $\langle$E#1$\rangle$; $\langle$S#2$\rangle$ $]\!] \to$ $[\![$ printFloat Te$\langle$E#1$\rangle$; Te$\langle$S#2$\rangle$ $]\!]$;
// 2. When $S_1.t$ and $S_2.ok$ are available then synthesize $S.ok = \text{Check}(\text{Float} = E_1.t) \wedge S_2.ok$.
$[\![$ printFloat $\langle$E#1↑t(#t1)$\rangle$; $\langle$S#2↑ok(#ok2)$\rangle$ $]\!]$↑ok($[\![$ $\langle$Bool CheckFloat(#t1)$\rangle$ $\wedge\langle$Bool#ok2$\rangle$ $]\!]$) ;

// $(S5) \quad S \to \epsilon$: No distribution necessary; synthesize $S.ok = \text{True}$.
$[\![$ Te $]\!]$ $\to$ $[\![]\!]$;
$[\![$ $]\!]$ ↑ok($[\![$T$]\!]$) ;

HACS Program (VI)

// Rules for E sort:
// – Synthesizes attribute $E.t$.
// – Inherited attribute $E.e$ is distributed by Te scheme (with helper Te2).
sort $\mathtt{E} \mid \uparrow\mathtt{t} \mid$ **scheme** $[\![\mathtt{Te}\ \langle\mathtt{E}\rangle]\!]\ \downarrow\mathtt{e}$;

// $(E1)\quad E \to E_1 + E_2$:
// 1. $E_1.e = E.e$ and $E_2.e = E.e$ recursively propagated (note use of parenthesis sugar).
$[\![\ \mathtt{Te}\ (\ \langle\mathtt{E\#1}\rangle + \langle\mathtt{E\#2}\rangle\ )\ ]\!] \to\ [\![\ (\mathtt{Te}\langle\mathtt{E\#1}\rangle) + (\mathtt{Te}\langle\mathtt{E\#2}\rangle)\ ]\!]$;
// 2. When $E_1.t$ and $E_2.t$ available then synthesize $E.t = \mathrm{Unif}(E_1.t, E_2.t)$.
$[\![\ \langle\mathtt{E\#1}\uparrow\mathtt{t(\#t1)}\rangle + \langle\mathtt{E\#2}\uparrow\mathtt{t(\#t2)}\rangle\ ]\!]\uparrow\mathtt{t(Unif(\#t1,\#t2))}$;

// $(E2)\quad E \to E_1 * E_2$:
// 1. $E_1.e = E.e$ and $E_2.e = E.e$ recursively propagated (note use of parenthesis sugar).
$[\![\ \mathtt{Te}\ (\ \langle\mathtt{E\#1}\rangle * \langle\mathtt{E\#2}\rangle\ )\ ]\!] \to\ [\![\ (\mathtt{Te}\langle\mathtt{E\#1}\rangle) * (\mathtt{Te}\langle\mathtt{E\#2}\rangle)\ ]\!]$;
// 2. When $E_1.t$ and $E_2.t$ available then synthesize $E.t = \mathrm{Unif}(E_1.t, E_2.t)$.
$[\![\ \langle\mathtt{E\#1}\uparrow\mathtt{t(\#t1)}\rangle + \langle\mathtt{E\#2}\uparrow\mathtt{t(\#t2)}\rangle\ ]\!]\uparrow\mathtt{t(Unif(\#t1,\#t2))}$;

HACS Program (VII)

// (*E*3) $E \rightarrow$ **int**:
// 1. Propagation leaf: set $E.t$ = Int directly.
$[\![$ Te $\langle$INT#1$\rangle$ $]\!] \rightarrow$ $[\![$ $\langle$INT#1$\rangle$ $]\!]\uparrow$t(Int) ;
// 2. Synthesize $E.t$ = Int.
$[\![$ $\langle$INT#1$\rangle$ $]\!]\uparrow$t(Int) ;

// (*E*4) $E \rightarrow$ **float**:
// 1. Propagation leaf: set $E.t$ = Float directly.
$[\![$ Te $\langle$FLOAT#1$\rangle$ $]\!]\rightarrow$ $[\![$ $\langle$FLOAT#1$\rangle$ $]\!]\uparrow$t(Float) ;
// 2. Synthesize $E.t$ = Float.
$[\![$ $\langle$FLOAT#1$\rangle$ $]\!]\uparrow$t(Float) ;

//(*E*5) $S \rightarrow$ **name**: There are two disjoint propagation cases:
// a. defined($E.e$, **name**.sym) so $E.t$ = lookup($E.e$, **name**.sym).
$[\![$ Te x $]\!]$ $\downarrow$e{x : #t} $\rightarrow$ $[\![$ x $]\!]$ $\uparrow$t(#t) ;
// b. $\neg$defined($E.e$, **name**.sym) so $E.t$ = TypeErr.
$[\![$ Te x $]\!]$ $\downarrow$e{$\neg$x} $\rightarrow$ $[\![$ x $]\!]$ $\uparrow$t(TypeErr) ;
}

HACS, Revisited

- Lexical and parsing specification (standard).
- Rewrite rules.
- Synthesis rules.

HACS Rewrite Rules with Explicit Propagation

$$F\left(\cdots \begin{matrix} A(\cdot)\uparrow a(\cdot) \\ X[\cdot]\uparrow x(\cdot) \end{matrix} \cdots\right)\downarrow c(\cdot) \to \cdots \begin{matrix} B(\cdot)\uparrow b(\cdot) \\ Y[\cdot]\uparrow y(\cdot) \end{matrix} \cdots \begin{matrix} G(\cdot)\downarrow g(\cdot) \\ Z[\cdot]\downarrow z(\cdot) \end{matrix} \cdots$$

becomes

$$F\left(\cdots \begin{matrix} A(\cdot) \\ X[\cdot] \end{matrix} \cdots\right)\downarrow c(\cdot) \to F'\left(\cdots \begin{matrix} \mathrm{Needs}_a(A(\cdot))) \\ \mathrm{Needs}_x(X[\cdot]) \end{matrix} \cdots\right)\downarrow c(\cdot)$$

$$F'\left(\cdots \begin{matrix} A(\cdot)\uparrow a(\cdot) \\ X[\cdot]\uparrow x(\cdot) \end{matrix} \cdots\right)\downarrow c(\cdot) \to \cdots \begin{matrix} B(\cdot)\uparrow b(\cdot) \\ Y[\cdot]\uparrow y(\cdot) \end{matrix} \cdots \begin{matrix} G(\cdot)\downarrow g(\cdot) \\ Z[\cdot]\downarrow z(\cdot) \end{matrix} \cdots$$

HACS Synthesis Rules with Explicit Propagation

$$A\Big(\cdots \begin{matrix} B(\cdot)\uparrow b(\cdot) \\ X[\cdot]\uparrow x(\cdot) \end{matrix} \cdots\Big)\uparrow a(\cdot)$$

becomes

$$\mathrm{Needs}_a\Big(A\Big(\cdots \begin{matrix} B(\cdot) \\ X[\cdot] \end{matrix} \cdots\Big)\Big) \to \mathrm{Collect}_R\Big(A\Big(\cdots \begin{matrix} \mathrm{Needs}_b(B(\cdot)) \\ \mathrm{Needs}_x(X[\cdot]) \end{matrix} \cdots\Big)\Big)$$

$$\mathrm{Collect}_R\Big(A\Big(\cdots \begin{matrix} B(\cdot)\uparrow b(\cdot) \\ X[\cdot]\uparrow x(\cdot) \end{matrix} \cdots\Big)\Big) \to A\Big(\cdots \begin{matrix} B(\cdot)\uparrow b(\cdot) \\ X[\cdot]\uparrow x(\cdot) \end{matrix} \cdots\Big)\uparrow a(\cdot)$$

with one R per $A\big(\cdots {\scriptstyle\begin{matrix} B(\cdot) \\ X[\cdot] \end{matrix}} \cdots\big)$.

First- and second-order term rewriting

First-order:

$$\begin{array}{rcl} \texttt{append}(\texttt{nil}, l) & \rightarrow & l \\ \texttt{append}(\texttt{cons}(x, l_1), l_2) & \rightarrow & \texttt{cons}(x, \texttt{append}(l_1, l_2)) \end{array}$$

First- and second-order term rewriting

First-order:

$$\begin{aligned}\mathtt{append}(\mathtt{nil}, l) &\rightarrow l \\ \mathtt{append}(\mathtt{cons}(x, l_1), l_2) &\rightarrow \mathtt{cons}(x, \mathtt{append}(l_1, l_2))\end{aligned}$$

Second-order:

$$\begin{aligned}\mathtt{map}(x.F(x), \mathtt{nil}) &\rightarrow \mathtt{nil} \\ \mathtt{map}(x.F(x), \mathtt{cons}(y, l)) &\rightarrow \mathtt{cons}(F(y), \mathtt{map}(x.F(x), l))\end{aligned}$$

First- and second-order term rewriting

First-order:

$$
\begin{aligned}
\mathtt{append}(\mathtt{nil}, l) &\rightarrow l \\
\mathtt{append}(\mathtt{cons}(x, l_1), l_2) &\rightarrow \mathtt{cons}(x, \mathtt{append}(l_1, l_2))
\end{aligned}
$$

Second-order:

$$
\begin{aligned}
\mathtt{map}(x.F(x), \mathtt{nil}) &\rightarrow \mathtt{nil} \\
\mathtt{map}(x.F(x), \mathtt{cons}(y, l)) &\rightarrow \mathtt{cons}(F(y), \mathtt{map}(x.F(x), l))
\end{aligned}
$$

Not permitted: x of function type!

Second-order Term Rewriting

$$
\begin{array}{rcl}
\mathtt{map}(x.F(x), \mathtt{nil}) & \to & \mathtt{nil} \\
\mathtt{map}(x.F(x), \mathtt{cons}(y, l)) & \to & \mathtt{cons}(F(y), \mathtt{map}(x.F(x), l)) \\
\mathtt{plus}(\mathtt{0}, y) & \to & y \\
\mathtt{plus}(\mathtt{s}(x), y) & \to & \mathtt{s}(\mathtt{plus}(x, y))
\end{array}
$$

Second-order Term Rewriting

$$\begin{array}{rcl}
\mathtt{map}(x.F(x), \mathtt{nil}) & \to & \mathtt{nil} \\
\mathtt{map}(x.F(x), \mathtt{cons}(y, l)) & \to & \mathtt{cons}(F(y), \mathtt{map}(x.F(x), l)) \\
\mathtt{plus}(\mathtt{0}, y) & \to & y \\
\mathtt{plus}(\mathtt{s}(x), y) & \to & \mathtt{s}(\mathtt{plus}(x, y))
\end{array}$$

Reduction:

$$\mathtt{map}(x.\mathtt{plus}(x, \mathtt{s}(x)), \mathtt{cons}(\mathtt{s}(z), \mathtt{cons}(\mathtt{0}, \mathtt{nil})))$$

Second-order Term Rewriting

$$\begin{array}{rcl}
\mathtt{map}(x.F(x), \mathtt{nil}) & \to & \mathtt{nil} \\
\mathtt{map}(x.F(x), \mathtt{cons}(y, l)) & \to & \mathtt{cons}(F(y), \mathtt{map}(x.F(x), l)) \\
\mathtt{plus}(\mathtt{0}, y) & \to & y \\
\mathtt{plus}(\mathtt{s}(x), y) & \to & \mathtt{s}(\mathtt{plus}(x, y))
\end{array}$$

Reduction:

$$\mathtt{map}(x.\mathtt{plus}(x, \mathtt{s}(x)), \mathtt{cons}(\mathtt{s}(z), \mathtt{cons}(\mathtt{0}, \mathtt{nil})))$$

Second-order Term Rewriting

$$\begin{array}{rcl}
\mathtt{map}(x.F(x), \mathtt{nil}) & \to & \mathtt{nil} \\
\mathtt{map}(x.F(x), \mathtt{cons}(y, l)) & \to & \mathtt{cons}(F(y), \mathtt{map}(x.F(x), l)) \\
\mathtt{plus}(\mathtt{0}, y) & \to & y \\
\mathtt{plus}(\mathtt{s}(x), y) & \to & \mathtt{s}(\mathtt{plus}(x, y))
\end{array}$$

Reduction:

$$\mathtt{map}(x.\mathtt{plus}(x, \mathtt{s}(x)), \mathtt{cons}(\mathtt{s}(z), \mathtt{cons}(\mathtt{0}, \mathtt{nil})))$$

$$\Rightarrow$$

$$\mathtt{cons}(\mathtt{plus}(\mathtt{s}(z), \mathtt{s}(\mathtt{s}(z))), \mathtt{map}(x.\mathtt{plus}(x, \mathtt{s}(x)), \mathtt{cons}(\mathtt{0}, \mathtt{nil})))$$

Second-order Term Rewriting

$$\begin{array}{rcl} \mathtt{map}(x.F(x), \mathtt{nil}) & \to & \mathtt{nil} \\ \mathtt{map}(x.F(x), \mathtt{cons}(y, l)) & \to & \mathtt{cons}(F(y), \mathtt{map}(x.F(x), l)) \\ \mathtt{plus}(\mathtt{0}, y) & \to & y \\ \mathtt{plus}(\mathtt{s}(x), y) & \to & \mathtt{s}(\mathtt{plus}(x, y)) \end{array}$$

Reduction:

$$\mathtt{cons}(\mathtt{plus}(\mathtt{s}(z), \mathtt{s}(\mathtt{s}(z))), \mathtt{map}(x.\mathtt{plus}(x, \mathtt{s}(x)), \mathtt{cons}(\mathtt{0}, \mathtt{nil})))$$

Second-order Term Rewriting

$$\begin{array}{rcl} \mathtt{map}(x.F(x), \mathtt{nil}) & \to & \mathtt{nil} \\ \mathtt{map}(x.F(x), \mathtt{cons}(y, l)) & \to & \mathtt{cons}(F(y), \mathtt{map}(x.F(x), l)) \\ \mathtt{plus}(\mathtt{0}, y) & \to & y \\ \mathtt{plus}(\mathtt{s}(x), y) & \to & \mathtt{s}(\mathtt{plus}(x, y)) \end{array}$$

Reduction:

$$\mathtt{cons}(\mathtt{plus}(\mathtt{s}(z), \mathtt{s}(\mathtt{s}(z))), \mathtt{map}(x.\mathtt{plus}(x, \mathtt{s}(x)), \mathtt{cons}(\mathtt{0}, \mathtt{nil})))$$

$$\Rightarrow$$

$$\mathtt{cons}(\mathtt{plus}(\mathtt{s}(z), \mathtt{s}(\mathtt{s}(z))), \mathtt{cons}(\mathtt{plus}(\mathtt{0}, \mathtt{s}(\mathtt{0})), \mathtt{map}(\ldots, \mathtt{nil})))$$

Attribute Mappings

Needed: things like:

Attribute Mappings

Needed: things like:

- $\texttt{e} = [x : \texttt{int}, y : \texttt{string}]$
- $\texttt{t} = \texttt{int}$
- $\texttt{jumps} = [3 : \texttt{some}(\texttt{code})]$

Attribute Mappings

Needed: things like:

- $\mathtt{e} = [x : \mathtt{int}, y : \mathtt{string}]$ (with $\mathtt{int}, \mathtt{string} : \mathtt{basictype}$)
- $\mathtt{t} = \mathtt{int}$ (of sort $\mathtt{basictype}$)
- $\mathtt{jumps} = [3 : \mathtt{some}(\mathtt{code})]$ (with value of sort $\mathtt{statement}$)

Attribute Mappings

Needed: things like:

- $\mathtt{e} = [x : \mathtt{int}, y : \mathtt{string}]$ (with $\mathtt{int}, \mathtt{string} : \mathtt{basictype}$)
- $\mathtt{t} = \mathtt{int}$ (of sort $\mathtt{basictype}$)
- $\mathtt{jumps} = [3 : \mathtt{some}(\mathtt{code})]$ (with value of sort $\mathtt{statement}$)

Attribute Declarations:

Attribute Mappings

Needed: things like:

- $\mathtt{e} = [x : \mathtt{int}, y : \mathtt{string}]$ (with $\mathtt{int}, \mathtt{string} : \mathtt{basictype}$)
- $\mathtt{t} = \mathtt{int}$ (of sort $\mathtt{basictype}$)
- $\mathtt{jumps} = [3 : \mathtt{some}(\mathtt{code})]$ (with value of sort $\mathtt{statement}$)

Attribute Declarations:

- $\mathtt{e} : \mathtt{expr} \Rightarrow \mathtt{basictype}$
- $\mathtt{jumps} : \mathtt{int} \Rightarrow \mathtt{statement}$

Attribute Mappings

Needed: things like:

- $\mathtt{e} = [x : \mathtt{int}, y : \mathtt{string}]$ (with $\mathtt{int}, \mathtt{string} : \mathtt{basictype}$)
- $\mathtt{t} = \mathtt{int}$ (of sort $\mathtt{basictype}$)
- $\mathtt{jumps} = [3 : \mathtt{some}(\mathtt{code})]$ (with value of sort $\mathtt{statement}$)

Attribute Declarations:

- $\mathtt{e} : \mathtt{expr} \Rightarrow \mathtt{basictype}$
- $\mathtt{jumps} : \mathtt{int} \Rightarrow \mathtt{statement}$
- $\mathtt{t} : \mathtt{unit} \Rightarrow \mathtt{basictype}$

Attribute Mappings

Needed: things like:

- $\mathtt{e} = [x : \mathtt{int}, y : \mathtt{string}]$ (with $\mathtt{int}, \mathtt{string} : \mathtt{basictype}$)
- $\mathtt{t} = \mathtt{int}$ (of sort $\mathtt{basictype}$)
- $\mathtt{jumps} = [3 : \mathtt{some}(\mathtt{code})]$ (with value of sort $\mathtt{statement}$)

Attribute Declarations:

- $\mathtt{e} : \mathtt{expr} \Rightarrow \mathtt{basictype}$
- $\mathtt{jumps} : \mathtt{int} \Rightarrow \mathtt{statement}$
- $\mathtt{t} : \mathtt{unit} \Rightarrow \mathtt{basictype}$

Attribute Maps:

Attribute Mappings

Needed: things like:

- $\mathtt{e} = [x : \mathtt{int}, y : \mathtt{string}]$ (with $\mathtt{int}, \mathtt{string} : \mathtt{basictype}$)
- $\mathtt{t} = \mathtt{int}$ (of sort $\mathtt{basictype}$)
- $\mathtt{jumps} = [3 : \mathtt{some}(\mathtt{code})]$ (with value of sort $\mathtt{statement}$)

Attribute Declarations:

- $\mathtt{e} : \mathtt{expr} \Rightarrow \mathtt{basictype}$
- $\mathtt{jumps} : \mathtt{int} \Rightarrow \mathtt{statement}$
- $\mathtt{t} : \mathtt{unit} \Rightarrow \mathtt{basictype}$

Attribute Maps:

- maps attribute names N with $N : A \Rightarrow B$ to mappings M with keys of sort A and values of sort B

Attribute Mappings

Needed: things like:

- $\mathtt{e} = [x : \mathtt{int}, y : \mathtt{string}]$ (with $\mathtt{int}, \mathtt{string} : \mathtt{basictype}$)
- $\mathtt{t} = \mathtt{int}$ (of sort $\mathtt{basictype}$)
- $\mathtt{jumps} = [3 : \mathtt{some}(\mathtt{code})]$ (with value of sort $\mathtt{statement}$)

Attribute Declarations:

- $\mathtt{e} : \mathtt{expr} \Rightarrow \mathtt{basictype}$
- $\mathtt{jumps} : \mathtt{int} \Rightarrow \mathtt{statement}$
- $\mathtt{t} : \mathtt{unit} \Rightarrow \mathtt{basictype}$

Attribute Maps:

- maps attribute names N with $N : A \Rightarrow B$ to mappings M with keys of sort A and values of sort B
- keys are variables or constants; values are terms

Attribute Mappings

Needed: things like:

- $\mathtt{e} = [x : \mathtt{int}, y : \mathtt{string}]$ (with $\mathtt{int}, \mathtt{string} : \mathtt{basictype}$)
- $\mathtt{t} = \mathtt{int}$ (of sort $\mathtt{basictype}$)
- $\mathtt{jumps} = [3 : \mathtt{some}(\mathtt{code})]$ (with value of sort $\mathtt{statement}$)

Attribute Declarations:

- $\mathtt{e} : \mathtt{expr} \Rightarrow \mathtt{basictype}$
- $\mathtt{jumps} : \mathtt{int} \Rightarrow \mathtt{statement}$
- $\mathtt{t} : \mathtt{unit} \Rightarrow \mathtt{basictype}$

Attribute Maps:

- maps attribute names N with $N : A \Rightarrow B$ to mappings M with keys of sort A and values of sort B
- keys are variables or constants; values are terms

$$[\mathtt{e} : \{x \mapsto \mathtt{int},\ y \mapsto \mathtt{String}\}, \mathtt{jumps} : \{3 : \mathtt{some}(\mathtt{code})\}]$$

Attributes and Sorts

Given: set of sorts

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols
  - $\mathit{syn}(\mathrm{S}) = \emptyset$ and $\mathit{syn}(\mathrm{E}) = \{\mathtt{t}\}$

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols
  - $\mathit{syn}(\mathrm{S}) = \emptyset$ and $\mathit{syn}(\mathrm{E}) = \{\mathtt{t}\}$
  - $+ : [\mathrm{E} \times \mathrm{E}] \Rightarrow \mathrm{E}^{\uparrow}$

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols
  - $syn(\mathrm{S}) = \emptyset$ and $syn(\mathrm{E}) = \{\mathtt{t}\}$
  - $+ : [\mathrm{E} \times \mathrm{E}] \Rightarrow \mathrm{E}^{\uparrow}$
  - if $s : K^{\uparrow}$ and $syn(K) = \{N_1, \ldots, N_k\}$, then $s \uparrow \{N_1 : \{\ldots\}, \ldots, N_k : \{\ldots\}\} : K$

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols
  - $syn(\mathrm{S}) = \emptyset$ and $syn(\mathrm{E}) = \{\mathtt{t}\}$
  - $+ : [\mathrm{E} \times \mathrm{E}] \Rightarrow \mathrm{E}^{\uparrow}$
  - if $s : K^{\uparrow}$ and $syn(K) = \{N_1, \ldots, N_k\}$, then $s \uparrow \{N_1 : \{\ldots\}, \ldots, N_k : \{\ldots\}\} : K$
  - example: $3 : \mathrm{E}^{\uparrow}$ but $3 \uparrow \{\mathtt{t} : \{\Box : \mathtt{int}\}\} : \mathrm{E}$

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols
  - $syn(\mathrm{S}) = \emptyset$ and $syn(\mathrm{E}) = \{\mathtt{t}\}$
  - $+ : [\mathrm{E} \times \mathrm{E}] \Rightarrow \mathrm{E}^{\uparrow}$
  - if $s : K^{\uparrow}$ and $syn(K) = \{N_1, \ldots, N_k\}$, then $s \uparrow \{N_1 : \{\ldots\}, \ldots, N_k : \{\ldots\}\} : K$
  - example: $3 : \mathrm{E}^{\uparrow}$ but $3 \uparrow \{\mathtt{t} : \{\Box : \mathtt{int}\}\} : \mathrm{E}$
- inherited sorts, associated with defined symbols

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols
  - $syn(\mathrm{S}) = \emptyset$ and $syn(\mathrm{E}) = \{\mathtt{t}\}$
  - $+ : [\mathrm{E} \times \mathrm{E}] \Rightarrow \mathrm{E}^{\uparrow}$
  - if $s : K^{\uparrow}$ and $syn(K) = \{N_1, \ldots, N_k\}$, then $s \uparrow \{N_1 : \{\ldots\}, \ldots, N_k : \{\ldots\}\} : K$
  - example: $3 : \mathrm{E}^{\uparrow}$ but $3 \uparrow \{\mathtt{t} : \{\square : \mathtt{int}\}\} : \mathrm{E}$
- inherited sorts, associated with defined symbols
  - $inh(\mathrm{S}) = \{\mathtt{e}\}$ and $inh(\mathrm{E}) = \{\mathtt{e}\}$

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols
  - $syn(\mathrm{S}) = \emptyset$ and $syn(\mathrm{E}) = \{\mathtt{t}\}$
  - $+ : [\mathrm{E} \times \mathrm{E}] \Rightarrow \mathrm{E}^{\uparrow}$
  - if $s : K^{\uparrow}$ and $syn(K) = \{N_1, \ldots, N_k\}$, then $s \uparrow \{N_1 : \{\ldots\}, \ldots, N_k : \{\ldots\}\} : K$
  - example: $3 : \mathrm{E}^{\uparrow}$ but $3 \uparrow \{\mathtt{t} : \{\Box : \mathtt{int}\}\} : \mathrm{E}$
- inherited sorts, associated with defined symbols
  - $inh(\mathrm{S}) = \{\mathtt{e}\}$ and $inh(\mathrm{E}) = \{\mathtt{e}\}$
  - $\mathrm{GS} : [\mathrm{S}] \Rightarrow \mathrm{S}^{\downarrow}$ and $\mathrm{GE} : [\mathrm{E}] \Rightarrow \mathrm{E}^{\downarrow}$

Attributes and Sorts

Given: set of sorts (e.g. S, E, Register, Basetype)

Associate: to each sort, two sets of attributes:

- synthesised sorts, associated with constructor symbols
  - $\mathit{syn}(\mathrm{S}) = \emptyset$ and $\mathit{syn}(\mathrm{E}) = \{\mathtt{t}\}$
  - $+ : [\mathrm{E} \times \mathrm{E}] \Rightarrow \mathrm{E}^{\uparrow}$
  - if $s : K^{\uparrow}$ and $\mathit{syn}(K) = \{N_1, \ldots, N_k\}$, then $s \uparrow \{N_1 : \{\ldots\}, \ldots, N_k : \{\ldots\}\} : K$
  - example: $3 : \mathrm{E}^{\uparrow}$ but $3 \uparrow \{\mathtt{t} : \{\square : \mathtt{int}\}\} : \mathrm{E}$
- inherited sorts, associated with defined symbols
  - $\mathit{inh}(\mathrm{S}) = \{\mathtt{e}\}$ and $\mathit{inh}(\mathrm{E}) = \{\mathtt{e}\}$
  - $\mathrm{GS} : [\mathrm{S}] \Rightarrow \mathrm{S}^{\downarrow}$ and $\mathrm{GE} : [\mathrm{E}] \Rightarrow \mathrm{E}^{\downarrow}$
  - example: $\mathrm{GE}(\mathrm{plus}(3 \uparrow \{\}, x) \uparrow \{\}) : \mathrm{E}^{\downarrow}$ but $\mathrm{GE}(\mathrm{plus}(3 \uparrow \{\}, x) \uparrow \{\}) \downarrow \{\mathtt{e} : \{x : \mathtt{Int}\}\} : \mathrm{E}$

Attributes in Rules

Needed:

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1} \rangle + \langle \mathrm{E\#2} \rangle]\!]{\downarrow}\mathrm{e} \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E\#1} \rangle]\!]{\downarrow}\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E\#2} \rangle]\!]{\downarrow}\mathrm{e}$

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified
  $[\![\mathrm{GE} \ \langle \mathrm{E\#1} \rangle \ + \ \langle \mathrm{E\#2} \rangle]\!] {\downarrow} \mathrm{e} \rightarrow [\![\mathrm{GE} \ \langle \mathrm{E\#1} \rangle]\!] {\downarrow} \mathrm{e} + [\![\mathrm{GE} \ \langle \mathrm{E\#2} \rangle]\!] {\downarrow} \mathrm{e}$
- testing whether a particular key is present

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified
  $[\![\mathrm{GE}\ \langle E\#1\rangle + \langle E\#2\rangle]\!]{\downarrow}e \rightarrow [\![\mathrm{GE}\ \langle E\#1\rangle]\!]{\downarrow}e + [\![\mathrm{GE}\ \langle E\#2\rangle]\!]{\downarrow}e$
- testing whether a particular key is present
  $[\![\mathrm{GE}\ \langle name\#\rangle]\!]{\downarrow}e\{\neg name\#\} \rightarrow \mathtt{error}$

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1} \rangle\ +\ \langle \mathrm{E\#2} \rangle]\!]{\downarrow}\mathrm{e} \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E\#1} \rangle]\!]{\downarrow}\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E\#2} \rangle]\!]{\downarrow}\mathrm{e}$
- testing whether a particular key is present
  $[\![\mathrm{GE}\ \langle \mathrm{name\#} \rangle]\!]{\downarrow}\mathrm{e}\{\neg \mathrm{name\#}\} \rightarrow \mathtt{error}$
- looking up the value associated to a given key

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle + \langle \mathrm{E\#2}\rangle]\!]{\downarrow}\mathrm{e} \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle]\!]{\downarrow}\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E\#2}\rangle]\!]{\downarrow}\mathrm{e}$
- testing whether a particular key is present
  $[\![\mathrm{GE}\ \langle \mathrm{name\#}\rangle]\!]{\downarrow}\mathrm{e}\{\neg\mathrm{name\#}\} \rightarrow \mathtt{error}$
- looking up the value associated to a given key
  $[\![\mathrm{GE}\ \langle \mathrm{name\#}\rangle]\!]{\downarrow}\mathrm{e}\{\mathrm{name\#{:}value\#}\}\ {\uparrow}\mathrm{t(value\#)}$

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1} \rangle + \langle \mathrm{E\#2} \rangle]\!]{\downarrow}\mathrm{e} \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E\#1} \rangle]\!]{\downarrow}\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E\#2} \rangle]\!]{\downarrow}\mathrm{e}$
- testing whether a particular key is present
  $[\![\mathrm{GE}\ \langle \mathrm{name\#} \rangle]\!]{\downarrow}\mathrm{e}\{\neg \mathrm{name\#}\} \rightarrow \mathtt{error}$
- looking up the value associated to a given key
  $[\![\mathrm{GE}\ \langle \mathrm{name\#} \rangle]\!]{\downarrow}\mathrm{e}\{\mathrm{name\#}{:}\mathrm{value\#}\}\ {\uparrow}\mathrm{t}(\mathrm{value\#})$
- adding key/value pairs to a given mapping

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified
  $[\![\mathrm{GE}\ \langle \mathrm{E}\#1\rangle\ +\ \langle \mathrm{E}\#2\rangle]\!]{\downarrow}\mathrm{e} \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E}\#1\rangle]\!]{\downarrow}\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E}\#2\rangle]\!]{\downarrow}\mathrm{e}$
- testing whether a particular key is present
  $[\![\mathrm{GE}\ \langle \mathrm{name}\#\rangle]\!]{\downarrow}\mathrm{e}\{\neg\mathrm{name}\#\} \rightarrow \mathtt{error}$
- looking up the value associated to a given key
  $[\![\mathrm{GE}\ \langle \mathrm{name}\#\rangle]\!]{\downarrow}\mathrm{e}\{\mathrm{name}\#\mathrm{:value}\#\}\ {\uparrow}\mathrm{t}(\mathrm{value}\#)$
- adding key/value pairs to a given mapping
  $[\![\langle \mathrm{Exp}\#1{\uparrow}\mathrm{t}(\#\mathrm{t1})\rangle\ +\ \langle \mathrm{Exp}\#2{\uparrow}\mathrm{t}(\#\mathrm{t2})\rangle]\!]\ {\uparrow}\mathrm{t}(\mathrm{Unif}(\#\mathrm{t1},\#\mathrm{t2}))$

Attributes in Rules

Needed:

- passing attribute mappings for given attributes on unmodified
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle + \langle \mathrm{E\#2}\rangle]\!]{\downarrow}\mathrm{e} \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle]\!]{\downarrow}\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E\#2}\rangle]\!]{\downarrow}\mathrm{e}$
- testing whether a particular key is present
  $[\![\mathrm{GE}\ \langle \mathrm{name\#}\rangle]\!]{\downarrow}\mathrm{e}\{\neg \mathrm{name\#}\} \rightarrow \mathtt{error}$
- looking up the value associated to a given key
  $[\![\mathrm{GE}\ \langle \mathrm{name\#}\rangle]\!]{\downarrow}\mathrm{e}\{\mathrm{name\#:value\#}\}\ {\uparrow}\mathrm{t(value\#)}$
- adding key/value pairs to a given mapping
  $[\![\langle \mathrm{Exp\#1}{\uparrow}\mathrm{t(\#t1)}\rangle + \langle \mathrm{Exp\#2}{\uparrow}\mathrm{t(\#t2)}\rangle]\!]$
  ${\uparrow}\mathrm{t}\{\square\mathrm{:Unif(\#t1,\#t2)}\}$

Attributes in Rules

Formally

In left-hand sides of rules:

Attributes in Rules

Formally

In left-hand sides of rules:

$$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[\mathrm{N}_1 : \{\mathrm{n}\#1 \mid \mathrm{x}_1 : \mathrm{val}\#1, \neg \mathrm{x}_2\}, \mathrm{N}_2 : \{\mathrm{n}\#2 \mid \neg\Box\}]$$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[\mathrm{N}_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, \mathrm{N}_2 : \{n\#2 \mid \neg\square\}]$$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\square\}]$$

In right-hand sides of rules:

Attributes in Rules

Formally

In left-hand sides of rules:

$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\square\}]$

In right-hand sides of rules:

$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\},$
$N_3 : \{x_2 : some(code)\}]$

Attributes in Rules

Formally

In left-hand sides of rules:

$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$

In right-hand sides of rules:

$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\},$
$N_3 : \{x_2 : some(code)\}]$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\square\}]$$

In right-hand sides of rules:

$$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\},$$
$$N_3 : \{x_2 : some(code)\}]$$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$$

In right-hand sides of rules:

$$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\},$$
$$N_3 \!: \{x_2 \!:\! some(code)\}]$$

Attributes in Rules

Formally

In left-hand sides of rules:

$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\square\}]$

In right-hand sides of rules:

$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\},$
$N_3 : \{x_2 : some(code)\}]$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$$

In right-hand sides of rules:

$$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\}, \\ N_3 : \{x_2 : some(code)\}]$$

Example:

Attributes in Rules

Formally

In left-hand sides of rules:

$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$

In right-hand sides of rules:

$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\},$
$N_3 : \{x_2 : some(code)\}]$

Example:

$[\![GE\ \langle name\#\rangle]\!]\ \downarrow e\{name\#:value\#\} \rightarrow$
$\langle name\#\rangle\ \uparrow t(value\#)$

Attributes in Rules

Formally

In left-hand sides of rules:

$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$

In right-hand sides of rules:

$[N_1 : \{n\#1 + x_2 : \mathrm{some(code)}\}, N_2 : \{n\#3\},$
$N_3\!: \{x_2\!:\!\mathrm{some(code)}\}]$

Example:

$\mathrm{GE(name\#)}{\downarrow}\mathrm{e}\{\mathrm{name\#\!:\!value\#}\} \rightarrow$
$\mathrm{name\#}{\uparrow}\mathrm{t(value\#)}$

Attributes in Rules

Formally

In left-hand sides of rules:

$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$

In right-hand sides of rules:

$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\},$
$N_3 \text{: } \{x_2 \text{:} some(code)\}]$

Example:

$GE(name\#){\downarrow}e\{name\#:value\#\} \rightarrow$
$name\#{\uparrow}t(value\#)$

Attributes in Rules

Formally

In left-hand sides of rules:

$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\square\}]$

In right-hand sides of rules:

$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\},$
$N_3 : \{x_2 : some(code)\}]$

Example:

$GE(name\#) \downarrow [e : \{emap\# \mid name\# : value\#\}] \rightarrow$
$name\# \uparrow t(value\#)$

Attributes in Rules

Formally

In left-hand sides of rules:

$$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$$

In right-hand sides of rules:

$$[N_1 : \{n\#1 + x_2 : some(code)\}, N_2 : \{n\#3\}, N_3\!: \{x_2\!:\!some(code)\}]$$

Example:

$$GE(name\#)\!\downarrow [e : \{emap\# \mid name\#\!:\!value\#\}] \rightarrow name\#{\uparrow}t(value\#)$$

Attributes in Rules

Formally

In left-hand sides of rules:

$[N_1 : \{n\#1 \mid x_1 : val\#1, \neg x_2\}, N_2 : \{n\#2 \mid \neg\Box\}]$

In right-hand sides of rules:

$[N_1 : \{n\#1 + x_2 : \text{some(code)}\}, N_2 : \{n\#3\},$
$N_3 : \{x_2 : \text{some(code)}\}]$

Example:

$\text{GE(name\#)}\downarrow [e : \{emap\# \mid name\#:value\#\}] \rightarrow$
$\text{name\#}\uparrow [t : \{\Box:value\#\}]$

Notes on the Translation

Notes on the Translation

- first inherited attribute notations are added everywhere

Notes on the Translation

- first inherited attribute notations are added everywhere
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle + \langle \mathrm{E\#2}\rangle]\!] \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle]\!] + [\![\mathrm{GE}\ \langle \mathrm{E\#2}\rangle]\!]$

Notes on the Translation

- first inherited attribute notations are added everywhere
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle + \langle \mathrm{E\#2}\rangle]\!]{\downarrow}\mathrm{e} \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle]\!]{\downarrow}\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E\#2}\rangle]\!]{\downarrow}\mathrm{e}$

Notes on the Translation

- first inherited attribute notations are added everywhere
  $\llbracket \text{GE } \langle \text{E\#1} \rangle + \langle \text{E\#2} \rangle \rrbracket{\downarrow}\text{e} \rightarrow \llbracket \text{GE } \langle \text{E\#1} \rangle \rrbracket{\downarrow}\text{e} + \llbracket \text{GE } \langle \text{E\#2} \rangle \rrbracket{\downarrow}\text{e}$
- synthesis rules are associated to a defined symbol

Notes on the Translation

- first inherited attribute notations are added everywhere
  $[\![\text{GE } \langle\text{E\#1}\rangle + \langle\text{E\#2}\rangle]\!]{\downarrow}\text{e} \rightarrow [\![\text{GE } \langle\text{E\#1}\rangle]\!]{\downarrow}\text{e} + [\![\text{GE } \langle\text{E\#2}\rangle]\!]{\downarrow}\text{e}$
- synthesis rules are associated to a defined symbol
  $[\![\langle\text{E\#1}{\uparrow}\text{t(\#t1)}\rangle + \langle\text{E\#2}{\uparrow}\text{t(\#t2)}\rangle]\!]\ {\uparrow}\text{t(Unif(\#t1,\#t2));}$

Notes on the Translation

- first inherited attribute notations are added everywhere
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle + \langle \mathrm{E\#2}\rangle]\!]\downarrow\mathrm{e} \rightarrow [\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle]\!]\downarrow\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E\#2}\rangle]\!]\downarrow\mathrm{e}$
- synthesis rules are associated to a defined symbol
  $\mathrm{SynE}(\langle \mathrm{E\#1}\uparrow\mathrm{t}(\mathrm{\#t1})\rangle + \langle \mathrm{E\#2}\uparrow\mathrm{t}(\mathrm{\#t2})\rangle) \rightarrow$
  $(\langle \mathrm{E\#1}\uparrow\mathrm{t}(\mathrm{\#t1})\rangle + \langle \mathrm{E\#2}\uparrow\mathrm{t}(\mathrm{\#t2})\rangle)\uparrow\mathrm{t}(\mathrm{Unif}(\mathrm{\#t1},\mathrm{\#t2}));$

Notes on the Translation

- first inherited attribute notations are added everywhere
  $[\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle + \langle \mathrm{E\#2}\rangle]\!]\downarrow\mathrm{e} \to [\![\mathrm{GE}\ \langle \mathrm{E\#1}\rangle]\!]\downarrow\mathrm{e} + [\![\mathrm{GE}\ \langle \mathrm{E\#2}\rangle]\!]\downarrow\mathrm{e}$
- synthesis rules are associated to a defined symbol
  $\mathrm{SynE}(\langle \mathrm{E\#1}\uparrow\mathrm{t(\#t1)}\rangle + \langle \mathrm{E\#2}\uparrow\mathrm{t(\#t2)}\rangle) \to$
  $(\langle \mathrm{E\#1}\uparrow\mathrm{t(\#t1)}\rangle + \langle \mathrm{E\#2}\uparrow\mathrm{t(\#t2)}\rangle)\uparrow\mathrm{t(Unif(\#t1,\#t2))};$
  (plus rules to put synE in place!)

Notes on the Translation

- first inherited attribute notations are added everywhere
  $[\![\text{GE } \langle\text{E\#1}\rangle + \langle\text{E\#2}\rangle]\!]\downarrow\text{e} \rightarrow [\![\text{GE } \langle\text{E\#1}\rangle]\!]\downarrow\text{e} + [\![\text{GE } \langle\text{E\#2}\rangle]\!]\downarrow\text{e}$
- synthesis rules are associated to a defined symbol
  $\text{SynE}(\langle\text{E\#1}\uparrow\text{t(\#t1)}\rangle + \langle\text{E\#2}\uparrow\text{t(\#t2)}\rangle) \rightarrow$
  $(\langle\text{E\#1}\uparrow\text{t(\#t1)}\rangle + \langle\text{E\#2}\uparrow\text{t(\#t2)}\rangle)\uparrow\text{t(Unif(\#t1,\#t2))};$
  (plus rules to put synE in place!)
- attribute matchings and copies are made explicit (as before)

Notes on the Translation

- first inherited attribute notations are added everywhere
  $[\![\text{GE } \langle \text{E\#1} \rangle + \langle \text{E\#2} \rangle]\!]\downarrow\text{e} \rightarrow [\![\text{GE } \langle \text{E\#1} \rangle]\!]\downarrow\text{e} + [\![\text{GE } \langle \text{E\#2} \rangle]\!]\downarrow\text{e}$
- synthesis rules are associated to a defined symbol
  $\text{SynE}(\langle \text{E\#1}\uparrow\text{t(\#t1)} \rangle + \langle \text{E\#2}\uparrow\text{t(\#t2)} \rangle) \rightarrow$
  $(\langle \text{E\#1}\uparrow\text{t(\#t1)} \rangle + \langle \text{E\#2}\uparrow\text{t(\#t2)} \rangle)\uparrow\text{t(Unif(\#t1,\#t2))};$
  (plus rules to put synE in place!)
- attribute matchings and copies are made explicit (as before)
- terms brought into term shapes

Overview

1. Compilers
2. Attribute Grammars
3. HACS by Example
4. Core HACS
5. Conclusions

Relevant questions

Relevant questions

- termination

Relevant questions

- termination $\checkmark$

Relevant questions

- termination ✓
- confluence

Relevant questions

- termination $\checkmark$
- confluence $\checkmark$

Relevant questions

- termination ✓
- confluence ✓
- quasi-reductivity (No Cookies)

Relevant questions

- termination ✓
- confluence ✓
- quasi-reductivity (No Cookies) ✓

Relevant questions

- termination ✓
- confluence ✓
- quasi-reductivity (No Cookies) ✓

  note: trickier than it may seem

$$
\begin{array}{rcl}
\mathtt{g}(\mathtt{0}) & \to & \mathtt{0} \\
\mathtt{g}(\mathtt{s}(\#Z)) & \to & \#Z \\
\mathtt{f}(x.\mathtt{0}) & \to & \mathtt{0} \\
\mathtt{f}(x.\mathtt{s}(\#F(x))) & \to & \mathtt{s}(\mathtt{0}) \\
\mathtt{f}(x.x) & \to & \mathtt{s}(\mathtt{s}(\mathtt{0}))
\end{array}
$$

Relevant questions

- termination ✓
- confluence ✓
- quasi-reductivity (No Cookies) ✓

  note: trickier than it may seem

$$\begin{array}{rcl}
\mathtt{g}(\mathtt{0}) & \to & \mathtt{0} \\
\mathtt{g}(\mathtt{s}(\#Z)) & \to & \#Z \\
\mathtt{f}(x.\mathtt{0}) & \to & \mathtt{0} \\
\mathtt{f}(x.\mathtt{s}(\#F(x))) & \to & \mathtt{s}(\mathtt{0}) \\
\mathtt{f}(x.x) & \to & \mathtt{s}(\mathtt{s}(\mathtt{0}))
\end{array}$$

Not quasi-reductive: $\mathtt{f}(x.\mathtt{g}(x))$ is blocked

Relevant questions

- termination ✓
- confluence ✓
- quasi-reductivity (No Cookies) ✓
  note: trickier than it may seem

$$\begin{array}{rcl} \mathtt{g}(\mathtt{0}) & \to & \mathtt{0} \\ \mathtt{g}(\mathtt{s}(\#Z)) & \to & \#Z \\ \mathtt{f}(x.\mathtt{0}) & \to & \mathtt{0} \\ \mathtt{f}(x.\mathtt{s}(\#F(x))) & \to & \mathtt{s}(\mathtt{0}) \\ \mathtt{f}(x.x) & \to & \mathtt{s}(\mathtt{s}(\mathtt{0})) \end{array}$$

  Not quasi-reductive: $\mathtt{f}(x.\mathtt{g}(x))$ is blocked
- complexity

Relevant questions

- termination ✓
- confluence ✓
- quasi-reductivity (No Cookies) ✓

  note: trickier than it may seem

$$\begin{array}{rcl} \mathtt{g}(\mathtt{0}) & \to & \mathtt{0} \\ \mathtt{g}(\mathtt{s}(\#Z)) & \to & \#Z \\ \mathtt{f}(x.\mathtt{0}) & \to & \mathtt{0} \\ \mathtt{f}(x.\mathtt{s}(\#F(x))) & \to & \mathtt{s}(\mathtt{0}) \\ \mathtt{f}(x.x) & \to & \mathtt{s}(\mathtt{s}(\mathtt{0})) \end{array}$$

  Not quasi-reductive: $\mathtt{f}(x.\mathtt{g}(x))$ is blocked
- complexity ✓

Relevant questions

- termination ✓
- confluence ✓
- quasi-reductivity (No Cookies) ✓

  note: trickier than it may seem

$$
\begin{array}{rcl}
\mathtt{g}(\mathtt{0}) & \to & \mathtt{0} \\
\mathtt{g}(\mathtt{s}(\#Z)) & \to & \#Z \\
\mathtt{f}(x.\mathtt{0}) & \to & \mathtt{0} \\
\mathtt{f}(x.\mathtt{s}(\#F(x))) & \to & \mathtt{s}(\mathtt{0}) \\
\mathtt{f}(x.x) & \to & \mathtt{s}(\mathtt{s}(\mathtt{0}))
\end{array}
$$

  Not quasi-reductive: $\mathtt{f}(x.\mathtt{g}(x))$ is blocked

- complexity ✓

All the common questions!

Relevant questions

- termination ✓
- confluence ✓
- quasi-reductivity (No Cookies) ✓

  note: trickier than it may seem

$$
\begin{array}{rcl}
\mathtt{g}(\mathtt{0}) & \to & \mathtt{0} \\
\mathtt{g}(\mathtt{s}(\#Z)) & \to & \#Z \\
\mathtt{f}(x.\mathtt{0}) & \to & \mathtt{0} \\
\mathtt{f}(x.\mathtt{s}(\#F(x))) & \to & \mathtt{s}(\mathtt{0}) \\
\mathtt{f}(x.x) & \to & \mathtt{s}(\mathtt{s}(\mathtt{0}))
\end{array}
$$

  Not quasi-reductive: $\mathtt{f}(x.\mathtt{g}(x))$ is blocked
- complexity ✓

All the common questions! (In almost pure form!)

Summary

- HACS user system: simple way to write compilers

Summary

- HACS user system: simple way to write compilers
- HACS core: compiler specified as a second-order TRS

Summary

- HACS user system: simple way to write compilers
- HACS core: compiler specified as a second-order TRS
- Potential: extending existing analysis techniques for TRSs!

Summary

- HACS user system: simple way to write compilers
- HACS core: compiler specified as a second-order TRS
- Potential: extending existing analysis techniques for TRSs!

Questions?