Fun with Attribute Grammars

Arie Middelkoop

April 8, 2010

Outline

- Attribute Grammars (AGs)
  - Theory: specification of semantics of programming languages
  - Practice: tree walks that decorate ASTs
- Motivation: Utrecht Haskell Compiler (UHC)
  - Makes heavy use of AGs
  - Problem: some tree walks cannot be described
- Use case: unification
  - Case distinction on multiple trees
  - (Order of evaluation concerning substitution)
- Visit Functions
  - AGs are a front-end for visit functions
  - This talk: another front-end for visit functions
  - (Static constraints on the life-time of attributes)

Sometimes programming is like...

But you rather want it like...

A-team sound here

Motivation

Utrecht Haskell Compiler

- Compiler for a complex language
- Goal: platform for experimentation with language features

- Architecture: long pipeline of transformations
- Each transformation step: analysis + actual transformation
- Implemented with (higher-order) attribute grammars

- Computation of attributes in terms of other attributes
  - Easy to add new attributes
  - Automatic ordering of computations
  - Separate specification
- Automatic generation of trivial computations
  - Code more robust against changes in the AST structure
- Sanity checks (e.g. cycle check)

Challenge

- Question: can we stretch AG formalism to cover an even larger part of the code base?
- Why: more concise code

Goal:

- Implement unification with AGs
- Challenge: AGs unsuitable to express simultaneous traversals over multiple ASTs
- Solution: decorate a virtual tree that represents the traversal itself

Unification with Type Rules

$$Ty\_Con\ s1 \equiv Ty\_Con\ s1\ \ \&\ \emptyset\ \textsc{m.con}$$

$$\frac{\tau_1 \equiv \tau_3\ \ \&\ e_1 \qquad \tau_2 \equiv \tau_4\ \ \&\ e_2}{Ty\_App\ \tau_1\ \tau_2 \equiv Ty\_App\ \tau_3\ \tau_4\ \ \&\ (e_1 +\!\!+ e_2)}\ \textsc{m.app}$$

$$\tau_1 \equiv \tau_2\ \ \&\ \{Err\_UnifyClash\}\ \textsc{m.clash}$$

Code starts like this: EH1, full unification

```
fitsIn :: Ty → Ty → FIOut
fitsIn τ1 τ2
  =  f τ1 τ2
  where
    res t                  = emptyFO {foTy = t}
    f Ty_Any t2            = res t2                                  -- m.any.l
    f t1 Ty_Any            = res t1                                  -- m.any.r
    f t1@(Ty_Con s1)                                                 -- m.con
      t2@(Ty_Con s2)
        | s1 ≡ s2          = res t2
    f t1@(Ty_App (Ty_App (Ty_Con c1) ta1) tr1)                       -- m.arrow
      t2@(Ty_App (Ty_App (Ty_Con c2) ta2) tr2)
        | hsnIsArrow c1 ∧ c1 ≡ c2
        = comp ta2 tr1 ta1 tr2 (λa r → [a] `mkArrow` r)
    f t1@(Ty_App tf1 ta1)                                            -- m.prod
      t2@(Ty_App tf2 ta2)
        = comp tf1 ta1 tf2 ta2 Ty_App
    f t1 t2                = err [Err_UnifyClash τ1 τ2 t1 t2]
    err e                  = emptyFO {foErrL = e}
    comp tf1 ta1 tf2 ta2 mkComp
        = foldr1 (λfo1 fo2 → if foHasErrs fo1 then fo1 else fo2)
                 [ffo, afo, res rt]
```

Before you know it... EH99, zoomed in on app-case

```
f fi t1@(Ty_App (Ty_App (Ty_Con c1) tpr1) tr1)
    t2@(Ty_App (Ty_App (Ty_Con c2) tpr2) tr2)
      | hsnIsArrow c1 ∧ c1 ≡ c2 ∧ ¬ (fioPredAsTy (fiFIOpts fi)) ∧ isJust mbfp
  = fromJust mbfp
  where (u', u1, u2, u3) = mkNewLevUID3 (fiUniq fi)
        prfPredScope     = fePredScope (fiEnv fi)
        mbfp             = fVarPred2 fP (fi {fiUniq = u'}) tpr1 tpr2
        mberr            = Just (errClash fi t1 t2)
        fP fi tpr1@(Ty_Pred _)   tpr2@(Ty_Pred _)
          = if foHasErrs pfo
            then Nothing
            else Just (foUpdTy ([foTy pfo] `mkArrow` foTy fo)
                       $ foUpdLRCoe (mkIdLRCoeWith n (CMetaVal_Dict Nothing))
                       $ foUpdLRTCoe (C.mkIdLRCoeWith n (C.MetaVal_Dict Nothi
                       $ fo)
          where pfo = fVar f (fi {fiFIOpts = predFIOpts}) tpr2 tpr1
                n   = uidHNm u2
                fo  = fVar ff (fofi pfo fi) tr1 tr2
        fP fi tpr1@(Ty_Pred pr1) (Ty_Impls (Impls_Tail iv2 ipo2))
          = Just (foUpdImplExplCoe iv2
                  (Impls_Cons iv2 pr1 (mkPrIdCHR u2) range ipo2 im2)
                  tpr1
                  (mkIdLRCoeWith n (CMetaVal_Dict Nothing))
                  (C.mkIdLRCoeWith n (C.MetaVal_Dict Nothing) (C.tyErr ("fitsIn.f
                  fo)
          where im2 = Impls_Tail u1 ipo2
```

Could we describe it as...

$$\dfrac{\mathit{Int} \equiv \mathit{Int} \ \& \emptyset \qquad \dfrac{\mathit{Bool} \equiv \mathit{Bool} \ \& \emptyset \qquad \mathit{Char} \equiv \mathit{Char} \ \& \emptyset}{\mathit{Bool} \to \mathit{Char} \equiv \mathit{Bool} \to \mathit{Char} \ \& \emptyset}}{\mathit{Int} \to \mathit{Bool} \to \mathit{Char} \equiv \mathit{Int} \to \mathit{Bool} \to \mathit{Char} \ \& \emptyset}$$

Could we describe it as...

$$\dfrac{\mathit{Int} \equiv \mathit{Int} \ \& \emptyset \qquad \dfrac{\mathit{Bool} \equiv \mathit{Bool} \ \& \emptyset \qquad \mathit{Char} \equiv \mathit{Char} \ \& \emptyset}{\mathit{Bool} \to \mathit{Char} \equiv \mathit{Bool} \to \mathit{Char} \ \& \emptyset}}{\mathit{Int} \to \mathit{Bool} \to \mathit{Char} \equiv \mathit{Int} \to \mathit{Bool} \to \mathit{Char} \ \& \emptyset}$$

- Dynamic construction of derivation tree
- Attribution of derivation tree

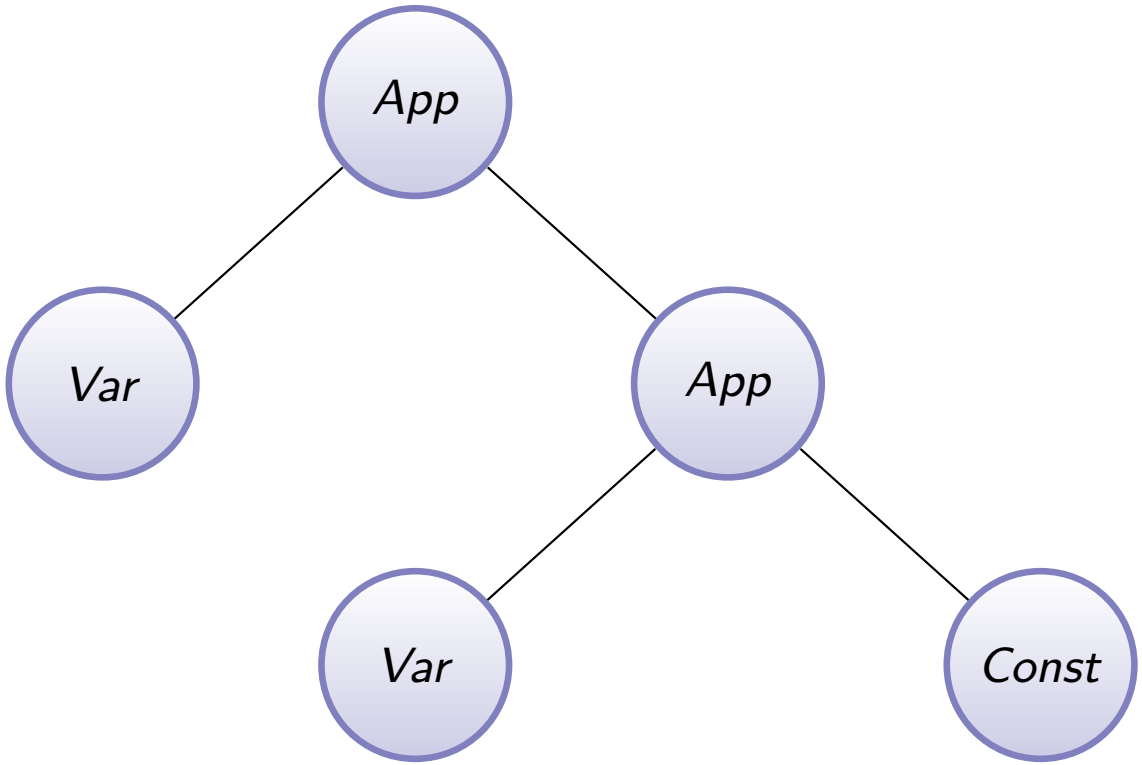
Background

Attribute Grammars

- Context-free grammar
  - Nonterminals
  - Productions
- Attributes
  - Parameters of nonterminals
  - Inherited attributes
  - Synthesized attributes
- Rules
  - Equations between attributes per production

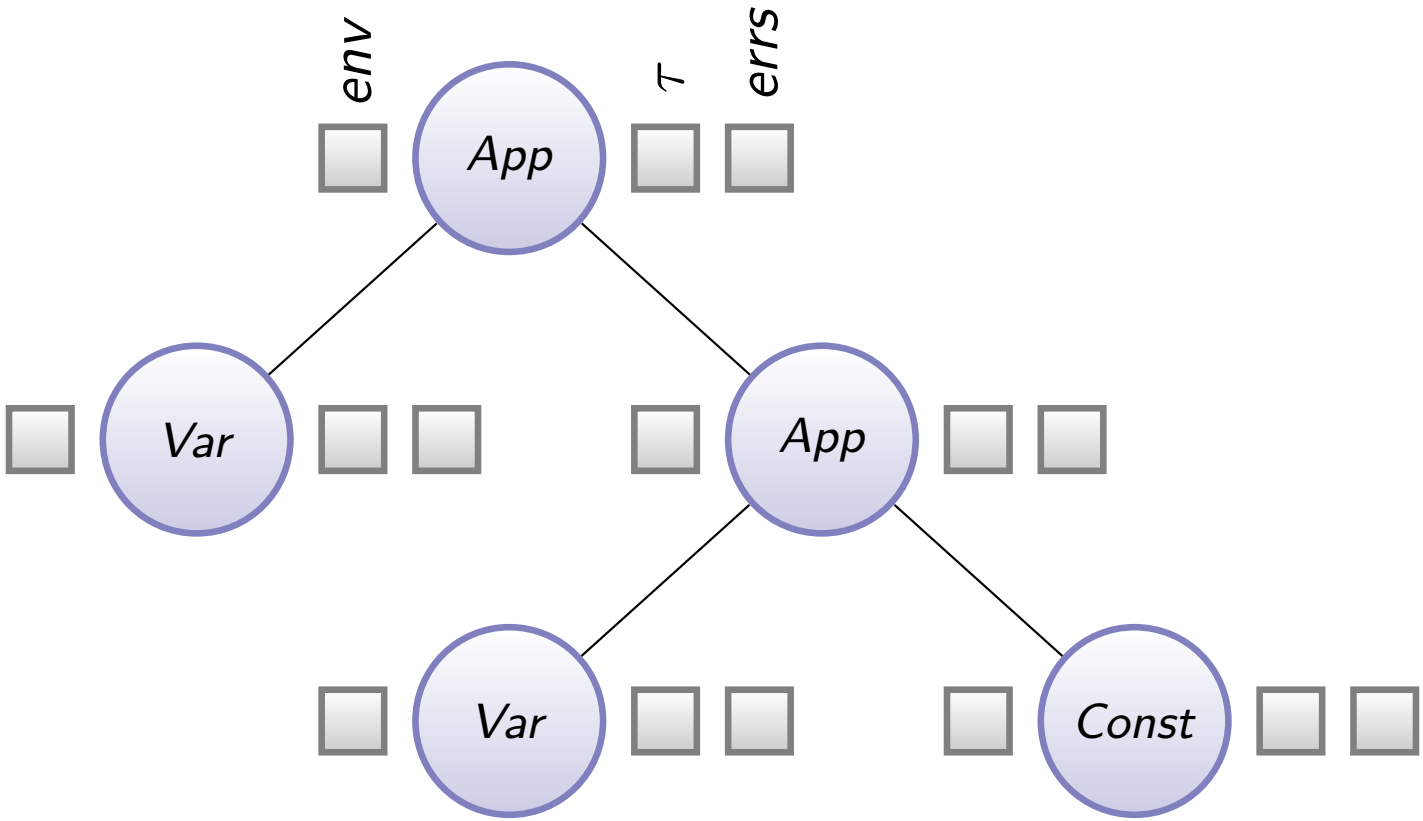
Traversing AST

- We use AGs to generate a traversal on an AST



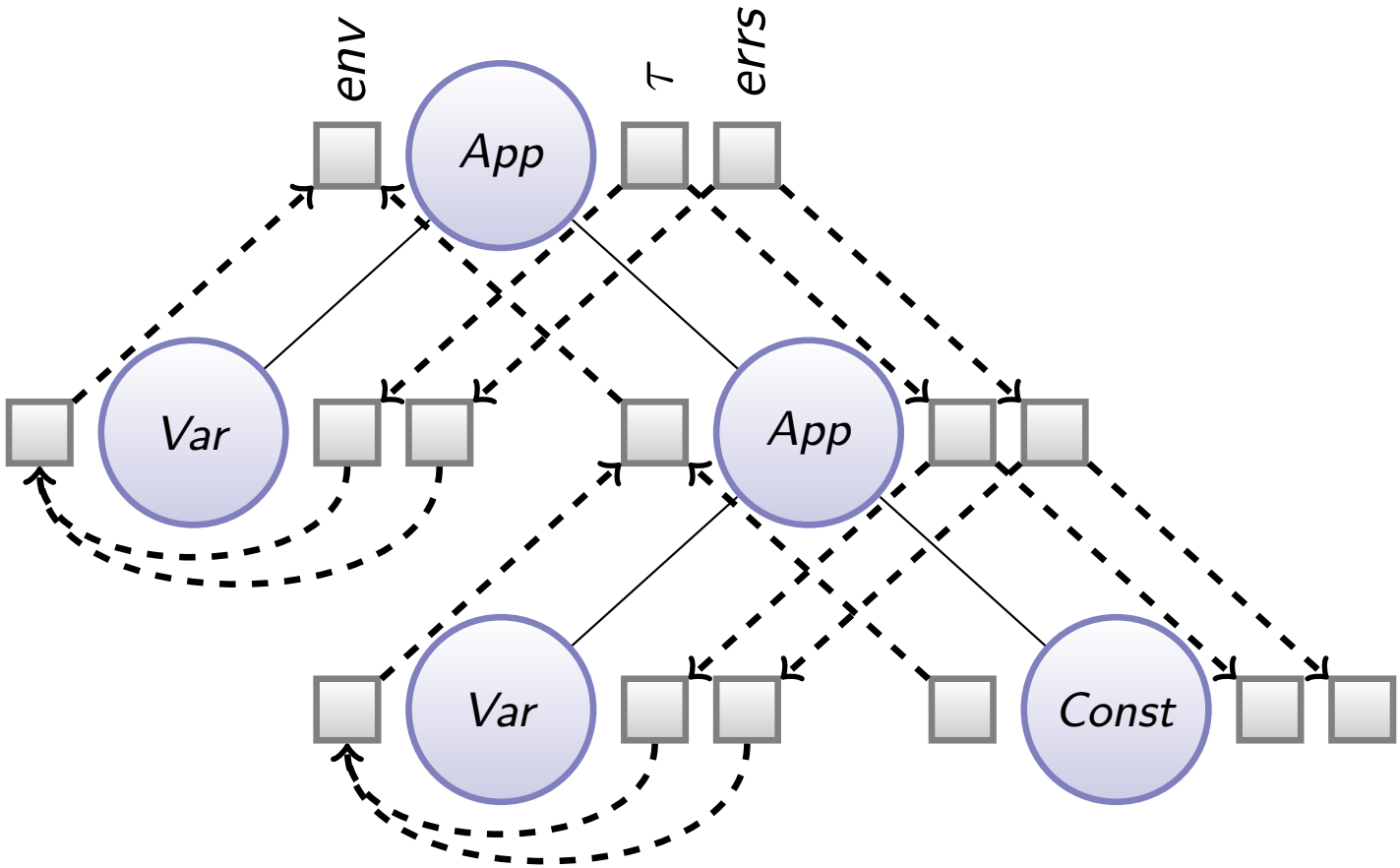
Traversing AST

- We use AGs to generate a traversal on an AST



Traversing AST

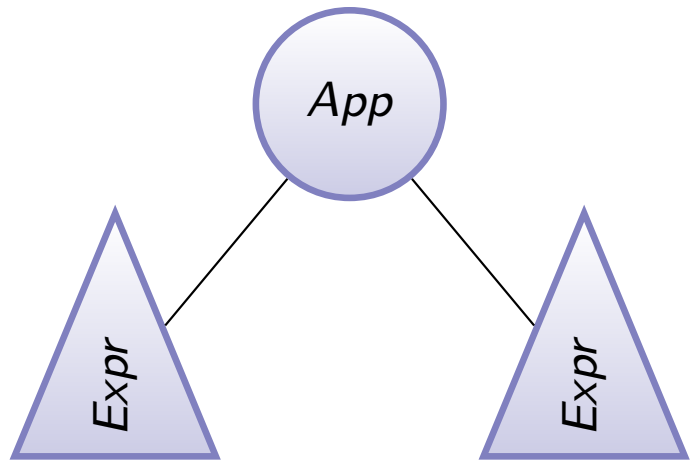
- We use AGs to generate a traversal on an AST



AG Programming Model

- Focus on AST instead of grammar.
- AG describes algorithm that decorates AST.

data *Expr*
  | *App* *fun* :: *Expr* *arg* :: *Expr*
  | *Var* *nm* :: *Ident*

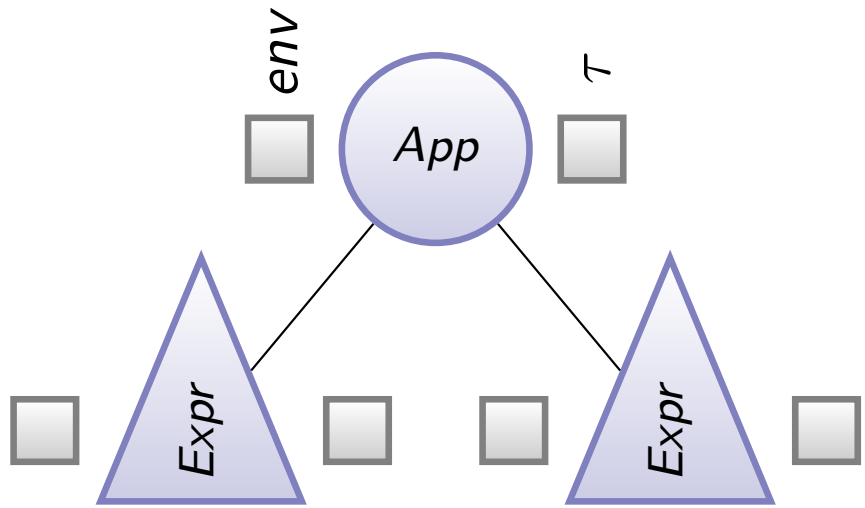


AG Programming Model

- Focus on AST instead of grammar.
- AG describes algorithm that decorates AST.

data *Expr*
  | *App* *fun* :: *Expr* *arg* :: *Expr*
  | *Var* *nm* :: *Ident*

attr *Expr* **inh** *env* :: *Env*
  syn τ :: *Ty*



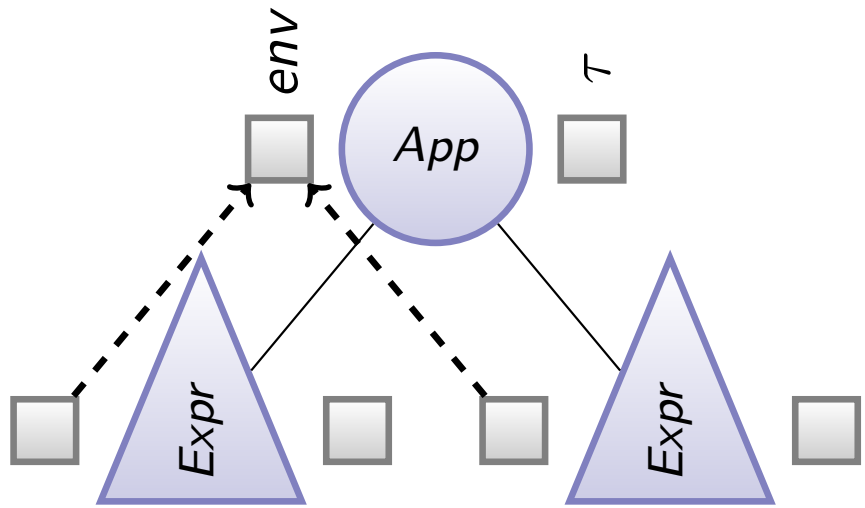
AG Programming Model

- Focus on AST instead of grammar.
- AG describes algorithm that decorates AST.

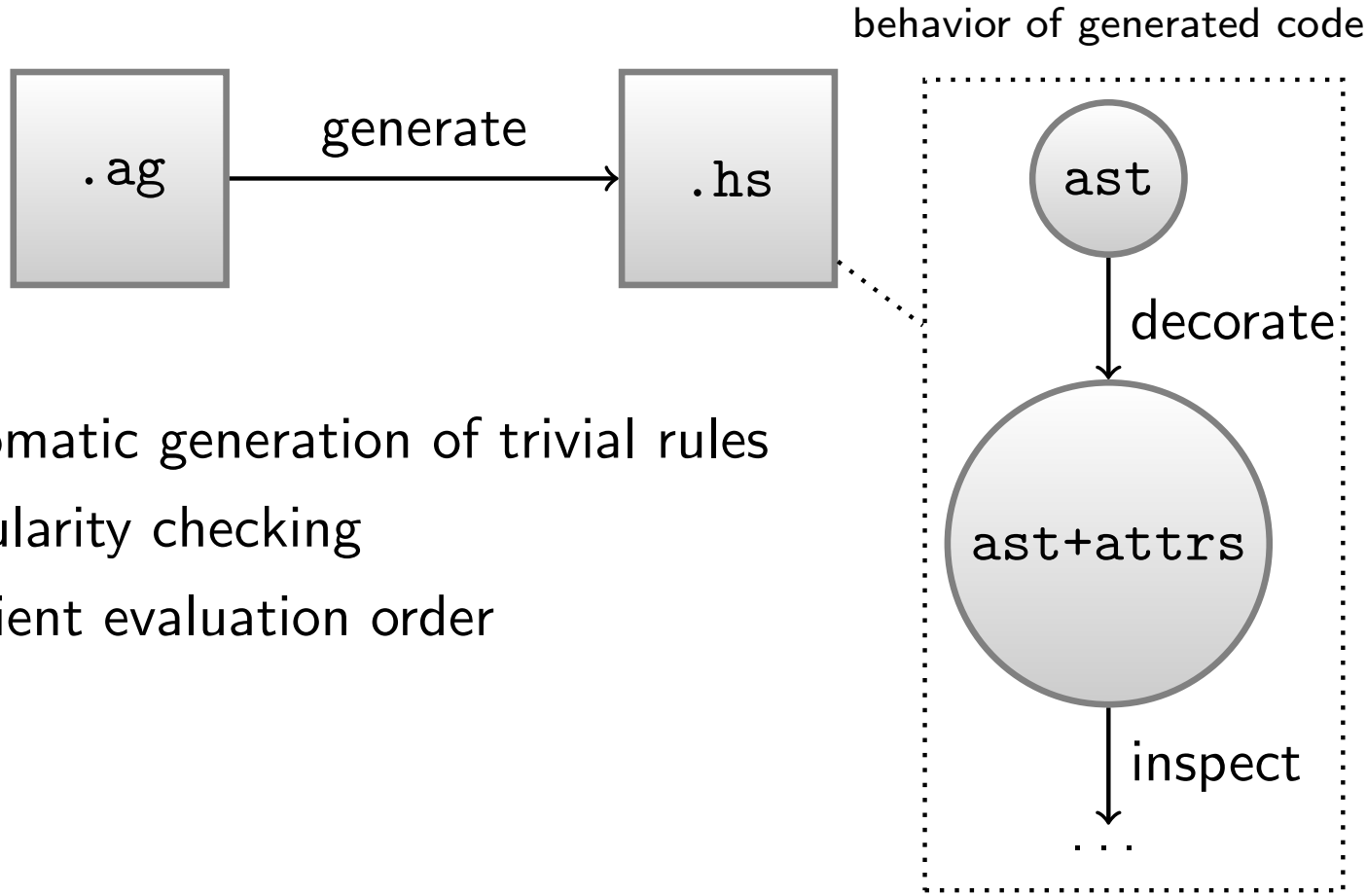
```
data Expr
   | App  fun :: Expr arg :: Expr
   | Var  nm :: Ident

attr Expr inh env :: Env
          syn τ   :: Ty

sem Expr
   | App  fun.env = this.env
          arg.env = this.env
   | Var  this.τ   = lookup loc.nm this.env
```



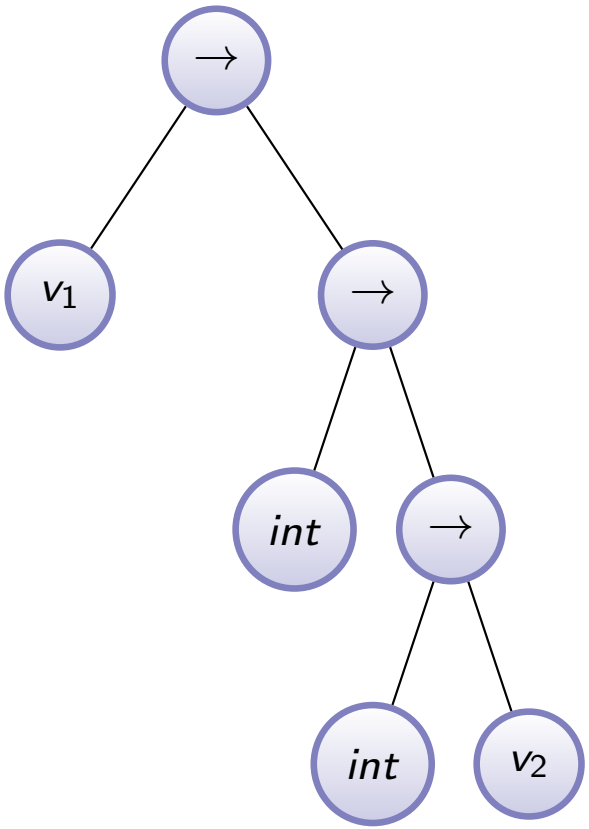
Compilation

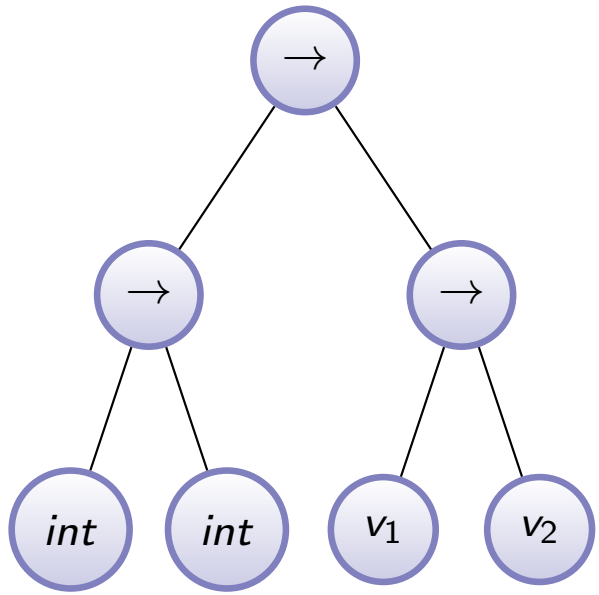


- Automatic generation of trivial rules
- Circularity checking
- Efficient evaluation order

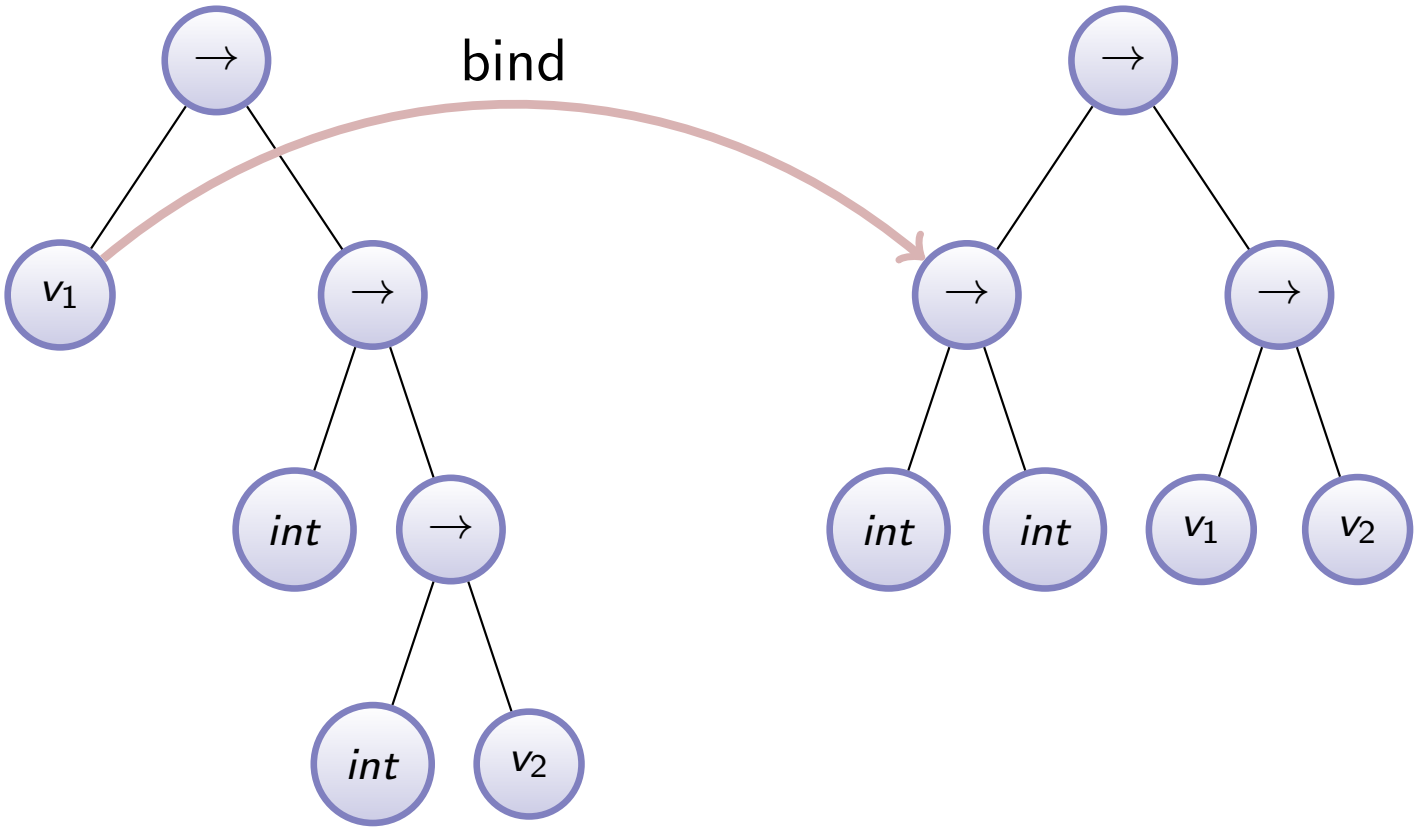
Problem

Unification

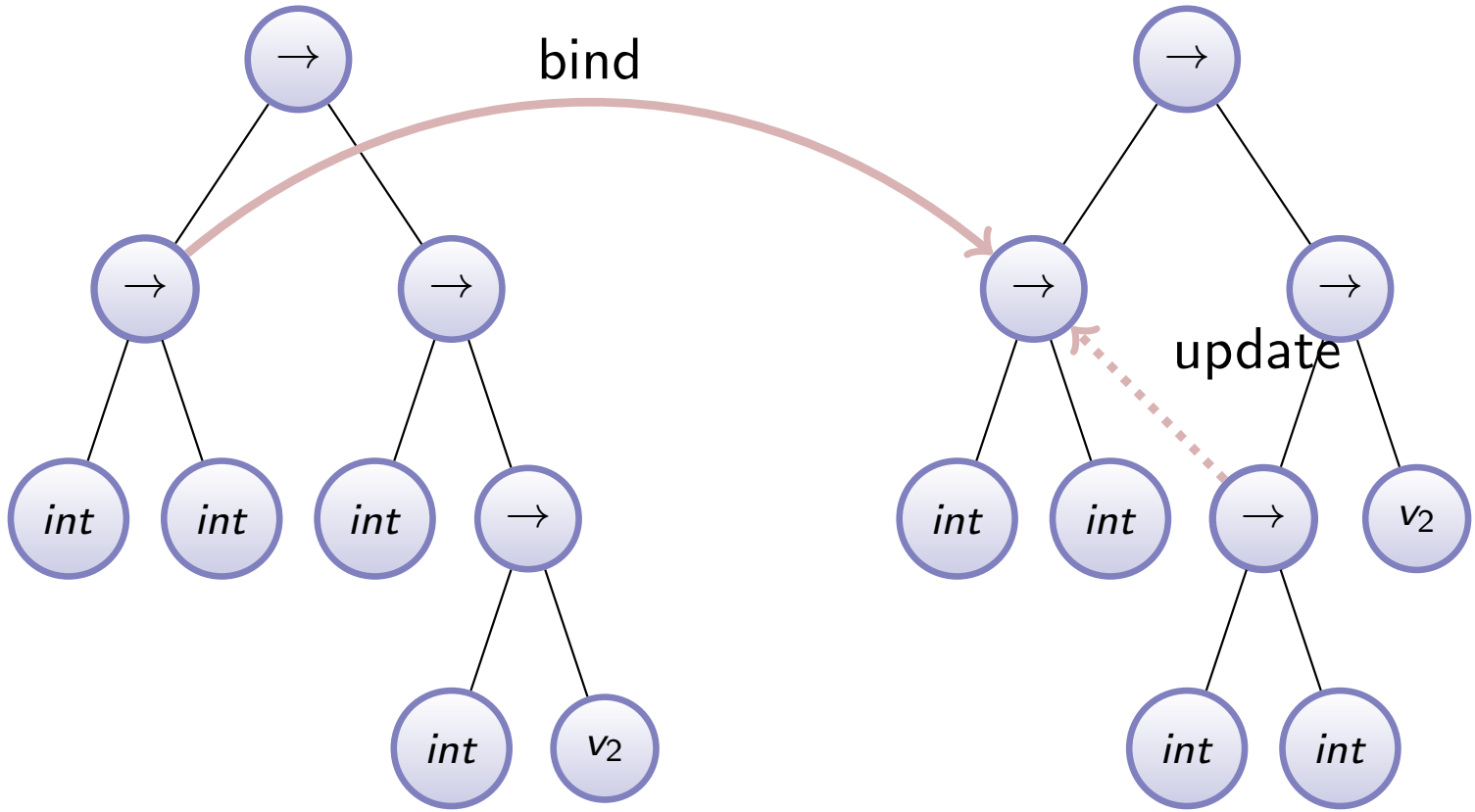




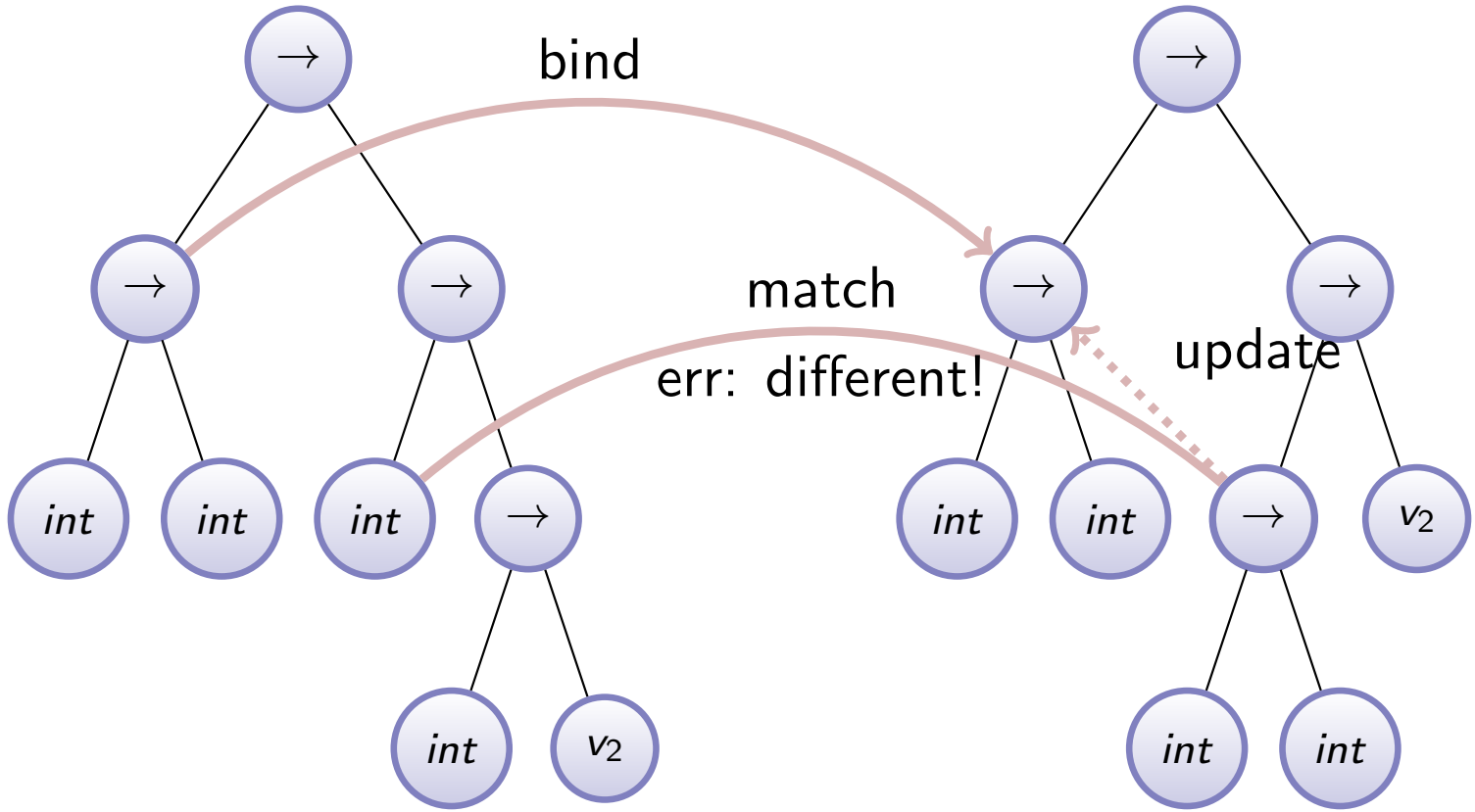
Unification



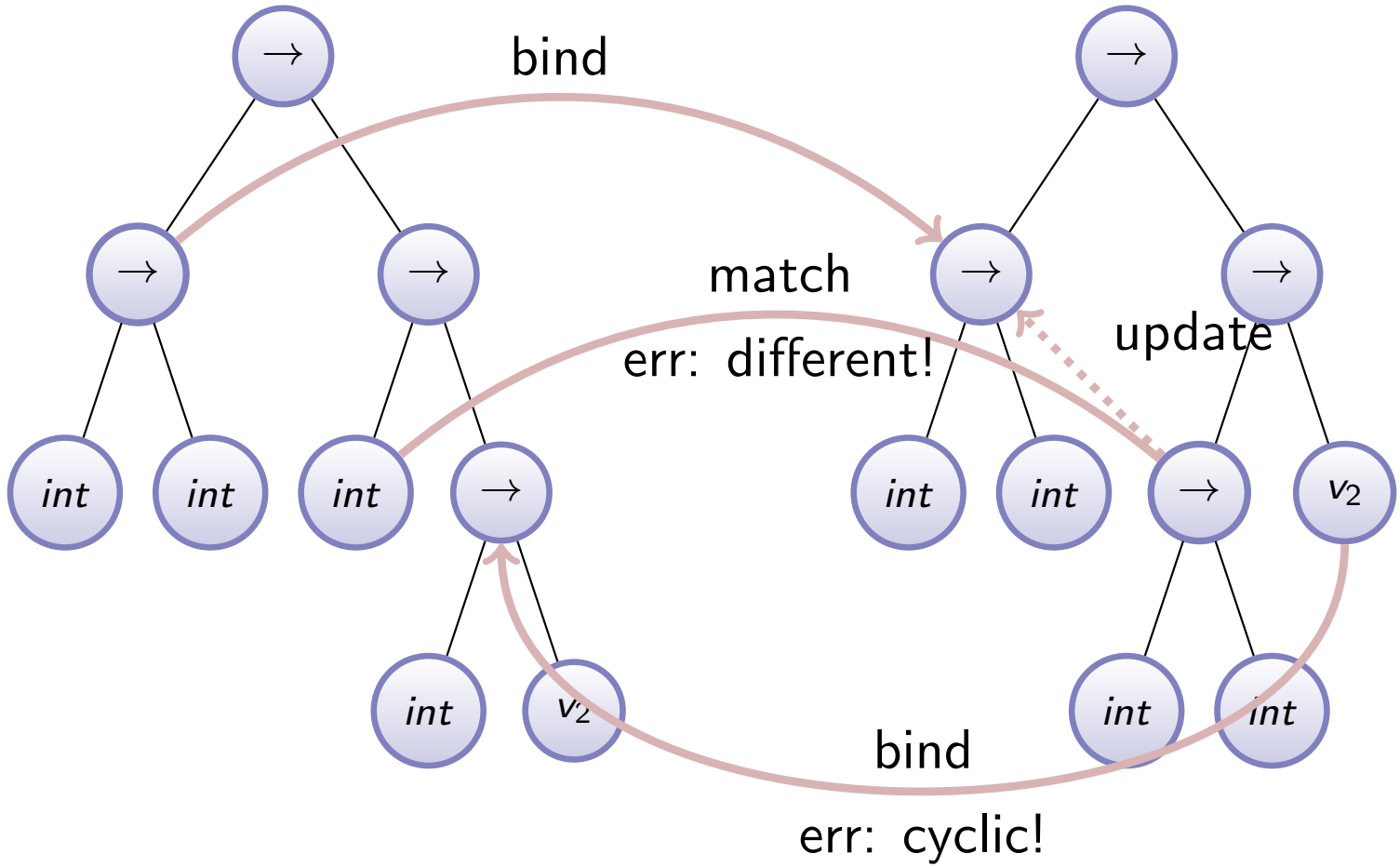
Unification



Unification



Unification



Unification with AGs?

- Unification in UHC has many inputs and outputs: type env, variance env, subst, type, errors, context, pred env, ...
- ... and a rich type language
- Large amount of boilerplate (mostly trivial copying)
- Caching, memoization
- AGs to the rescue?

Unification with AGs?

- Unification in UHC has many inputs and outputs: type env, variance env, subst, type, errors, context, pred env, ...
- ... and a rich type language
- Large amount of boilerplate (mostly trivial copying)
- Caching, memoization
- AGs to the rescue?

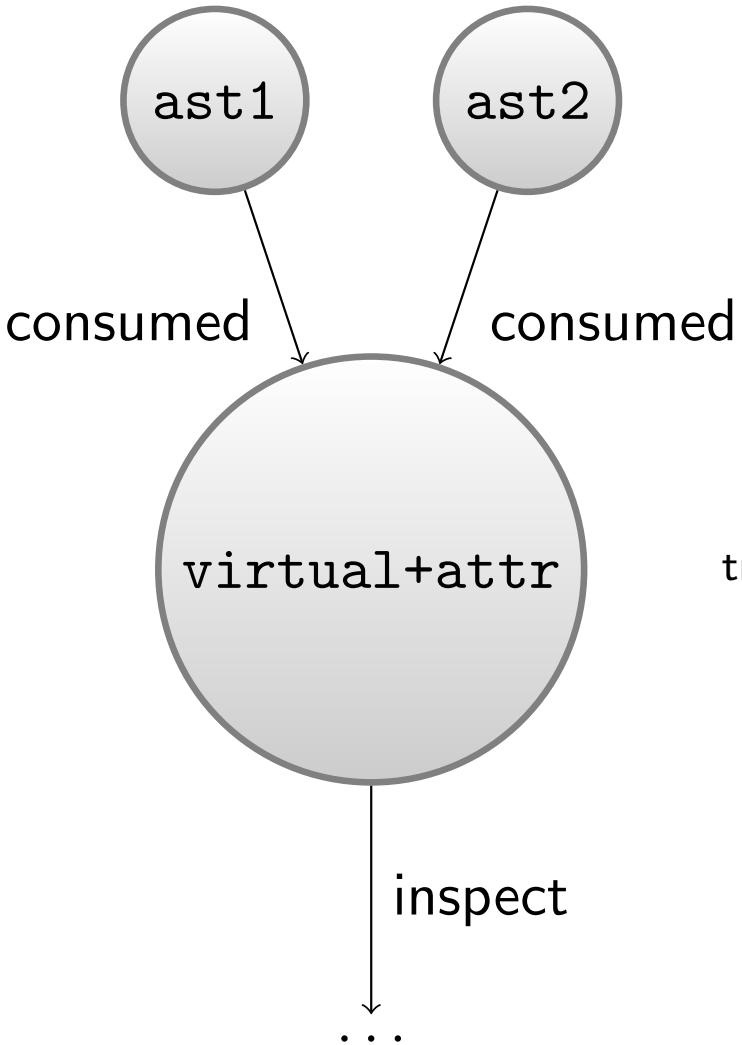
Challenges:

- Simultaneous traversal of two ASTs
- ASTs depend on substitution, which changes during the computation

Idea

Construct virtual tree representing the traversal

- AG evaluation: we already have a tree, then decorate it
- Now: build and decorate the tree at the same time



tree representing traversal of `ast1` and `ast2`

Virtual Constructors: clauses

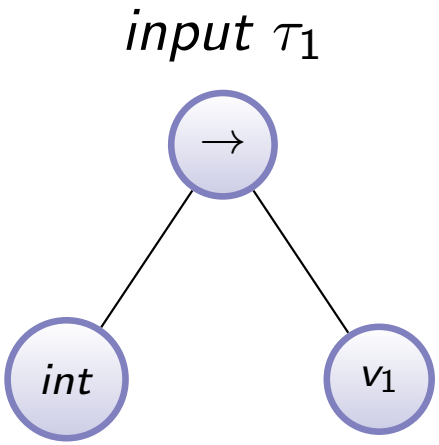
- Each virtual constructor is a separate function
  - Function has access to parameters of parent
  - Function constructs virtual subtrees
  - Function may inspect values of attributes
  - Function may fail
- These function make up the *clauses* of the actual function
- Evaluation: take the results of the first clause that succeeds

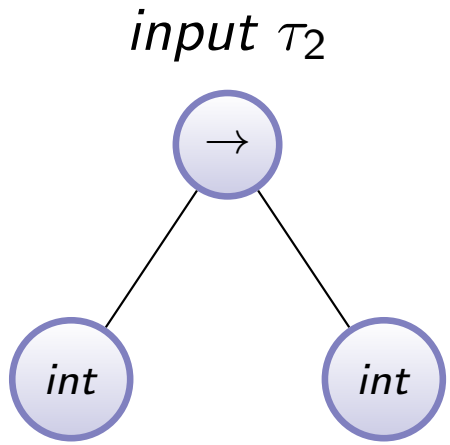
$$
\begin{aligned}
\mathit{unify} = \quad & \mathit{unify\_arrows} \\
& `\mathit{mappend}` \ \mathit{unify\_var} \\
& `\mathit{mappend}` \ \mathit{unify\_const} \\
& `\mathit{mappend}` \ \mathit{unify\_mismatch}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{unify\_arrows} &= \ldots \\
\mathit{unify\_var} &= \ldots \\
\mathit{unify\_const} &= \ldots \\
\mathit{unify\_mismatch} &= \ldots
\end{aligned}
$$

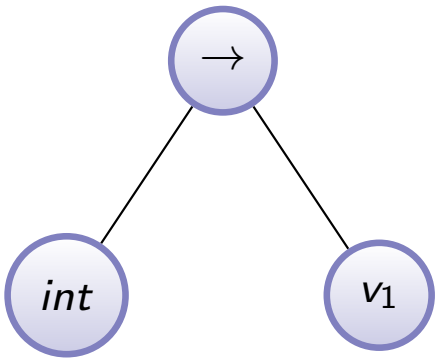
Scenario 1: Growing the tree

Growing the tree



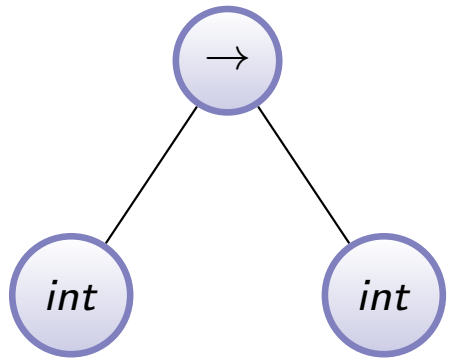


Growing the tree

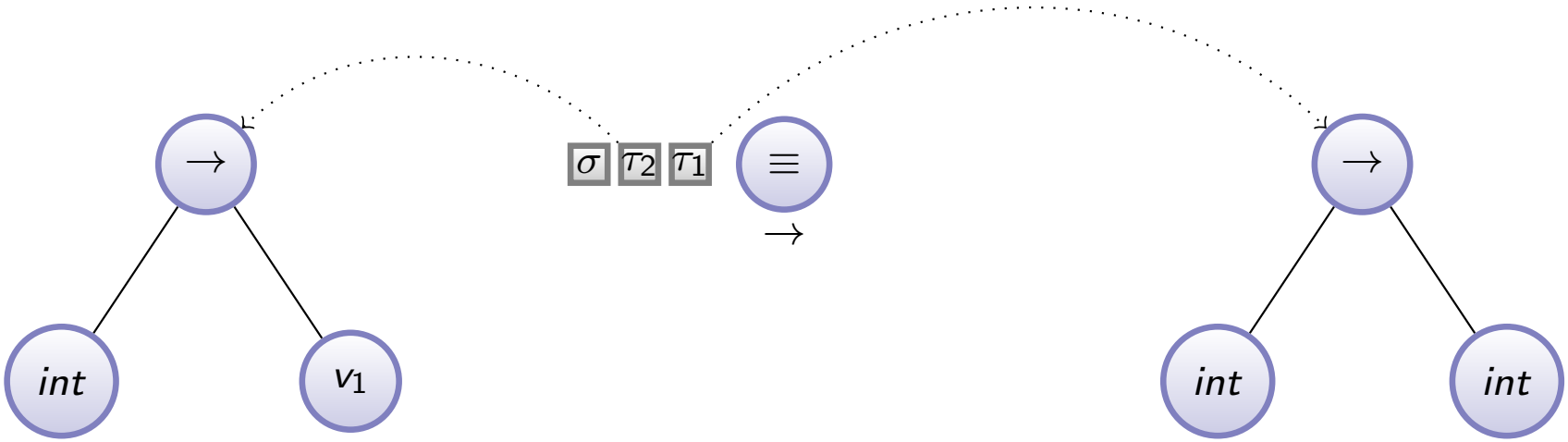


root

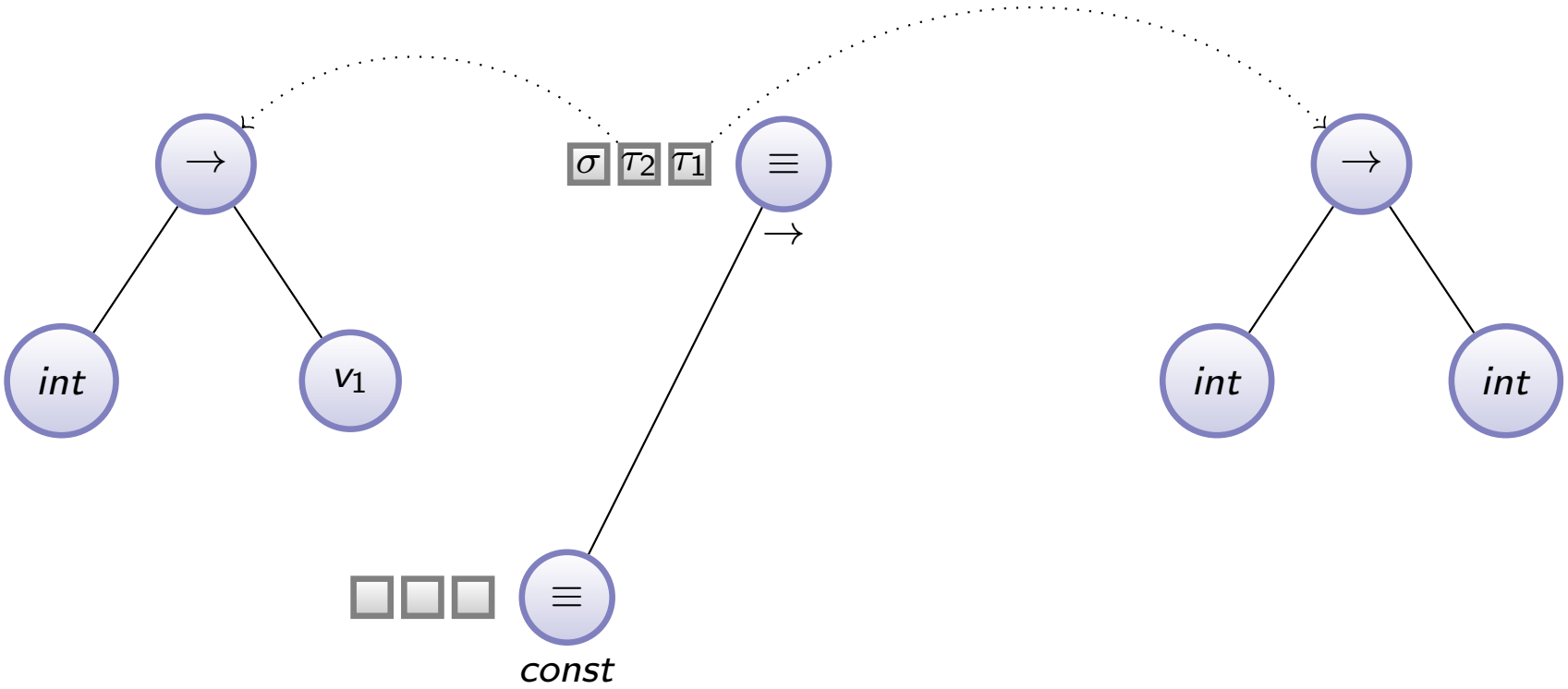
σ τ_2 τ_1 $\equiv$



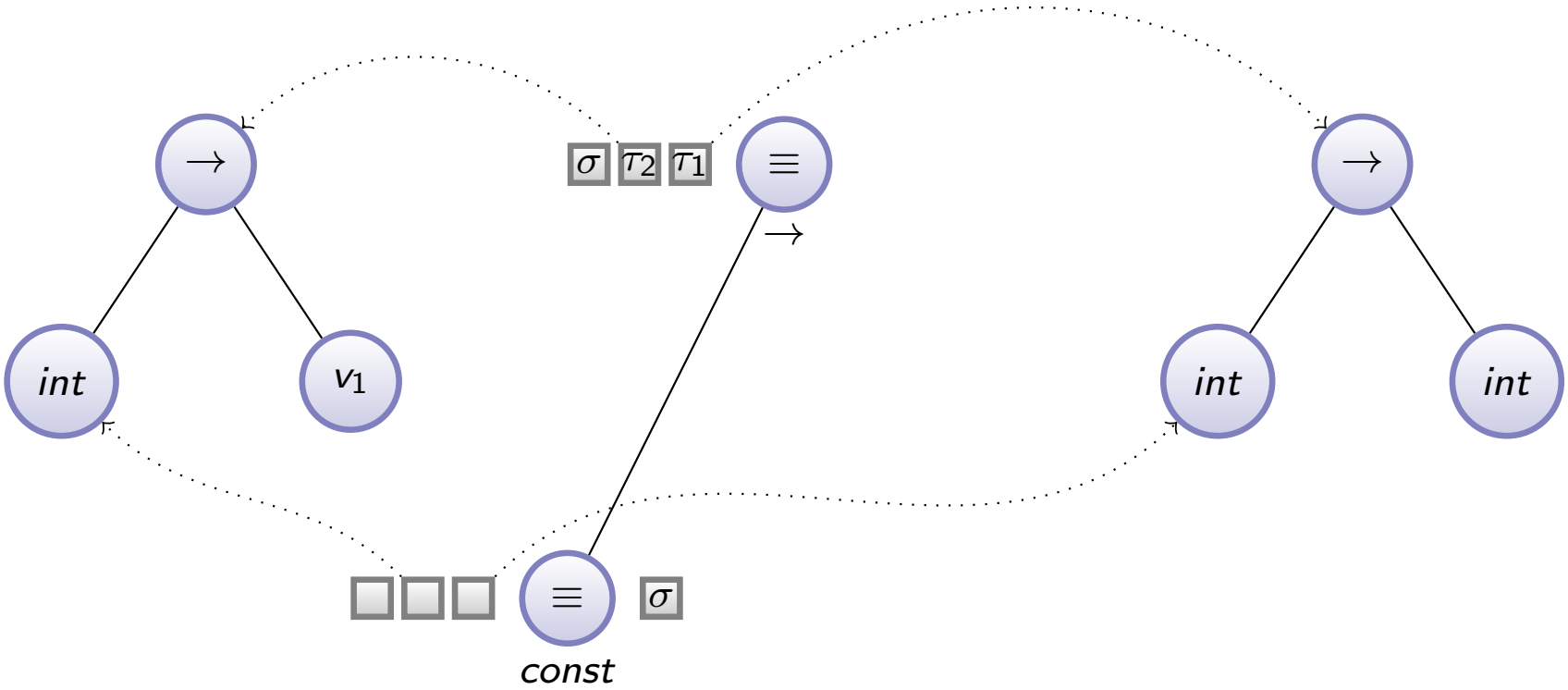
Growing the tree



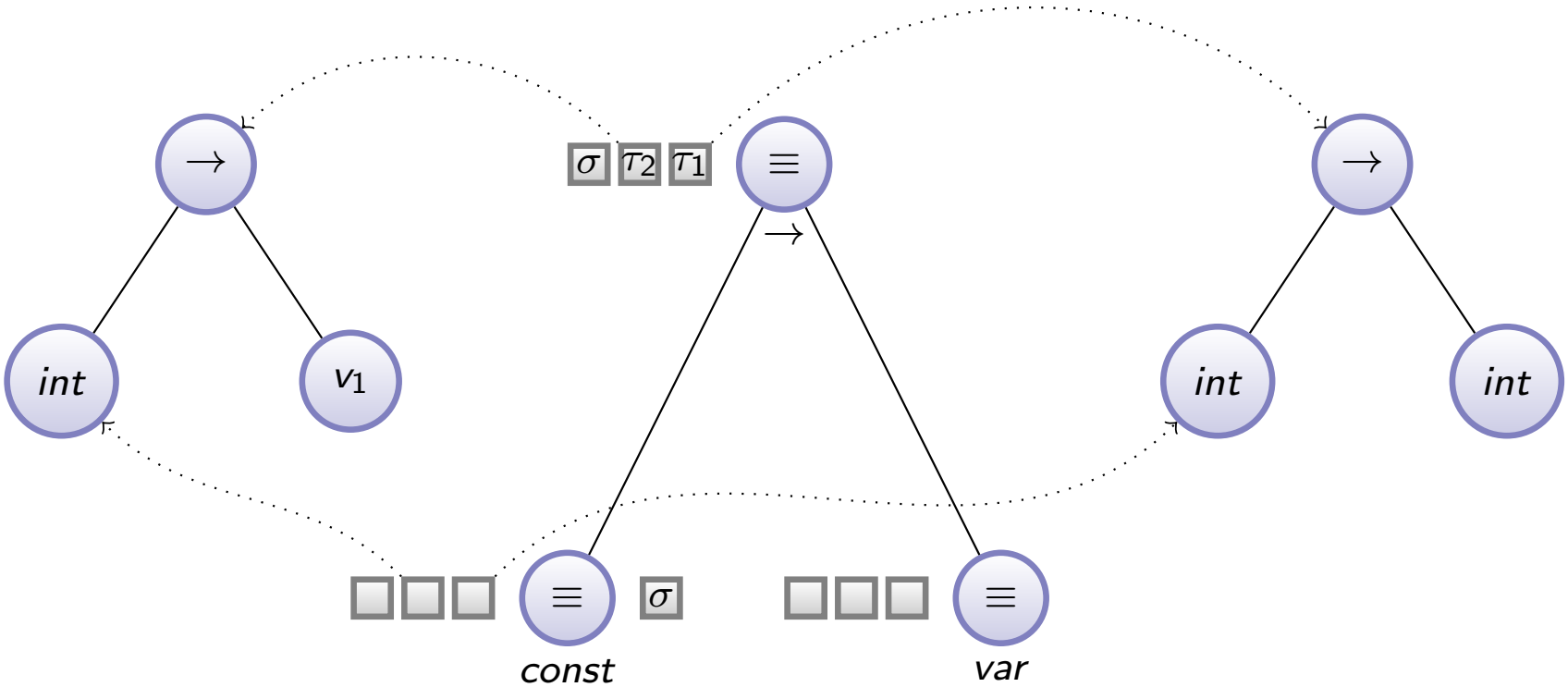
Growing the tree



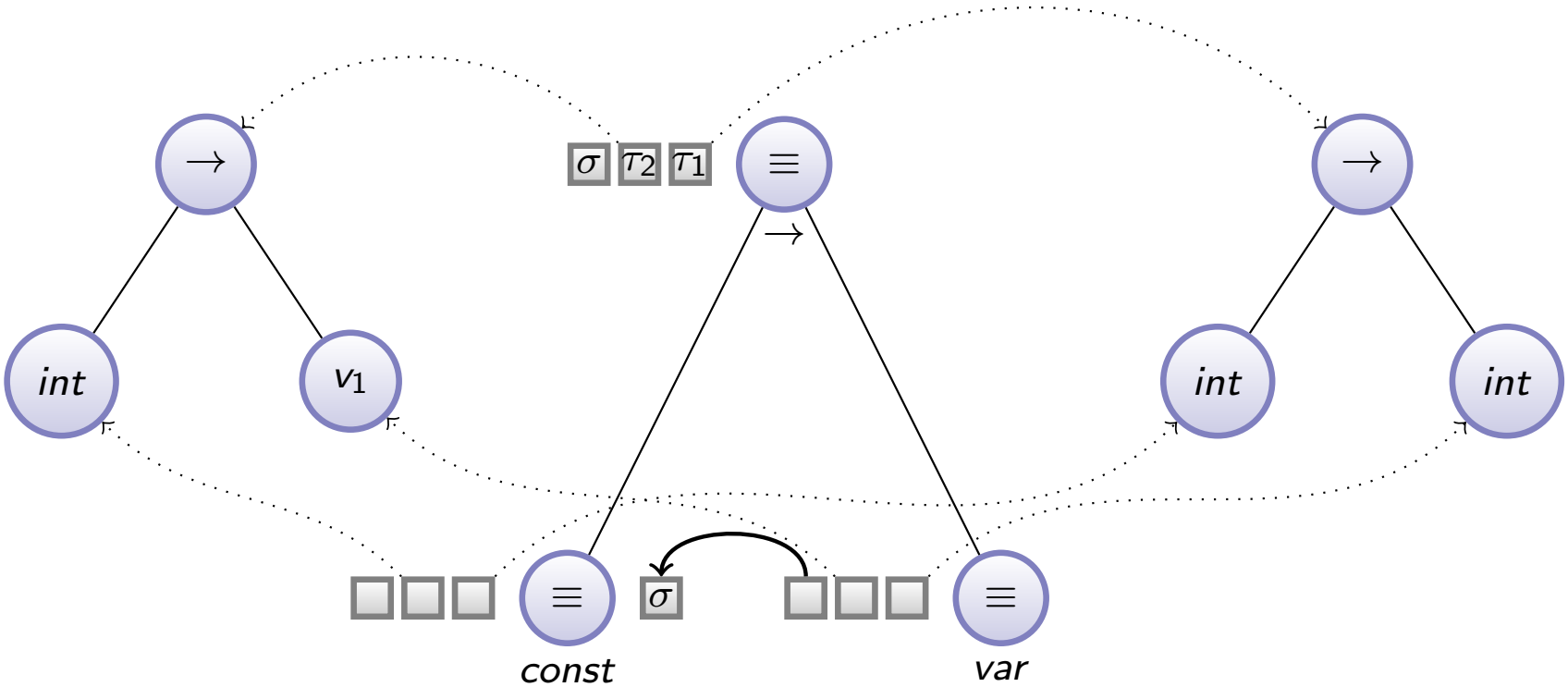
Growing the tree



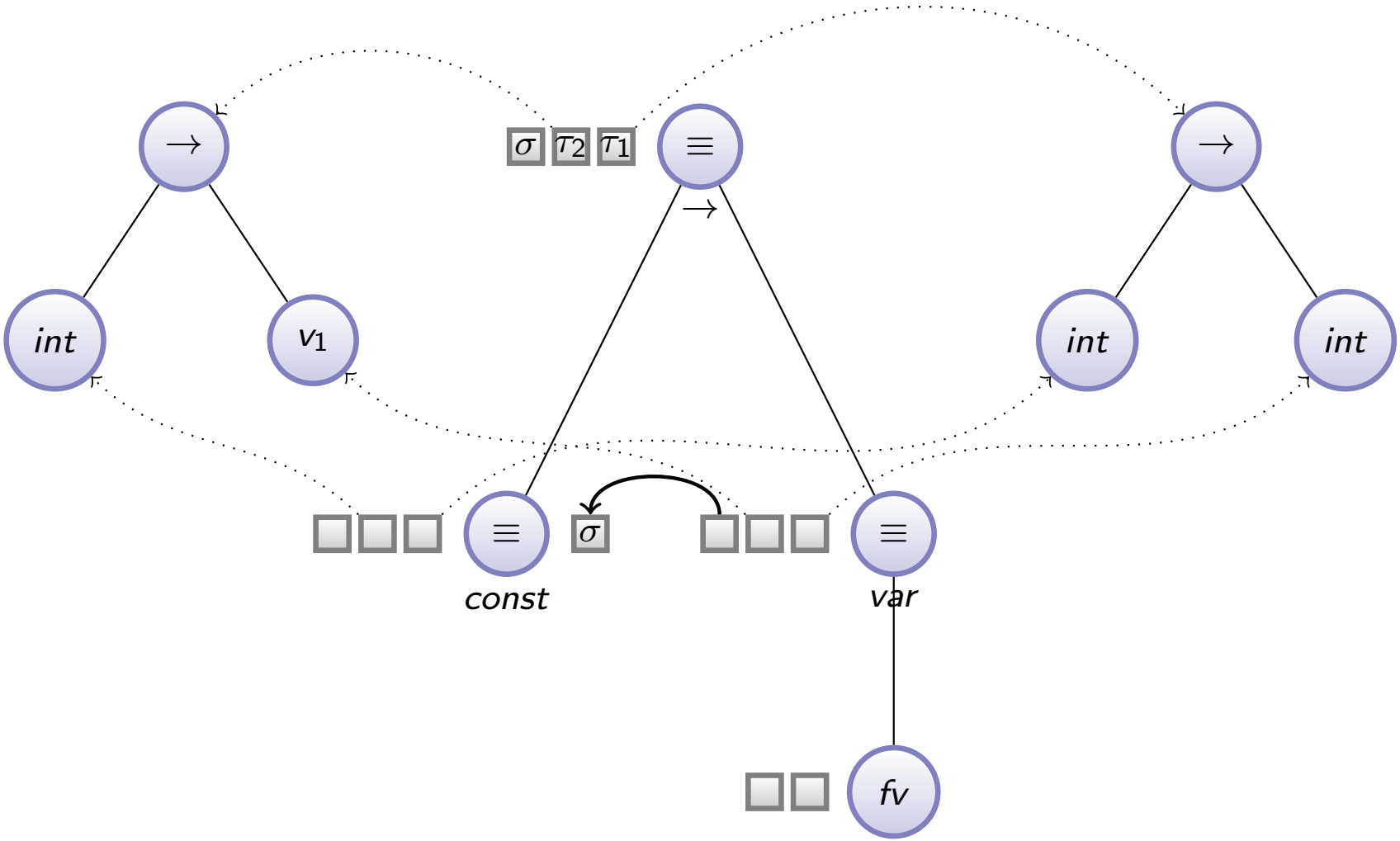
Growing the tree



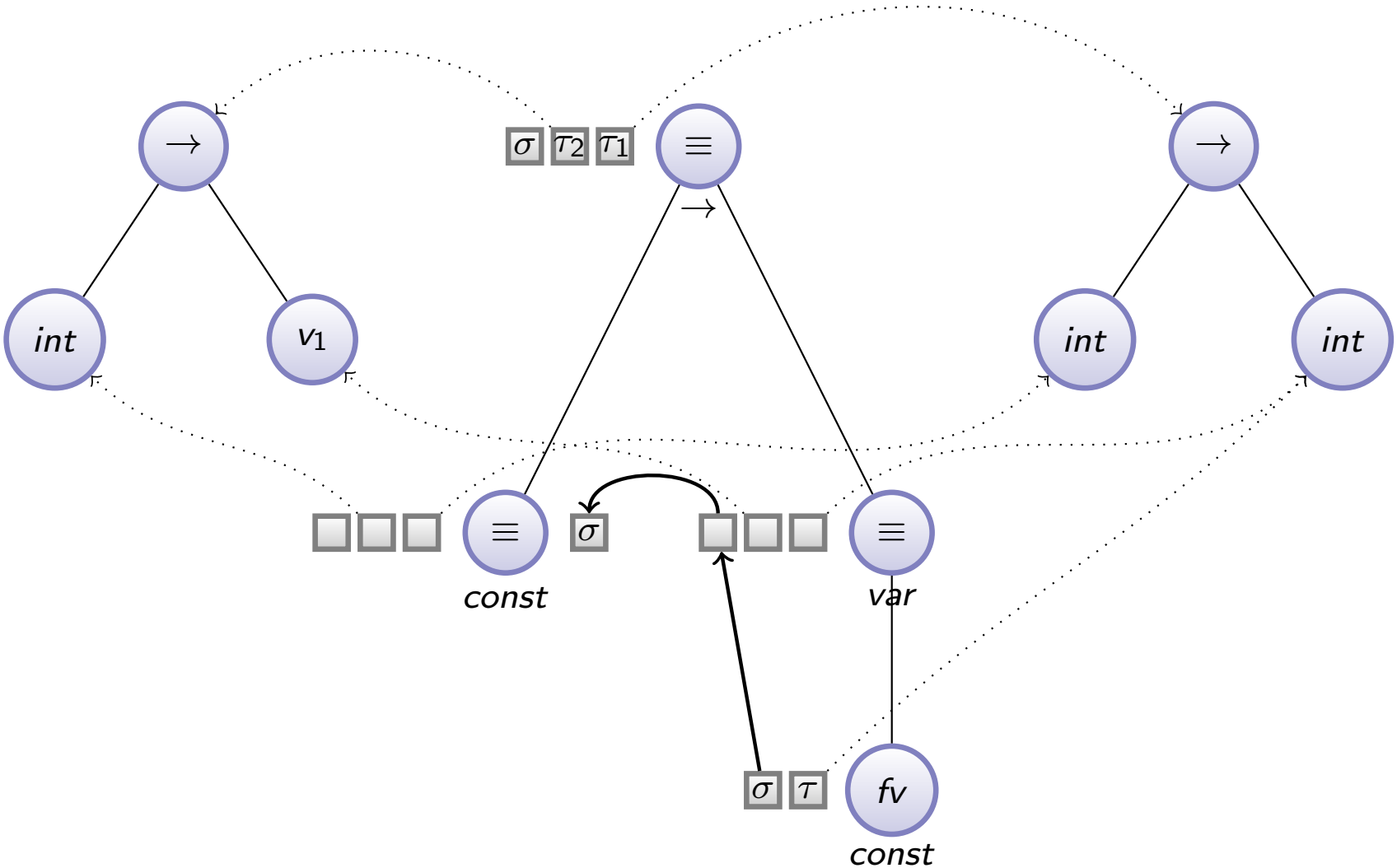
Growing the tree



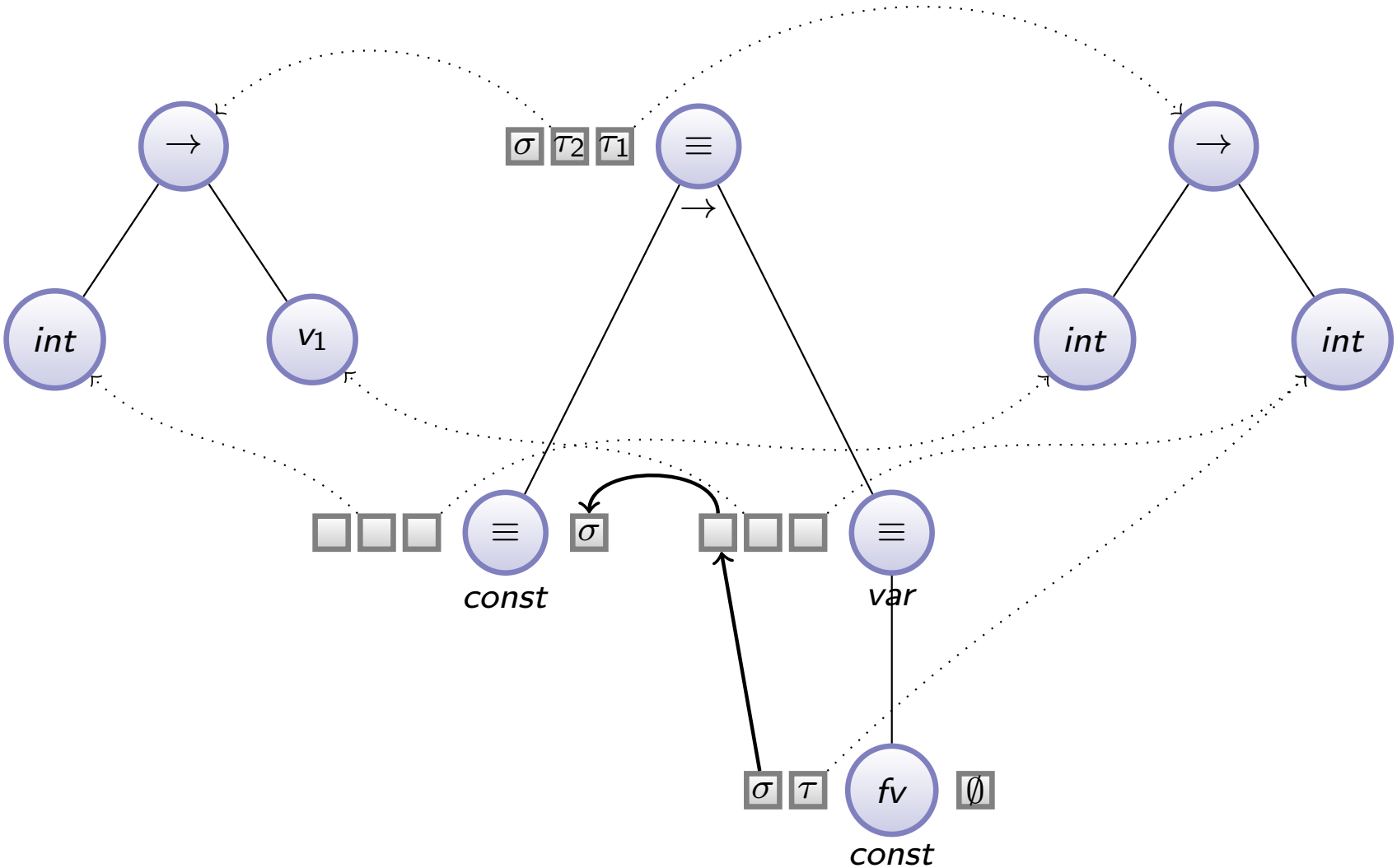
Growing the tree



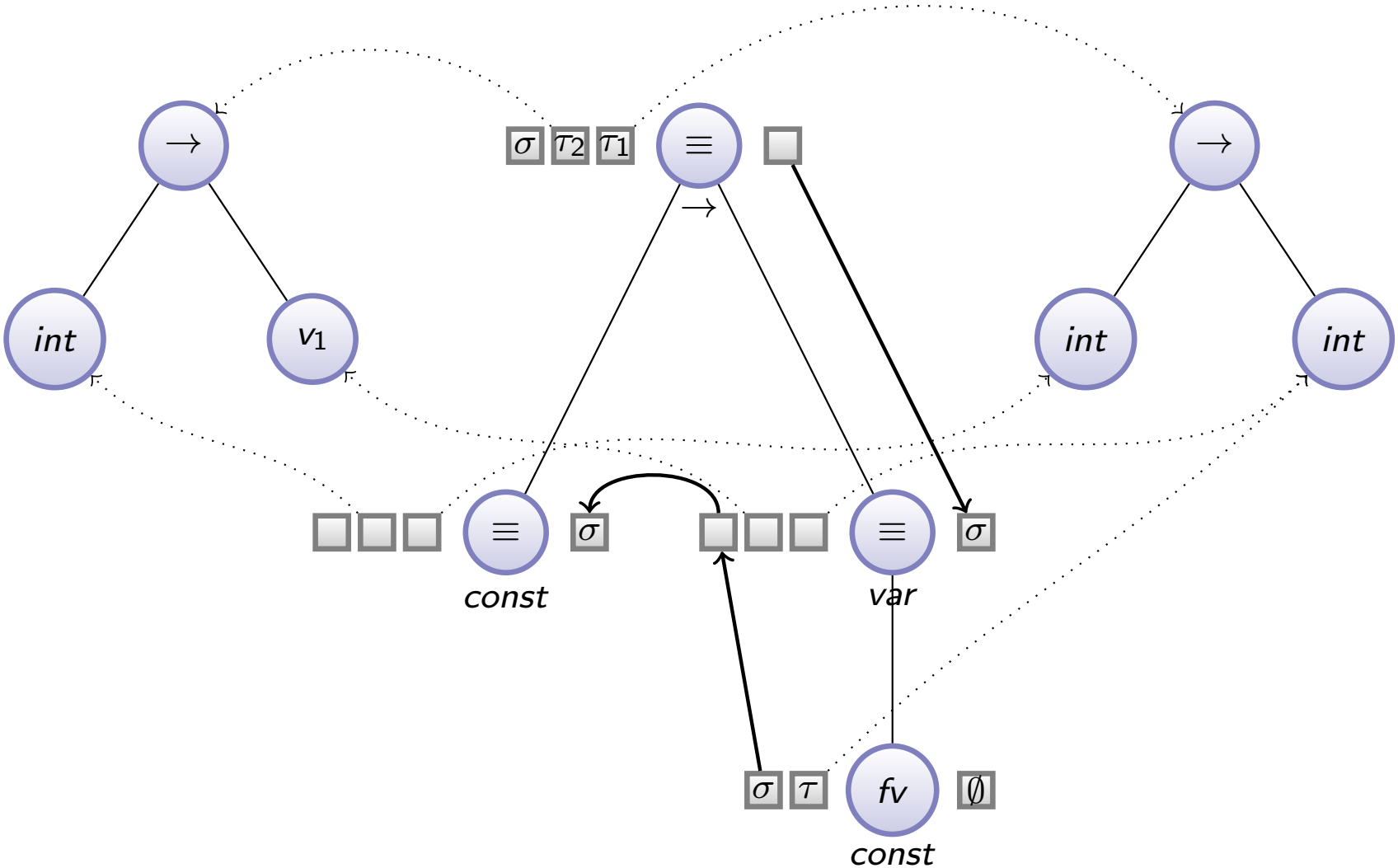
Growing the tree



Growing the tree

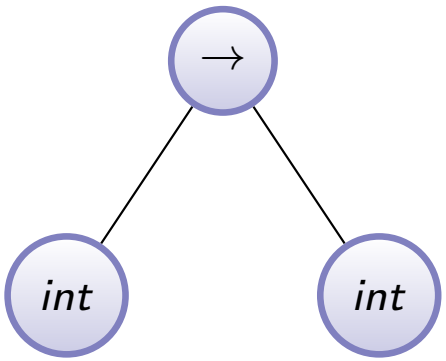


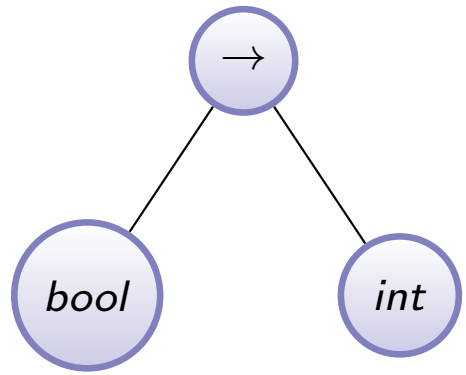
Growing the tree



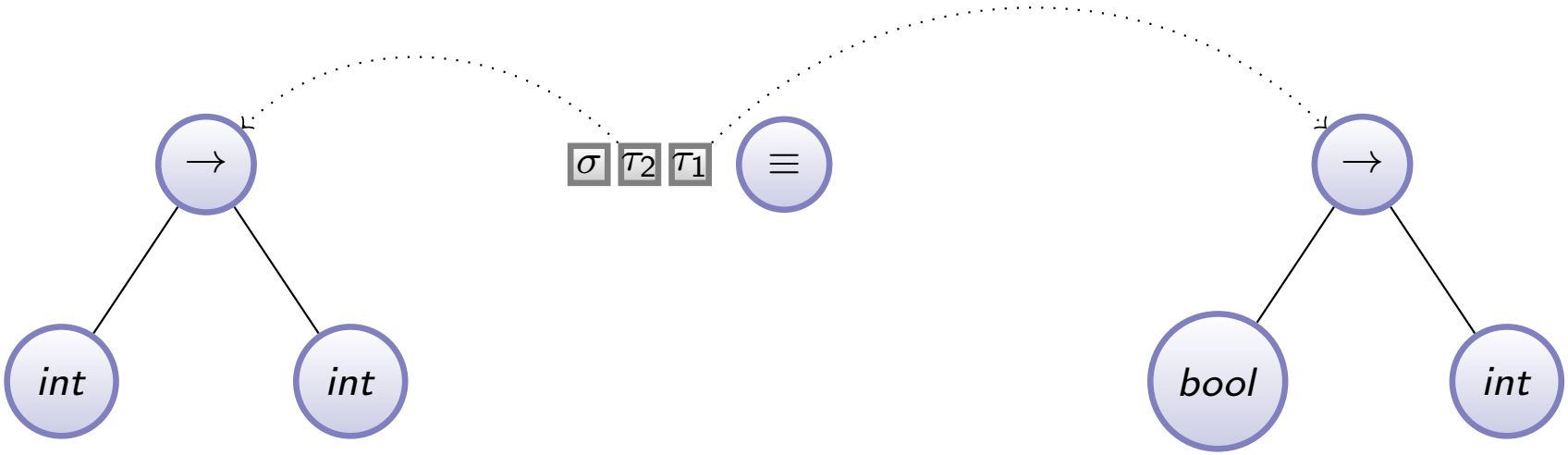
Scenario 2: Context dependent attribution

Context dependent attribution

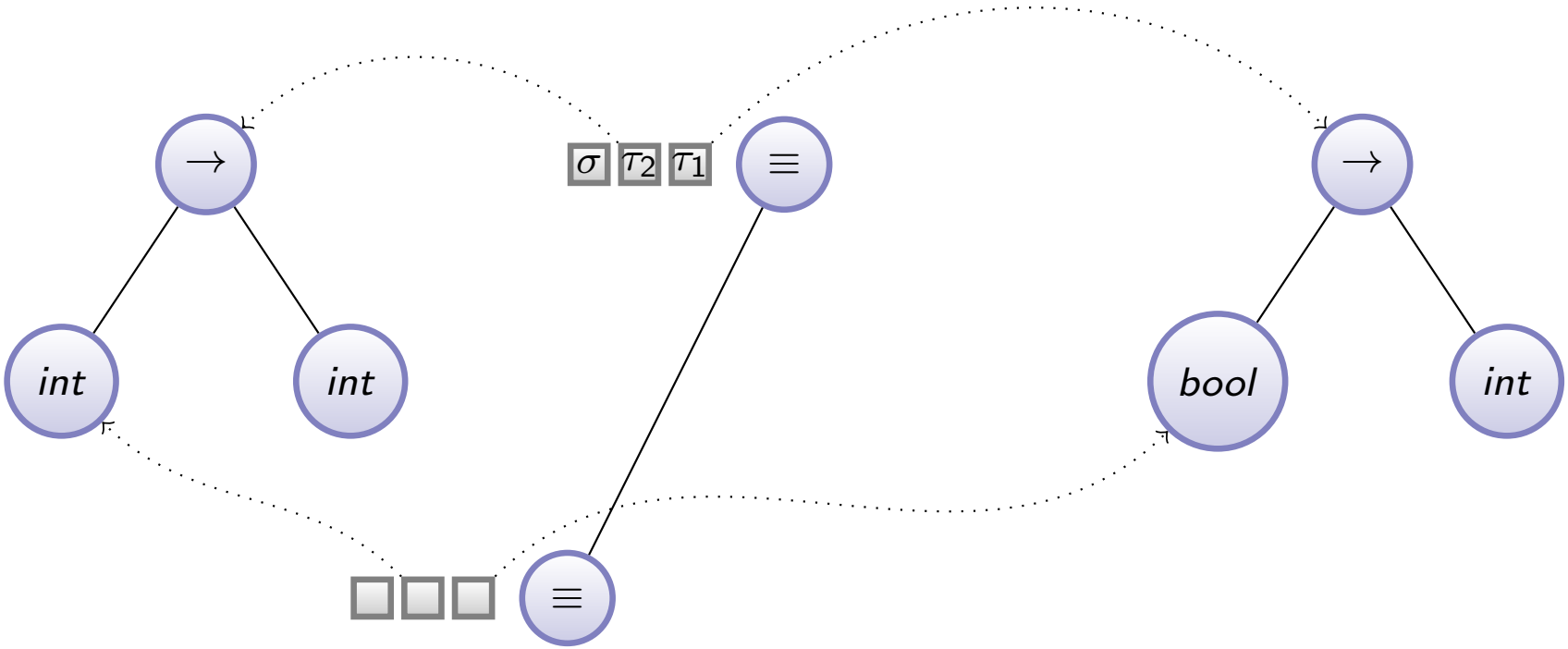




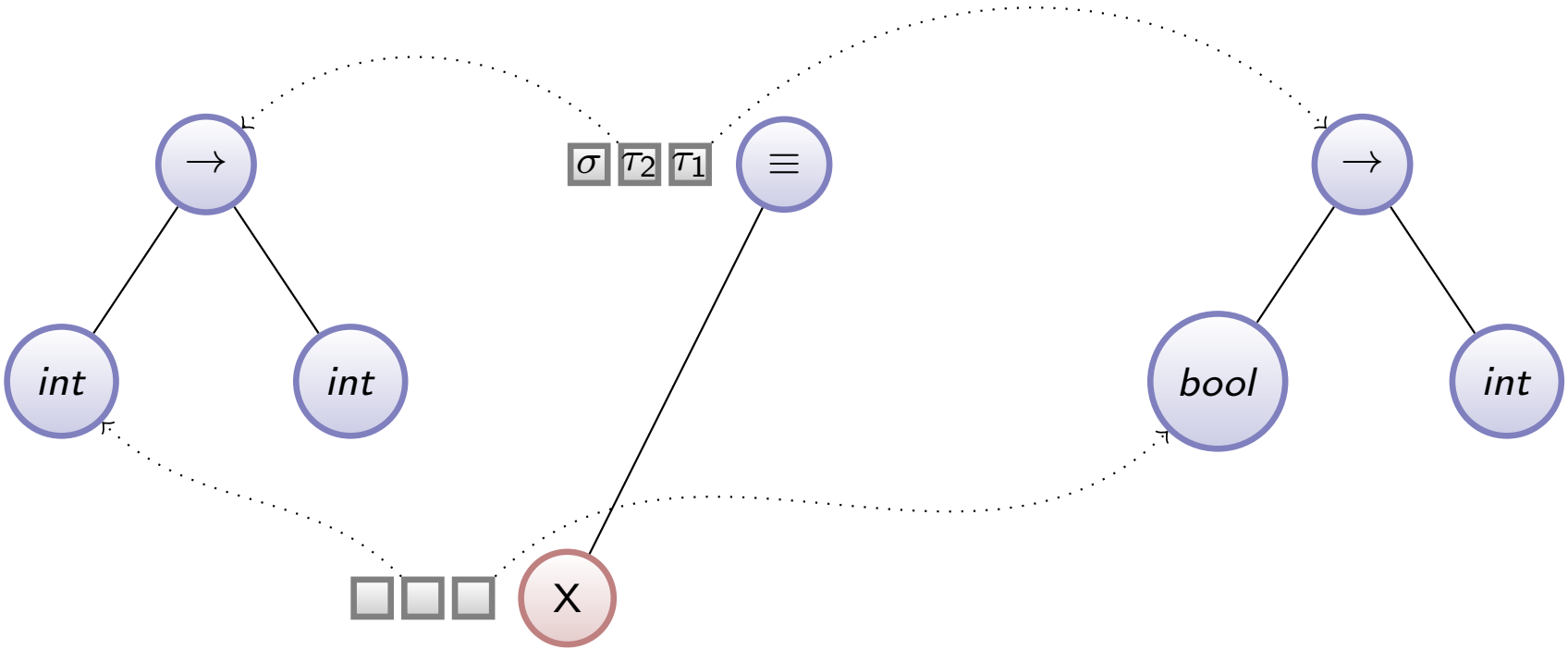
Context dependent attribution



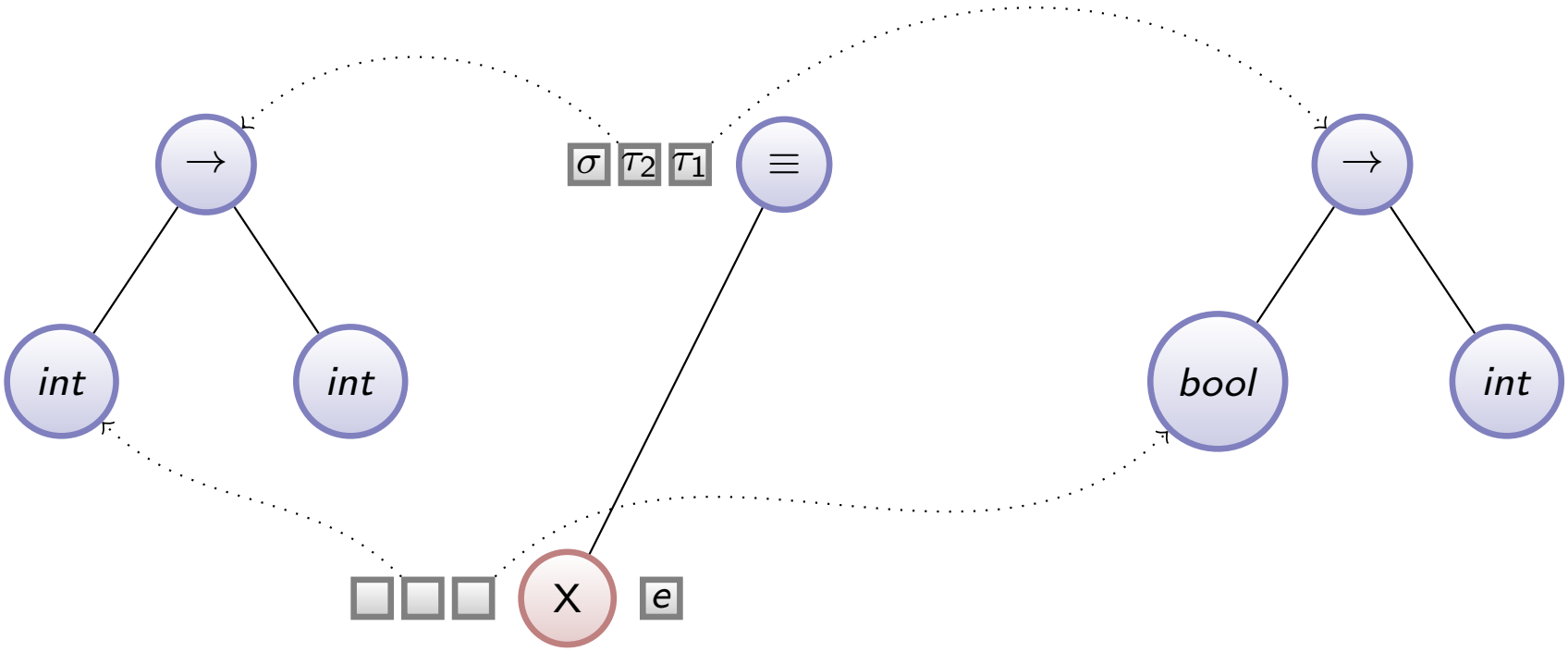
Context dependent attribution



Context dependent attribution



Context dependent attribution



errors instead of an improved substitution

Some Aspects of the Implementation

Simplification

- Assumption: simplified context dependent attribution
- Success and failure contexts only
- No values for synthesized attributes upon failure
- A visit to a subtree must result into success otherwise the clause fails
- Selection of clauses: pick the first one that succeeds

Attributed Trees as Functions of Attributes - type

$$\textbf{type } I\_Unify = Ty \to Ty \to Subst \to Maybe\ (Subst, Errs)$$

Attributed Trees as Functions of Attributes - type

$$\textbf{type } I\_Unify = Ty \to Ty \to Subst \to Maybe\ (Subst, Errs)$$

itf *Unify*
- **inh** τ_1 :: *Ty*
- **inh** τ_2 :: *Ty*
- **inh** σ :: *Subst*
- **syn** σ :: *Subst*
- **syn** *errs* :: *Errs*

Attributed Trees as Functions of Attributes - body

$$unify :: I\_Unify$$
$$unify = unify\_arr\ `mappend`\ ...$$

Attributed Trees as Functions of Attributes - clauses

$$
\begin{aligned}
&\mathit{unify\_arr} \\
&\quad = \ \textbf{sem} :: \mathit{Unify} \\
&\qquad\quad \textbf{match}\ (\mathit{Ty\_Arr}\ \mathit{loc.a}\ \mathit{loc.b}) = \mathit{this}.\tau_1 \\
&\qquad\quad \textbf{match}\ (\mathit{Ty\_Arr}\ \mathit{loc.c}\ \mathit{loc.d}) = \mathit{this}.\tau_2 \\
&\qquad\quad \textbf{child}\ \mathit{left} : \mathit{Unify} = \mathit{unify} \\
&\qquad\quad \mathit{left}.\tau_1 = \mathit{loc.a} \\
&\qquad\quad \mathit{left}.\tau_2 = \mathit{loc.b} \\
&\qquad\quad \mathit{this.errs} = \mathit{left.errs} +\!\!+ \mathit{right.errs}
\end{aligned}
$$

Attributed Trees as Functions of Attributes - clauses

$$
\begin{aligned}
&\mathit{unify\_arr} \\
&\quad = \ \textbf{sem} :: \mathit{Unify} \\
&\qquad \textbf{match}\ (\mathit{Ty\_Arr}\ \mathit{loc.a}\ \mathit{loc.b}) = \mathit{this}.\tau_1 \\
&\qquad \textbf{match}\ (\mathit{Ty\_Arr}\ \mathit{loc.c}\ \mathit{loc.d}) = \mathit{this}.\tau_2 \\
&\qquad \textbf{child}\ \mathit{left} : \mathit{Unify} = \mathit{unify} \\
&\qquad \mathit{left}.\tau_1 = \mathit{loc.a} \\
&\qquad \mathit{left}.\tau_2 = \mathit{loc.b} \\
&\qquad \mathit{this.errs} = \mathit{left.errs} +\!\!+ \mathit{right.errs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unify\_arr}\ \tau_1\ \tau_2\ \mathit{inSubst} = \\
&\quad \textbf{case}\ \tau_1\ \textbf{of} \qquad \textbf{case}\ \tau_2\ \textbf{of} \\
&\qquad (\mathit{Ty\_Arr}\ a\ b)\ (\mathit{Ty\_Arr}\ c\ d) \\
&\quad (\mathit{subst1}, \mathit{errs1}) = \mathit{unify}\ a\ c \\
&\quad (\mathit{subst2}, \mathit{errs2}) = \mathit{unify}\ b\ d \\
&\quad \mathit{outSubst} = ...\mathit{inSubst} \\
&\quad \mathit{outErrs} = ... \\
&\quad \rightarrow \mathit{Just}\ (\mathit{outSubst}, \mathit{outErrs}) \\
&\quad \rightarrow \mathit{Nothing}
\end{aligned}
$$

Pattern Matching and Evaluation Order

$$
\begin{aligned}
&\mathit{unify\_arr}\ \tau_1\ \tau_2\ \mathit{inSubst} = \\
&\quad \textbf{case}\ \tau_1\ \textbf{of} \\
&\qquad (\mathit{Ty\_Arr}\ a\ b) \\
&\quad \textbf{case}\ \tau_2\ \textbf{of} \\
&\qquad (\mathit{Ty\_Arr}\ c\ d)
\end{aligned}
$$

- Clause selection based on pattern matches on attributes
- The order of the pattern matches influences (lazy) evaluation
- Dependency analysis

Multi-visit

Scenario:

- Select a clause based on some inherited attributes, i.e. the var-clause for unification
- Compute some attributes, i.e. free variables
- Refine clause selection based on other attributes, i.e. occur-check
- Compute more attributes, i.e. errors on failure, new substitution on success

Or variations on this theme

Implementation based on Visit Functions

A visit function is:

- Function that takes some inherited attributes, finds a matching clause, computes some synthesized attributes and a continuation visit-function for the remainder of the visits.
- Originally intended as functional backend for AGs

- Consider the ASTs as additional inherited attributes
- Visit function pattern matches on inherited attribute to find a clause
- Call recursively, etc. to visit children

- We need a front-end for visit functions!

Front-end

itf *Check*
 inh τ_1, τ_2 :: *Ty*
 syn σ :: *Subst*
 tail :: *Analyze*

itf *Analyze*
 inh *finalSubst* :: *Subst*
 syn *errs* :: *Ty*

unify = **sem** :: *Check*
 match (*Arr a b*) = *this*.τ_1
 match (*Arr c d*) = *this*.τ_2
 child *left* = *unify*
 visit *left* : *Check*
 left.τ_1 = *a*
 left.τ_2 = *c*
 this.σ = *left*.σ ‘*union*‘ *right*.σ
 tail sem :: *Analyze*
 visit *left* : *Analyze*
 left.finalSubst = *this.finalSubst*
 this.errs = *left.errs* ++ *right.errs*
 ‘*mappend*‘ **sem** :: *Check* ...

Comparison to AGs

Similar:

- Model with attributes and rules
- Conveniences: trivial rule insertion, sanity checks, AG optimizations

To be done explicitly what you get normally for free with AGs:

- Manually select clauses
- Partition attributes over multiple visits
- Explicitly indicate what function to use for the children

The usual dependency analysis for AGs can be used to augment a partial specification of visit function to a full specification.

Current Work

- Fun with Higher-Order Attribute Grammars

```
data Tree ...
data View ...
sem Tree
  | Node inst.view : T_View
         inst.view = sem_View_Node' (...)
         inst.inh = ...
         ... = inst.syn
```

- Nested Attribute Grammars

```
sem Tree
  | Node lhs.view = sem View
                        -- own inh/syn attrs
                        -- + attrs of outer sem
                  -- own attrs + attrs of view
```

Conclusion

- Problem: AGs not suitable for traversals over multiple ASTs
- Solution: decorate a virtual tree that represents the traversal itself

Use AGs for the implementation of:

- Type rules that are not syntax directed (but e.g. type directed)
- Bi-directional type rules
- Search algorithms with attributed search trees
- Constraint solving
- Actually, any Haskell function

- Intended to improve the code of the UHC type checker
- Implemented as a preprocessor for Haskell
- UHC project: `http://www.cs.uu.nl/~ariem/`