universität innsbruck

Summer Term 2026

Program Verification

Part 4 – Checking Well-Definedness of Functional Programs

René Thiemann

Department of Computer Science

Overview

- recall: a functional program is well-defined if
  - it is pattern disjoint,
  - it is pattern complete, and
  - $\hookrightarrow$ is terminating
- well-definedness is prerequisite for standard model, for derived theorems, ...
- task: given a functional program as input, ensure well-definedness
  - known: type-checking algorithm
  - missing: algorithm for type-inference
  - missing: algorithm for deciding pattern disjointness
  - missing: algorithm for deciding pattern completeness
  - missing: methods to ensure termination
- all of these missing parts will be covered in this chapter

Type-Checking with Implicit Variables

Type-Inference

- structure of functional programs
  - data-type definitions
  - function definitions: type of new function + defining equations
  - not mentioned: type of variables
- in proseminar: work-around via fixed scheme which does not scale
  - singleton characters get type Nat
  - words ending in "s" get type List
- aim: infer suitable type of variables automatically
- example: given FP

$$\begin{aligned}&\mathsf{append} : \mathsf{List} \times \mathsf{List} \to \mathsf{List}\\&\mathsf{append}(\mathsf{Cons}(x, y), z) = \mathsf{Cons}(x, \mathsf{append}(y, z))\\&\mathsf{append}(\mathsf{Nil}, x) = x\end{aligned}$$

we should be able to infer that $x : \mathsf{Nat}$, $y : \mathsf{List}$ and $z : \mathsf{List}$ in the first equation, whereas $x : \mathsf{List}$ in the second equation

Type-Checking for Single Equations

- known: type-checking algorithm (we omit the error message in `assert` and `failure`)
  `typeCheck :: Sig -> Vars -> Term -> CheckS Type`
  - `type Sig = FSym -> CheckS ([Type], Type)` – Σ
  - `type Vars = Var -> CheckS Type` – $\mathcal{V}$
  - `typeCheck` takes Σ and $\mathcal{V}$ and delivers type of term
- we want a function that works in the other direction: it gets an intended type as input, and delivers a suitable type for the variables
  `inferType :: Sig -> Type -> Term -> CheckS [(Var,Type)]`
- then type-checking an equation without explicit `Vars` is possible

```
typeCheckEqn :: Sig -> (Term, Term) -> CheckS ()
typeCheckEqn sigma (Var x, r) = failure
typeCheckEqn sigma (l @ (Fun f _), r) = do
  (_,ty) <- sigma f
  vars <- inferType sigma ty l
  tyR <- typeCheck sigma (\ x -> lookup x vars) r
  assert (ty == tyR)
```


Type-Inference Algorithm

- note: upcoming algorithm only infers types of variables
  (in polymorphic setting often also type of function symbols is inferred)

```
inferType :: Sig -> Type -> Term -> CheckS [(Var,Type)]
inferType sigma ty (Var x) = return [(x,ty)]
inferType sigma ty (Fun f ts) = do
  (tysIn,tyOut) <- sigma f
  assert (length tysIn == length ts)
  assert (tyOut == ty)
  varsL <- mapM (\ (tyi, ti) -> inferType sigma tyi ti) (zip tysIn ts)
  let vars = nub (concat varsL) -- nub removes duplicates
  assert (distinct (map fst vars))
  return vars

distinct :: Eq a => [a] -> Bool
distinct xs = length (nub xs) == length xs
```


Soundness of Type-Inference Algorithm

- properties
  - if $t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ then $\mathit{inferType}\ \Sigma\ \tau\ t = \mathit{return}\ (\mathcal{V} \cap \mathcal{V}\mathit{ars}(t))$
  - if $\mathit{inferType}\ \Sigma\ \tau\ t = \mathit{return}\ \mathcal{V}$ then
    - $\mathcal{V}$ is well-defined (no conflicting variable assignments) and
    - $t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$
- properties can be shown in similar way to type-checking algorithm, cf. slides 2/37–44
- note that 'if $t \in \mathcal{T}(\Sigma,\mathcal{V})_\tau$ then $\mathit{inferType}\ \Sigma\ \tau\ t \neq \mathit{failure}$' is a property which is not strong enough when performing induction

Processing Functional Programs

Processing Functional Programs

- aim: write program which takes
  - functional program as input (data type definitions + function definitions)
  - checks the syntactic requirements
  - stores the relevant information in some internal representation
  - later: also checks well-definedness
- such a program is essential part of a compiler
- program should be easy to verify

Recall: Data Type Definitions

- given: set of types $\mathcal{T}y$, signature $\Sigma = \mathcal{C} \uplus \mathcal{D}$
- each data type definition has the following form

$$\begin{aligned} \text{data } \tau = {} & c_1 : \tau_{1,1} \times \ldots \times \tau_{1,m_1} \to \tau \\ & \mid \ldots \\ & \mid c_n : \tau_{n,1} \times \ldots \times \tau_{n,m_n} \to \tau \end{aligned}$$

where

  - $\tau \notin \mathcal{T}y$ — fresh type name
  - $c_1, \ldots, c_n \notin \Sigma$ and $c_i \neq c_j$ for $i \neq j$ — fresh and distinct constructor names
  - each $\tau_{i,j} \in \{\tau\} \cup \mathcal{T}y$ — only known types
  - exists c_i such that $\tau_{i,j} \in \mathcal{T}y$ for all j — non-recursive constructor
- effect: add new type and new constructors
  - $\mathcal{T}y := \mathcal{T}y \cup \{\tau\}$
  - $\mathcal{C} := \mathcal{C} \cup \{c_1 : \tau_{1,1} \times \ldots \times \tau_{1,m_1} \to \tau, \ldots, c_n : \tau_{n,1} \times \ldots \times \tau_{n,m_n} \to \tau\}$

Existing Encoding of Part 2: Signatures and Terms

```
type Check e = ... -- Maybe or Either e

type Type = String
type Var  = String
type FSym = String
type Vars = Var -> CheckS Type
type FSymInfo = ([Type], Type)
type Sig = FSym -> CheckS FSymInfo

data Term = Var Var | Fun FSym [Term]
```


Encoding Functional Programs in Haskell

```
-- input: unchecked data-type definitions and function definitions
data DataDefinition = Data Type [(FSym, FSymInfo)]
data FunctionDefinition = ... -- later
type FunctionalProg =
  ([DataDefinition], [FunctionDefinition])

-- internal representation
type SigList = [(FSym, FSymInfo)]  -- signatures as list
type Defs = SigList                -- list of defined symbols
type Cons = SigList                -- list of constructors
type Equations = [(Term, Term)]    -- all function equations
-- all combined in Haskell-type; it also stores known types
data ProgInfo = ProgInfo [Type] Cons Defs Equations

-- checking single data type definition
processDataDefinition ::
  ProgInfo -> DataDefinition -> CheckS ProgInfo
```

Checking a Single Data Definitions

```
processDataDefinition
    (ProgInfo tys cons defs eqs)
    (Data ty newCs)
 = do
    assert (not (elem ty tys))
    let newTys = ty : tys
    assert (distinct (map fst newCs))
    assert (all (\ (c,_) -> all (/= c) (map fst (cons ++ defs))) newCs)
    assert (all (\ (_,(tysIn,tyOut)) ->
      tyOut == ty &&
      all (\ ty -> elem ty newTys) tysIn) newCs)
    assert (any
      (\ (_,(tysIn,_)) -> all (/= ty) tysIn) newCs)
    return (ProgInfo newTys (newCs ++ cons) defs eqs)
```


Checking Several Data Definitions

- processing many data definitions can be easily done by using `foldM`: predefined monadic version of `foldl`

```
foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
foldM f e [] = return e
foldM f e (x : xs) = do
  d <- f e x
  foldM f d xs

processDataDefinition ::
  ProgInfo -> DataDefinition -> CheckS ProgInfo
processDataDefinition = ... -- previous slide

processDataDefinitions ::
  ProgInfo -> [DataDefinition] -> CheckS ProgInfo
processDataDefinitions = foldM processDataDefinition
```


Checking Function Definitions w.r.t. Slide 3/15

```
data FunctionDefinition = Function
  FSym           -- name of function
  FSymInfo       -- type of function
  [(Term,Term)]  -- equations

processFunctionDefinition
  :: ProgInfo -> FunctionDefinition -> CheckS ProgInfo
processFunctionDefinition = ... -- exercise

processFunctionDefinitions ::
  ProgInfo -> [FunctionDefinition] -> CheckS ProgInfo
processFunctionDefinitions =
  foldM processFunctionDefinition
```


Checking Functional Programs

```
initialProgInfo = ProgInfo [] [] [] []

processProgram :: FunctionalProg -> CheckS ProgInfo
processProgram (dataDefs, funDefs) = do
  pi <- processDataDefinitions initialProgInfo dataDefs
  processFunctionDefinitions pi funDefs
```

Current State

- `processProgram :: FunctionalProg -> CheckS ProgInfo` is Haskell program to check user provided functional programs, whether they adhere to the specification of functional programs w.r.t. slides 3/4 and 3/15
- its functional style using error monads permits us to easily verify its correctness
  - no induction required
  - based on assumption that builtin functions behave correctly, e.g., `all`, `any`, `nub`, ...
- missing: checks for well-defined functional programs w.r.t. slide 3/45

Checking Pattern Disjointness

Deciding Pattern Disjointness

- program is pattern disjoint if for all $f : \tau_1 \times \cdots \times \tau_n \to \tau \in \mathcal{D}$ and all $t_1 \in \mathcal{T}(\mathcal{C})_{\tau_1}, \ldots, t_n \in \mathcal{T}(\mathcal{C})_{\tau_n}$ there is at most one equation $\ell = r$ in the program, such that ℓ matches $f(t_1, \ldots, t_n)$
- in proseminar: pattern disjointness is equivalent to the following condition: for each pair of distinct equations $\ell_1 = r_1$ and $\ell_2 = r_2$, ℓ_1 and a variable renamed variant of ℓ_2 do not unify
- key missing part for checking pattern disjointness is an algorithm for unification:

  given two terms s and t, decide $\exists \sigma.\, s\sigma = t\sigma$

Unification Algorithm of Martelli and Montanari

- input: unification problem $U = \{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$
- question: is U solvable, i.e., does there exist a solution σ, a substitution satisfying $\forall i \in \{1, \ldots, n\}.\, s_i\sigma = t_i\sigma$
- two different kinds of output:
  - unification problem in solved form:

    $\{x_1 \stackrel{?}{=} v_1, \ldots, x_m \stackrel{?}{=} v_m\}$ with distinct x_j's and $x_j \notin \mathcal{V}(v_i)$

    solved forms can be interpreted as substitution

    $$\sigma(x) = \begin{cases} v_i, & \text{if } x = x_i \\ x, & \text{otherwise} \end{cases}$$

    and this σ will be solution of U
  - $\bot$, indicating that U is not solvable
- algorithm itself is build via one-step relation $\rightsquigarrow$ which is applied as long as possible

Unification Algorithm of Martelli and Montanari, continued

- input: unification problem $U = \{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$
- output: solution of U via solved form or $\bot$, indicating unsolvability
- algorithm applies $\rightsquigarrow$ as long as possible; $\rightsquigarrow$ is defined as

$$U \cup \{t \stackrel{?}{=} t\} \rightsquigarrow U \qquad \text{(delete)}$$

$$U \cup \{F(u_1, \ldots, u_k) \stackrel{?}{=} F(v_1, \ldots, v_k)\} \rightsquigarrow U \cup \{u_1 \stackrel{?}{=} v_1, \ldots, v_k \stackrel{?}{=} v_k\} \qquad \text{(decompose)}$$

$$U \cup \{F(u_1, \ldots, u_k) \stackrel{?}{=} G(v_1, \ldots, v_\ell)\} \rightsquigarrow \bot, \text{ if } F \neq G \vee k \neq \ell \qquad \text{(clash)}$$

$$U \cup \{F(\ldots) \stackrel{?}{=} x\} \rightsquigarrow U \cup \{x \stackrel{?}{=} F(\ldots)\} \qquad \text{(swap)}$$

$$U \cup \{x \stackrel{?}{=} F(\ldots)\} \rightsquigarrow \bot, \text{ if } x \in \mathcal{V}ars(F(\ldots)) \qquad \text{(occurs check)}$$

$$U \cup \{x \stackrel{?}{=} t\} \rightsquigarrow U\{x/t\} \cup \{x \stackrel{?}{=} t\}, \quad \text{if } x \notin \mathcal{V}ars(t) \text{ and } x \text{ occurs in } U \qquad \text{(eliminate)}$$

notation $U\{x/t\}$: apply substitution $\{x/t\}$ on all terms in U (lhs + rhs)

Correctness of Unification Algorithm

- we only state properties (proofs: see term rewriting lecture)
  - $\rightsquigarrow$ terminates
  - normal form of $\rightsquigarrow$ is $\bot$ or a solved form
  - whenever $U \rightsquigarrow V$, then U and V have same solutions
  - in total: to solve unification problem U
    - determine some normal form V of U
    - if $V = \bot$ then U is unsolvable
    - otherwise, V represents a substitution that is a solution to U
- note that $\rightsquigarrow$ is not confluent
  - $\{x \stackrel{?}{=} y, y \stackrel{?}{=} x\} \overset{x/y}{\rightsquigarrow} \{x \stackrel{?}{=} y, y \stackrel{?}{=} y\} \rightsquigarrow \{x \stackrel{?}{=} y\}$
  - $\{x \stackrel{?}{=} y, y \stackrel{?}{=} x\} \overset{y/x}{\rightsquigarrow} \{x \stackrel{?}{=} x, y \stackrel{?}{=} x\} \rightsquigarrow \{y \stackrel{?}{=} x\}$

Correctness of an Implementation of a (Unification) Algorithm

- any concrete implementation will make choices
  - preference of rules
  - selection of pairs from U
  - representation of sets U
  - (pivot-selection in quicksort)
  - (order of edges in graph-/tree-traversals)
  - ...
- task: how to ensure that implementation is sound
- solution: refinement proof
  - aim: reuse correctness of abstract algorithm ($\rightsquigarrow$)
  - define relation between representations in concrete and abstract algorithm (this was called alignment before and done informally)
  - show that concrete algorithm has less behaviour, i.e., every result of concrete (deterministic) algorithm can be related to some result of (non-deterministic) abstract algorithm
  - benefit: clear separation between
    - soundness of abstract algorithm (solves unification problems)
    - soundness of implementation (implements abstract algorithm)

A Concrete Implementing of the Unification Algorithm

```
subst :: Var -> Term -> Term -> Term
subst x t = applySubst (\ y -> if y == x then t else Var y)

unify :: [(Term, Term)] -> CheckS [(Var, Term)]
unify u = unifyMain u []

unifyMain :: [(Term, Term)] -> [(Var,Term)] -> CheckS [(Var, Term)]
unifyMain [] v = return v                                  -- return solved form
unifyMain ((Fun f ts, Fun g ss) : u) v = do
  assert (f == g && length ts == length ss)                -- clash
  unifyMain (zip ts ss ++ u) v                             -- decompose
unifyMain ((Fun f ts, x) : u) v =
  unifyMain ((x, Fun f ts) : u) v                          -- swap
unifyMain ((Var x, t) : u) v =
  if Var x == t then unifyMain u v                         -- delete
  else do
    assert (not (elem x (varsTerm t)))                     -- occurs check
    unifyMain                                              -- eliminate
      (map ( \ (l,r) -> (subst x t l, subst x t r)) u)
      ((x,t) : map ( \ (y, s) -> (y, subst x t s)) v)
```

Notes on Implementation

- it is non-trivial to prove soundness of implementation, since there are several differences w.r.t. $\rightsquigarrow$
  - $unifyMain$ takes two parameters u and v
    - these represent one unification problem $u \cup v$
  - rule-application is not tried on v, only on u
    - we need to know that v is in normal form w.r.t. $\rightsquigarrow$
  - in (occurs check)-rule, the algorithm has no test that rhs is function application
    - we need to show that this will follow from other conditions
  - in (elimination)-rule, the algorithm substitutes only in rhss of v
    - we need to know that substituting in lhss of v has no effect
  - in (elimination)-rule, the algorithm does not check that x occurs in remaining problem
    - we need to check that consequences don't harm

Soundness via Refinement: Setting up the Relation

- relation $\sim$ formally aligns parameters of concrete algorithm (u and v) with parameters of abstract algorithm (U); $\sim$ also includes invariants of implementation
  - set converts list to set, we identify $s \stackrel{?}{=} t$ with (s, t)
  - $(u, v) \sim U$ iff
    - $U = set\ u \cup set\ v$,
    - $set\ v$ is in normal form w.r.t. $\rightsquigarrow$ (notation: $set\ v \in NF(\rightsquigarrow)$), and
    - for all $(x, t) \in set\ v$: x does not occur in u
- since alignment between concrete and abstract parameters is specified formally, alignment properties of auxiliary functions can also be made formal
  - $set\ (x : xs) = \{x\} \cup set\ xs$
  - $set\ (xs +\!\!+\ ys) = set\ xs \cup set\ ys$
  - $set\ (zip\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n]) = \{(x_1, y_1), \ldots, (x_n, y_n)\}$
  - $set\ (map\ f\ [x_1, \ldots, x_n]) = \{f\ x_1, \ldots, f\ x_n\}$
  - $subst\ x\ t\ s = s\{x/t\}$
  - ...

  these properties can be proven formally and also be applied formally (although we don't do it in the upcoming proof)

Soundness via Refinement: Main Statement

- we need a function to align results in the algorithm to unification problems: check-to-unification problem (ctu)

$$ctu\ (return\ w) = set\ w \quad \text{and} \quad ctu\ (failure\ e) = \bot$$

- property: whenever $(u, v) \sim U$ and $unifyMain\ u\ v = res$ then $U \rightsquigarrow^!\ ctu\ res$
- once property is established, we can prove that implementation solves unification problems
  - assume input u, i.e., invocation of $unify\ u$ which yields result res
  - hence, $unifyMain\ u\ [] = res$
  - moreover, $(u, []) \sim set\ u$ by definition of $\sim$
  - via property conclude $set\ u \rightsquigarrow^!\ ctu\ res$
  - at this point apply correctness of $\rightsquigarrow$: $ctu\ res$ is the correct answer to the unification problem $set\ u$

Proving the Refinement Property

- property $P(u, v, U)$: $(u, v) \sim U \wedge unifyMain\ u\ v = res \longrightarrow U \rightsquigarrow^!\ ctu\ res$
- $(u, v) \sim U \longleftrightarrow U = set\ u \cup set\ v \wedge set\ v \in NF(\rightsquigarrow) \wedge \forall (x, t) \in set\ v.\ x \notin \mathcal{V}ars(u)$
- we prove the property $P(u, v, U)$ by induction on u and v w.r.t. the algorithm for arbitrary U, i.e., we consider all left-hand sides and can assume that the property holds for all recursive calls;
  induction w.r.t. algorithm gives partial correctness result (assumes termination)
- in the lecture, we will cover a simple, a medium, and the hardest case
- case 1 (arguments [] and v):
  - we have to prove $P([], v, U)$, so assume
    (*) $([], v) \sim U$ and
    (**) $unifyMain\ []\ v = res$
  - from (*) conclude $U = set\ v$ and $set\ v \in NF(\rightsquigarrow)$
  - from (**) conclude $res = return\ v$ and hence, $ctu\ res = set\ v$
  - we have to show $U \rightsquigarrow^!\ ctu\ res$, i.e., $set\ v \rightsquigarrow^!\ set\ v$ which is satisfied since $set\ v \in NF(\rightsquigarrow)$

- $P(u,v,U)$: $(u,v) \sim U \wedge \mathit{unifyMain}\ u\ v = \mathit{res} \longrightarrow U \rightsquigarrow^!\ \mathit{ctu}\ \mathit{res}$
- $(u,v) \sim U \longleftrightarrow U = \mathit{set}\ u \cup \mathit{set}\ v \wedge \mathit{set}\ v \in \mathit{NF}(\rightsquigarrow) \wedge \forall (x,t) \in \mathit{set}\ v.\ x \notin \mathcal{V}\mathit{ars}(u)$

case 2 (arguments $(F(ts), G(ss)) : u$ and v)

- we have to prove $P((F(ts), G(ss)) : u, v, U)$, so assume

  (*) $((F(ts), G(ss)) : u, v) \sim U$ and
  (**) $\mathit{unifyMain}\ ((F(ts), G(ss)) : u)\ v = \mathit{res}$

- consider sub-cases
  - $\neg(F = G \wedge \mathit{length}\ ts = \mathit{length}\ ss)$:
    - from (**) conclude $\mathit{ctu}\ \mathit{res} = \bot$
    - from (*) conclude $F(ts) \stackrel{?}{=} G(ss) \in U$ and hence $U \rightsquigarrow \bot$ by (clash)
    - consequently, $U \rightsquigarrow^!\ \mathit{ctu}\ \mathit{res}$
  - $F = G \wedge \mathit{length}\ ts = \mathit{length}\ ss$:
    - from (**) conclude $\mathit{res} = \mathit{unifyMain}\ ((F(ts), G(ss)) : u)\ v = \mathit{unifyMain}\ (\mathit{zip}\ ts\ ss +\!\!+ u)\ v$
    - from (*) and alignment for zip and $+\!\!+$ conclude $U = \{F(ts) \stackrel{?}{=} G(ss)\} \cup \mathit{set}\ u \cup \mathit{set}\ v$ and hence $U \rightsquigarrow \mathit{set}\ (\mathit{zip}\ ts\ ss +\!\!+ u) \cup \mathit{set}\ v =: V$ by (decompose)
    - we get $P(\mathit{zip}\ ts\ ss +\!\!+ u, v, V)$ as IH; $(\mathit{zip}\ ts\ ss +\!\!+ u, v) \sim V$ follows from (*), so $U \rightsquigarrow V \rightsquigarrow^!\ \mathit{ctu}\ \mathit{res}$

- $P(u,v,U)$: $(u,v) \sim U \wedge \mathit{unifyMain}\ u\ v = \mathit{res} \longrightarrow U \rightsquigarrow^!\ \mathit{ctu}\ \mathit{res}$
- $(u,v) \sim U \longleftrightarrow U = \mathit{set}\ u \cup \mathit{set}\ v \wedge \mathit{set}\ v \in \mathit{NF}(\rightsquigarrow) \wedge \forall (x,t) \in \mathit{set}\ v.\ x \notin \mathcal{V}\mathit{ars}(u)$

case 4 (arguments $(x,t) : u$ and v)

- we have to prove $P((x,t) : u, v, U)$, so assume

  (*) $((x,t) : u, v) \sim U$ and
  (**) $\mathit{unifyMain}\ ((x,t) : u)\ v = \mathit{res}$

- consider sub-cases (where the red part is not triggered by structure of algorithm)
  - $x \neq t \wedge x \notin \mathcal{V}\mathit{ars}(t) \wedge x$ occurs in $\mathit{set}\ u \cup \mathit{set}\ v$:
    - define $u' = \mathit{map}\ (\lambda(l,r).\ (\mathit{subst}\ x\ t\ l, \mathit{subst}\ x\ t\ r))\ u$
    - define $v' = \mathit{map}\ (\lambda(y,s).\ (y, \mathit{subst}\ x\ t\ s))\ v$
    - define $V = (\mathit{set}\ u \cup \mathit{set}\ v)\{x/t\} \cup \{x \stackrel{?}{=} t\}$
    - from (**) conclude $\mathit{res} = \mathit{unifyMain}\ ((x,t) : u)\ v = \mathit{unifyMain}\ u'\ ((x,t) : v')$
    - from IH conclude $P(u', (x,t) : v', V)$ and hence, $(u', (x,t) : v') \sim V \longrightarrow V \rightsquigarrow^!\ \mathit{ctu}\ \mathit{res}$
    - for proving $U \rightsquigarrow^!\ \mathit{ctu}\ \mathit{res}$ it hence suffices to show $(u', (x,t) : v') \sim V$ and $U \rightsquigarrow V$
    - $U \stackrel{(*)}{=} \{x \stackrel{?}{=} t\} \cup \mathit{set}\ u \cup \mathit{set}\ v \rightsquigarrow (\mathit{set}\ u \cup \mathit{set}\ v)\{x/t\} \cup \{x \stackrel{?}{=} t\} = V$ by (eliminate) because of preconditions

- $(u,v) \sim U \longleftrightarrow U = \mathit{set}\ u \cup \mathit{set}\ v \wedge \mathit{set}\ v \in \mathit{NF}(\rightsquigarrow) \wedge \forall (x,t) \in \mathit{set}\ v.\ x \notin \mathcal{V}\mathit{ars}(u)$

case 4 (arguments $(x,t) : u$ and v)

- we have to prove $P((x,t) : u, v, U)$, so assume (*) $((x,t) : u, v) \sim U$ and ...
  and consider sub-case $x \neq t \wedge x \notin \mathcal{V}\mathit{ars}(t) \wedge x$ occurs in $\mathit{set}\ u \cup \mathit{set}\ v$:
  - define $u' = \mathit{map}\ (\lambda(l,r).\ (\mathit{subst}\ x\ t\ l, \mathit{subst}\ x\ t\ r))\ u$
  - define $v' = \mathit{map}\ (\lambda(y,s).\ (y, \mathit{subst}\ x\ t\ s))\ v$
  - define $V = (\mathit{set}\ u \cup \mathit{set}\ v)\{x/t\} \cup \{x \stackrel{?}{=} t\}$
  - we still need to show $(u', (x,t) : v') \sim V$
  - since (*) holds, we know $\forall (y,s) \in \mathit{set}\ v.\ x \neq y$
  - hence, $v' = \mathit{map}\ (\lambda(y,s).\ (\mathit{subst}\ x\ t\ y, \mathit{subst}\ x\ t\ s))\ v$
  - so, $V = (\mathit{set}\ u)\{x/t\} \cup \{x \stackrel{?}{=} t\} \cup (\mathit{set}\ v)\{x/t\} = \mathit{set}\ u' \cup \mathit{set}\ ((x,t) : v')$
  - we show $\forall (y,s) \in \mathit{set}\ ((x,t) : v').\ y \notin \mathcal{V}\mathit{ars}(u')$ as follows:
    $x \notin \mathcal{V}\mathit{ars}(u')$ since $x \notin \mathcal{V}\mathit{ars}(t)$; and if $(y,s) \in \mathit{set}\ v'$, then $(y,s') \in \mathit{set}\ v$ for some s' and by (*) we conclude $y \notin \mathcal{V}\mathit{ars}((x,t) : u)$; thus, $y \notin \mathcal{V}\mathit{ars}((\mathit{set}\ u)\{x/t\}) = \mathcal{V}\mathit{ars}(u')$
  - we finally show $\mathit{set}\ ((x,t) : v') \in \mathit{NF}(\rightsquigarrow)$: so, assume to the contrary that some step is applicable; by the shape of $\mathit{set}\ ((x,t) : v')$ we know that the step can only be (eliminate), (delete) or (occurs check); all of these cases result in a contradiction by using the available facts

Proving the Refinement Property

- remaining cases: similar, cf. exercises
- summary
  - non-trivial implementation of abstract unification algorithm $\rightsquigarrow$
  - optimizations required additional invariants, encoded in refinement relation
  - proof of correctness can be done formally
    - induction + case analysis proof uses mostly the structure of the Haskell code;
      exception: case analysis on "x occurs in $\mathit{set}\ u \cup \mathit{set}\ v$"
    - most cases can easily be solved, after having identified suitable invariants
    - fully reuse correctness of $\rightsquigarrow$
  - we only proved partial correctness
  - termination of implementation: consider lexicographic measure

$$(\underbrace{|\mathcal{V}\mathit{ars}(\mathit{set}\ u)|}_{(\mathit{eliminate})}, \underbrace{|u|}_{(\mathit{decomp}),(\mathit{delete})}, \underbrace{\mathit{length}\ [x \mid (t, \mathit{Var}\ x) \leftarrow u]}_{(\mathit{swap})})$$

Checking Pattern Completeness

Pattern Problems

- reminder: program is pattern complete, if for all $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \mathcal{D}$ and all $t_i \in \mathcal{T}(\mathcal{C})_{\tau_i}$ there is some lhs that matches $f(t_1, \ldots, t_n)$
- algorithm considers more generic shape
  - matching problems mp consist of pairs of terms (ℓ, t) where
    - ℓ is (a subterm of) some lhs
    - t is a term, representing the set of all its constructor ground instances, e.g., $t = f(x_1, \ldots, x_n)$
    - semantics: find one substitution σ such that $\ell\sigma = t\gamma$ for all $(\ell, t) \in mp$ where γ is some constructor ground substitution
  - pattern problems pp consist of multiple matching problems
    - semantics: disjunction, i.e., find one suitable matching problem
    - reason for disjunction: a term t might be matched by arbitrary lhs
    - initially: $pp = \{\{(\ell_1, t)\}, \ldots, \{(\ell_n, t)\}\}$ for lhss $\ell_1, \ldots, \ell_n$
  - sets of pattern problems P consist of several pattern problems
    - semantics: conjunction
    - reason: consider different ground instances and different defined function symbols
    - initial set of pattern problems: $P_{init} = \{\{\{(\ell, f(x_1, \ldots, x_n))\} \mid \ell \text{ is lhs of } f\text{-eqn.}\} \mid f \in \mathcal{D}\}$
  - overall semantics: P is complete iff

$$\forall pp \in P.\, \forall \gamma : \mathcal{V} \to \mathcal{T}(\mathcal{C}).\, \exists mp \in pp.\, \exists \sigma.\, \forall (\ell, t) \in mp.\, \ell\sigma = t\gamma$$

Reformulation of Pattern Completeness of Programs

- definitions of previous slide (omitting types)
  - program is pattern complete iff for all $f \in \mathcal{D}$ and all $t_i \in \mathcal{T}(\mathcal{C})$ there is some lhs that matches $f(t_1, \ldots, t_n)$
  - $P_{init} = \{\{\{(\ell, f(x_1, \ldots, x_n))\} \mid \ell \text{ is lhs of } f\text{-equation}\} \mid f \in \mathcal{D}\}$
  - P is complete iff $\forall pp \in P.\, \forall \gamma : \mathcal{V} \to \mathcal{T}(\mathcal{C}).\, \exists mp \in pp.\, \exists \sigma.\, \forall (\ell, t) \in mp.\, \ell\sigma = t\gamma$
- corollary: program is pattern complete iff P_{init} is complete

Task: determine completeness of pattern problems

- algorithm modifies matching problems and (sets of) pattern problems
- special problems: $\bot$ represents a non-solvable matching problem and an incomplete set of pattern problems, and $\top$ represents a complete pattern problem
- here: only consider linear pattern problems, i.e., problems where variables in lhss of programs occur at most once

Transforming Matching and Pattern Problems

$$\{(F(\ell_1, \ldots, \ell_n), F(t_1, \ldots, t_n))\} \uplus mp \rightharpoonup \{(\ell_1, t_1), \ldots, (\ell_n, t_n)\} \cup mp \qquad \text{(decompose)}$$

$$\{(x, t)\} \uplus mp \rightharpoonup mp \qquad \text{(match)}$$

$$\{(F(\ldots), G(\ldots))\} \uplus mp \rightharpoonup \bot \quad \text{if } F \neq G \qquad \text{(clash)}$$

$$\{mp\} \uplus pp \rightharpoonup\!\!\!\rightharpoonup \{mp'\} \cup pp \quad \text{if } mp \rightharpoonup mp' \qquad \text{(simp-mp)}$$

$$\{\bot\} \uplus pp \rightharpoonup\!\!\!\rightharpoonup pp \qquad \text{(remove-mp)}$$

$$\{\varnothing\} \uplus pp \rightharpoonup\!\!\!\rightharpoonup \top \qquad \text{(success)}$$

$$\{pp\} \uplus P \rightharpoonup\!\!\!\!\rightharpoonup \{pp'\} \cup P \quad \text{if } pp \rightharpoonup\!\!\!\rightharpoonup pp' \qquad \text{(simp-pp)}$$

$$\{\varnothing\} \uplus P \rightharpoonup\!\!\!\!\rightharpoonup \bot \qquad \text{(failure)}$$

$$\{\top\} \uplus P \rightharpoonup\!\!\!\!\rightharpoonup P \qquad \text{(remove-pp)}$$

$$\{pp\} \uplus P \rightharpoonup\!\!\!\!\rightharpoonup Inst(pp, x) \cup P \quad \text{if } mp \in pp \text{ and } (F(\ldots), x) \in mp \qquad \text{(instantiate)}$$

where $Inst(pp, x)$ contains a pattern problem $pp\sigma_{x,c}$ for each constructor c where

- $x : \tau$ and $c : \tau_1 \times \cdots \times \tau_n \to \tau$ and $x_1 : \tau_1, \ldots, x_n : \tau_n$ are fresh, and
- $pp\sigma_{x,c}$ is obtained from pp by replacing each pair (ℓ, t) by $(\ell, t\{x/c(x_1, \ldots, x_n)\})$

Example

consider

$$\begin{aligned} \mathsf{data}\ \mathsf{Bool} = \mathsf{True} : \mathsf{Bool} \mid \mathsf{False} : \mathsf{Bool} \\ \ell_1 := \mathsf{conj}(\mathsf{True}, \mathsf{True}) = \ldots \\ \ell_2 := \mathsf{conj}(\mathsf{False}, y) = \ldots \\ \ell_3 := \mathsf{conj}(x, \mathsf{False}) = \ldots \end{aligned}$$

then we have

$$\begin{aligned} P_{init} &= \{\{(\ell_1, \mathsf{conj}(x_1, x_2))\}, \{(\ell_2, \mathsf{conj}(x_1, x_2))\}, \{(\ell_3, \mathsf{conj}(x_1, x_2))\}\} \\ &\rightsquigarrow^* \{\{(\mathsf{True}, x_1), (\mathsf{True}, x_2)\}, \{(\mathsf{False}, x_1), (y, x_2)\}, \{(x, x_1), (\mathsf{False}, x_2)\}\} \\ &\rightsquigarrow^* \{\{(\mathsf{True}, x_1), (\mathsf{True}, x_2)\}, \{(\mathsf{False}, x_1)\}, \{(\mathsf{False}, x_2)\}\} \\ &\rightsquigarrow \{\{(\mathsf{True}, \mathsf{True}), (\mathsf{True}, x_2)\}, \{(\mathsf{False}, \mathsf{True})\}, \{(\mathsf{False}, x_2)\}\}, \\ &\quad \{\{(\mathsf{True}, \mathsf{False}), (\mathsf{True}, x_2)\}, \{(\mathsf{False}, \mathsf{False})\}, \{(\mathsf{False}, x_2)\}\} \\ &\rightsquigarrow^* \{\{(\mathsf{True}, x_2)\}, \bot, \{(\mathsf{False}, x_2)\}\},\ \{\bot, \varnothing, \{(\mathsf{False}, x_2)\}\} \\ &\rightsquigarrow^* \{\{(\mathsf{True}, x_2)\}, \{(\mathsf{False}, x_2)\}\} \\ &\rightsquigarrow \{\{(\mathsf{True}, \mathsf{True})\}, \{(\mathsf{False}, \mathsf{True})\}\},\ \{\{(\mathsf{True}, \mathsf{False})\}, \{(\mathsf{False}, \mathsf{False})\}\} \rightsquigarrow^* \varnothing \end{aligned}$$

Example

consider

$$\begin{aligned} \mathsf{data}\ \mathsf{Bool} = \mathsf{True} : \mathsf{Bool} \mid \mathsf{False} : \mathsf{Bool} \\ \ell_1 := \mathsf{conj}(\mathsf{True}, \mathsf{True}) = \ldots \\ \ell_2 := \mathsf{conj}(\mathsf{False}, y) = \ldots \end{aligned}$$

then we have

$$\begin{aligned} P_{init} &= \{\{(\ell_1, \mathsf{conj}(x_1, x_2))\}, \{(\ell_2, \mathsf{conj}(x_1, x_2))\}\} \\ &\rightsquigarrow^* \{\{(\mathsf{True}, x_1), (\mathsf{True}, x_2)\}, \{(\mathsf{False}, x_1)\}\} \\ &\rightsquigarrow \{\{(\mathsf{True}, \mathsf{True}), (\mathsf{True}, x_2)\}, \{(\mathsf{False}, \mathsf{True})\}, \\ &\quad \{\{(\mathsf{True}, \mathsf{False}), (\mathsf{True}, x_2)\}, \{(\mathsf{False}, \mathsf{False})\}\} \\ &\rightsquigarrow^* \{\{(\mathsf{True}, x_2)\}, \bot\},\ \{\varnothing, \bot\}\} \\ &\rightsquigarrow^* \{\{(\mathsf{True}, x_2)\}\} \\ &\rightsquigarrow \{\{(\mathsf{True}, \mathsf{True})\}\},\ \{\{(\mathsf{True}, \mathsf{False})\}\} \rightsquigarrow^* \{\top, \varnothing\} \rightsquigarrow \bot \end{aligned}$$

Partial Correctness of $\rightsquigarrow$

- theorem: whenever $P \rightsquigarrow Q$, then P is complete iff Q is complete
- corollary: if $P \rightsquigarrow^* \varnothing$ then P is complete,
  and if $P \rightsquigarrow^* \bot$ then P is not complete
- definition: P is complete iff
  $\forall pp \in P.\, \forall \gamma : \mathcal{V} \to \mathcal{T}(\mathcal{C}).\ \underbrace{\exists mp \in pp.\, \exists \sigma.\, \forall (\ell, t) \in mp.\, \ell\sigma = t\gamma}_{=:\psi}$
- proof of theorem by case analysis on the various rules
  - (clash): first inline rule to $\{\{(F(\ldots), G(\ldots))\} \uplus mp\} \uplus pp\} \uplus P \rightsquigarrow \{pp\} \cup P$, if $F \neq G$
    - by definition of completeness and structure of rule it suffices to show that completeness is preserved by rule
      $$\underbrace{\{\{(F(\ldots), G(\ldots))\} \uplus mp\}}_{=:mp'} \uplus pp \rightsquigarrow pp$$
    - hence, it suffices to show that ψ is not satisfied when choosing mp' in the existential quantifier $\exists mp \in pp. \ldots$
    - but this property is easy to see, since $\ell\sigma = t\gamma$ is never satisfied if (ℓ, t) is $(F(\ldots), G(\ldots))$
  - many other rules are similar, exceptions are (match) and (instantiate)

Partial Correctness of $\rightsquigarrow$, continued

- definition: P is complete iff
  $\forall pp \in P.\, \forall \gamma : \mathcal{V} \to \mathcal{T}(\mathcal{C}).\, \exists mp \in pp.\, \exists \sigma.\, \forall (\ell, t) \in mp.\, \ell\sigma = t\gamma$
- proof continued
  - (instantiate): $\{pp\} \uplus P \rightsquigarrow Inst(pp, x) \cup P$, where $x : \tau$, τ has constructors $c_1, \ldots, c_n$, and $\sigma_i = \{x/c_i(x_1, \ldots, x_k)\}$ for fresh x_i, and $Inst(pp, x) = \{pp\sigma_i \mid 1 \leq i \leq n\}$
    - we only consider one direction of the proof: we assume that $Inst(pp, x)$ is complete and prove that pp is complete
    - to this end, consider an arbitrary constructor ground substitution γ
    - since γ is constructor ground, we know $\gamma(x) = c_i(t_1, \ldots, t_k)$ for some constructor c_i and constructor ground terms $t_1, \ldots, t_k$
    - define $\gamma'(y) = \begin{cases} t_j, & \text{if } y = x_j \\ \gamma(y), & \text{otherwise} \end{cases}$
    - γ' is well-defined since the x_j's are distinct, and γ' is a constructor ground substitution
    - note that $t\gamma = t\gamma_i\gamma'$ for all terms t that occur in pp since the x_j's are fresh
    - by completeness of $Inst(pp, x)$ there must be some $mp \in pp\gamma_i$ and σ such that $\forall (\ell, t) \in mp.\, \ell\sigma = t\gamma'$
    - hence, there is some $mp \in pp$ and σ such that $\forall (\ell, t) \in mp.\, \ell\sigma = t\gamma_i\gamma'$
    - together with $t\gamma = t\gamma_i\gamma'$ we conclude that pp is complete

Correctness of $\rightsquigarrow$, Missing Parts

- already proven
  - if $P \rightsquigarrow^* \varnothing$ then P is complete
  - if $P \rightsquigarrow^* \bot$ then P is not complete
- open: termination of $\rightsquigarrow$
- open: can $\rightsquigarrow$ get stuck?

$\rightsquigarrow$ Cannot Get Stuck

$$\{(F(\ell_1,\ldots,\ell_n), F(t_1,\ldots,t_n))\} \uplus mp \rightharpoonup \{(\ell_1,t_1),\ldots,(\ell_n,t_n)\} \cup mp \quad \text{(decompose)}$$
$$\{(x,t)\} \uplus mp \rightharpoonup mp \quad \text{(match)}$$
$$\{(F(\ldots), G(\ldots))\} \uplus mp \rightharpoonup \bot \quad \text{if } F \neq G \quad \text{(clash)}$$
$$\{mp\} \uplus pp \rightharpoonup \{mp'\} \cup pp \quad \text{if } mp \rightharpoonup mp' \quad \text{(simp-mp)}$$
$$\{\bot\} \uplus pp \rightharpoonup pp \quad \text{(remove-mp)}$$
$$\{\varnothing\} \uplus pp \rightharpoonup \top \quad \text{(success)}$$
$$\{pp\} \uplus P \rightsquigarrow \{pp'\} \cup P \quad \text{if } pp \rightharpoonup pp' \quad \text{(simp-pp)}$$
$$\{\varnothing\} \uplus P \rightsquigarrow \bot \quad \text{(failure)}$$
$$\{\top\} \uplus P \rightsquigarrow P \quad \text{(remove-pp)}$$
$$\{pp\} \uplus P \rightsquigarrow Inst(pp,x) \cup P \quad \text{if } mp \in pp \text{ and } (F(\ldots),x) \in mp \quad \text{(instantiate)}$$

- lemma: whenever P is well-typed and in normal form w.r.t. $\rightsquigarrow$, then $P \in \{\varnothing, \bot\}$
- proof: by a large case-analysis

Termination of $\rightsquigarrow$

$$\{pp\} \uplus P \rightsquigarrow \{pp'\} \cup P \quad \text{if } pp \rightharpoonup pp' \quad \text{(simp-pp)}$$
$$\{\varnothing\} \uplus P \rightsquigarrow \bot \quad \text{(failure)}$$
$$\{\top\} \uplus P \rightsquigarrow P \quad \text{(remove-pp)}$$
$$\{pp\} \uplus P \rightsquigarrow Inst(pp,x) \cup P \quad \text{if } mp \in pp \text{ and } (F(\ldots),x) \in mp \quad \text{(instantiate)}$$

- define $|\ell - t|$ as a measure of difference of ℓ and t
  - $|\ell - x|$ = number of function symbols in ℓ
  - $|F(\ell_1,\ldots,\ell_n) - F(t_1,\ldots,t_n)| = \sum_i |\ell_i - t_i|$
  - $|\ell - t| = 0$, in all other cases
- map each pattern problem pp to number $|pp| = \sum_{mp \in pp, (\ell,t) \in mp} |\ell - t|$
- map each set of pattern problem P to multiset $\{|pp| \mid pp \in P\}$
- this multiset decreases in (instantiate) and is not increased in the other $\rightsquigarrow$-rules (multiset decrease: $M \cup N >^{mul} M \cup N'$ if $N \neq \varnothing$ and $\forall y \in N'.\, \exists x \in N.\, x > y$)
- hence (instantiate) cannot be applied infinitely often
- since the remaining rules also terminate, $\rightsquigarrow$ must terminate

Implementation and Complexity of $\rightsquigarrow$

- clearly, $\rightsquigarrow$ is formulated abstractly
- a concrete implementation has to use a concrete representation for matching- and pattern problems; it has to resolve non-determinism, e.g., order of rules, selection of instantiation variables, etc.
- theorem: deciding pattern completeness is co-NP-hard
- consequence: worst-case complexity on required number of $\rightsquigarrow$-steps unlikely to be sub-exponential
- fully verified implementation exists
- currently fastest known algorithm for pattern completeness, developed for this lecture

Summary on Pattern Completeness

- pattern completeness of functional programs is decidable:

  program is pattern complete iff $P_{init} \twoheadrightarrow^{!} \varnothing$
- two possible extensions
  - generation of counter-examples
  - handling of non-linear pattern problems
- partial correctness was proven via invariant of $\twoheadrightarrow$
- termination of $\twoheadrightarrow$ was shown via multisets and a dedicated measure
- termination proof was tricky, definitely required human interaction
- in contrast: upcoming part is on automated termination proving

Termination – Dependency Pairs

Termination of Programs

- the question of termination is a famous problem
  - Turing showed that "halting problem" is undecidable
  - halting problem
    - question: does program (Turing machine) terminate on given input
    - problem is semi-decidable: positive instances can always be identified
    - algorithm: just simulate the program and then say "yes, terminates"
- we here consider universal termination, i.e., termination on all inputs
- universal termination is not even semi-decidable
- despite theoretical limits: often termination can be proven automatically

Termination of Functional Programs

- for termination, we mainly consider functional programs which are pattern-disjoint; hence, $\hookrightarrow$ is confluent
- consequence: it suffices to prove innermost termination, i.e., the restriction of $\hookrightarrow$ such that arguments t_i will be fully evaluated before evaluating a function invocation $f(t_1, \ldots, t_n)$
- example without confluence

$$\begin{aligned} \mathsf{f}(\mathsf{True}, \mathsf{False}, x) &= \mathsf{f}(x, x, x) \\ \mathsf{f}(\ldots, \ldots, x) &= x \qquad \text{(all other cases)} \\ \mathsf{coin} &= \mathsf{True} \\ \mathsf{coin} &= \mathsf{False} \end{aligned}$$

- both f and coin terminate if seen as separate programs
- program is innermost terminating, but not terminating in general

$$\mathsf{f}(\mathsf{True}, \mathsf{False}, \mathsf{coin}) \hookrightarrow \mathsf{f}(\mathsf{coin}, \mathsf{coin}, \mathsf{coin}) \hookrightarrow^2 \mathsf{f}(\mathsf{True}, \mathsf{False}, \mathsf{coin}) \hookrightarrow \ldots$$

Subterm Relation and Innermost Evaluation

- define $\rhd$ as the strict subterm relation and $\trianglerighteq$ as its reflexive closure

$$\frac{}{F(t_1,\ldots,t_n) \rhd t_i} \qquad \frac{t_i \rhd s}{F(t_1,\ldots,t_n) \rhd s}$$

- innermost evaluation $\xhookrightarrow{i}$ is defined similar to one-step evaluation $\hookrightarrow$

$$\frac{s_i \xhookrightarrow{i} t_i}{F(s_1,\ldots,s_i,\ldots,s_n) \xhookrightarrow{i} F(s_1,\ldots,t_i,\ldots,s_n)} \text{ rewrite in context}$$

$$\frac{\ell = r \text{ is equation in program} \quad \forall s \lhd \ell\sigma.\ s \in NF(\hookrightarrow)}{\ell\sigma \xhookrightarrow{i} r\sigma} \text{ root step}$$

- example

$$\mathsf{f}(\mathsf{True},\mathsf{False},\mathsf{coin}) \not\xhookrightarrow{i} \mathsf{f}(\mathsf{coin},\mathsf{coin},\mathsf{coin})$$

since $\mathsf{coin} \lhd \mathsf{f}(\mathsf{True},\mathsf{False},\mathsf{coin})$ and $\mathsf{coin} \notin NF(\hookrightarrow)$

Strong Normalization

- relation $\succ$ is strongly normalizing, written $SN(\succ)$, if there is no infinite sequence

$$a_1 \succ a_2 \succ a_3 \succ \ldots$$

- strong normalization is other notion for termination
- strong normalization of a relation is equivalent to soundness of induction principle w.r.t. that relation;
  the following two conditions are equivalent
  - $SN(\succ)$
  - $\forall P.\ (\forall x.\ (\forall y.\ x \succ y \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall x.\ P\ x)$
- equivalence shows why it is possible to perform induction w.r.t. algorithm for terminating programs

Termination Analysis with Dependency Pairs

- aim: prove $SN(\xhookrightarrow{i})$
- only reason for potential non-termination: recursive calls
- for each recursive call of equation $f(t_1,\ldots,t_n) = \ell = r \trianglerighteq f(s_1,\ldots,s_n)$ build one dependency pair with fresh (constructor) symbol $f^\sharp$:

$$f^\sharp(t_1,\ldots,t_n) \to f^\sharp(s_1,\ldots,s_n)$$

define DP as the set of all dependency pairs

- example program for Ackermann function has three dependency pairs

$$\begin{aligned}
\mathsf{ack}(\mathsf{Zero},y) &= \mathsf{Succ}(y)\\
\mathsf{ack}(\mathsf{Succ}(x),\mathsf{Zero}) &= \mathsf{ack}(x,\mathsf{Succ}(\mathsf{Zero}))\\
\mathsf{ack}(\mathsf{Succ}(x),\mathsf{Succ}(y)) &= \mathsf{ack}(x,\mathsf{ack}(\mathsf{Succ}(x),y))\\
\mathsf{ack}^\sharp(\mathsf{Succ}(x),\mathsf{Zero}) &\to \mathsf{ack}^\sharp(x,\mathsf{Succ}(\mathsf{Zero}))\\
\mathsf{ack}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) &\to \mathsf{ack}^\sharp(x,\mathsf{ack}(\mathsf{Succ}(x),y))\\
\mathsf{ack}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) &\to \mathsf{ack}^\sharp(\mathsf{Succ}(x),y)
\end{aligned}$$

Termination Analysis with Dependency Pairs, continued

- dependency pairs provide characterization of termination
- definition: let $P \subseteq DP$; a P-chain is a possible infinite sequence

$$s_1\sigma_1 \to t_1\sigma_1 \xhookrightarrow{i}{}^* s_2\sigma_2 \to t_2\sigma_2 \xhookrightarrow{i}{}^* s_3\sigma_3 \to t_3\sigma_3 \xhookrightarrow{i}{}^* \ldots$$

such that all $s_i \to t_i \in P$ and all $s_i\sigma_i \in NF(\hookrightarrow)$

  - $s_i\sigma_i \to t_i\sigma_i$ represent the "main" recursive calls that may lead to non-termination
  - $t_i\sigma_i \xhookrightarrow{i}{}^* s_{i+1}\sigma_{i+1}$ corresponds to evaluation of arguments of recursive calls
- theorem: $SN(\xhookrightarrow{i})$ iff there is no infinite DP-chain
- advantage of dependency pairs
  - in infinite chain, non-terminating recursive calls are always applied at the root
  - simplifies termination analysis

Example of Evaluation and Chain

$$\begin{aligned}
\mathsf{minus}(x, \mathsf{Zero}) &= x \\
\mathsf{minus}(\mathsf{Succ}(x), \mathsf{Succ}(y)) &= \mathsf{minus}(x, y) \\
\mathsf{div}(\mathsf{Zero}, \mathsf{Succ}(y)) &= \mathsf{Zero} \\
\mathsf{div}(\mathsf{Succ}(x), \mathsf{Succ}(y)) &= \mathsf{Succ}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{Succ}(y))) \\
\mathsf{minus}^\sharp(\mathsf{Succ}(x), \mathsf{Succ}(y)) &\to \mathsf{minus}^\sharp(x, y) \\
\mathsf{div}^\sharp(\mathsf{Succ}(x), \mathsf{Succ}(y)) &\to \mathsf{div}^\sharp(\mathsf{minus}(x, y), \mathsf{Succ}(y))
\end{aligned}$$

- example innermost evaluation

$$\begin{aligned}
&\mathsf{div}(\mathsf{Succ}(\mathsf{Zero}), \mathsf{Succ}(\mathsf{Zero})) \\
&\xhookrightarrow{i} \mathsf{Succ}(\mathsf{div}(\mathsf{minus}(\mathsf{Zero}, \mathsf{Zero}), \mathsf{Succ}(\mathsf{Zero}))) \\
&\xhookrightarrow{i} \mathsf{Succ}(\mathsf{div}(\mathsf{Zero}, \mathsf{Succ}(\mathsf{Zero}))) \\
&\xhookrightarrow{i} \mathsf{Succ}(\mathsf{Zero})
\end{aligned}$$

  and its (partial) representation as DP-chain

$$\begin{aligned}
&\mathsf{div}^\sharp(\mathsf{Succ}(\mathsf{Zero}), \mathsf{Succ}(\mathsf{Zero})) \\
&\to \mathsf{div}^\sharp(\mathsf{minus}(\mathsf{Zero}, \mathsf{Zero}), \mathsf{Succ}(\mathsf{Zero})) \\
&\xhookrightarrow{i}{}^* \mathsf{div}^\sharp(\mathsf{Zero}, \mathsf{Succ}(\mathsf{Zero}))
\end{aligned}$$

Proving Termination

- global approaches
  - try to find one termination argument that no infinite chain exists
- iterative approaches
  - identify dependency pairs that are harmless, i.e., cannot be used infinitely often in a chain
  - remove harmless dependency pairs from set of dependency pairs
  - until no dependency pairs are left
- we focus on iterative approaches, in particular those that are incremental
  - incremental: a termination proof of some function stays valid if later on other functions are added to the program
  - incremental termination proving is not possible in general case (for non-confluent programs), consider coin-example on slide 48

Termination – Subterm Criterion

A First Termination Technique – The Subterm Criterion

- the subterm criterion works as follows
  - let $P \subseteq DP$
  - choose $f^\sharp$, a symbol of arity n
  - choose some argument position $i \in \{1, \ldots, n\}$
  - demand $s_i \trianglerighteq t_i$ for all $f^\sharp(s_1, \ldots, s_n) \to f^\sharp(t_1, \ldots, t_n) \in P$
  - define $P_\triangleright = \{f^\sharp(s_1, \ldots, s_n) \to f^\sharp(t_1, \ldots, t_n) \in P \mid s_i \triangleright t_i\}$
  - then for proving absence of infinite P-chains it suffices to prove absence of infinite $P \setminus P_\triangleright$-chains, i.e., one can remove all pairs in $P_\triangleright$
- observations
  - easy to test: just find argument position i such that each i-th argument of all $f^\sharp$-dependency pairs decreases w.r.t. $\trianglerighteq$ and then remove all strictly decreasing pairs
  - incremental method: adding other dependency pairs for $g^\sharp$ later on will have no impact
  - can be applied iteratively
  - fast, but limited power

Subterm Criterion – Example

- consider a program with the following set of dependency pairs

$$\mathsf{ack}^\sharp(\mathsf{Succ}(x),\mathsf{Zero}) \to \mathsf{ack}^\sharp(x,\mathsf{Succ}(\mathsf{Zero})) \quad (1)$$
$$\mathsf{ack}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) \to \mathsf{ack}^\sharp(x,\mathsf{ack}(\mathsf{Succ}(x),y)) \quad (2)$$
$$\mathsf{ack}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) \to \mathsf{ack}^\sharp(\mathsf{Succ}(x),y) \quad (3)$$
$$\mathsf{minus}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) \to \mathsf{minus}^\sharp(x,y) \quad (4)$$
$$\mathsf{div}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) \to \mathsf{div}^\sharp(\mathsf{minus}(x,y),\mathsf{Succ}(y)) \quad (5)$$
$$\mathsf{plus}^\sharp(\mathsf{Succ}(x),y) \to \mathsf{plus}^\sharp(y,x) \quad (6)$$

- it is easy to remove (4) by choosing any argument of $\mathsf{minus}^\sharp$
- we can remove (1) and (2) by choosing argument 1 of $\mathsf{ack}^\sharp$
- afterwards we can remove (3) by choosing argument 2 of $\mathsf{ack}^\sharp$
- it is not possible to remove any of the remaining dependency pairs (5) and (6) by the subterm criterion

Subterm Criterion – Soundness Proof

- assume the chosen parameters in the subterm criterion are $f^\sharp$ and i
- it suffices to prove that there is no infinite chain

$$s_1\sigma_1 \to t_1\sigma_1 \xrightarrow{\trianglerighteq i}{}^* s_2\sigma_2 \to t_2\sigma_2 \xrightarrow{\trianglerighteq i}{}^* s_3\sigma_3 \to t_3\sigma_3 \xrightarrow{\trianglerighteq i}{}^* \dots$$

  such that all $s_j \to t_j \in P$, all s_j and t_j have $f^\sharp$ as root and there are infinitely many $s_j \to t_j \in P_\triangleright$; perform proof by contradiction
- hence all $s_j \to t_j$ are of the form $f^\sharp(s_{j,1},\dots,s_{j,n}) \to f^\sharp(t_{j,1},\dots,t_{j,n})$
- from condition $s_{j,i} \trianglerighteq t_{j,i}$ of criterion conclude $s_{j,i}\sigma_j \trianglerighteq t_{j,i}\sigma_j$ and if $s_j \to t_j \in P_\triangleright$ then $s_{j,i} \triangleright t_{j,i}$ and thus $s_{j,i}\sigma_j \triangleright t_{j,i}\sigma_j$
- we further know $t_{j,i}\sigma_j \xrightarrow{\trianglerighteq i}{}^* s_{j+1,i}\sigma_{j+1}$ since $f^\sharp$ is a constructor
- this implies $t_{j,i}\sigma_j = s_{j+1,i}\sigma_{j+1}$ since $t_{j,i}\sigma_j \in NF(\hookrightarrow)$ as $t_{j,i}\sigma_j \trianglelefteq s_{j,i}\sigma_j \triangleleft f^\sharp(s_{j,1}\sigma_j,\dots,s_{j,n}\sigma_j) = s_j\sigma_j \in NF(\hookrightarrow)$
- obtain an infinite sequence with infinitely many $\triangleright$; this is a contradiction to $SN(\triangleright)$

$$s_{1,i}\sigma_1 \trianglerighteq t_{1,i}\sigma_1 = s_{2,i}\sigma_2 \trianglerighteq t_{2,i}\sigma_2 = s_{3,i}\sigma_3 \trianglerighteq t_{3,i}\sigma_3 = \dots$$

Termination – Size-Change Principle

The Size-Change Principle

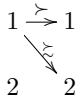
- the size-change principle abstracts decreases of arguments into size-change graphs
- size-change graph
  - let $f^\sharp$ be a symbol of arity n
  - a size-change graph for $f^\sharp$ is a bipartite graph $G = (V, W, E)$
  - the nodes are $V = \{1_{in},\dots,n_{in}\}$ and $W = \{1_{out},\dots,n_{out}\}$
  - E is a set of directed edges between in- and out-nodes labelled with $\succ$ or $\succsim$
  - the size-change graph G of a dependency pair $f^\sharp(s_1,\dots,s_n) \to f^\sharp(t_1,\dots,t_n)$ defines E as follows
    - $i_{in} \xrightarrow{\succ} j_{out} \in E$ whenever $s_i \triangleright t_j$ (strict decrease)
    - $i_{in} \xrightarrow{\succsim} j_{out} \in E$ whenever $s_i = t_j$ (weak decrease)
- in representation, in-nodes are on the left, out-nodes are on the right, and subscripts are omitted

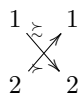
Example – Size-Change Graphs

- consider the following dependency pairs; they include permutations that cannot be solved by the subterm criterion

$$\mathsf{f}^\sharp(\mathsf{Succ}(x), y) \to \mathsf{f}^\sharp(x, \mathsf{Succ}(x)) \quad (7)$$
$$\mathsf{f}^\sharp(x, \mathsf{Succ}(y)) \to \mathsf{f}^\sharp(y, x) \quad (8)$$

- obtain size-change graphs that contain more information than just the size-decrease in one argument, as we had in subterm criterion

$G_{(7)}$: 

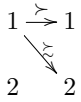
$G_{(8)}$: 

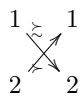
Multigraphs and Concatenation

- graphs can be glued together, tracing size-changes in chains, i.e., subsequent dependency pairs
- definition: let $\mathcal{G}$ be a set of size-change graphs for the same symbol $f^\sharp$; then the set of multigraphs for $f^\sharp$ is defined as follows
  - every $G \in \mathcal{G}$ is a multigraph
  - whenever there are multigraphs G_1 and G_2 with edges E_1 and E_2 then also the concatenated graph $G = G_1 \bullet G_2$ is a multigraph; here, the edges of E of G are defined as
    - if $i \to j \in E_1$ and $j \to k \in E_2$, then $i \to k \in E$
    - if at least one of the edges $i \to j$ and $j \to k$ is labeled with $\succ$ then $i \to k$ is labeled with $\succ$, otherwise with $\succsim$
    - if the previous rules would produce two edges $i \overset{\succ}{\to} k$ and $i \overset{\succsim}{\to} k$, then only the former is added to E
- a multigraph G is maximal if $G = G \bullet G$
- since there are only finitely many possible sets of edges, the set of multigraphs is finite and can easily be computed

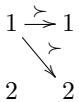
Example – Multigraphs

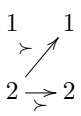
- consider size-change graphs

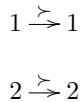
$G_{(7)}$: 

$G_{(8)}$: 

- this leads to three maximal multigraphs

$G_{(7)} \bullet G_{(8)}$: 

$G_{(8)} \bullet G_{(7)}$: 

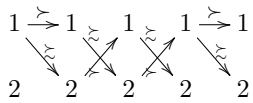
$G_{(8)} \bullet G_{(8)}$: 

- and a non-maximal multigraph

$G_{(8)} \bullet G_{(8)} \bullet G_{(8)}$: 

Size-Change Termination

- instead of multigraphs, one can also glue two graphs G_1 and G_2 by just identifying the out-nodes of G_1 with the in-nodes of G_2, defined as $G_1 \circ G_2$; in this way it is also possible to consider an infinite sequence of graphs $G_1 \circ G_2 \circ G_3 \circ \dots$
- example:

$G_{(7)} \circ G_{(8)} \circ G_{(8)} \circ G_{(7)}$: 

- definition: a set $\mathcal{G}$ of size-change graph is size-change terminating iff for every infinite concatenation of graphs of $\mathcal{G}$ there is a path with infinitely many $\overset{\succ}{\to}$-edges
- theorem: let P be a set of dependency pairs for symbol $f^\sharp$ and $\mathcal{G}$ be the corresponding size-change graphs; if $\mathcal{G}$ is size-change terminating, then there is no infinite P-chain
- the proof is mostly identical to the one of the subterm criterion

Deciding Size-Change Termination

- definition: a set $\mathcal{G}$ of size-change graph is size-change terminating iff for every infinite concatenation of graphs of $\mathcal{G}$ there is a path with infinitely many $\overset{\succ}{\to}$-edges
- checking size-change termination directly is not possible
- still, size-change termination is decidable
- theorem: let $\mathcal{G}$ be a set of size-change graphs; the following two properties are equivalent
  1. $\mathcal{G}$ is size-change terminating
  2. every maximal multigraph of $\mathcal{G}$ contains an edge $i \overset{\succ}{\to} i$
- although the above theorem only gives rise to an EXPSPACE-algorithm, size-change termination is in PSPACE;
  in fact, size-change termination is PSPACE-complete
- despite the high theoretical complexity class, for sets of size-change graphs arising from usual algorithms, the number of multigraphs is rather low

Proof of Theorem

- the direction that size-change termination implies the property on maximal multigraphs can be done in a straight-forward way
- the other direction is much more advanced and relies upon Ramsey's theorem in its infinite version

Proof of Theorem: Easy Direction (1. implies 2.)

- assume that $\mathcal{G}$ is size-change terminating, and consider any maximal graph G
- since G is a multigraph, it can be written as $G = G_1 \bullet \ldots \bullet G_n$ with each $G_i \in \mathcal{G}$
- consider infinite graph $G_1 \circ \ldots \circ G_n \circ G_1 \circ \ldots \circ G_n \circ \ldots$
- because of size-change termination, this graph contains path with infinitely many $\overset{\succ}{\to}$-edges
- hence $G \circ G \circ \ldots$ also has a path with infinitely many $\overset{\succ}{\to}$-edges
- on this path some index i must be visited infinitely often
- hence there is a path of length k such that $G \circ G \circ \ldots \circ G$ (k-times) contains a path from the leftmost argument i to the rightmost argument i with at least one $\overset{\succ}{\to}$-edge
- consequently $G \bullet G \bullet \ldots \bullet G$ (k-times) contains an edge $i \overset{\succ}{\to} i$
- by maximality, $G = G \bullet G \bullet \ldots \bullet G$, and thus G contains an edge $i \overset{\succ}{\to} i$

Ramsey's Theorem

- definition: given set X and $n \in \mathbb{N}$, we define $X^{(n)}$ as the set of all subsets of X of size n; formally:
$$X^{(n)} = \{Z \mid Z \subseteq X \wedge |Z| = n\}$$
- Ramsey's Theorem – Infinite Version
  - let $n \in \mathbb{N}$
  - let C be a finite set of colors
  - let X be an infinite set
  - let c be a coloring of the size n sets of X, i.e., $c : X^{(n)} \to C$
  - theorem: there exists an infinite subset $Y \subseteq X$ such that all size n sets of Y have the same color

Proof of Theorem: Hard Direction (2. implies 1.)

- consider some arbitrary infinite graph $G_0 \circ G_1 \circ G_2 \circ \ldots$
- for $n < m$ define $G_{n,m} = G_n \bullet \ldots \bullet G_{m-1}$
- by Ramsey's theorem there is an infinite set $I \subseteq \mathbb{N}$ such that $G_{n,m}$ is always the same graph G for all $n, m \in I$ with $n < m$
  ($n = 2$, C = multigraphs, $X = \mathbb{N}$, $c(\{n,m\}) = G_{min\{n,m\},max\{n,m\}}$)
- G is maximal: for $n_1 < n_2 < n_3$ with $\{n_1, n_2, n_3\} \subseteq I$, we have $G_{n_1,n_3} = G_{n_1} \bullet \ldots \bullet G_{n_2-1} \bullet G_{n_2} \bullet \ldots \bullet G_{n_3-1} = G_{n_1,n_2} \bullet G_{n_2,n_3}$, and thus $G = G \bullet G$
- by assumption, G contains edge $i \overset{\succ}{\to} i$
- let $I = \{n_1, n_2, \ldots\}$ with $n_1 < n_2 < \ldots$ and obtain

$$\begin{aligned} & G_0 \circ G_1 \circ \ldots \\ = {} & G_0 \circ \ldots \circ G_{n_1-1} \circ G_{n_1} \circ \ldots \circ G_{n_2-1} \circ G_{n_2} \circ \ldots \circ G_{n_3-1} \circ \ldots \\ \sim {} & G_0 \circ \ldots \circ G_{n_1-1} \circ G \qquad\qquad \circ G \qquad\qquad \circ \ldots \end{aligned}$$

so that edge $i \overset{\succ}{\to} i$ of G delivers path with infinitely many $\overset{\succ}{\to}$-edges

Proof of Ramsey's Theorem

- Ramsey's Theorem – Infinite Version
  - let $n \in \mathbb{N}$
  - let C be a finite set of colors
  - let X be an infinite set
  - let c be a coloring of the size n sets of X, i.e., $c : X^{(n)} \to C$
  - theorem: there exists an infinite subset $Y \subseteq X$ such that all size n sets of Y have the same color
- proof of Ramsey's theorem is interesting
  - it is simple, in that it only uses standard induction on n with arbitrary c and X
  - it is complex, in that it uses a non-trivial construction in the step-case, in particular applying the IH infinitely often
- base case $n = 0$ is trivial, since there is only one size-0 set: the empty set

Proof of Ramsey's Theorem – Step Case $n = m + 1$

- define $X_0 = X$
- pick an arbitrary element a_0 of X_0
- define $Y_0 = X_0 \setminus \{a_0\}$; define coloring $c' : Y_0^{(m)} \to C$ as $c'(Z) = c(Z \cup \{a_0\})$
- IH yields infinite subset $X_1 \subseteq Y_0$ such that all size m sets of X_1 have the same color c_0 w.r.t. c'
- hence, $c(\{a_0\} \cup Z) = c_0$ for all $Z \in X_1^{(m)}$
- next pick an arbitrary element a_1 of X_1 to obtain infinite set $X_2 \subseteq X_1 \setminus \{a_1\}$ such that $c(\{a_1\} \cup Z) = c_1$ for all $Z \in X_2^{(m)}$
- by iterating this obtain elements $a_0, a_1, a_2, \ldots$, colors $c_0, c_1, c_2 \ldots$ and sets $X_0, X_1, X_2, \ldots$ satisfying the above properties
- since C is finite there must be some color d in the infinite list $c_0, c_1, \ldots$ that occurs infinitely often; define $Y = \{a_i \mid c_i = d\}$
- Y has desired properties since all size n sets of Y have color d: if $Z \in Y^{(n)}$ then Z can be written as $\{a_{i_1}, \ldots, a_{i_n}\}$ with $i_1 < \ldots < i_n$; hence, $Z = \{a_{i_1}\} \cup Z'$ with $Z' \in X_{i_1+1}^{(m)}$, i.e., $c(Z) = c_{i_1} = d$

Summary of Size-Change Principle

- size-change principle abstracts dependency pairs into set of size-change graphs
- if no critical graph exists (multigraph without edge $i \overset{\succ}{\to} i$), termination is proven
- soundness relies upon Ramsey's theorem
- subsumes subterm criterion in the following sense:
  if all DPs can be deleted by subterm criterion, then also size-change principle is successful
- still no handling of defined symbols in dependency pairs as in

$$\mathsf{div}^\sharp(\mathsf{Succ}(x), \mathsf{Succ}(y)) \to \mathsf{div}^\sharp(\mathsf{minus}(x, y), \mathsf{Succ}(y))$$

Termination – Reduction Pairs

Reduction Pairs

- recall definition: P-chain is sequence

$$s_1\sigma_1 \to t_1\sigma_1 \overset{i}{\hookrightarrow}^* s_2\sigma_2 \to t_2\sigma_2 \overset{i}{\hookrightarrow}^* s_3\sigma_3 \to t_3\sigma_3 \overset{i}{\hookrightarrow}^* \dots$$

  such that all $s_i \to t_i \in P$ and all $s_i\sigma_i \in NF(\hookrightarrow)$
- previously we used $\rhd$ on $s_i \to t_i$ to ensure decrease $s_i\sigma_i \rhd t_i\sigma_i$
- previously we used $s_i\sigma \in NF(\hookrightarrow)$ and $\trianglerighteq$ to turn $\overset{i}{\hookrightarrow}^*$ into $=$
- now generalize $\rhd$ to strongly normalizing relation $\succ$
- now demand $\ell \succsim r$ for equations to ensure decrease $t_i\sigma_i \succsim s_{i+1}\sigma_{i+1}$
- definition: reduction pair $(\succ, \succsim)$ is pair of relations such that
  - $SN(\succ)$
  - $\succsim$ is transitive
  - $\succ$ and $\succsim$ are compatible: $\succ \circ \succsim \subseteq \succ$
  - both $\succ$ and $\succsim$ are closed under substitutions: $s \;(\succsim)\; t \longrightarrow s\sigma \;(\succsim)\; t\sigma$
  - $\succsim$ is closed under contexts: $s \succsim t \longrightarrow F(\dots, s, \dots) \succsim F(\dots, t, \dots)$
  - note: $\succ$ does not have to be closed under contexts

Applying Reduction Pairs

- recall definition: P-chain is sequence

$$s_1\sigma_1 \to t_1\sigma_1 \overset{i}{\hookrightarrow}^* s_2\sigma_2 \to t_2\sigma_2 \overset{i}{\hookrightarrow}^* s_3\sigma_3 \to t_3\sigma_3 \overset{i}{\hookrightarrow}^* \dots$$

  such that all $s_i \to t_i \in P$ and all $s_i\sigma \in NF(\hookrightarrow)$
- demand $s \succsim t$ for all $s \to t \in P$ to ensure $s_i\sigma_i \succsim t_i\sigma_i$
- demand $\ell \succsim r$ for all equations to ensure $t_i\sigma_i \succsim s_{i+1}\sigma_{i+1}$
- define $P_\succ = \{s \to t \in P \mid s \succ t\}$
- effect: pairs in $P_\succ$ cannot be applied infinitely often and can therefore be removed
- theorem: if there is an infinite P-chain, then there also is an infinite $P \setminus P_\succ$-chain

Example

- remaining termination problem

$$\begin{aligned}
\mathsf{minus}(x, \mathsf{Zero}) &= x\\
\mathsf{minus}(\mathsf{Succ}(x), \mathsf{Succ}(y)) &= \mathsf{minus}(x, y)\\
\mathsf{div}(\mathsf{Zero}, \mathsf{Succ}(y)) &= \mathsf{Zero}\\
\mathsf{div}(\mathsf{Succ}(x), \mathsf{Succ}(y)) &= \mathsf{Succ}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{Succ}(y)))\\
\mathsf{div}^\sharp(\mathsf{Succ}(x), \mathsf{Succ}(y)) &\to \mathsf{div}^\sharp(\mathsf{minus}(x, y), \mathsf{Succ}(y))
\end{aligned}$$

- constraints

$$\begin{aligned}
\mathsf{minus}(x, \mathsf{Zero}) &\succsim x\\
\mathsf{minus}(\mathsf{Succ}(x), \mathsf{Succ}(y)) &\succsim \mathsf{minus}(x, y)\\
\mathsf{div}(\mathsf{Zero}, \mathsf{Succ}(y)) &\succsim \mathsf{Zero}\\
\mathsf{div}(\mathsf{Succ}(x), \mathsf{Succ}(y)) &\succsim \mathsf{Succ}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{Succ}(y)))\\
\mathsf{div}^\sharp(\mathsf{Succ}(x), \mathsf{Succ}(y)) &\succ \mathsf{div}^\sharp(\mathsf{minus}(x, y), \mathsf{Succ}(y))
\end{aligned}$$

Usable Symbols and Usable Equations

$$\mathsf{div}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) \to \mathsf{div}^\sharp(\mathsf{minus}(x,y),\mathsf{Succ}(y))$$

- requiring $\ell \gtrsim r$ for all program equations $\ell = r$ is quite demanding
  - not incremental, i.e., adding other functions later will invalidate proof
  - not necessary, i.e., argument evaluation in example DP only requires minus
- definition: the usable symbols $\mathcal{US}$ w.r.t. a set P are those defined symbols that occur in rhss of P (no $\sharp$-symbols(!)), or that occur in rhss of defining equations of usable symbols;
  - $\mathcal{US}(x) = \varnothing$
  - $\mathcal{US}(F(t_1,\ldots,t_n)) = \mathcal{US}(t_1) \cup \ldots \cup \mathcal{US}(t_n) \cup \begin{cases} \{F\}, & \text{if } F \in \mathcal{D} \\ \varnothing, & \text{otherwise} \end{cases}$
  - assuming $\mathcal{E}$ is the set of equations of a program, then $\mathcal{US}(P)$ is defined by

$$\frac{s \to t \in P}{\mathcal{US}(t) \subseteq \mathcal{US}(P)} \qquad \frac{f \in \mathcal{US}(P) \quad f(\ldots) = r \in \mathcal{E}}{\mathcal{US}(r) \subseteq \mathcal{US}(P)}$$

- the usable equations of P are $\mathcal{U}(P) = \{f(\ldots) = r \in \mathcal{E} \mid f \in \mathcal{US}(P)\}$
- theorem: whenever $t_i\sigma_i \xhookrightarrow{\text{i}}{}^* s_{i+1}\sigma_{i+1}$ in chain, then only $\mathcal{U}(\{s_i \to t_i\})$ can be used

Applying Reduction Pairs with Usable Equations

- recall definition: P-chain is sequence

$$s_1\sigma_1 \to t_1\sigma_1 \xhookrightarrow{\text{i}}{}^* s_2\sigma_2 \to t_2\sigma_2 \xhookrightarrow{\text{i}}{}^* s_3\sigma_3 \to t_3\sigma_3 \xhookrightarrow{\text{i}}{}^* \ldots$$

  such that all $s_i \to t_i \in P$ and all $s_i\sigma \in NF(\hookrightarrow)$
- choose a symbol $f^\sharp$ and define $P_{f^\sharp} = \{s \to t \in P \mid root\ s = f^\sharp\}$
- demand $s \gtrsim t$ for all $s \to t \in P_{f^\sharp}$
- demand $\ell \gtrsim r$ for all $\ell = r \in \mathcal{U}(P_{f^\sharp})$
- define $P_\succ = \{s \to t \in P_{f^\sharp} \mid s \succ t\}$
- effect: pairs in $P_\succ$ cannot be applied infinitely often and can therefore be removed
- theorem: if there is an infinite P-chain, then there also is an infinite $P \setminus P_\succ$-chain

Example with Usable Equations

- remaining termination problem

$$\begin{aligned}
\mathsf{minus}(x,\mathsf{Zero}) &= x \\
\mathsf{minus}(\mathsf{Succ}(x),\mathsf{Succ}(y)) &= \mathsf{minus}(x,y) \\
\mathsf{div}(\mathsf{Zero},\mathsf{Succ}(y)) &= \mathsf{Zero} \\
\mathsf{div}(\mathsf{Succ}(x),\mathsf{Succ}(y)) &= \mathsf{Succ}(\mathsf{div}(\mathsf{minus}(x,y),\mathsf{Succ}(y)))
\end{aligned}$$

$$\mathsf{div}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) \to \mathsf{div}^\sharp(\mathsf{minus}(x,y),\mathsf{Succ}(y))$$

- $\mathcal{US}(P_{\mathsf{div}^\sharp}) = \{\mathsf{minus}\}$, so constraints are

$$\begin{aligned}
\mathsf{minus}(x,\mathsf{Zero}) &\gtrsim x \\
\mathsf{minus}(\mathsf{Succ}(x),\mathsf{Succ}(y)) &\gtrsim \mathsf{minus}(x,y) \\
\mathsf{div}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) &\succ \mathsf{div}^\sharp(\mathsf{minus}(x,y),\mathsf{Succ}(y))
\end{aligned}$$

- because of usable equations, applying reduction pairs becomes incremental: new function definitions won't increase usable equations of DPs of previously defined equations

Remaining Problem

- given constraints

$$\begin{aligned}
\mathsf{minus}(x,\mathsf{Zero}) &\gtrsim x \\
\mathsf{minus}(\mathsf{Succ}(x),\mathsf{Succ}(y)) &\gtrsim \mathsf{minus}(x,y) \\
\mathsf{div}^\sharp(\mathsf{Succ}(x),\mathsf{Succ}(y)) &\succ \mathsf{div}^\sharp(\mathsf{minus}(x,y),\mathsf{Succ}(y))
\end{aligned}$$

  find a suitable reduction pair such that these constraints are satisfied
- many such reduction pairs are available (cf. term rewriting lecture)
  - Knuth–Bendix order (constraint solving is in P)
  - recursive path order (NP-complete)
  - polynomial interpretations (undecidable)
    - powerful
    - intuitive
    - automatable
  - matrix interpretations (undecidable)
  - weighted path order (undecidable)

Polynomial Interpretation

- interpret each n-ary symbol F as polynomial $p_F(x_1, \ldots, x_n)$
- variables in polynomials range over $\mathbb{N}$ and polynomials have to be weakly monotone

$$x_i \geq y_i \longrightarrow p_F(x_1, \ldots, x_i, \ldots, x_n) \geq p_F(x_1, \ldots, y_i, \ldots, x_n)$$

  sufficient criterion: forbid subtraction and negative numbers in p_F
- interpretation is lifted to terms by composing polynomials

$$[\![x]\!] = x$$
$$[\![F(t_1, \ldots, t_n)]\!] = p_F([\![t_1]\!], \ldots, [\![t_n]\!])$$

- $\succsim_{(\sim)}$ is defined as

$$s \succsim_{(\sim)} t \text{ iff } \forall \vec{x} \in \mathbb{N}^*.\ [\![s]\!] \geq_{(=)} [\![t]\!]$$

- $(\succ, \succsim)$ is a reduction pair, e.g.,
  - $SN(\succ)$ follows from strong-normalization of $>$ on $\mathbb{N}$
  - $\succsim$ is closed under contexts since each p_F is weakly monotone

Example – Polynomial Interpretation

- given constraints

$$\mathsf{minus}(x, \mathsf{Zero}) \succsim x$$
$$\mathsf{minus}(\mathsf{Succ}(x), \mathsf{Succ}(y)) \succsim \mathsf{minus}(x, y)$$
$$\mathsf{div}^\sharp(\mathsf{Succ}(x), \mathsf{Succ}(y)) \succ \mathsf{div}^\sharp(\mathsf{minus}(x, y), \mathsf{Succ}(y))$$

  and polynomial interpretation

$$p_{\mathsf{minus}}(x_1, x_2) = x_1$$
$$p_{\mathsf{Zero}} = 2$$
$$p_{\mathsf{Succ}}(x_1) = 1 + x_1$$
$$p_{\mathsf{div}^\sharp}(x_1, x_2) = x_1 + 3x_2$$

  we obtain polynomial constraints

$$[\![\mathsf{minus}(x, \mathsf{Zero})]\!] = x \geq x = [\![x]\!]$$
$$[\![\mathsf{minus}(\mathsf{Succ}(x), \mathsf{Succ}(y))]\!] = 1 + x \geq x = [\![\mathsf{minus}(x, y)]\!]$$
$$[\![\mathsf{div}^\sharp(\mathsf{Succ}\ldots)]\!] = 4 + x + 3y > 3 + x + 3y = [\![\mathsf{div}^\sharp(\mathsf{minus}\ldots)]\!]$$

Solving Polynomial Constraints

- each polynomial constraint over $\mathbb{N}$ can be brought into simple form "$p \geq 0$" for some polynomial p
  - replace $p_1 > p_2$ by $p_1 \geq p_2 + 1$
  - replace $p_1 \geq p_2$ by $p_1 - p_2 \geq 0$
- the question of "$p \geq 0$" over $\mathbb{N}$ is undecidable (Hilbert's 10th problem)
- approximation via absolute positiveness: if all coefficients of p are non-negative, then $p \geq 0$ for all instances over $\mathbb{N}$
- division example has trivial constraints

original	simplified
$x \geq x$	$0 \geq 0$
$1 + x \geq x$	$1 \geq 0$
$4 + x + 3y > 3 + x + 3y$	$0 \geq 0$

Finding Polynomial Interpretations

- in division example, interpretation was given on slides
- aim: search for suitable interpretation
- approach: perform everything symbolically

Symbolic Polynomial Interpretations

- fix shape of polynomial, e.g., linear

$$p_F(x_1, \ldots, x_n) = F_0 + F_1 x_1 + \cdots + F_n x_n$$

where the F_i are symbolic coefficients

- $$p_{\mathsf{minus}}(x_1, x_2) = x_1$$
$$p_{\mathsf{Zero}} = 2$$
$$p_{\mathsf{Succ}}(x_1) = 1 + x_1$$
$$p_{\mathsf{div}^\sharp}(x_1, x_2) = x_1 + 3x_2$$

concrete interpretation above becomes symbolic

$$p_{\mathsf{minus}}(x_1, x_2) = \mathsf{m}_0 + \mathsf{m}_1 x_1 + \mathsf{m}_2 x_2$$
$$p_{\mathsf{Zero}} = \mathsf{Z}_0$$
$$p_{\mathsf{Succ}}(x_1) = \mathsf{S}_0 + \mathsf{S}_1 x_1$$
$$p_{\mathsf{div}^\sharp}(x_1, x_2) = \mathsf{d}_0 + \mathsf{d}_1 x_1 + \mathsf{d}_2 x_2$$

Symbolic Polynomial Constraints

- given constraints

$$\mathsf{minus}(x, \mathsf{Zero}) \succsim x$$
$$\mathsf{minus}(\mathsf{Succ}(x), \mathsf{Succ}(y)) \succsim \mathsf{minus}(x, y)$$
$$\mathsf{div}^\sharp(\mathsf{Succ}(x), \mathsf{Succ}(y)) \succ \mathsf{div}^\sharp(\mathsf{minus}(x, y), \mathsf{Succ}(y))$$

- obtain symbolic polynomial constraints

$$\mathsf{m}_0 + \mathsf{m}_1 x + \mathsf{m}_2 \mathsf{Z}_0 \geq x$$
$$\mathsf{m}_0 + \mathsf{m}_1(\mathsf{S}_0 + \mathsf{S}_1 x) + \mathsf{m}_2(\mathsf{S}_0 + \mathsf{S}_1 y) \geq \mathsf{m}_0 + \mathsf{m}_1 x + \mathsf{m}_2 y$$
$$\mathsf{d}_0 + \mathsf{d}_1(\mathsf{S}_0 + \mathsf{S}_1 x) + \mathsf{d}_2(\mathsf{S}_0 + \mathsf{S}_1 y) > \mathsf{d}_0 + \mathsf{d}_1(\mathsf{m}_0 + \mathsf{m}_1 x + \mathsf{m}_2 y) + \mathsf{d}_2(\mathsf{S}_0 + \mathsf{S}_1 y)$$

- and simplify to

$$(\mathsf{m}_0 + \mathsf{m}_2 \mathsf{Z}_0) + (\mathsf{m}_1 - 1)x \geq 0$$
$$(\mathsf{m}_1 \mathsf{S}_0 + \mathsf{m}_2 \mathsf{S}_0) + (\mathsf{m}_1 \mathsf{S}_1 - \mathsf{m}_1)x + (\mathsf{m}_2 \mathsf{S}_1 - \mathsf{m}_2)y \geq 0$$
$$(\mathsf{d}_1 \mathsf{S}_0 - \mathsf{d}_1 \mathsf{m}_0 - 1) + (\mathsf{d}_1 \mathsf{S}_1 - \mathsf{d}_1 \mathsf{m}_1)x + (-\mathsf{d}_1 \mathsf{m}_2)y \geq 0$$

Absolute Positiveness – Symbolic Example

- on symbolic polynomial constraints

$$(\mathsf{m}_0 + \mathsf{m}_2 \mathsf{Z}_0) + (\mathsf{m}_1 - 1)x \geq 0$$
$$(\mathsf{m}_1 \mathsf{S}_0 + \mathsf{m}_2 \mathsf{S}_0) + (\mathsf{m}_1 \mathsf{S}_1 - \mathsf{m}_1)x + (\mathsf{m}_2 \mathsf{S}_1 - \mathsf{m}_2)y \geq 0$$
$$(\mathsf{d}_1 \mathsf{S}_0 - \mathsf{d}_1 \mathsf{m}_0 - 1) + (\mathsf{d}_1 \mathsf{S}_1 - \mathsf{d}_1 \mathsf{m}_1)x + (-\mathsf{d}_1 \mathsf{m}_2)y \geq 0$$

absolute positiveness works as before; obtain constraints

$$\begin{array}{lll} \mathsf{m}_0 + \mathsf{m}_2 \mathsf{Z}_0 \geq 0 & \mathsf{m}_1 - 1 \geq 0 & \\ \mathsf{m}_1 \mathsf{S}_0 + \mathsf{m}_2 \mathsf{S}_0 \geq 0 & \mathsf{m}_1 \mathsf{S}_1 - \mathsf{m}_1 \geq 0 & \mathsf{m}_2 \mathsf{S}_1 - \mathsf{m}_2 \geq 0 \\ \mathsf{d}_1 \mathsf{S}_0 - \mathsf{d}_1 \mathsf{m}_0 - 1 \geq 0 & \mathsf{d}_1 \mathsf{S}_1 - \mathsf{d}_1 \mathsf{m}_1 \geq 0 & -\mathsf{d}_1 \mathsf{m}_2 \geq 0 \end{array}$$

- at this point, use solver for integer arithmetic to find suitable coefficients (in $\mathbb{N}$)
- popular choice: SMT solver for integer arithmetic where one has to add constraints $\mathsf{m}_0 \geq 0, \mathsf{m}_1 \geq 0, \mathsf{m}_2 \geq 0, \mathsf{S}_0 \geq 0, \mathsf{S}_1 \geq 0, \mathsf{Z}_0 \geq 0, \ldots$

Constraint Solving by Hand – Example

- original constraints

$$\begin{array}{lll} \mathsf{m}_0 + \mathsf{m}_2 \mathsf{Z}_0 \geq 0 & \mathsf{m}_1 - 1 \geq 0 & \\ \mathsf{m}_1 \mathsf{S}_0 + \mathsf{m}_2 \mathsf{S}_0 \geq 0 & \mathsf{m}_1 \mathsf{S}_1 - \mathsf{m}_1 \geq 0 & \mathsf{m}_2 \mathsf{S}_1 - \mathsf{m}_2 \geq 0 \\ \mathsf{d}_1 \mathsf{S}_0 - \mathsf{d}_1 \mathsf{m}_0 - 1 \geq 0 & \mathsf{d}_1 \mathsf{S}_1 - \mathsf{d}_1 \mathsf{m}_1 \geq 0 & -\mathsf{d}_1 \mathsf{m}_2 \geq 0 \end{array}$$

- delete trivial constraints

$$\begin{array}{lll} & \mathsf{m}_1 - 1 \geq 0 & \\ & \mathsf{m}_1 \mathsf{S}_1 - \mathsf{m}_1 \geq 0 & \mathsf{m}_2 \mathsf{S}_1 - \mathsf{m}_2 \geq 0 \\ \mathsf{d}_1 \mathsf{S}_0 - \mathsf{d}_1 \mathsf{m}_0 - 1 \geq 0 & \mathsf{d}_1 \mathsf{S}_1 - \mathsf{d}_1 \mathsf{m}_1 \geq 0 & -\mathsf{d}_1 \mathsf{m}_2 \geq 0 \end{array}$$

- conclusions

$$\begin{array}{lll} \mathsf{m}_1 \geq 1 & \mathsf{d}_1 \geq 1 & \\ \mathsf{S}_0 \geq 1 & \mathsf{S}_1 \geq 1 & \\ \mathsf{m}_2 = 0 & \mathsf{S}_1 \geq \mathsf{m}_1 & \mathsf{m}_0 = 0 \end{array}$$

Constraint Solving by SMT-Solver – Example

- original constraints

$$\begin{array}{lll} m_0 + m_2 Z_0 \geq 0 & m_1 - 1 \geq 0 & \\ m_1 S_0 + m_2 S_0 \geq 0 & m_1 S_1 - m_1 \geq 0 & m_2 S_1 - m_2 \geq 0 \\ d_1 S_0 - d_1 m_0 - 1 \geq 0 & d_1 S_1 - d_1 m_1 \geq 0 & -d_1 m_2 \geq 0 \end{array}$$

- encode as SMT problem in file `division.smt2`

```
(set-logic QF_NIA)
(declare-fun m0 () Int) ... (declare-fun d2 () Int)
(assert (>= m0 0)) ... (assert (>= d2 0))
(assert (>= (+ m0 (* m2 Z0)) 0))
...
(assert (>= (* (- 1) d1 m2) 0))
(check-sat)
(get-model)
(exit)
```


Constraint Solving by SMT-Solver – Example Continued

- invoke SMT solver, e.g., Microsoft's open source solver Z3

```
cmd> z3 division.smt2
sat
(model
  (define-fun d1 () Int 8)
  (define-fun S1 () Int 15)
  (define-fun S0 () Int 8)
  (define-fun Z0 () Int 0)
  (define-fun m2 () Int 0)
  (define-fun m1 () Int 12)
  (define-fun m0 () Int 4)
  (define-fun d2 () Int 0)
  (define-fun d0 () Int 0)
)
```

- parse result to obtain polynomial interpretation

Constraint Solving by SMT-Solver – Scepticism

- polynomial interpretation found by SMT solving approach is generated by complex (potentially buggy) tool
- however, termination is essential for well-defined programs, i.e., in particular to derive correct theorems
- solution: certification
  - search for interpretation can be done in arbitrary untrusted way
  - write simple trusted checker that certifies whether concrete interpretation indeed satisfies all constraints
  - like solving NP-complete problem: positive answer can easily be verified
- in fact, this approach is heavily used in termination proving
  - untrusted tools: AProVE, T_TT_2, Terminator, ...
  - trusted checker: CeTA; soundness formally proven in Isabelle/HOL
  - certification will be handled in more detail in Part 7 of this lecture

Summary

- pattern-completeness and pattern-disjointness are decidable
- termination proving can be done via
  - dependency pairs
  - subterm criterion
  - size-change termination
  - polynomial interpretation
- termination proving often performed with help of SMT solvers
- increase reliability via certification: checking of generated proofs