



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
 - \leftrightarrow 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

$$\text{append} : \text{List} \times \text{List} \rightarrow \text{List}$$

$$\text{append}(\text{Cons}(x, y), z) = \text{Cons}(x, \text{append}(y, z))$$

$$\text{append}(\text{Nil}, x) = x$$

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

Interlude: Maybe-Type for Errors

- recall type-checking algorithm

```
typeCheck :: Sig -> Vars -> Term -> Maybe Type
```

```
typeCheck sigma vars (Var x) = vars x
```

```
typeCheck sigma vars (Fun f ts) = do
```

```
  (tysIn,tyOut) <- sigma f
```

```
  tysTs <- mapM (typeCheck sigma vars) ts
```

```
  if tysTs == tysIn then return tyOut else Nothing
```

- Maybe-type is only one possibility to represent computational results with failure
- let us abstract from concrete Maybe-type:

- introduce new type Check to represent a result or failure

```
type Check a = Maybe a
```

- function return :: a -> Check a to produce successful results

- function to raise a failure

```
failure :: Check a
```

```
failure = Nothing
```

- convenience function: asserting a property

```
assert :: Bool -> Check ()
```

```
assert p = if p then return () else failure
```

Making Type-Checking More Abstract

- original type-checking algorithm

```

typeCheck :: Sig -> Vars -> Term -> Maybe Type
typeCheck sigma vars (Var x) = vars x
typeCheck sigma vars (Fun f ts) = do
  (tysIn,tyOut) <- sigma f
  tysTs <- mapM (typeCheck sigma vars) ts
  if tysTs == tysIn then return tyOut else Nothing

```

- with new abstract types and functions

```

typeCheck :: Sig -> Vars -> Term -> Check Type
typeCheck sigma vars (Var x) = vars x
typeCheck sigma vars (Fun f ts) = do
  (tysIn,tyOut) <- sigma f
  tysTs <- mapM (typeCheck sigma vars) ts
  assert (tysTs == tysIn)
  return tyOut

```

- advantage: readability, change `Check`-type easily

Back to Type-Checking and Type-Inference

- known: type-checking algorithm

```
typeCheck :: Sig -> Vars -> Term -> Check Type
```

- `type Sig = FSym -> Check ([Type], Type) - Σ`
 - `type Vars = Var -> Check 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 -> Check [(Var,Type)]
```

- then type-checking an equation without explicit `Vars` is possible

```
typeCheckEqn :: Sig -> (Term, Term) -> Check ()
```

```
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 -> Check [(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 (\ (ty, t) -> inferType sigma ty t) (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 $\text{inferType } \Sigma \ \tau \ t = \text{return } (\mathcal{V} \cap \text{Vars}(t))$
 - if $\text{inferType } \Sigma \ \tau \ t = \text{return } \mathcal{V}$ 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 2/35–42](#)
- note that ‘if $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ then $\text{inferType } \Sigma \ \tau \ t \neq \text{failure}$ ’ is a property which is not strong enough when performing induction

Changing the Error Monad

Weakness of Maybe-Type for Errors

- situation: several functions for checking properties of terms, equations, which can be assembled to check functional programs w.r.t. slides 3/4 (data-type definitions), 3/15 (function definitions) and partly 3/45 (well-definedness)
 - `inferType :: Sig -> Type -> Term -> Check [(Var,Type)]`
 - `typeCheck :: Sig -> Vars -> Term -> Check Type`
 - `typeCheckEqn :: Sig -> (Term, Term) -> Check ()`
- problem: if checks are not successful, we just get result `Nothing`
- desired: **informative error message** why a functional program is refused
- possible solution: use more verbose error type than `Maybe`
`type Check a = Either String a`

Changing Implementation of Interface

- current interface for error type
 - `type Check a = Maybe a`
 - function `return :: a -> Check a`
 - function `assert :: Bool -> Check ()`
 - function `failure :: Check a`
 - do-blocks, monadic-functions such as `mapM`, etc.
- it is actually easy to change to `Either`-type for errors
 - `type Check a = Either String a`
 - `return`, do-blocks and `mapM` are unchanged, since these are part of generic monad interface
 - functions `assert` and `failure` need to be changed, since they now require error messages
 - `failure :: String -> Check a`
`failure = Left`
 - `assert :: Bool -> String -> Check ()`
`assert p err = if p then return () else failure err`

Changing Algorithms for Checking Properties

- adapting algorithms often only requires additional error messages

- before change (`type Check a = Maybe a`)

```
typeCheck :: Sig -> Vars -> Term -> Check Type
```

```
typeCheck sigma vars (Var x) = vars x
```

```
typeCheck sigma vars (Fun f ts) = do
```

```
  (tysIn,tyOut) <- sigma f
```

```
  tysTs <- mapM (typeCheck sigma vars) ts
```

```
  assert (tysTs == tysIn)
```

```
  return tyOut
```

- after change (`type Check a = Either String a`)

```
typeCheck :: Sig -> Vars -> Term -> Check Type
```

```
typeCheck sigma vars (Var x) = ...
```

```
typeCheck sigma vars t@(Fun f ts) = do
```

```
  ...
```

```
  assert (tysTs == tysIn) (show t ++ " ill-typed")
```

```
  ...
```

Changing Algorithms for Checking Properties, Continued

- example requiring more changes; with `type Check a = Maybe a`

```

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)
      
```
- new version with `type Check a = Either String a`

```

typeCheckEqn sigma (Var x, r) = failure "var as lhs"
typeCheckEqn sigma (l @ (Fun f _), r) = do
  ...
  tyR <- typeCheck sigma (\ x -> lookup x vars) r
  assert (ty == tyR) "types of lhs and rhs don't match"
      
```
- problem: `lookup` produces `Maybe`, not `Either String`
- solution: use `maybeToEither :: e -> Maybe a -> Either e a`

Fixed Type-Checking Algorithm with Error Messages

```

import Data.Either.Utils -- for maybeToEither
-- import requires MissingH lib; if not installed, define it yourself:
-- maybeToEither e Nothing = Left e
-- maybeToEither _ (Just x) = return x

typeCheckEqn sigma (Var x, r) = failure "var as lhs"
typeCheckEqn sigma (l @ (Fun f _), r) = do
  (_,ty) <- sigma f
  vars <- inferType sigma ty l
  tyR <- typeCheck
    sigma
    (\ x -> maybeToEither
      (x ++ " is unknown variable")
      (lookup x vars))
    r
  assert (ty == tyR) "types of lhs and rhs don't match"

```

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

$$\begin{array}{l} \text{data } \tau = c_1 : \tau_{1,1} \times \dots \times \tau_{1,m_1} \rightarrow \tau \\ \quad \quad \quad | \dots \\ \quad \quad \quad | c_n : \tau_{n,1} \times \dots \times \tau_{n,m_n} \rightarrow \tau \end{array}$$

where

- $\tau \notin \mathcal{T}_y$ fresh type name
- $c_1, \dots, c_n \notin \Sigma$ 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
 - $\mathcal{T}_y := \mathcal{T}_y \cup \{\tau\}$
 - $\mathcal{C} := \mathcal{C} \cup \{c_1 : \tau_{1,1} \times \dots \times \tau_{1,m_1} \rightarrow \tau, \dots, c_n : \tau_{n,1} \times \dots \times \tau_{n,m_n} \rightarrow \tau\}$

Existing Encoding of Part 2: Signatures and Terms

```
type Check a = ... -- Maybe a or Either String a
```

```
type Type = String
```

```
type Var = String
```

```
type FSym = String
```

```
type Vars = Var -> Check Type
```

```
type FSymInfo = ([Type], Type)
```

```
type Sig = FSym -> Check 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 -> Check 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 -> Check ProgInfo
processDataDefinition = ... -- previous slide
```

```
processDataDefinitions ::
  ProgInfo -> [DataDefinition] -> Check 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 -> Check ProgInfo
processFunctionDefinition = ... -- exercise

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

Checking Functional Programs

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

```
processProgram :: FunctionalProg -> Check ProgInfo
```

```
processProgram (dataDefs, funDefs) = do
```

```
  pi <- processDataDefinitions initialProgInfo dataDefs
```

```
  processFunctionDefinitions pi funDefs
```


Current State

- `processProgram :: FunctionalProg -> Check 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 $f : \tau_1 \times \dots \times \tau_n \rightarrow \tau \in \mathcal{D}$ and all $t_1 \in \mathcal{T}(\mathcal{C})_{\tau_1}, \dots, t_n \in \mathcal{T}(\mathcal{C})_{\tau_n}$ there is at most one equation $\ell = r$ in the program, such that ℓ matches $f(t_1, \dots, t_n)$
- in proseminar it was proven that 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 $U = \{s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n\}$
- question: is U **solvable**, i.e., does there exist a solution σ , a substitution satisfying $\forall i \in \{1, \dots, n\}. s_i\sigma = t_i\sigma$
- two different kinds of output:

- unification problem in **solved form**:

$$\{x_1 \stackrel{?}{=} v_1, \dots, x_m \stackrel{?}{=} v_m\} \text{ with distinct } x_j\text{'s}$$

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

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

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

$$U \cup \{f(u_1, \dots, u_k) \stackrel{?}{=} f(v_1, \dots, v_k)\} \rightsquigarrow U \cup \{u_1 \stackrel{?}{=} v_1, \dots, v_k \stackrel{?}{=} v_k\} \quad \text{(decompose)}$$

$$U \cup \{f(u_1, \dots, u_k) \stackrel{?}{=} g(v_1, \dots, v_\ell)\} \rightsquigarrow \perp, \text{ if } f \neq g \vee k \neq \ell \quad \text{(clash)}$$

$$U \cup \{f(\dots) \stackrel{?}{=} x\} \rightsquigarrow U \cup \{x \stackrel{?}{=} f(\dots)\} \quad \text{(swap)}$$

$$U \cup \{x \stackrel{?}{=} f(\dots)\} \rightsquigarrow \perp, \text{ if } x \in \mathcal{Vars}(f(\dots)) \quad \text{(occurs check)}$$

$$U \cup \{x \stackrel{?}{=} t\} \rightsquigarrow U\{x/t\} \cup \{x \stackrel{?}{=} t\}, \quad \text{(eliminate)}$$

if $x \notin \mathcal{Vars}(t)$ and x occurs in U

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 \perp 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 $V = \perp$ then U is unsolvable
 - otherwise, V represents a substitution that is a solution to U
- note that \rightsquigarrow is not confluent
 - $\{x \stackrel{?}{=} y, y \stackrel{?}{=} x\} \xrightarrow{x/y} \{x \stackrel{?}{=} y, y \stackrel{?}{=} y\} \rightsquigarrow \{x \stackrel{?}{=} y\}$
 - $\{x \stackrel{?}{=} y, y \stackrel{?}{=} x\} \xrightarrow{y/x} \{x \stackrel{?}{=} x, y \stackrel{?}{=} x\} \rightsquigarrow \{y \stackrel{?}{=} x\}$

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

unifyMain :: [(Term, Term)] -> [(Var,Term)] -> Check [(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 (x `elem` 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
 - $set\ (x : xs) = \{x\} \cup set\ xs$
 - $set\ (xs ++ ys) = set\ xs \cup set\ ys$
 - $set\ (zip\ [x_1, \dots, x_n]\ [y_1, \dots, y_n]) = \{(x_1, y_1), \dots, (x_n, y_n)\}$
 - $set\ (map\ f\ [x_1, \dots, x_n]) = \{f\ x_1, \dots, f\ x_n\}$
 - $subst\ x\ t\ s = s\{x/t\}$
 - ...

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

- define $setMaybe\ Nothing = \perp$, $setMaybe\ (Just\ w) = set\ w$
- **property**: whenever $(u, v) \sim U$ and $unifyMain\ u\ v = res$ then $U \rightsquigarrow^! setMaybe\ res$
- 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 $set\ u \rightsquigarrow^! setMaybe\ res$
 - at this point apply correctness of \rightsquigarrow :
 $setMaybe\ res$ is the correct answer to the unification problem $set\ u$

Proving the Refinement Property

- property $P(u, v, U)$: $(u, v) \sim U \wedge \text{unifyMain } u \ v = \text{res} \longrightarrow U \rightsquigarrow^! \text{setMaybe res}$
- $(u, v) \sim U \iff U = \text{set } u \cup \text{set } v \wedge \text{set } v \in NF(\rightsquigarrow) \wedge \forall (x, t) \in \text{set } v. x \notin \text{Vars}(u)$
- 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
 - (**) $\text{unifyMain } [] \ v = \text{res}$
 - from (*) conclude $U = \text{set } v$ and $\text{set } v \in NF(\rightsquigarrow)$
 - from (**) conclude $\text{res} = \text{Just } v$ and hence, $\text{setMaybe res} = \text{set } v$
 - we have to show $U \rightsquigarrow^! \text{setMaybe res}$, i.e., $\text{set } v \rightsquigarrow^! \text{set } v$ which is satisfied since $\text{set } v \in NF(\rightsquigarrow)$

- $P(u, v, U): (u, v) \sim U \wedge \text{unifyMain } u \ v = \text{res} \longrightarrow U \rightsquigarrow^! \text{setMaybe } \text{res}$
- $(u, v) \sim U \iff U = \text{set } u \cup \text{set } v \wedge \text{set } v \in \text{NF}(\rightsquigarrow) \wedge \forall (x, t) \in \text{set } v. x \notin \text{Vars}(u)$

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

(**) $\text{unifyMain } ((f(ts), g(ss)) : u) \ v = \text{res}$

- consider sub-cases

- $\neg(f = g \wedge \text{length } ts = \text{length } ss)$:

- from (**) conclude $\text{setMaybe } \text{res} = \perp$
- from (*) conclude $f(ts) \stackrel{?}{=} g(ss) \in U$ and hence $U \rightsquigarrow \perp$ by (clash)
- consequently, $U \rightsquigarrow^! \text{setMaybe } \text{res}$

- $f = g \wedge \text{length } ts = \text{length } ss$:

- from (**) conclude $\text{res} = \text{unifyMain } ((f(ts), g(ss)) : u) \ v = \text{unifyMain } (\text{zip } ts \ ss \ ++ \ u) \ v$
- from (*) and alignment for zip and ++ conclude $U = \{f(ts) \stackrel{?}{=} g(ss)\} \cup \text{set } u \cup \text{set } v$ and hence $U \rightsquigarrow \text{set } (\text{zip } ts \ ss \ ++ \ u) \cup \text{set } v =: V$ by (decompose)
- we get $P(\text{zip } ts \ ss \ ++ \ u, v, V)$ as IH; $(\text{zip } ts \ ss \ ++ \ u, v) \sim V$ follows from (*), so $U \rightsquigarrow V \rightsquigarrow^! \text{setMaybe } \text{res}$

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

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

(**) $\text{unifyMain } ((x, t) : u) \ v = \text{res}$

- consider sub-cases (where the red part is not triggered by structure of algorithm)

- $x \neq t \wedge x \notin \text{Vars}(t) \wedge x$ **occurs in set $u \cup \text{set } v$:**

- define $u' = \text{map } (\lambda(l, r). (\text{subst } x \ t \ l, \text{subst } x \ t \ r)) \ u$

- define $v' = \text{map } (\lambda(y, s). (y, \text{subst } x \ t \ s)) \ v$

- define $V = (\text{set } u \cup \text{set } v)\{x/t\} \cup \{x \stackrel{?}{=} t\}$

- from (**) conclude $\text{res} = \text{unifyMain } ((x, t) : u) \ v = \text{unifyMain } u' \ ((x, t) : v')$

- from IH conclude $P(u', (x, t) : v', V)$ and hence, $(u', (x, t) : v') \sim V \longrightarrow V \rightsquigarrow^! \text{setMaybe } \text{res}$

- for proving $U \rightsquigarrow^! \text{setMaybe } \text{res}$ it hence suffices to show $(u', (x, t) : v') \sim V$ and $U \rightsquigarrow V$

- $U \stackrel{(*)}{=} \{x \stackrel{?}{=} t\} \cup \text{set } u \cup \text{set } v \rightsquigarrow (\text{set } u \cup \text{set } v)\{x/t\} \cup \{x/t\} = V$

by (eliminate) because of preconditions

- $(u, v) \sim U \iff U = \text{set } u \cup \text{set } v \wedge \text{set } v \in NF(\rightsquigarrow) \wedge \forall (x, t) \in \text{set } v. x \notin \text{Vars}(u)$

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 $x \neq t \wedge x \notin \text{Vars}(t) \wedge x$ occurs in $\text{set } u \cup \text{set } v$:

- define $u' = \text{map } (\lambda(l, r). (\text{subst } x \ t \ l, \text{subst } x \ t \ r)) \ u$
- define $v' = \text{map } (\lambda(y, s). (y, \text{subst } x \ t \ s)) \ v$
- define $V = (\text{set } u \cup \text{set } v)\{x/t\} \cup \{x \stackrel{?}{=} t\}$
- we still need to show $(u', (x, t) : v') \sim V$
- since (*) holds, we know $\forall (y, s) \in \text{set } v. x \neq y$
- hence, $v' = \text{map } (\lambda(y, s). (\text{subst } x \ t \ y, \text{subst } x \ t \ s)) \ v$
- so, $V = (\text{set } u)\{x/t\} \cup \{x \stackrel{?}{=} t\} \cup (\text{set } v)\{x/t\} = \text{set } u' \cup \text{set } ((x, t) : v')$
- we show $\forall (y, s) \in \text{set } ((x, t) : v'). y \notin \text{Vars}(u')$ as follows:
 $x \notin \text{Vars}(u')$ since $x \notin \text{Vars}(t)$; and if $(y, s) \in \text{set } v'$, then $(y, s') \in \text{set } v$ for some s' and by (*) we conclude $y \notin \text{Vars}((x, t) : u)$; thus, $y \notin \text{Vars}((\text{set } u)\{x/t\}) = \text{Vars}(u')$
- we finally show $\text{set } ((x, t) : v') \in NF(\rightsquigarrow)$: so, assume to the contrary that some step is applicable; by the shape of $\text{set } ((x, t) : v')$ 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 $set\ u \cup set\ v$ ”
 - 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

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

Checking Pattern Completeness

Pattern Problems

- reminder: program is pattern complete, if for all $f : \tau_1 \times \dots \times \tau_n \rightarrow \tau \in \mathcal{D}$ and all $t_i \in \mathcal{T}(\mathcal{C})_{\tau_i}$ there is some lhs that matches $f(t_1, \dots, t_n)$
- algorithm considers more generic shape
 - **matching problems** mp consist of pairs of terms (t, ℓ) where
 - t is a term, representing the set of all its constructor ground instances, e.g., $t = f(x_1, \dots, x_n)$
 - ℓ is (a subterm of) some lhs
 - semantics: find one substitution γ such that $t = \ell\gamma$ for all $(t, \ell) \in mp$
 - reason: decomposition of terms
 - **pattern problems** pp consist of multiple matching problems
 - semantics: disjunction, i.e., find one suitable matching problem
 - reason: a term t might be matched by arbitrary lhs
 - initially: $pp = \{\{(t, \ell_1)\}, \dots, \{(t, \ell_n)\}\}$ for lhss ℓ_1, \dots, ℓ_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**: $P_{init} = \{\{(f(x_1, \dots, x_n), \ell)\} \mid \ell \text{ is lhs of } f\text{-eqn.}\} \mid f \in \mathcal{D}\}$
 - overall semantics: P is **complete** iff

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

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 $f(t_1, \dots, t_n)$
 - $P_{init} = \{ \{ \{ (f(x_1, \dots, x_n), \ell) \} \mid \ell \text{ is lhs of } f\text{-equation} \} \mid f \in \mathcal{D} \}$
 - P is complete iff $\forall pp \in P. \forall \sigma : \mathcal{V} \rightarrow \mathcal{T}(\mathcal{C}). \exists mp \in pp. \exists \gamma. \forall (t, \ell) \in mp. t\sigma = \ell\gamma$
- **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: \perp 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

$\{(f(t_1, \dots, t_n), f(\ell_1, \dots, \ell_n))\} \uplus mp \rightarrow \{(t_1, \ell_1), \dots, (t_n, \ell_n)\} \cup mp$	(decompose)
$\{(t, x)\} \uplus mp \rightarrow mp$	(match)
$\{(f(\dots), g(\dots))\} \uplus mp \rightarrow \perp$ if $f \neq g$	(clash)
$\{mp\} \uplus pp \rightsquigarrow \{mp'\} \cup pp$ if $mp \rightarrow mp'$	(simp-mp)
$\{\perp\} \uplus pp \rightsquigarrow pp$	(remove-mp)
$\{\emptyset\} \uplus pp \rightsquigarrow \top$	(success)
$\{pp\} \uplus P \rightsquigarrow \{pp'\} \cup P$ if $pp \rightsquigarrow pp'$	(simp-pp)
$\{\emptyset\} \uplus P \rightsquigarrow \perp$	(failure)
$\{\top\} \uplus P \rightsquigarrow P$	(remove-pp)
$\{pp\} \uplus P \rightsquigarrow Inst(pp, x) \cup P$ if $mp \in pp$ and $(x, f(\dots)) \in mp$	(instantiate)

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

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

Example

consider

`data Bool = True : Bool | False : Bool`

$\ell_1 := \text{conj}(\text{True}, \text{True}) = \dots$

$\ell_2 := \text{conj}(\text{False}, y) = \dots$

$\ell_3 := \text{conj}(x, \text{False}) = \dots$

then we have

$$\begin{aligned}
 P_{init} &= \{ \{ \{ (\text{conj}(x_1, x_2), \ell_1) \}, \{ (\text{conj}(x_1, x_2), \ell_2) \}, \{ (\text{conj}(x_1, x_2), \ell_3) \} \} \} \\
 &\multimap^* \{ \{ \{ (x_1, \text{True}), (x_2, \text{True}) \}, \{ (x_1, \text{False}), (x_2, y) \}, \{ (x_1, x), (x_2, \text{False}) \} \} \} \\
 &\multimap^* \{ \{ \{ (x_1, \text{True}), (x_2, \text{True}) \}, \{ (x_1, \text{False}) \}, \{ (x_2, \text{False}) \} \} \} \\
 &\multimap \{ \{ \{ (\text{True}, \text{True}), (x_2, \text{True}) \}, \{ (\text{True}, \text{False}) \}, \{ (x_2, \text{False}) \} \}, \\
 &\quad \{ \{ (\text{False}, \text{True}), (x_2, \text{True}) \}, \{ (\text{False}, \text{False}) \}, \{ (x_2, \text{False}) \} \} \} \\
 &\multimap^* \{ \{ \{ (x_2, \text{True}) \}, \perp, \{ (x_2, \text{False}) \} \}, \{ \perp, \emptyset, \{ (x_2, \text{False}) \} \} \} \\
 &\multimap^* \{ \{ \{ (x_2, \text{True}) \}, \{ (x_2, \text{False}) \} \} \} \\
 &\multimap \{ \{ \{ (\text{True}, \text{True}) \}, \{ (\text{True}, \text{False}) \} \}, \{ \{ (\text{False}, \text{True}) \}, \{ (\text{False}, \text{False}) \} \} \} \multimap^* \emptyset
 \end{aligned}$$

Example

consider

```
data Bool = True : Bool | False : Bool
```

```
ℓ1 := conj(True, True) = ...
```

```
ℓ2 := conj(False, y) = ...
```

then we have

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

Partial Correctness of \multimap

- **theorem**: whenever $P \multimap Q$, then P is complete iff Q is complete
- **corollary**: if $P \multimap^* \emptyset$ then P is complete, and if $P \multimap^* \perp$ then P is not complete
- **definition**: P is complete iff

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

- **proof of theorem** by case analysis on the various rules
 - (clash): first inline rule to $\{\{\{(f(\dots), g(\dots))\} \uplus mp\} \uplus pp\} \uplus P \multimap \{pp\} \cup P$, if $f \neq g$
 - by definition of completeness and structure of rule it suffices to show that completeness is preserved by rule

$$\underbrace{\{\{(f(\dots), g(\dots))\} \uplus mp\} \uplus pp}_{=:mp'} \multimap pp$$

- hence, it suffices to show that ψ is not satisfied when choosing mp' in the existential quantifier $\exists mp \in pp. \dots$
 - but this property is easy to see, since $t\sigma = \ell\gamma$ is never satisfied if (t, ℓ) is $(f(\dots), g(\dots))$
- many other rules are similar, exceptions are (match) and (instantiate)

Partial Correctness of \multimap , continued

- definition: P is complete iff

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

- **proof continued**

- (instantiate): $\{pp\} \uplus P \multimap Inst(pp, x) \cup P$, where $x : \tau$, τ has constructors c_1, \dots, c_n , and $\sigma_i = \{x/c_i(x_1, \dots, x_k)\}$ 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 σ
 - since σ is constructor ground, we know $\sigma(x) = c_i(t_1, \dots, t_k)$ for some constructor c_i and constructor ground terms t_1, \dots, t_k
 - define $\sigma'(y) = \begin{cases} t_j, & \text{if } y = x_j \\ \sigma(y), & \text{otherwise} \end{cases}$
 - σ' is well-defined since the x_j 's are distinct, and σ' is a constructor ground substitution
 - note that $t\sigma = t\sigma_i\sigma'$ 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 $mp \in pp\sigma_i$ and γ such that $\forall (t, \ell) \in mp. t\sigma' = \ell\gamma$
 - hence, there is some $mp \in pp$ and γ such that $\forall (t, \ell) \in mp. t\sigma_i\sigma' = \ell\gamma$
 - together with $t\sigma = t\sigma_i\sigma'$ we conclude that pp is complete

Correctness of \rightsquigarrow , Missing Parts

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

— Cannot Get Stuck

$\{(f(t_1, \dots, t_n), f(\ell_1, \dots, \ell_n))\} \uplus mp \rightarrow \{(t_1, \ell_1), \dots, (t_n, \ell_n)\} \cup mp$	(decompose)
$\{(t, x)\} \uplus mp \rightarrow mp$	(match)
$\{(f(\dots), g(\dots))\} \uplus mp \rightarrow \perp$ if $f \neq g$	(clash)
$\{mp\} \uplus pp \rightarrow \{mp'\} \cup pp$ if $mp \rightarrow mp'$	(simp-mp)
$\{\perp\} \uplus pp \rightarrow pp$	(remove-mp)
$\{\emptyset\} \uplus pp \rightarrow \top$	(success)
$\{pp\} \uplus P \rightarrow \{pp'\} \cup P$ if $pp \rightarrow pp'$	(simp-pp)
$\{\emptyset\} \uplus P \rightarrow \perp$	(failure)
$\{\top\} \uplus P \rightarrow P$	(remove-pp)
$\{pp\} \uplus P \rightarrow Inst(pp, x) \cup P$ if $mp \in pp$ and $(x, f(\dots)) \in mp$	(instantiate)

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

Termination of \multimap

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

$$\{\emptyset\} \uplus P \multimap \perp \quad (\text{failure})$$

$$\{\top\} \uplus P \multimap P \quad (\text{remove-pp})$$

$$\{pp\} \uplus P \multimap Inst(pp, x) \cup P \quad \text{if } mp \in pp \text{ and } (x, f(\dots)) \in mp \quad (\text{instantiate})$$

- define $|\ell - t|$ as a measure of difference of ℓ and t
 - $|\ell - x| = \text{number of function symbols in } \ell$
 - $|f(\ell_1, \dots, \ell_n) - f(t_1, \dots, t_n)| = \sum_i |\ell_i - t_i|$
 - $|\ell - t| = 0$, in all other cases
- map each pattern problem pp to number $|pp| = \sum_{mp \in pp, (t, \ell) \in mp} |\ell - t|$
- 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 \multimap -rules
(multiset decrease: $M \cup N >^{mul} M \cup N'$ if $N \neq \emptyset$ and $\forall y \in N'. \exists x \in N. x > y$)
- hence (instantiate) cannot be applied infinitely often
- since the remaining rules also terminate, \multimap 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 $P_{init} \multimap! \emptyset$

- two possible extensions
 - generation of counter-examples
 - handling of non-linear pattern problems
- partial correctness was proven via invariant of \multimap
- termination of \multimap 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 $f(t_1, \dots, t_n)$
- example without confluence

$$f(\text{True}, \text{False}, x) = f(x, x, x)$$

$$f(\dots, \dots, x) = x \quad (\text{all other cases})$$

$$\text{coin} = \text{True}$$

$$\text{coin} = \text{False}$$

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

$$f(\text{True}, \text{False}, \text{coin}) \hookrightarrow f(\text{coin}, \text{coin}, \text{coin}) \hookrightarrow^2 f(\text{True}, \text{False}, \text{coin}) \hookrightarrow \dots$$

Subterm Relation and Innermost Evaluation

- define \triangleright as the strict **subterm relation** and \trianglerighteq as its reflexive closure

$$\frac{}{F(t_1, \dots, t_n) \triangleright t_i} \qquad \frac{t_i \triangleright s}{F(t_1, \dots, t_n) \triangleright s}$$

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

$$\frac{s_i \overset{i}{\hookrightarrow} t_i}{F(s_1, \dots, s_i, \dots, s_n) \overset{i}{\hookrightarrow} F(s_1, \dots, t_i, \dots, s_n)} \text{ rewrite in context}$$

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

- example

$$f(\text{True}, \text{False}, \text{coin}) \not\overset{i}{\hookrightarrow} f(\text{coin}, \text{coin}, \text{coin})$$

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

Strong Normalization

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

$$a_1 \succ a_2 \succ a_3 \succ \dots$$

- 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 $SN(\hookrightarrow)$
- only reason for potential non-termination: recursive calls
- for each recursive call of equation $f(t_1, \dots, t_n) = \ell = r \triangleright f(s_1, \dots, s_n)$ build one **dependency pair** with fresh (constructor) symbol f^\sharp :

$$f^\sharp(t_1, \dots, t_n) \rightarrow f^\sharp(s_1, \dots, s_n)$$

define DP as the set of all dependency pairs

- example program for Ackermann function has three dependency pairs

$$\text{ack}(\text{Zero}, y) = \text{Succ}(y)$$

$$\text{ack}(\text{Succ}(x), \text{Zero}) = \text{ack}(x, \text{Succ}(\text{Zero}))$$

$$\text{ack}(\text{Succ}(x), \text{Succ}(y)) = \text{ack}(x, \text{ack}(\text{Succ}(x), y))$$

$$\text{ack}^\sharp(\text{Succ}(x), \text{Zero}) \rightarrow \text{ack}^\sharp(x, \text{Succ}(\text{Zero}))$$

$$\text{ack}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{ack}^\sharp(x, \text{ack}(\text{Succ}(x), y))$$

$$\text{ack}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{ack}^\sharp(\text{Succ}(x), y)$$

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

$$s_1\sigma_1 \rightarrow t_1\sigma_1 \xrightarrow{\hookrightarrow^*} s_2\sigma_2 \rightarrow t_2\sigma_2 \xrightarrow{\hookrightarrow^*} s_3\sigma_3 \rightarrow t_3\sigma_3 \xrightarrow{\hookrightarrow^*} \dots$$

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

- $s_i\sigma_i \rightarrow t_i\sigma_i$ represent the “main” recursive calls that may lead to non-termination
- $t_i\sigma_i \xrightarrow{\hookrightarrow^*} s_{i+1}\sigma_{i+1}$ corresponds to evaluation of arguments of recursive calls
- **theorem**: $SN(\hookrightarrow)$ 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

$$\text{minus}(x, \text{Zero}) = x$$

$$\text{minus}(\text{Succ}(x), \text{Succ}(y)) = \text{minus}(x, y)$$

$$\text{div}(\text{Zero}, \text{Succ}(y)) = \text{Zero}$$

$$\text{div}(\text{Succ}(x), \text{Succ}(y)) = \text{Succ}(\text{div}(\text{minus}(x, y), \text{Succ}(y)))$$

$$\text{minus}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{minus}^\sharp(x, y)$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

- example innermost evaluation

$$\text{div}(\text{Succ}(\text{Zero}), \text{Succ}(\text{Zero}))$$

$$\stackrel{i}{\hookrightarrow} \text{Succ}(\text{div}(\text{minus}(\text{Zero}, \text{Zero}), \text{Succ}(\text{Zero})))$$

$$\stackrel{i}{\hookrightarrow} \text{Succ}(\text{div}(\text{Zero}, \text{Succ}(\text{Zero})))$$

$$\stackrel{i}{\hookrightarrow} \text{Succ}(\text{Zero})$$

and its (partial) representation as *DP*-chain

$$\text{div}^\sharp(\text{Succ}(\text{Zero}), \text{Succ}(\text{Zero}))$$

$$\rightarrow \text{div}^\sharp(\text{minus}(\text{Zero}, \text{Zero}), \text{Succ}(\text{Zero}))$$

$$\stackrel{i}{\hookrightarrow}^* \text{div}^\sharp(\text{Zero}, \text{Succ}(\text{Zero}))$$

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 56

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 $i \in \{1, \dots, n\}$
 - **demand** $s_i \succeq t_i$ for all $f^\sharp(s_1, \dots, s_n) \rightarrow f^\sharp(t_1, \dots, t_n) \in P$
 - define $P_{\triangleright} = \{f^\sharp(s_1, \dots, s_n) \rightarrow f^\sharp(t_1, \dots, t_n) \in P \mid s_i \triangleright t_i\}$
 - 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. \succeq 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

$$\text{ack}^\sharp(\text{Succ}(x), \text{Zero}) \rightarrow \text{ack}^\sharp(x, \text{Succ}(\text{Zero})) \quad (1)$$

$$\text{ack}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{ack}^\sharp(x, \text{ack}(\text{Succ}(x), y)) \quad (2)$$

$$\text{ack}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{ack}^\sharp(\text{Succ}(x), y) \quad (3)$$

$$\text{minus}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{minus}^\sharp(x, y) \quad (4)$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y)) \quad (5)$$

$$\text{plus}^\sharp(\text{Succ}(x), y) \rightarrow \text{plus}^\sharp(y, x) \quad (6)$$

- it is easy to remove (4) by choosing any argument of minus^\sharp
- we can remove (1) and (2) by choosing argument 1 of ack^\sharp
- afterwards we can remove (3) by choosing argument 2 of 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

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

such that all $s_j \rightarrow t_j \in P$, all s_j and t_j have f^\sharp as root and there are infinitely many $s_j \rightarrow t_j \in P_\triangleright$; perform proof by contradiction

- hence all $s_j \rightarrow t_j$ are of the form $f^\sharp(s_{j,1}, \dots, s_{j,n}) \rightarrow f^\sharp(t_{j,1}, \dots, t_{j,n})$
- 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 $s_j \rightarrow t_j \in P_\triangleright$ then $s_{j,i} \triangleright t_{j,i}$ and thus $s_{j,i}\sigma_j \triangleright t_{j,i}\sigma_j$
- we further know $t_{j,i}\sigma_j \xrightarrow{i}^* s_{j+1,i}\sigma_{j+1}$ 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 \succeq
 - the size-change graph G of a dependency pair $f^\sharp(s_1, \dots, s_n) \rightarrow f^\sharp(t_1, \dots, t_n)$ defines E as follows
 - $i_{in} \xrightarrow{\succ} j_{out} \in E$ whenever $s_i \triangleright t_j$ (strict decrease)
 - $i_{in} \xrightarrow{\succeq} j_{out} \in E$ 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

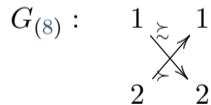
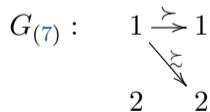
Example – Size-Change Graphs

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

$$f^\sharp(\text{Succ}(x), y) \rightarrow f^\sharp(x, \text{Succ}(x)) \quad (7)$$

$$f^\sharp(x, \text{Succ}(y)) \rightarrow f^\sharp(y, x) \quad (8)$$

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

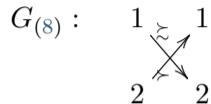
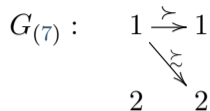


Multigraphs and Concatenation

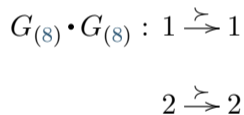
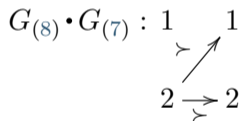
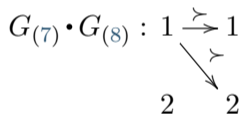
- graphs can be glued together, tracing size-changes in chains, i.e., subsequent dependency pairs
- definition: let \mathcal{G} be a set of size-change graphs for the same symbol f^\sharp ; then the set of **multigraphs** for f^\sharp is defined as follows
 - every $G \in \mathcal{G}$ is a multigraph
 - 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 $i \rightarrow j \in E_1$ and $j \rightarrow k \in E_2$, then $i \rightarrow k \in E$
 - if at least one of the edges $i \rightarrow j$ and $j \rightarrow k$ is labeled with \succ then $i \rightarrow k$ is labeled with \succ , otherwise with \succsim
 - if the previous rules would produce two edges $i \xrightarrow{\succ} k$ and $i \xrightarrow{\succsim} k$, 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

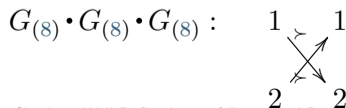
- consider size-change graphs



- this leads to three maximal multigraphs

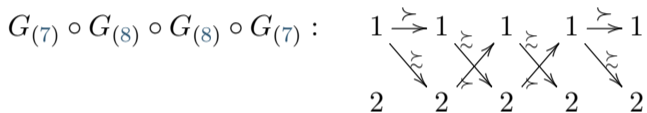


- and a non-maximal multigraph



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:



- **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 $\xrightarrow{\gamma}$ -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 $\xrightarrow{\gamma}$ -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 $i \xrightarrow{\gamma} i$
- 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 $G = G_1 \cdot \dots \cdot G_n$ with each $G_i \in \mathcal{G}$
- consider infinite graph $G_1 \circ \dots \circ G_n \circ G_1 \circ \dots \circ G_n \circ \dots$
- because of size-change termination, this graph contains path with infinitely many $\xrightarrow{\gamma}$ -edges
- hence $G \circ G \circ \dots$ also has a path with infinitely many $\xrightarrow{\gamma}$ -edges
- on this path some index i must be visited infinitely often
- hence there is a path of length k such that $G \circ G \circ \dots \circ G$ (k -times) contains a path from the leftmost argument i to the rightmost argument i with at least one $\xrightarrow{\gamma}$ -edge
- consequently $G \cdot G \cdot \dots \cdot G$ (k -times) contains an edge $i \xrightarrow{\gamma} i$
- by maximality, $G = G \cdot G \cdot \dots \cdot G$, and thus G contains an edge $i \xrightarrow{\gamma} i$

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., $c : X^{(n)} \rightarrow C$
- **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 $G_0 \circ G_1 \circ G_2 \circ \dots$
- for $n < m$ define $G_{n,m} = G_n \cdot \dots \cdot G_{m-1}$
- 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 $n < m$
 $(n = 2, C = \text{multigraphs}, X = \mathbb{N}, c(\{n, m\}) = G_{\min\{n,m\}, \max\{n,m\}})$
- G is maximal: for $n_1 < n_2 < n_3$ with $\{n_1, n_2, n_3\} \subseteq I$, we have
 $G_{n_1, n_3} = G_{n_1} \cdot \dots \cdot G_{n_2-1} \cdot G_{n_2} \cdot \dots \cdot G_{n_3-1} = G_{n_1, n_2} \cdot G_{n_2, n_3}$, and thus $G = G \cdot G$
- by assumption, G contains edge $i \xrightarrow{\succ} i$
- let $I = \{n_1, n_2, \dots\}$ with $n_1 < n_2 < \dots$ and obtain

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

so that edge $i \xrightarrow{\succ} i$ of G delivers path with infinitely many $\xrightarrow{\succ}$ -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., $c : X^{(n)} \rightarrow C$
 - 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 $c' : Y_0^{(m)} \rightarrow C$ 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, \dots , colors $c_0, c_1, c_2 \dots$ and sets X_0, X_1, X_2, \dots satisfying the above properties
- since C is finite there must be some color d in the infinite list c_0, c_1, \dots 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 $\{a_{i_1}, \dots, a_{i_n}\}$ with $i_1 < \dots < i_n$; 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 $i \xrightarrow{\succ} i$), 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

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

Termination – Reduction Pairs

Reduction Pairs

- recall definition: P -chain is sequence

$$s_1\sigma_1 \rightarrow t_1\sigma_1 \xrightarrow{\hookrightarrow^*} s_2\sigma_2 \rightarrow t_2\sigma_2 \xrightarrow{\hookrightarrow^*} s_3\sigma_3 \rightarrow t_3\sigma_3 \xrightarrow{\hookrightarrow^*} \dots$$

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

- previously we used \triangleright on $s_i \rightarrow t_i$ to ensure decrease $s_i\sigma_i \triangleright t_i\sigma_i$
- previously we used $s_i\sigma \in NF(\hookrightarrow)$ and \succeq to turn \hookrightarrow^* into $=$
- now generalize \triangleright 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: $s \xrightarrow{\succ} t \implies s\sigma \xrightarrow{\succ} t\sigma$
 - \succsim is closed under contexts: $s \succsim t \implies F(\dots, s, \dots) \succsim F(\dots, t, \dots)$
 - note: \succ does not have to be closed under contexts

Applying Reduction Pairs

- recall definition: P -chain is sequence

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

such that all $s_i \rightarrow t_i \in P$ and all $s_i\sigma \in NF(\hookrightarrow)$

- demand $s \succsim t$ for all $s \rightarrow t \in P$ 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 $P_\succ = \{s \rightarrow t \in P \mid s \succ t\}$
- 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

$$\text{minus}(x, \text{Zero}) = x$$

$$\text{minus}(\text{Succ}(x), \text{Succ}(y)) = \text{minus}(x, y)$$

$$\text{div}(\text{Zero}, \text{Succ}(y)) = \text{Zero}$$

$$\text{div}(\text{Succ}(x), \text{Succ}(y)) = \text{Succ}(\text{div}(\text{minus}(x, y), \text{Succ}(y)))$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

- constraints

$$\text{minus}(x, \text{Zero}) \rightsquigarrow x$$

$$\text{minus}(\text{Succ}(x), \text{Succ}(y)) \rightsquigarrow \text{minus}(x, y)$$

$$\text{div}(\text{Zero}, \text{Succ}(y)) \rightsquigarrow \text{Zero}$$

$$\text{div}(\text{Succ}(x), \text{Succ}(y)) \rightsquigarrow \text{Succ}(\text{div}(\text{minus}(x, y), \text{Succ}(y)))$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \succ \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

Usable Equations

$$\text{div}^\#(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{div}^\#(\text{minus}(x, y), \text{Succ}(y))$$

- requiring $\ell \succsim r$ 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 only requires **minus**
- definition: the **usable equations** \mathcal{U} w.r.t. a set P are program equations of those symbols that occur in P or that are invoked by (other) usable equations; formally, let \mathcal{E} be set of equations of program, let $\text{root}(f(\dots)) = f$; then \mathcal{U} is defined as

$$\frac{s \rightarrow t \in P \quad t \triangleright u \quad \ell = r \in \mathcal{E} \quad \text{root } u = \text{root } \ell}{\ell = r \in \mathcal{U}}$$

$$\frac{\ell' = r' \in \mathcal{U} \quad r' \triangleright u \quad \ell = r \in \mathcal{E} \quad \text{root } u = \text{root } \ell}{\ell = r \in \mathcal{U}}$$

- observation whenever $t_i \sigma_i \xrightarrow{c_i}^* s_{i+1} \sigma_{i+1}$ in chain, then only usable equations of $\{s_i \rightarrow t_i\}$ can be used

Applying Reduction Pairs with Usable Equations

- recall definition: P -chain is sequence

$$s_1\sigma_1 \rightarrow t_1\sigma_1 \xrightarrow{c_i^*} s_2\sigma_2 \rightarrow t_2\sigma_2 \xrightarrow{c_i^*} s_3\sigma_3 \rightarrow t_3\sigma_3 \xrightarrow{c_i^*} \dots$$

such that all $s_i \rightarrow t_i \in P$ and all $s_i\sigma \in NF(\hookrightarrow)$

- choose a symbol $f^\#$ and define $P_{f^\#} = \{s \rightarrow t \in P \mid \text{root } s = f^\#\}$
- demand $s \succsim t$ for all $s \rightarrow t \in P_{f^\#}$
- demand $l \succsim r$ for all $l = r \in \mathcal{U}$ where \mathcal{U} are usable equations w.r.t. $P_{f^\#}$
- define $P_\succ = \{s \rightarrow t \in P_{f^\#} \mid s \succ t\}$
- 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

$$\text{minus}(x, \text{Zero}) = x$$

$$\text{minus}(\text{Succ}(x), \text{Succ}(y)) = \text{minus}(x, y)$$

$$\text{div}(\text{Zero}, \text{Succ}(y)) = \text{Zero}$$

$$\text{div}(\text{Succ}(x), \text{Succ}(y)) = \text{Succ}(\text{div}(\text{minus}(x, y), \text{Succ}(y)))$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \rightarrow \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

- constraints

$$\text{minus}(x, \text{Zero}) \succsim x$$

$$\text{minus}(\text{Succ}(x), \text{Succ}(y)) \succsim \text{minus}(x, y)$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \succ \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

- 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

$$\text{minus}(x, \text{Zero}) \rightsquigarrow x$$

$$\text{minus}(\text{Succ}(x), \text{Succ}(y)) \rightsquigarrow \text{minus}(x, y)$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \succ \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

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 $p_F(x_1, \dots, x_n)$
- variables in polynomials range over \mathbb{N} and polynomials have to be **weakly monotone**

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

sufficient criterion: forbid subtraction and negative numbers in p_F

- interpretation is lifted to terms by composing polynomials

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket F(t_1, \dots, t_n) \rrbracket &= p_F(\llbracket t_1 \rrbracket, \dots, \llbracket t_n \rrbracket) \end{aligned}$$

- (\succsim) is defined as

$$s \succsim t \text{ iff } \forall \vec{x} \in \mathbb{N}^*. \llbracket s \rrbracket (\geq) \llbracket t \rrbracket$$

- (\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

$$\text{minus}(x, \text{Zero}) \succsim x$$

$$\text{minus}(\text{Succ}(x), \text{Succ}(y)) \succsim \text{minus}(x, y)$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \succ \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

and polynomial interpretation

$$p_{\text{minus}}(x_1, x_2) = x_1$$

$$p_{\text{Zero}} = 2$$

$$p_{\text{Succ}}(x_1) = 1 + x_1$$

$$p_{\text{div}^\sharp}(x_1, x_2) = x_1 + 3x_2$$

we obtain polynomial constraints

$$\llbracket \text{minus}(x, \text{Zero}) \rrbracket = x \geq x = \llbracket x \rrbracket$$

$$\llbracket \text{minus}(\text{Succ}(x), \text{Succ}(y)) \rrbracket = 1 + x \geq x = \llbracket \text{minus}(x, y) \rrbracket$$

$$\llbracket \text{div}^\sharp(\text{Succ} \dots) \rrbracket = 4 + x + 3y > 3 + x + 3y = \llbracket \text{div}^\sharp(\text{minus} \dots) \rrbracket$$

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

$$p_F(x_1, \dots, x_n) = F_0 + F_1x_1 + \dots + F_nx_n$$

where the F_i are symbolic coefficients

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

concrete interpretation above becomes symbolic

$$p_{\text{minus}}(x_1, x_2) = m_0 + m_1x_1 + m_2x_2$$

$$p_{\text{Zero}} = Z_0$$

$$p_{\text{Succ}}(x_1) = S_0 + S_1x_1$$

$$p_{\text{div}\#}(x_1, x_2) = d_0 + d_1x_1 + d_2x_2$$

Symbolic Polynomial Constraints

- given constraints

$$\text{minus}(x, \text{Zero}) \succsim x$$

$$\text{minus}(\text{Succ}(x), \text{Succ}(y)) \succsim \text{minus}(x, y)$$

$$\text{div}^\sharp(\text{Succ}(x), \text{Succ}(y)) \succ \text{div}^\sharp(\text{minus}(x, y), \text{Succ}(y))$$

- obtain symbolic polynomial constraints

$$m_0 + m_1x + m_2Z_0 \geq x$$

$$m_0 + m_1(S_0 + S_1x) + m_2(S_0 + S_1y) \geq m_0 + m_1x + m_2y$$

$$d_0 + d_1(S_0 + S_1x) + d_2(S_0 + S_1y) > d_0 + d_1(m_0 + m_1x + m_2y) + d_2(S_0 + S_1y)$$

- and simplify to

$$(m_0 + m_2Z_0) + (m_1 - 1)x \geq 0$$

$$(m_1S_0 + m_2S_0) + (m_1S_1 - m_1)x + (m_2S_1 - m_2)y \geq 0$$

$$(d_1S_0 - d_1m_0 - 1) + (d_1S_1 - d_1m_1)x + (-d_1m_2)y \geq 0$$

Absolute Positiveness – Symbolic Example

- on symbolic polynomial constraints

$$(m_0 + m_2 Z_0) + (m_1 - 1)x \geq 0$$

$$(m_1 S_0 + m_2 S_0) + (m_1 S_1 - m_1)x + (m_2 S_1 - m_2)y \geq 0$$

$$(d_1 S_0 - d_1 m_0 - 1) + (d_1 S_1 - d_1 m_1)x + (-d_1 m_2)y \geq 0$$

absolute positiveness works as before; obtain constraints

$$m_0 + m_2 Z_0 \geq 0$$

$$m_1 - 1 \geq 0$$

$$m_1 S_0 + m_2 S_0 \geq 0$$

$$m_1 S_1 - m_1 \geq 0$$

$$m_2 S_1 - m_2 \geq 0$$

$$d_1 S_0 - d_1 m_0 - 1 \geq 0$$

$$d_1 S_1 - d_1 m_1 \geq 0$$

$$-d_1 m_2 \geq 0$$

- 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 $m_0 \geq 0, m_1 \geq 0, m_2 \geq 0, S_0 \geq 0, S_1 \geq 0, Z_0 \geq 0, \dots$

Constraint Solving by Hand – Example

- original constraints

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

- delete trivial constraints

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

- conclusions

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

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, TTT₂, Terminator, ...
 - trusted checker: CeTA; soundness formally proven in Isabelle/HOL

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