universität innsbruck

SS 2021

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
  - known: algorithm for checking pattern disjointness
  - missing: algorithm for type-inference
  - 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

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

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

Interlude: Maybe-Type for Errors

- recall type-checking algorithm (variable case omitted)

```
type_check :: Sig -> Vars -> Term -> Maybe Type
type_check sigma vars (Fun f ts) = do
  (tys_in,ty_out) <- sigma f
  tys_ts <- mapM (type_check sigma vars) ts
  if tys_ts == tys_in then return ty_out 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

```
type_check :: Sig -> Vars -> Term -> Maybe Type
type_check sig vars (Var x) = vars x
type_check sigma vars (Fun f ts) = do
  (tys_in,ty_out) <- sigma f
  tys_ts <- mapM (type_check sigma vars) ts
  if tys_ts == tys_in then return ty_out else Nothing
```

- with new abstract types and functions

```
type_check :: Sig -> Vars -> Term -> Check Type
type_check sig vars (Var x) = vars x
type_check sigma vars (Fun f ts) = do
  (tys_in,ty_out) <- sigma f
  tys_ts <- mapM (type_check sigma vars) ts
  assert (tys_ts == tys_in)
  return ty_out
```

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

Back to Type-Checking and Type-Inference

- known: type-checking algorithm
  ```
  type_check :: Sig -> Vars -> Term -> Check Type
  ```
  - `type Sig = FSym -> Check ([Type], Type)` – Σ
  - `type Vars = Var -> Check Type` – $\mathcal{V}$
  - `type_check` 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
  ```
  infer_type :: Sig -> Type -> Term -> Check [(Var,Type)]
  ```
- then type-checking an equation without explicit `Vars` is possible
  ```
  type_check_eqn :: Sig -> (Term, Term) -> Check ()
  type_check_eqn sigma (Var x, r) = failure
  type_check_eqn sigma (l @ (Fun f _), r) = do
    (_,ty) <- sigma f
    vars <- infer_type sigma ty l
    ty_r <- type_check sigma (\ x -> lookup x vars) r
    assert (ty == ty_r)
  ```

Type-Inference Algorithm

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

```
infer_type :: Sig -> Type -> Term -> Check [(Var,Type)]
infer_type sig ty (Var x) = return [(x,ty)]
infer_type sig ty (Fun f ts) = do
  (tys_in,ty_out) <- sig f
  assert (length tys_in == length ts)
  assert (ty_out == ty)
  vars_l <- mapM (\ (ty, t) -> infer_type sig ty t) (zip tys_in ts)
  let vars = nub (concat vars_l) -- 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 $\mathit{infer\_type}\ \Sigma\ \tau\ t = \mathit{return}\ (\mathcal{V} \cap \mathcal{V}ars(t))$
  - if $\mathit{infer\_type}\ \Sigma\ \tau\ t = \mathit{return}\ \mathcal{V}$ 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 2/35–42
- note that 'if $t \in \mathcal{T}(\Sigma, \mathcal{V})_\tau$ then $\mathit{infer\_type}\ \Sigma\ \tau\ t \neq \mathit{failure}$' 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 wrt. slides 3/4 (data-type definitions), 3/15 (function definitions) and partly 3/45 (well-definedness)
  - `infer_type :: Sig -> Type -> Term -> Check [(Var,Type)]`
  - `type_check :: Sig -> Vars -> Term -> Check Type`
  - `type_check_eqn :: 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`)

```
type_check :: Sig -> Vars -> Term -> Check Type
type_check sigma vars (Var x) = vars x
type_check sigma vars (Fun f ts) = do
  (tys_in,ty_out) <- sigma f
  tys_ts <- mapM (type_check sigma vars) ts
  assert (tys_ts == tys_in)
  return ty_out
```

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

```
type_check :: Sig -> Vars -> Term -> Check Type
type_check sigma vars (Var x) = ...
type_check sigma vars t@(Fun f ts) = do
  ...
  assert (tys_ts == tys_in) (show t ++ " ill-typed")
  ...
```

Changing Algorithms for Checking Properties, Continued

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

```
type_check_eqn sigma (Var x, r) = failure
type_check_eqn sigma (l @ (Fun f _), r) = do
  (_,ty) <- sigma f
  vars <- infer_type sigma ty l
  ty_r <- type_check sigma (\ x -> lookup x vars) r
  assert (ty == ty_r)
```

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

```
type_check_eqn sigma (Var x, r) = failure "var as lhs"
type_check_eqn sigma (l @ (Fun f _), r) = do
  ...
  ty_r <- type_check sigma (\ x -> lookup x vars) r
  assert (ty == ty_r) "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

type_check_eqn sigma (Var x, r) = failure "var as lhs"
type_check_eqn sigma (l @ (Fun f _), r) = do
  (_,ty) <- sigma f
  vars <- infer_type sigma ty l
  ty_r <- type_check
     sigma
     (\ x -> maybeToEither
         (x ++ " is unknown variable")
         (lookup x vars))
     r
  assert (ty == ty_r) "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

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

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 a = ... -- Maybe a or Either String a

type Type = String
type Var  = String
type FSym = String
type Vars = Var -> Check Type
type FSym_Info = ([Type], Type)
type Sig = FSym -> Check FSym_Info

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

New Auxiliary Function for Error Monad

```
is_result :: Check a => Bool -- True if argument is not an error
is_result Nothing = False   or   is_result (Left _) = False
is_result _ = True               is_result _ = True
```

Encoding Functional Programs in Haskell

```
-- input: unchecked data-type definitions and function definitions
data Data_Definition = Data Type [(FSym, FSym_Info)]
data Function_Definition = ... -- later
type Functional_Prog =
  ([Data_Definition], [Function_Definition])

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

-- checking single data type definition
process_data_definition ::
  Prog_Info -> Data_Definition -> Check Prog_Info
```

Checking a Single Data Definitions

```
process_data_definition
    (Prog_Info tys cons defs eqs)
    (Data ty new_cs)
 = do
    assert (not (elem ty tys))
    let new_tys = ty : tys
    let sigma = sig_list_to_sig (cons ++ defs)
    assert (distinct (map fst new_cs))
    assert (all
       (\ (c,_) -> not (is_result (sigma c))) new_cs)
    assert (all (\ (_,(tys_in,ty_out)) ->
      ty_out == ty &&
      all (\ ty -> elem ty new_tys) tys_in) new_cs)
    assert (any
      (\ (_,(tys_in,_)) -> all (/= ty) tys_in) new_cs)
    return (Prog_Info new_tys (new_cs ++ 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

process_data_definition ::
  Prog_Info -> Data_Definition -> Check Prog_Info
process_data_definition = ... -- previous slide

process_data_definitions ::
  Prog_Info -> [Data_Definition] -> Check Prog_Info
process_data_definitions = foldM process_data_definition
```

Checking Function Definitions wrt. Slide 3/15

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

process_function_definition
  :: Prog_Info -> Function_Definition -> Check Prog_Info
process_function_definition = ... -- exercise

process_function_definitions ::
  Prog_Info -> [Function_Definition] -> Check Prog_Info
process_function_definitions =
  foldM process_function_definition
```

Checking Functional Programs

```
initial_prog_info = Prog_Info [] [] [] []

process_program :: Functional_Prog -> Check Prog_Info
process_program (data_defs, fun_defs) = do
  pi <- process_data_definitions initial_prog_info data_defs
  process_function_definitions pi fun_defs
```

Current State

- `process_program :: Functional_Prog -> Check Prog_Info` is Haskell program to check user provided functional programs, whether they adhere to the specification of functional programs wrt. slides 3/4 and 3/15
- its functional style using error monads permits 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 wrt. 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 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 $U = \{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$
- 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:
    $$\{x_1 \stackrel{?}{=} v_1, \ldots, x_m \stackrel{?}{=} v_m\} \text{ with distinct } x_j\text{'s}$$
    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
  - $\bot$, 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 $U = \{s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n\}$
- output: solution of U via solved form or $\bot$, indicating unsolvability
- algorithm applies $\rightsquigarrow$ as long as possible; $\rightsquigarrow$ is defined as

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

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

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

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

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

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

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
  - $\{x \stackrel{?}{=} y, y \stackrel{?}{=} x\} \stackrel{x/y}{\rightsquigarrow} \{x \stackrel{?}{=} y, y \stackrel{?}{=} y\} \rightsquigarrow \{x \stackrel{?}{=} y\}$
  - $\{x \stackrel{?}{=} y, y \stackrel{?}{=} x\} \stackrel{y/x}{\rightsquigarrow} \{x \stackrel{?}{=} x, y \stackrel{?}{=} x\} \rightsquigarrow \{y \stackrel{?}{=} x\}$

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 = apply_subst (\ y -> if y == x then t else Var y)

unify :: [(Term, Term)] -> Maybe [(Var, Term)]
unify u = unify_main u []

unify_main :: [(Term, Term)] -> [(Var,Term)] -> Maybe [(Var, Term)]
unify_main [] v = Just v                              -- return solved form
unify_main ((Fun f ts, Fun g ss) : u) v =
  if f == g && length ts == length ss
  then unify_main (zip ts ss ++ u) v                  -- decompose
  else Nothing                                        -- clash
unify_main ((Fun f ts, x) : u) v =
  unify_main ((x, Fun f ts) : u) v                    -- swap
unify_main ((Var x, t) : u) v =
  if Var x == t then unify_main u v                   -- delete
  else if x `elem` vars_term t then Nothing           -- occurs check
  else unify_main                                     -- 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

- non-trivial to prove soundness of implementation, since there are several differences wrt. $\rightsquigarrow$
  - $unify\_main$ 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 wrt. $\rightsquigarrow$
  - 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 algorithms 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 wrt. $\rightsquigarrow$ (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
  - $set\ (x : xs) = \{x\} \cup set\ xs$
  - $set\ (xs \mathbin{++} ys) = set\ xs \cup set\ ys$
  - $set\ (zip\ [x_1, \dots, x_n]\ [y_1, \dots, y_n]) = \{(x_1, y_1), \dots, (x_n, y_n)\}$
  - $set\ (map\ f\ [x_1, \dots, x_n]) = \{f\ x_1, \dots, f\ x_n\}$
  - $subst\ x\ t\ s = s\{x/t\}$
  - ...

  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 $set\_maybe\ Nothing = \bot$, $set\_maybe\ (Just\ w) = set\ w$
- property: whenever $(u, v) \sim U$ and $unify\_main\ u\ v = res$ then $U \rightsquigarrow^! set\_maybe\ res$
- 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, $unify\_main\ u\ [] = res$
  - moreover, $(u, []) \sim set\ u$ by definition of $\sim$
  - via property conclude $set\ u \rightsquigarrow^! set\_maybe\ res$
  - at this point apply correctness of $\rightsquigarrow$:
    $set\_maybe\ res$ is the correct answer to the unification problem $set\ u$

Proving the Refinement Property

- property $P(u, v, U)$: $(u, v) \sim U \land unify\_main\ u\ v = res \longrightarrow U \rightsquigarrow^! set\_maybe\ res$
- $(u, v) \sim U \longleftrightarrow U = set\ u \cup set\ v \land set\ v \in NF(\rightsquigarrow) \land \forall (x, t) \in set\ v.\ x \notin \mathcal{V}ars(u)$
- we prove the property $P(u, v, U)$ by induction on u and v wrt. 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 wrt. 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
    (**) $unify\_main\ []\ v = res$
  - from (*) conclude $U = set\ v$ and $set\ v \in NF(\rightsquigarrow)$
  - from (**) conclude $res = Just\ v$ and hence, $set\_maybe\ res = set\ v$
  - we have to show $U \rightsquigarrow^! set\_maybe\ res$, i.e., $set\ v \rightsquigarrow^! set\ v$ which is satisfied since $set\ v \in NF(\rightsquigarrow)$

- $P(u, v, U)$: $(u, v) \sim U \land \mathit{unify\_main}\ u\ v = \mathit{res} \longrightarrow U \leadsto^! \mathit{set\_maybe}\ \mathit{res}$
- $(u, v) \sim U \longleftrightarrow U = \mathit{set}\ u \cup \mathit{set}\ v \land \mathit{set}\ v \in NF(\leadsto) \land \forall (x, t) \in \mathit{set}\ v.\ x \notin \mathcal{V}ars(u)$

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
  (**) $\mathit{unify\_main}\ ((f(ts), g(ss)) : u)\ v = \mathit{res}$

- consider sub-cases
  - $\neg(f = g \land \mathit{length}\ ts = \mathit{length}\ ss)$:
    - from (**) conclude $\mathit{set\_maybe}\ \mathit{res} = \bot$
    - from (*) conclude $f(ts) \stackrel{?}{=} g(ss) \in U$ and hence $U \leadsto \bot$ by (clash)
    - consequently, $U \leadsto^! \mathit{set\_maybe}\ \mathit{res}$
  - $f = g \land \mathit{length}\ ts = \mathit{length}\ ss$:
    - from (**) conclude $\mathit{res} = \mathit{unify\_main}\ ((f(ts), g(ss)) : u)\ v = \mathit{unify\_main}\ (\mathit{zip}\ ts\ ss \mathbin{++} u)\ v$
    - from (*) and alignment for zip and $\mathbin{++}$ conclude $U = \{f(ts) \stackrel{?}{=} g(ss)\} \cup \mathit{set}\ u \cup \mathit{set}\ v$ and hence $U \leadsto \mathit{set}\ (\mathit{zip}\ ts\ ss \mathbin{++} u) \cup \mathit{set}\ v =: V$ by (decompose)
    - we get $P(\mathit{zip}\ ts\ ss \mathbin{++} u, v, V)$ as IH; $(\mathit{zip}\ ts\ ss \mathbin{++} u, v) \sim V$ follows from (*), so $U \leadsto V \leadsto^! \mathit{set\_maybe}\ \mathit{res}$

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

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
  (**) $\mathit{unify\_main}\ ((x,t) : u)\ v = \mathit{res}$

- 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 $\mathit{res} = \mathit{unify\_main}\ ((x,t) : u)\ v = \mathit{unify\_main}\ u'\ ((x,t) : v')$
    - from IH conclude $P(u', (x,t) : v', V)$ and hence, $(u', (x,t) : v') \sim V \longrightarrow V \rightsquigarrow^! \mathit{set\_maybe}\ \mathit{res}$
    - for proving $U \rightsquigarrow^! \mathit{set\_maybe}\ \mathit{res}$ it hence suffices to show $(u', (x,t) : v') \sim V$ and $U \rightsquigarrow V$
    - $U \stackrel{(*)}{=} \{x \stackrel{?}{=} t\} \cup \mathit{set}\ u \cup \mathit{set}\ v \rightsquigarrow (\mathit{set}\ u \cup \mathit{set}\ v)\{x/t\} \cup \{x/t\} = V$
      by (eliminate) because of preconditions

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

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

- case 4 (arguments $(x, t) : u$ and v)
  - other sub-cases: exercise
- case 3 (arguments $(f(ss), x) : u$ and v): exercise
- 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

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

Checking Pattern Completeness

Checking Pattern Completeness

- 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 $f(t_1, \ldots, t_n)$
- idea of abstract algorithm
  - a pattern problem is a set P of pairs (t, L) where
    - t is a term, representing the set of all its constructor ground instances
    - L is a set of left-hand sides that potentially match instances of t
  - initially, $P = \{(f(x_1, \ldots, x_n), \text{set of all lhss of } f\text{-equations}) \mid f \in \mathcal{D}\}$
  - whenever some left-hand side $\ell \in L$ cannot match any instance of t anymore, it can be removed
  - whenever L becomes empty, then no instance of t can be matched
  - whenever all constructor ground instances of t are matched by L, then (t, L) can be removed from P
  - when P becomes empty, pattern completeness should be guaranteed
  - if none of the above is applicable, we instantiate t
- initial task: think about exact statement, what kind of property of pattern problem we are investigating (similar to definition of solution of unification problem)

Semantics of Pattern Problems

- in the following algorithm and proofs, we always consider type-correct terms and substitutions wrt. $\Sigma = \mathcal{C} \cup \mathcal{D}$, but do not mention this explicitly
- a pattern problem is a set P of pairs (t, L) consisting of a term t and a set of terms L
- P is complete if for all $(t, L) \in P$ and all constructor ground substitutions σ there is some $\ell \in L$ that matches $t\sigma$
- obviously, $P = \varnothing$ is complete
- we define $\bot$ as additional pattern problem, which is not complete
- define $L_{init,f}$ as the set of all lhss of f-equations of the program
- define $P_{init} = \{(f(x_1, \ldots, x_n), L_{init,f}) \mid f \in \mathcal{D}\}$
- consequence: program is pattern complete iff P_{init} is complete

Deciding Completeness of Pattern Problems

- we develop abstract algorithm that is similar to abstract unification algorithm, it is defined via a one step relation $\rightharpoonup$ that transforms pattern problems into equivalent simpler problems
- it uses the matching algorithm of slides 3/23–29 (with detailed error results) as auxiliary algorithm
- $P \cup \{(t, \{\ell\} \cup L)\} \rightharpoonup P$, if ℓ matches t (match)
- $P \cup \{(t, \{\ell\} \cup L)\} \rightharpoonup P \cup \{(t, L)\}$, if $match\ \ell\ t$ clashes (clash)
- $P \cup \{(t, \varnothing)\} \rightharpoonup \bot$ (fail)
- $P \cup \{(t, L)\} \rightharpoonup P \cup \{(t\sigma_1, L), \dots, (t\sigma_n, L)\}$, if (split)
  - $\ell \in L$ and $match\ \ell\ t$ results in fun-var-conflict with variable x
  - the type of x is τ
  - τ has n constructors $c_1, \dots, c_n$
  - $\sigma_i = \{x/c_i(x_1, \dots, x_k)\}$ where k is the arity of c_i and the x_i's are distinct fresh variables

Example

consider

$$\mathsf{data}\ \mathsf{Bool} = \mathsf{True} : \mathsf{Bool} \mid \mathsf{False} : \mathsf{Bool}$$

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

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

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

then we have

$$\begin{aligned}
P_{init} &= \{(\mathsf{conj}(x_1, x_2), \{\ell_1, \ell_2, \ell_3\})\} \\
&\rightharpoonup \{(\mathsf{conj}(\mathsf{True}, x_2), \{\ell_1, \ell_2, \ell_3\}), (\mathsf{conj}(\mathsf{False}, x_2), \{\ell_1, \ell_2, \ell_3\})\} \\
&\rightharpoonup \{(\mathsf{conj}(\mathsf{True}, x_2), \{\ell_1, \ell_3\}), (\mathsf{conj}(\mathsf{False}, x_2), \{\ell_1, \ell_2, \ell_3\})\} \\
&\rightharpoonup \{(\mathsf{conj}(\mathsf{True}, x_2), \{\ell_1, \ell_3\}), (\mathsf{conj}(\mathsf{False}, x_2), \{\ell_2, \ell_3\})\} \\
&\rightharpoonup \{(\mathsf{conj}(\mathsf{True}, x_2), \{\ell_1, \ell_3\})\} \\
&\rightharpoonup \{(\mathsf{conj}(\mathsf{True}, \mathsf{True}), \{\ell_1, \ell_3\}), (\mathsf{conj}(\mathsf{True}, \mathsf{False}), \{\ell_1, \ell_3\})\} \\
&\rightharpoonup \{(\mathsf{conj}(\mathsf{True}, \mathsf{False}), \{\ell_1, \ell_3\})\} \\
&\rightharpoonup \varnothing
\end{aligned}$$

Example

consider

$$\mathsf{data}\ \mathsf{Bool} = \mathsf{True} : \mathsf{Bool} \mid \mathsf{False} : \mathsf{Bool}$$

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

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

then we have

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

Partial Correctness of $\rightharpoonup$

- definition: P is complete if for all $(t, L) \in P$ and all constructor ground substitutions σ there is some $\ell \in L$ that matches $t\sigma$
- theorem: whenever $P \rightharpoonup Q$, then P is complete iff Q is complete
- corollary: if $P \rightharpoonup^* \varnothing$ then P is complete,
  and if $P \rightharpoonup^* \bot$ then P is not complete
- proof of theorem
  - (match): $P \cup \{(t, \{\ell\} \cup L)\} \rightharpoonup P$, if ℓ matches t
    - we only have to show that $\{(t, \{\ell\} \cup L)\}$ is complete, i.e., for all constructor ground substitutions σ there must be some $\ell' \in \{\ell\} \cup L$ that matches $t\sigma$
    - since ℓ matches t, we know that $t = \ell\gamma$ for some substitution γ
    - consequently $t\sigma = (\ell\gamma)\sigma = \ell(\gamma\sigma)$, i.e., ℓ matches $t\sigma$ and obviously $\ell \in \{\ell\} \cup L$
  - (fail): $P \cup \{(t, \varnothing)\} \rightharpoonup \bot$
    - both matching problems are not complete: $\bot$ by definition, and for $(t, \varnothing)$ there obviously isn't any $\ell \in \varnothing$ which matches $t\sigma$

Partial Correctness of $\rightharpoonup$, continued

- definition: P is complete if for all $(t, L) \in P$ and all constructor ground substitutions σ there is some $\ell \in L$ that matches $t\sigma$
- proof continued
  - (clash): $P \cup \{(t, \{\ell\} \cup L)\} \rightharpoonup P \cup \{(t, L)\}$, if $\mathit{match}\ \ell\ t$ clashes
    - if suffices to show that ℓ cannot match any instance of t, i.e., $\mathit{match}\ \ell\ (t\sigma)$ will always fail
    - to this end we require an auxiliary property of the matching algorithm
    - for a matching problem M, define $M\sigma = \{(\ell, r\sigma) \mid (\ell, r) \in M\}$, i.e., where σ is applied on rhss, and $\bot\sigma = \bot$
    - lemma: whenever M is transformed into M' by rule (decompose) or (clash), then $M\sigma$ is transformed into $M'\sigma$ by the same rule
    - hence, since $\mathit{match}\ \ell\ t$ clashes, we conclude that $\mathit{match}\ \ell\ (t\sigma)$ clashes

Partial Correctness of $\rightharpoonup$, final part

- definition: P is complete if for all $(t, L) \in P$ and all constructor ground substitutions σ there is some $\ell \in L$ that matches $t\sigma$
- proof continued
  - (split): $P \cup \{(t, L)\} \rightharpoonup P \cup \{(t\sigma_1, L), \dots, (t\sigma_n, L)\}$, 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
    - we only consider one direction of the proof: we assume that the rhs of $\rightharpoonup$ is complete and prove that the lhs is complete
    - to this end, consider an arbitrary constructor ground substitution σ and we have to show that $t\sigma$ is matched by some element of L
    - 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 $\gamma(y) = \begin{cases} t_j, & \text{if } y = x_j \\ \sigma(y), & \text{otherwise} \end{cases}$
    - γ is well-defined since the x_j's are distinct
    - γ is a constructor ground substitution
    - $t\sigma = t\sigma_i\gamma$ since the x_j's are fresh
    - since $(t\sigma_i, L)$ is an element of the rhs of $\rightharpoonup$ and the assumed completeness, we conclude that there is some element of L that matches $(t\sigma_i)\gamma$ and consequently, also $t\sigma$

Correctness of $\rightharpoonup$, Missing Parts

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

$\rightharpoonup$ Cannot Get Stuck

- $P \cup \{(t, \{\ell\} \cup L)\} \rightharpoonup P$, if ℓ matches t (match)
- $P \cup \{(t, \{\ell\} \cup L)\} \rightharpoonup P \cup \{(t, L)\}$, if $match\ \ell\ t$ results in clash (clash)
- $P \cup \{(t, \varnothing)\} \rightharpoonup \bot$ (fail)
- $P \cup \{(t, L)\} \rightharpoonup P \cup \{(t\sigma_1, L), \dots, (t\sigma_n, L)\}$, if (split)
  - $\ell \in L$ and $match\ \ell\ t$ results in fun-var-conflict with variable x and ...
- lemma: whenever P is in normal form wrt. $\rightharpoonup$ and for all $(t, L) \in P$ and all $\ell \in L$, the lhs ℓ is linear, then $P \in \{\varnothing, \bot\}$
- proof by contradiction
  - assume P is such a normal form, $P \notin \{\varnothing, \bot\}$
  - hence, $(t, L) \in P$ for some t and L
  - since (fail) is not applicable, $L \neq \varnothing$, i.e., $\ell \in L$ for some ℓ
  - as (match) is not applicable, $match\ \ell\ t$ must fail
  - as (clash) and (split) are not applicable the failure can only be a var-clash
  - however, a var-clash cannot occur since ℓ is linear

Termination of $\rightharpoonup$

- $P \cup \{(t, \{\ell\} \cup L)\} \rightharpoonup P$, if ℓ matches t (match)
- $P \cup \{(t, \{\ell\} \cup L)\} \rightharpoonup P \cup \{(t, L)\}$, if $match\ \ell\ t$ clashes (clash)
- $P \cup \{(t, \varnothing)\} \rightharpoonup \bot$ (fail)
- $P \cup \{(t, L)\} \rightharpoonup P \cup \{(t\sigma_1, L), \dots, (t\sigma_n, L)\}$, if (split)
  $\ell \in L$ and $match\ \ell\ t$ results in fun-var-conflict with variable x and ...
- clearly, $\rightharpoonup$ without (split) terminates as in every step the size of the pattern problem is reduced
- argumentation that also (split) cannot be applied infinitely often
  - a fun-var-conflict between t and $\ell \in L$ occurs iff the subterm of t at position p is a variable x, but the subterm ℓ at position p is a function application
  - the effect of (split) is that the variable x becomes a constructor, so there is no fun-var-conflict of $t\sigma_i$ with any lhs at position p any more
  - hence, when (split)ting over-and-over again, all possible fun-var-conflicts move to deeper positions
  - since the depths of the conflict positions are bounded by the sizes of the terms in L, all fun-var-conflicts eventually disappear, so that (split) is no longer applicable

Implementing $\rightharpoonup$

- a direct implementation of $\rightharpoonup$ mainly faces two problems (exercise)
  - handling of fresh variable
  - figuring out constructors in (split)
- direct: matching algorithm is started from scratch every time
- an optimized implementation should try to reuse previous runs of matching algorithm after applying (split)
- this will require changes in the interface of matching algorithm

Summary on Pattern Completeness

- pattern completeness of functional programs is decidable:

  $$\text{program is pattern complete iff } P_{init} \rightharpoonup^! \varnothing$$

- partial correctness was proven via invariant of $\rightharpoonup$
- proof required additional properties of matching algorithm
- termination of $\rightharpoonup$ was shown informally
- formal proof would require further properties of matching algorithm
- termination proof was tricky, definitely requiring 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

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

- 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 $\unrhd$ as its reflexive closure

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

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

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

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

- example

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

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 is also equivalent to induction;
  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 wrt. algorithm for terminating programs

Termination Analysis with Dependency Pairs

- aim: prove $SN(\stackrel{i}{\hookrightarrow})$
- only reason for potential non-termination: recursive calls
- for each recursive call of eqn. $f(t_1, \ldots, t_n) = \ell = r \trianglerighteq f(s_1, \ldots, s_n)$ 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

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

  such that all $s_i \to t_i \in P$ and all $s_i\sigma_i \in NF(\hookrightarrow)$
  - $s_i\sigma_i \to t_i\sigma_i$ represent the "main" recursive calls that may lead to non-termination
  - $t_i\sigma_i \overset{i}{\hookrightarrow}^* s_{i+1}\sigma_{i+1}$ corresponds to evaluation of arguments of recursive calls
- theorem: $SN(\overset{i}{\hookrightarrow})$ 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

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

- example innermost evaluation

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

  and its (partial) representation as DP-chain

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

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 57

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 $s_i \trianglerighteq t_i$ for all $f^\sharp(s_1, \dots, s_n) \to f^\sharp(t_1, \dots, t_n) \in P$
  - define $P_\triangleright = \{f^\sharp(s_1, \dots, s_n) \to f^\sharp(t_1, \dots, t_n) \in P \mid s_i \triangleright t_i\}$
  - 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 wrt. $\trianglerighteq$ 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

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

  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 $s_j \to t_j \in P_\rhd$; 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 $s_{j,i} \unrhd t_{j,i}$ of criterion conclude $s_{j,i}\sigma_j \unrhd t_{j,i}\sigma_j$ and if $s_j \to t_j \in P_\rhd$ then $s_{j,i} \rhd t_{j,i}$ and thus $s_{j,i}\sigma_j \rhd t_{j,i}\sigma_j$
- we further know $t_{j,i}\sigma_j \overset{\mathsf{i}}{\hookrightarrow}^* s_{j+1,i}\sigma_{j+1}$ 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 $t_{j,i}\sigma_j \unlhd s_{j,i}\sigma_j \lhd f^\sharp(s_{j,1}\sigma_j, \dots, s_{j,n}\sigma_j) = s_j\sigma_j \in NF(\hookrightarrow)$
- obtain an infinite sequence with infinitely many $\rhd$; this is a contradiction to $SN(\rhd)$

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

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 $V = \{1_{in}, \ldots, n_{in}\}$ and $W = \{1_{out}, \ldots, n_{out}\}$
  - 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 $f^\sharp(s_1, \ldots, s_n) \to f^\sharp(t_1, \ldots, t_n)$ defines E as follows
    - $i_{in} \xrightarrow{\succ} j_{out} \in E$ whenever $s_i \rhd t_j$ (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

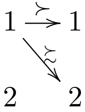
- consider the following dependency pairs; they include permutations that cannot be solved by the subterm criterion

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

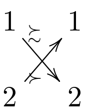
$$\mathsf{f}^\sharp(x, \mathsf{Succ}(y)) \to \mathsf{f}^\sharp(y, x) \tag{8}$$

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

$G_{(7)}$:



$G_{(8)}$:

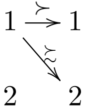


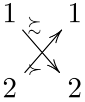
Multigraphs and Concatenation

- 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 $i \stackrel{\succ}{\to} k$ and $i \stackrel{\succsim}{\to} k$, 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

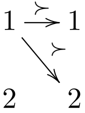
Example – Multigraphs

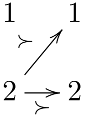
- consider size-change graphs

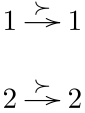
$G_{(7)}$:


$G_{(8)}$:


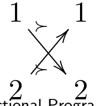
- this leads to three maximal multigraphs

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


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


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


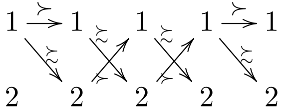
- and a non-maximal multigraph

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


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 $\xrightarrow{\succ}$-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 $\xrightarrow{\succ}$-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 $i \xrightarrow{\succ} i$
- 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 $\xrightarrow{\succ}$-edges
- hence $G \circ G \circ \ldots$ also has a path with infinitely many $\xrightarrow{\succ}$-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 $\xrightarrow{\succ}$-edge
- consequently $G \bullet G \bullet \ldots \bullet G$ (k-times) contains an edge $i \xrightarrow{\succ} i$
- by maximality, $G = G \bullet G \bullet \ldots \bullet G$, and thus G contains an edge $i \xrightarrow{\succ} i$

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 $G_0 \circ G_1 \circ G_2 \circ \dots$
- for $n < m$ define $G_{n,m} = G_n \bullet \dots \bullet G_{m-1}$
- 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 $G_{n_1,n_3} = G_{n_1} \bullet \dots \bullet G_{n_2-1} \bullet G_{n_2} \bullet \dots \bullet G_{n_3-1} = G_{n_1,n_2} \bullet G_{n_2,n_3}$, and thus $G = G \bullet G$
- by assumption, G contains edge $i \overset{\succ}{\to} i$
- let $I = \{n_1, n_2, \dots\}$ with $n_1 < n_2 < \dots$ and obtain

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

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 $i \xrightarrow{\succ} i$), termination is proven
- soundness relies upon Ramsey's theorem
- subsumes subterm criterion
- 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: $s \underset{(\sim)}{\succ} t \longrightarrow s\sigma \underset{(\sim)}{\succ} t\sigma$
  - $\succsim$ 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

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

  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

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

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

- constraints

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

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

Usable Equations

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

- 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 $\mathcal{U}$ wrt. 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 $root\ (f(\dots)) = f$; then $\mathcal{U}$ is defined as

$$\frac{s \to t \in P \quad t \trianglerighteq u \quad \ell = r \in \mathcal{E} \quad root\ u = root\ \ell}{\ell = r \in \mathcal{U}}$$

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

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

Applying Reduction Pairs with Usable Equations

- recall definition: P-chain is sequence

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

  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 $s \succsim t$ for all $s \to t \in P_{f^\sharp}$
- demand $\ell \succsim r$ for all $l = r \in \mathcal{U}$ where $\mathcal{U}$ are usable equations wrt. $P_{f^\sharp}$
- 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

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

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

- constraints

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

- 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

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

  find a suitable reduction pair such that these constraints are satisfied
- many such reductions pair 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 $p_F(x_1, \ldots, x_n)$
- polynomials are over $\mathbb{N}$ and 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

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

- $\underset{(\sim)}{\succ}$ is defined as

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

- $(\succ, \succsim)$ 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

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

  concrete interpretation above becomes symbolic

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

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

$$(m_0 + m_2 Z_0) + (m_1 - 1)x \geq 0$$
$$(m_1 S_0 + m_2 S_0) + (m_1 S_1 - m_1)x + (m_2 S_1 - m_2)y \geq 0$$
$$(d_1 S_0 - d_1 m_0 - 1) + (d_1 S_1 - d_1 m_1)x + (-d_1 m_2)y \geq 0$$

absolute positiveness works as before; obtain constraints

$$m_0 + m_2 Z_0 \geq 0 \qquad m_1 - 1 \geq 0$$
$$m_1 S_0 + m_2 S_0 \geq 0 \qquad m_1 S_1 - m_1 \geq 0 \qquad m_2 S_1 - m_2 \geq 0$$
$$d_1 S_0 - d_1 m_0 - 1 \geq 0 \qquad d_1 S_1 - d_1 m_1 \geq 0 \qquad -d_1 m_2 \geq 0$$

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

Constraint Solving by Hand – Example

- original constraints

$$m_0 + m_2 Z_0 \geq 0 \qquad m_1 - 1 \geq 0$$

$$m_1 S_0 + m_2 S_0 \geq 0 \qquad m_1 S_1 - m_1 \geq 0 \qquad m_2 S_1 - m_2 \geq 0$$

$$d_1 S_0 - d_1 m_0 - 1 \geq 0 \qquad d_1 S_1 - d_1 m_1 \geq 0 \qquad -d_1 m_2 \geq 0$$

- delete trivial constraints

$$m_1 - 1 \geq 0$$

$$m_1 S_1 - m_1 \geq 0 \qquad m_2 S_1 - m_2 \geq 0$$

$$d_1 S_0 - d_1 m_0 - 1 \geq 0 \qquad d_1 S_1 - d_1 m_1 \geq 0 \qquad -d_1 m_2 \geq 0$$

- conclusions

$$m_1 \geq 1 \qquad d_1 \geq 1$$

$$S_0 \geq 1 \qquad S_1 \geq 1$$

$$m_2 = 0 \qquad S_1 \geq m_1 \qquad m_0 = 0$$

Constraint Solving by SMT-Solver – Example

- original constraints

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

- 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