A Context Aware Type System for Tree-Like Data Structures

by

Akhmetzhan Kussainov

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

June 2025

Author .....
Department of Computer Science
7th May, 2025

Certified by .....
Hans de Nivelle
Associate Professor
Thesis Supervisor

Accepted by .....
Yelyzaveta Arkhangelsky
Dean, School of Engineering and Digital Sciences

A Context Aware Type System for Tree-Like Data Structures

by

Akhmetzhan Kussainov

Submitted to the Department of Computer Science
on 7th May, 2025, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In this work, we design a context aware type system for an imperative programming language where all data are tree structures. Tree-like data structures can have different forms with distinct fields based on the way they are constructed. Our type system can determine the forms of data trees from a program context and ensure the correctness of field accesses and function applications. During type checking, the program context is approximated as a function that maps each program point to a set of possible program states. Indexing information is also approximated with special array range states and relative order of indices. This novel treatment of indices allows us to reason about the state of an array at some index without storing the states of individual elements. Once the program approximation is computed, the type-checking algorithm ensures the correctness of function and field applications. Function and field applications are valid if their preconditions are satisfied by the current program state. A program is well-typed if all function applications and field accesses can be resolved to exactly one valid overload in each possible program state. Otherwise, our type-checking algorithm concludes that the program is ill-typed.

Thesis Supervisor: Hans de Nivelle
Title: Associate Professor

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Professor Hans de Nivelle, for his invaluable guidance, continuous support, and unwavering encouragement throughout the course of my research. His insightful feedback, patience, and commitment to excellence have significantly shaped the quality and direction of this thesis.

I am also sincerely thankful to my co-supervisor, Professor Benjamin Tyler, whose expertise, thoughtful suggestions, and collaborative spirit greatly contributed to the development of my work.

I would like to extend my sincere gratitude to my external examiner, Professor Witold Charatonics at the University of Wroclaw, for taking the time to review my thesis and provide thoughtful, constructive feedback.

Their support and constructive criticism helped me refine my ideas and approach the research from different perspectives. Together, their mentorship has been instrumental in my academic journey, and I am truly grateful for the opportunity to work under their supervision.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this project, we design a context aware type system for an experimental imperative programming language where all data are trees. Data trees can have different forms with distinct field layouts based on the way they are constructed. Thus, deciding whether an operation applied on a data tree is valid requires knowing its form. In order to determine the form of a data tree, we approximate the program behaviour to copture the way data are constructed and modified at runtime. Therefore, our type system requires approximated context information to type check a program. The approximation of program context is a function which maps every program point to a set of possible program states. A single program point can be mapped to multiple program states since different computations can end up in the same program point with different values for variables. Moreover, transitions to a next possible state becomes non-deterministic since two states in the current program point can transition to different states. Therefore, computational model of the approximated program is non-deterministic. Once the program context is approximated, out type checking algorithm can verify correctness of operations applied on data trees. A program is well-typed if all function applications can be resolved to exactly one valid overload in each possible state. If there is no valid overload then the program is ill-typed. If there are several valid overloads then the function application is ambiguous and we can conclude ill-type or choose one overload based on some priority rules.

This project is an extension of the TreeGen project developed by Professor Hans de

Nivelle [16]. TreeGen can generate tree-like data structures in C++ with automatic memory management from inductive type definitions. The type system developed in this project allows to resolve function overloads on tree data structures based on the program context. Without it, selecting overloads requires dynamic type casting to subclasses in object-oriented programming or pattern matching in functional programming. Previously, in TreeGen, this task required creating instance of appropriate view obejct.

Let's consider a tree data structure that represents simple propositional logic, denoted as `Prop`. Propositional formulas of type `Prop` can have four distinct forms. Atom is a character array and negation is an unary operation with subtree of of type `Prop`. Implication and equivalence share the same form which is a binary operation on `Prop`'s. Conjuction and disjunction operations also share the same form with unrestricted arities. We define the propositional formulas as follows:

Example 1.0.1.
```
typedef Prop = {
    ?atom            => [ c : char ];
    ?not             => sub : Prop;
    ?implies, ?equiv => sub1 : Prop, sub2 : Prop;
    ?or, ?and        => [ sub : Prop ];
};

typedef Atom = ( ?atom, all( char ) );
typedef Literal = Atom || ( ?not, Atom );
typedef ArrowFree = Atom || (?not, ArrowFree)
                    || ( ?or || ?and, all( ArrowFree ) );
typedef NNF = Literal || ( ?or || ?and, all( NNF ) );
```

Along with the base type `Prop`, we have defined several adjective types. These adjectives express additional constraints on the base type. Note that, tuples of arbitrary arity are only constructors allowed in adjective definitions (e.g. (?not, Atom)). `Atom` represents propositional formulas only consisting of atoms. `Literal` represents for-

mulas consisting of atoms and their negations. These two adjectives are not useful on their own but, rather, they are used as parts of other adjectives to modularize type definitions. `ArrowFree` represents formulas consisting of atoms, negations, disjunctions, and conjuctions. It is used to ensure that subformulas involving implication and equivalence are replaced by their logical identities. `NNF` represents formulas in normal negation form.

Let's now consider a function that transforms an arbitrary propositional formula into its negation normal form. We first define `make_arrowfree` for prepocessing formulas to eliminate implication and equivalence operations in subterms. Next we define `make_nnf_pos` and `make_nnf_neg` which transform formulas without implication and equivalence operations into NNF formulas with positive and negative polarities, respectively. All three functions recursively iterate on the form of the argument formula `f` where the selector field `f.sel` is used to distinguish different forms.

Example 1.0.2.
```
Prop && ArrowFree make_arrowfree( f : Prop ) {
   switch( f. sel ) {
   case ?atom:
      return f;
   case ?not:
      return ( ?not, make_arrowfree( f. sub ));
   case ?implies:
      return ( ?or, ( ?not, make_arrowfree( f. sub1 )),
                    ( make_arrowfree( s. sub2 )) );
   case ?equiv:
      var s1 = make_arrowfree( f. sub1 );
      var s2 = make_arrowfree( f. sub2 );
      return ( ?and, ( ?or, ( ?not, s1 ), s2 ),
                     ( ?or, s1, ( ?not, s2 )) );

   case ?or, ?and:
      var f2 = ( f. sel, ( ));
```

```
      for( i : u64 = 0; i < f. length; ++ i )
         f2. push_back( make_arrowfree( f. sub[i] ) );
      return f2;
}

Prop && NNF make_nnf_pos( f : Prop && ArrowFree ) {
   switch( f. sel ) {
   case ?atom:
      return f;
   case ?not:
      return make_nnf_neg( f. sub );
   case ?or, ?and:
      var f2 = ( f. sel( ));
      for( i : u64 = 0; i < f. length; ++ i )
         f2. push_back( make_nnf_pos( f. sub[i] ));
      return f2;
   }
}

Prop && NNF make_nnf_neg( f : Prop && ArrowFree ) {
   switch( f. sel ) {
   case ?atom:
      return ( ?neg, f );
   case ?not:
      return make_nnf_pos( f. sub );
   case ?or, ?and:
      var s = ?or;
      if( f. sel( ) == ?or )
         s = ?and;
```

```
        var f2 = ( s, ( ));
        for( i : u64 = 0; i < f. length; ++ i )
            f2. push_back( make_nnf_neg( f. sub[i] ));
        return f2;
    }
}
```

Note that functions' argument and return types are annotated with `ArrowFree` and `NNF` to enforce desired behaviours. That is, `make_arrowfree` is well-typed if and only if there exists a proof that given an arbitrary propositional formula, all possible executions of the function always construct formula which does not contain implication and equivalence operations in its subterms. Similarly, `make_nnf_pos` and `make_nnf_neg` are well-typed if and only if there exists a proof that these functions always construct formulas in negation normal form.

In practice, we do not construct proofs, instead we approximate possible program states and check for correctness of function applications. First, we construct an intermediate representation for a given program. Usually, intermediate representation are constructed after type checking. However, our type system requires inspection of intermediate representation before type checking since forms of data trees can depend on loop invariants. Along with the intermediate representation, data type definitions are compiled into a deterministic tree automaton which is used to determine the state of a given data term. The tree automaton finitely partitions the domain of all data terms into equivalence classes where any two elements of the same class have the same properties. A field map and function map associate the field and function names with state conditions which specify a set of states a data term should have for a function or field application to be valid. A field map and a function map are constructed from the data type definitions. The field map associates identifiers to field possitions and state conditions while the function map associates identifiers to function state conditions. Indexing information is approximated with special array range states and relative order of indices. This range interpretation allows to reason about the state of an array at some index without storing states of inividual elements. Once all

structures are constructed, we approximate the program. Here, an approximation is a function that maps each program point to a set of possible states. The field and function names can be overloaded, then all overload candidates are inseartend into the intermediate representation as a special branch statemets. Function and field applications are valid if there is exactly one overload whose precondtion is satisfied by the current program state. Otherwise, our type checking algorithm concludes that the program is ill-typed.

Contribution of this project is a novel set based abstraction of arrays. This approximation allows to infer the state of an array at some index without storing the states of individual elements. The index information can be recovered from a carefully desingned interpretation of array ranges.

This project is related to the studies in abstract interpretations and set based approximations.

Chapter 2

Related Work

In their early days, abstract interpretation and set-based approximation were considered distinct techniques due to the non-iterative nature of the least model computation, unlike the least fixed point computation. The equivalence between these two program approximation techniques was established by Patrick Cousot and Radhia Cousot [10].

The earliest application of sets in program analysis is presented in the work of Reynolds [17]. He presents a procedure for computing recursive set definitions for LISP data structures. The data definitions are inferred from the program behaviour by approximating the variables with sets of possible values. This program approximation is expressed as a system of set constraints that captures the program's data flow behaviour. The set constraints are generated from the data flow equations by replacing expressions consisting of atoms and functions with sets and set constructors (and destructors). In order to determine data set definitions, in [17], the procedure computes the least fixed point of the system of set constraints. The least fixed point is computed by iteratively transforming the system of set constraints into a system where all subterms are directly involved in the construction of data. The transformation is achieved by eliminating so-called, analytical subterms which correspond to the branching in the original flow equations.

A uniform formal definition of abstract interpretation is presented in [9] Patrick Cousot and Radhia Cousot. Here, the authors introduce static semantics later referred

to as collecting semantics of a program. The collecting semantics are defined as functions from program points to sets of states (or environments) that can arise at that point during execution. Essentially, collecting semantics are static summary of the program behaviour. In [9], Cousot and Cousot provide a formal framework that unifies different program analysis as abstract interpretations over an abstract domain which is connected to the domain of collecting program semantics with a Galois connection. An abstract domain is an approximation of a program's collecting semantics with finite ascending chains.

Both [17] and [9] are based on the fundamental remark by Floyd [11] that static program properties can be expressed in terms of mapping from program points to sets of states.

In [15], Jones and Muchnick design a shape analysis of LISP data structures. Similar to the [17], here data flow equations are compiled into a system of set constraints and the least model of the system is computed. However, here, the least model computation is based on the regular tree grammar where sets are generated by trees and constraints between sets are production rules in the grammar. The solution to the least model represents a class of all possible shapes a LISP data structure can assume during runtime.

The set constraints used in [17] and [15] are limited in expressiveness as the only allowed set operation is function projection. Other related works improve upon these two works by allowing different combinations of set operations.

Although [17] and [15] are the first works to use sets in program analysis, this technique is only recognized as set-based approximation in later works by Heintze and Jaffar [13], [14]. In [13], Heintze and Jaffar formulate the set-based approximation of the logic program based on the cartesian closure of available substitutions. The set-based approximation is formalized as the approximation technique that ignores all inter-variable dependencies where variables are associated with sets of values. An approximated meaning of a logic program is the least fixed point of a function τ_P from/to interpretations that represent all possible evaluations of the program. The τ_P function updates a given interpretation by applying substitutions from the

cartesian closure of available substitutions in the current interpretation. The cartesian closure of available substitutions allows us to consider the variables over a set of terms instead of an individual term. The τ_P can be finitely represented as a system of set constraint where the least fixed point of the function corresponds to the least model of the system. The least model of the system is captured by the reduction algorithm which extends previous results in set constraint resolution by dealing with constraints consisting of arbitrary combinations of union and intersection operations. In [13], authors argue that approximations based on the cartesian closure over the set of substitutions are more accurate than alternative approximations based on a set of substitutions and closure over the set of terms. The former is too restricted to cover all interpretations while the latter enlarges the program with redundant interpretations.

The main contribution of [14] is a decision procedure for a class of definite set constraints. A definite constraint $a \subseteq exp$ is a restricted form of the constraint where a consists of constants, variables, and functions while exp is a set expression without the complement operation. The procedure presented here outputs an explicit representation of the least model of a system of constraints where set expression can contain an arbitrary combination of union, intersection, and projection operations. Similar to [15], in [14], a system of set constraints is finitely represented as set expression grammar (regular tree grammar). A system of set constraints is represented in more explicit set expression grammar where testing for emptiness and containment is convenient. Then the decision procedure transforms the set expression grammar into term grammar which is used to verify set constraints. Additionally, the decision procedure in [14] can output an explicit representation of the least model of a system of solved form set constraints. This class of constraints is used in [17] and [15].

Next steps are taken by Aiken, Murphy and Wimmer [3], [2], [4] by implementing regular tree expressions, designing a constraint resolution algorithm that can deal with set complements, and applying set-based approximation to type inference problems. [3] provides a static type system for FL (Function Level programming language designed by IBM) based on regular trees similar to [15] and [14]. Here, types are sets of normal form expressions represented as regular trees where "x of type T" means x

can be evaluated to some expression in T. Types are represented by the regular trees referred to in [3] as type expressions. Type expression can contain an arbitrary combination of type constructors, union, intersection, and fixed point operations. The exact semantics of type expressions are defined in terms of rewrite rules. According to [3], this operational formulation of types results in greater precision in inferring types of higher-order recursive functions.

[2] presents an implementation of regular tree expression used in [15], [14], and [3] along with high-level descriptions of emptyness and inclusion test. Although regular trees are a natural way of representing sets of trees which are often used in program analysis, the actual implementation uses the leaf-linear system of equation [2]. In particular, the fixed point computation for the regular tree expressions is more convenient to represent as a solution of the system of set equations. Thus emptiness and containment test algorithms are defined as sequences of transformations on set equations. Along with the high-level description of the algorithms [2] provides an implementation of regular tree operation in a leaf-linear system of equations.

In [4], Aiken and Wimmer provide a decision procedure for set constraints consisting of standard set operations. It extendes the results of [17], [15], [13] and [3] by handling unrestricted union, intersection and complement operations. The algorithm is a conservative transformation on set constraints which preserves the original set of solutions. Given a system of set constraints with unrestricted set operations, the algorithm transforms the system to a solved form set constraints for which it is guaranteed to have a solution if and only if the original system is satisfiable. Otherwise, the algorithm reports an inconsistency in the system. Complexity analysis in [4] shows that verifying the consistency of a system of set constraints is in NEXPTIME.

In [7], Charatonik and Pacholski prove that the satisfiability problem of set constraints with projection operation is NEXPTIME-complete. It extends the previous work by Aiken et al. [1]. In [8], Charatonik and Podelski determined a complexity class for set constraints consisting of constructors and set intersections. The authors proved that solving a system of set constraints consisting of intersections and constructions is DEXPTIME-complete. The same complexity characterization holds for

the negative set constraints as well. Moreover, authors in [8] established the equivalence between set constraints with intersection and definite set constraints introduced by Heintze and Jaffar in [14]. Consequently, the complexity class of the satisfiability problem of a system of definite set constraints is DEXPTIME-complete.

[5] applies results in solving set constraints to general type inference problems. Here, deciding whether a program is well-typed is reduced to verifying the consistency of a system of type inclusion constraints. Types in this setting are subsets of some semantic domain represented by the type expression. That is, an expression is of a type if the approximated meaning of the expression is in the type. A type expression consists of union, intersection, function types, type constructors, $\perp$, and $\top$ types for least and universal types. Type expressions are the same as the set expressions in [4] with the exception of function types and let-polymorphism. In [6], the type language is extended with conditional types $X?Y$ to represent conditional branching statements. A similar expression is used in [17] and [15] to handle $if...else$ expressions.

[12] contributes in two ways. First, it provides an algorithm for solving a system of set constraints in $O(n^3)$ time. Second, more importantly, it presents a uniform formalization of set-based approximation. In [12], set-based approximation ignores all dependencies between variables and considers variables over a domain of sets instead of individual values. This allows for the uniform formulation of approximation theory, unlike previous attempts where the approximation is defined over some abstract domain such that the approximated set of values corresponds to the fixed point computation. This notion of approximation is used in the authors' previous works [13], [14] and it is similar to the approximation used in Aikens soft type system presented in [5], [6]. According to the author, set-based approximation is based on operational semantics while Aiken's approximation is based on denotational semantics. In [12], the author considers analysis for a call-by-value higher-order functional language (ML) but the provided techniques can be adapted for any programming language. The formulation of the set-based approximation starts with the definition of a set operational semantics based on the operational semantics of ML. The set semantics modifies a program environment to map expressions to a set of values. Then the approximation

is the least model of a system of constraints defined in terms of the set operational semantics.

Chapter 3

Data Terms

In this section, we describe the main concern of our type system which are data terms that can have different forms based on the way they are constructed. All data in the programming language are constructed from primitive types using either a stuple constructor, or an array constructor.

Primitive types are basic data types such as numbers and characters common to many programming languages. The elements of these primitive types are singleton trees that can occur at the leaves of other data trees. Among these primitive data types, 64-bit integers are used for indexing arrays of data trees.

Since the data trees are defined in terms of elements of primitive types, we first define primitive data types. Note that the primitive types are not the complete type system used in our programming language.

Definition 3.0.1. *We use the following primitive types:*

- **never**, **unit**,
- **bool**, **char**, **selector**,
- **u64**, **integer**, **double**.

We call **bool**, **char**, *and* **selector** tag types, *because they can be used as tags for selecting options. We assume that* **u64**, **char** *and* **bool** *are ordered types with the*

usual order. Type **never** *has no inhabitants, and type* **unit** *has one inhabitant, which is called* trivial.

Similar to C enumerators, data of **bool**, **char**, and **selector** types can be used to tag options. In fact, **selector** is a global enumerator type since it is annoying to have different enumerator types for each type definition. Thus, these three types togather are called **tag types** in the report. We assume **u64**, **char** and **bool** are ordered with the usual order.

Definition 3.0.2. *We define the set of* data terms $\mathcal{D}$ *by recursion as follows:*

- *If d is an element of one of the primitive types listed in Definition 3.0.1, then $d \in \mathcal{D}$.*
- *If $d_1, \ldots, d_n$ with $n \geq 0$ are data trees, then $(d_1, \ldots, d_n) \in D$ and $[d_1, \ldots, d_n] \in D$. We call $(d_1, \ldots, d_n)$ a* tuple*, and $[d_1, \ldots, d_n]$ an* array*.*

The difference between $(d_1, \ldots, d_n)$ and $[d_1, \ldots, d_n]$ is that the former represents a fixed sized tuple, while the latter represents an array. In a program, a term of $(d_1, \ldots, d_n)$ can be indexed only by constants between 1 and n, because field selection is always fixed. At the same time there is no a priori upper bound on the size of an array, and arrays can be indexed by variables, so that is not a priori known which element will be accessed at a given point in the code.

Definition 3.0.3. *For a data term t, we define* $\mathbf{width}(t)$ *as the maximal size of a tuple occurring in t. If t contains no tuples, then* $\mathbf{width}(t) = 0$.

Because a program is always finite, there is an upperbound that the size of a tuple can have, for a given program. Hence, there always exists a natural number n, s.t. all data terms that the program operates on, have width at most n.

We explain the role of selectors: In case a type has options, selector can be used for determining the current option. For example, an optional **u64** can be represented either by (?none), or by $(?\text{just}, i)$, with i of type **u64**. A union type consisting of **char** of **double** can be represented by $(?\text{first}, c)$ or $(?\text{second}, d)$. If one defines logical formulas a a type, selectors are used for representing the logical operators.

In the literature, tree expressions are defined with a finite set of constructors ordered by their arity. In Definition 3.0.2, we use tuples of data trees to express arbitrary constructors of any arity. Although the tuples allow for unrestricted construction, in our programming language we consider tuples starting with an element of some tag type as a valid data tree construction. The tag element specifies an option of the data and determines the constrcutor parameters as consequent elements of the tuple expression.

Chapter 4

Abstraction Automata

In order to approximate data terms, we use deterministic tree automata. We run the tree automata on the values of variables, and replace them by the resulting states. In this way, a program state can be represented by a mapping of variables to states of a tree automaton.

In order to obtain the automaton, we translate preconditions of possible overload candidates into a tree automaton. After that, we merge the different tree automata into a single automaton, so that we know for a term which combination of preconditions of overload candidates, it satisfies. In order to avoid losing all information for terms that are not a subterm of a precondition, we extend the automata with *overflow states* that will preserve some information. The construction of the automata is a separate topic not covered in this thesis. Here we will simply assume that the automata are given.

Definition 4.0.1. *We define a* deterministic tree automaton (DTA) $\mathcal{A}$ *as a triple* (Q, δ, N), *where* Q *is a finite (nonempty) set of states, and* δ *is a transition function with the following properties:*

- *If* c *is a constant of one of the primitive types defined in Definition 3.0.1, then* $\delta(c) \in Q$.

- *If* $q_1, \ldots, q_n \in Q$ *and* $0 \leq n \leq N$, *then* $\delta(q_1, \ldots, q_n) \in Q$.

- $\delta() \in Q$,
- *If* $q_1, q_2 \in Q$, *then* $\delta(q_1; q_2) \in Q$.

The intended meaning of ; *is multiset insertion, with the inserted element to the right. Technically,* δ *should be viewed as distinct overloads, namely* δ_T *for primitive type* T, $\delta_()$ *for tuples,* $\delta_\emptyset$ *for the empty multiset, and* $\delta_;$ *for multiset insertion. W.r.t to multiset insertion,* δ *must have the following property, denoting irrelevance of insertion order:*

$$\delta(\delta(q_1; q_2); q_3) = \delta(\delta(q_1; q_3); q_2).$$

Arrays are always treated as multisets. In order to obtain a state of an array, one first obtains the states of its members. After that, one can start with $\delta_\emptyset$, and apply $\delta(;)$ to insert the elements one by one. The condition on δ enforces independence of insertion order. In an earlier version of Definition 4.0.1, we used set semantics by adding an other condition on $\delta_;$ stating that second insertions have no additional effect: $\delta(\delta(q_1, q_2); q_2) = \delta(q_1; q_2)$. We use multisets, because it is easy, and it increases expressivity. All we had to do, is remove the axiom $\delta(\delta(q_1; q_2); q_2) = \delta(q_1; q_2)$. Multisets can contain repeated elements, but since we are always using a finite number of states, the number of repetitions that can be distinguished in a multiset, must be finite. We can expressive properties like : There must be at least 3 occurrences of ..., but not properties of the form: there are more occurrences of ... than of ...

Using DTA $\mathcal{A}$, every term can be mapped to a state of $\mathcal{A}$. We will call the resulting function $Q_\mathcal{A}$:

Definition 4.0.2. *For a deterministic tree automaton* $\mathcal{A} = (Q, \delta, N)$, *we define a function* $Q_\mathcal{A}$ *from data terms to* Q *by recursion as follows:*

- *If* c *is a constant of primitive type, then* $Q_\mathcal{A}(c) = \delta(c)$.
- $Q_\mathcal{A}(\ (d_1, \ldots, d_n)\ ) = \delta(\ Q_\mathcal{A}(d_1), \ldots, Q_\mathcal{A}(d_n)\ )$.
- $Q_\mathcal{A}(\ [\,]\ ) = \delta(\ )$.
- *If* $n > 0$, *then* $Q_\mathcal{A}(\ [d_1, \ldots, d_n]\ ) = \delta(\ Q_\mathcal{A}(\ [d_1, \ldots, d_{n-1}]\ );\ Q_\mathcal{A}(d_n)\ )$.

In the last case, the order in which $d_1, \ldots, d_n$ appear in the array, does not matter.

During type checking, we will use $Q_{\mathcal{A}}$ as abstraction function. In order to make this possible, we need to find a DTA that preserves enough information, so that confirmed type correctness can be trusted, and at the same time, incorrect rejections do not happen too often.

Instead of first defininining a DTA $\mathcal{A}$, and defining $Q_{\mathcal{A}}$ from the automaton, one can also define first $Q_{\mathcal{A}}$, and the construct the automaton, assuming that $Q_{\mathcal{A}}$ permutes with tuple construction, and with multiset insertion.

Theorem 4.0.3. *Let Q be a finite set. Let Q^* be a function from data terms to Q, s.t. Q^* is defined for all data terms containing no tuple with more than N elements. Assume that Q^* satisfies the following properties:*

- *If $n \leq N$, and $t_1, \ldots, t_n,\ t'_1, \ldots, t'_n$ are sequences of data terms, s.t.*

  $$Q^*(t_1) = Q^*(t'_1), \ldots, Q^*(t_n) = Q^*(t'_n),$$

  then

  $$Q^*(\ (t_1, \ldots, t_n)\ ) = Q^*(\ (t'_1, \ldots, t'_n)\ ).$$

- *For every n, for every sequence $t_1, \ldots, t_n$ of data terms, for every permutation $t'_1, \ldots, t'_n$ of $t_1, \ldots, t_n$, we have*

  $$Q^*(\ [t_1, \ldots, t_n]\ ) = Q^*(\ [t'_1, \ldots, t'_n]\ ).$$

- *For every $n \geq 0$, for every two sequences $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$ of data terms, if*

  $$Q^*(t_1) = Q^*(t'_1), \ldots, Q^*(t_n) = Q^*(t'_n),$$

  then

  $$Q^*(\ [t_1, \ldots, t_n]\ ) = Q^*(\ [t'_1, \ldots, t'_n]\ ).$$

There exists a DTA $\mathcal{A} = (Q, \delta, N)$, s.t. $Q^* = Q_{\mathcal{A}}$.

Remember that in data terms, [ ] denotes sequence construction, hence the second condition is not redundant.

Example 4.0.4. *These are two examples that control the depth of traversal and the collection of sub-results:*

CUTOFF(k): *Let* \$ *be constant symbol that occurs nowhere else, and which does not have a type. For a term t, let $Q_{\mathcal{A}}(t)$ be obtained from t by replacing every subterm at depth k by* \$.

TAG(k): *Let $Q_{\mathcal{A}}(t)$ be obtained from t by first replacing every subterm at depth k by* \$. *After that, replace every subterm that does not contain a constant of type* **selector**, **bool** *or* **char** *by* \$.

Each of the functions listed in Example 4.0.4 satisfies the conditions of Theorem 4.0.3.

When running an automaton, we like all states equally of course, but some we may like a bit more than others, for example those that represent a precondition of one of the overload candidates:

Definition 4.0.5. *Let $\mathcal{A} = (Q, \delta, N)$ be a DTA. A* unary state property *F is a unary predicate on Q. A* binary state property *F is a binary predicate on Q.*

In principle, a state property can also be defined as a subset of F, but this suggests that the elements of F are enumerated, while in our application, F always has more structure. Technically, predicates and sets are the same.

Definition 4.0.6. *Let $\mathcal{A} = (Q, \delta, N)$. Assume that $N' > N$. The result of* inflating *$\mathcal{A}$ to N' is the DTA $\mathcal{A}' = (Q', \delta', N')$, obtained by*

- *$Q' = \{Q\} \cup \{q'\}$, with $q' \notin Q$ a new state.*
- *If $n \leq N$, then $\delta'(q_1, \ldots, q_n) = \delta(q_1, \ldots, q_n)$. If $N < n \leq N'$, then $\delta(q_1, \ldots, q_n) = q'$.*

Definition 4.0.7. *Let $\mathcal{A}_1 = (Q_1, \delta_1, N_1)$ and $\mathcal{A}_2 = (Q_2, \delta_2, N_2)$ be deterministic tree automata. We define the product automaton $\mathcal{A}_1 \times \mathcal{A}_2 = (\, Q_1 \times Q_2,\ \delta',\ \mathbf{max}(N_1, N_2)\,)$*

as follows: If $N_1 \neq N_2$*, then first inflate the smaller automata to* $\max(N_1, N_2)$*. After that, construct* δ' *as follows:*

- *For a constant* c *of primitive type,* $\delta'(c) = (\,\delta_1(c), \delta_2(c)\,)$.
- $\delta'(\,(q_1, q_1'), \ldots, (q_n, q_n')\,) = (\,\delta_1(q_1, \ldots, q_n),\ \delta_2(q_1', \ldots, q_n')\,)$.
- $\delta'() = (\,\delta_1(), \delta_2()\,)$.
- $\delta'(\,(q_1, q_1'); (q_2, q_2')\,) = \delta(\,(q_1; q_2), (q_1', q_2')\,)$.

Using product, all Boolean combinations of state conditions can be decided. States of deterministic tree automata finitely partition the set of all data terms and multisets into equivalence classes. Since equivalence classes are formed by the states of tree automata, each class corresponds to some combination of properties or the absence of properties (e.g. q_{def}). Equivalence classes can be divided into three general groups:

- Classes that correspond to preconditions of overload candidates. Here, any two data terms from the same class satisfy the same preconditions.
- Classes of data terms that do not satisfy any of the preconditions but they present important properties for analysis. For example, given a data term t that satisfies a precondition p. If we put t into some tuple $(t, \ldots)$ such that it does not satisfy any precondition, and then extract t back again we should be able to deduce that it still satisfies p. These classes are formed by overflow states.

Example 4.0.8. *Lets consider a deterministic tree automaton* $\mathcal{A}_{\text{Prop}} = (Q, \delta, 3)$ *for recognizing data terms of* **Prop** *type in Example 1.0.1. First, we define transitions*

for constants of **selector** *and* **char** *types.*

$$
\begin{aligned}
\delta(?\text{atom}) &= q_{\text{atom}} \\
\delta(?\text{not}) &= q_{\text{not}} \\
\delta(?\text{implies}) &= q_{\text{implies,equiv}} \\
\delta(?\text{equiv}) &= q_{\text{implies,equiv}} \\
\delta(?\text{or}) &= q_{\text{or,and}} \\
\delta(?\text{and}) &= q_{\text{or,and}} \\
\delta(c) &= q_{\text{char}}
\end{aligned}
$$

Note that the last transition is actually a scheme of transitions for all characters. Next, we define transitions for different forms of **Prop**. *Assuming that* $\mathcal{A}_{\text{Prop}}$ *is required to distinguish different forms of* **Prop**, *we use* $q_{\text{Prop1}}, \ldots, q_{\text{Prop4}}$ *to represent all four distinct forms.*

$$
\begin{aligned}
&\delta(\ q_{\text{atom}},\ q_{\text{char}^\forall}\ ) && = q_{\text{Prop1}} \\
&\delta(\ q_{\text{not}},\ q_{\text{Prop1}}\ ) && = q_{\text{Prop2}} \\
&\vdots \\
&\delta(\ q_{\text{not}},\ q_{\text{Prop4}}\ ) && = q_{\text{Prop2}} \\
&\delta(\ q_{\text{implies,equiv}},\ q_{\text{Prop1}},\ q_{\text{Prop1}}\ ) && = q_{\mathit{Prop}3} \\
&\vdots \\
&\delta(\ q_{\text{implies,equiv}},\ q_{\text{Prop4}},\ q_{\text{Prop4}}\ ) && = q_{\mathit{Prop}3} \\
&\delta(\ q_{\text{or,and}},\ q_{\text{Prop}^\forall}\ ) && = q_{\text{Prop4}}
\end{aligned}
$$

Note that the forms represented by states q_{Prop2} *and* q_{Prop3} *with sub terms of* **Prop** *type now require separate transitions for each possible combination of forms of* **Prop**. *Also states* $q_{\text{char}^\forall}$ *and* $q_{\text{Prop}^\forall}$ *represent multiset of data terms of* **char** *and* **Prop** *types,*

respectively. Finally, we define transition for multi sets of **char** *and* **Prop** *types.*

$$
\begin{aligned}
\delta(\ ) &= q_{\emptyset} \\
\delta(q_{\text{char}}\ ;\ q_{\emptyset}) &= q_{\text{char}^{\forall}} \\
\delta(q_{\text{char}}\ ;\ q_{\text{char}^{\forall}}) &= q_{\text{char}^{\forall}} \\
\delta(q_{Prop1}\ ;\ q_{\emptyset}) &= q_{\text{Prop}^{\forall}} \\
&\vdots \\
\delta(q_{Prop4}\ ;\ q_{\emptyset}) &= q_{\text{Prop}^{\forall}} \\
\delta(q_{Prop1}\ ;\ q_{\text{Prop}^{\forall}}) &= q_{\text{Prop}^{\forall}} \\
&\vdots \\
\delta(q_{Prop4}\ ;\ q_{\text{Prop}^{\forall}}) &= q_{\text{Prop}^{\forall}}
\end{aligned}
$$

Note that we designate a separate state for the empty set since it can be both a multiset of **char***'s and multiset of* **Prop***'s. The state of a multiset depends on the state of the first insearted element.*

The automaton in Example 4.0.8 togather with a helper function $Q_{\mathcal{A}_{\text{Prop}}}$ can be used to decide whether an instance of accessing a field of **Prop** type is valid. Note that the definition of the transition function δ in Example 4.0.8 is completed by mapping all other data terms and multisets to a default state q_{def}.

Example 4.0.9. *Given a helper fucntion $Q_{\mathcal{A}_{\text{Prop}}}$ for the automaton $\mathcal{A}_{\text{Prop}}$ in Example 4.0.8:*

- *d.c is valid iff* $Q_{\mathcal{A}_{\text{Prop}}}(d) = q_{\text{Prop1}}$
- *d.sub is valid iff* $Q_{\mathcal{A}_{\text{Prop}}}(d) = q_{\text{Prop2}}$
- $d.sub_1$ *and* $d.sub_2$ *are valid iff* $Q_{\mathcal{A}_{\text{Prop}}}(d) = q_{\text{Prop3}}$
- $d.sub[i]$ *is valid iff* $Q_{\mathcal{A}_{\text{Prop}}}(d) = q_{\text{Prop4}}$

In practice, we constract several DTAs for all function and field overload candidates preconditions. These DTAs are combined togather into a single DTA which can be used to determine which combination of preconditions a data term satisfies.

Field and function maps associate function and field names to state conditions on the states of the DTA. A state condition specifies a class of data terms that satisfy the precondition of the corresponding overload candidate.

Chapter 5

Field and Function Maps

In this section, we define function and field maps that defines different overload candidates and associats then with state conditions. Given a DTA which is capable of capturing necessary properties of data terms in its states, a state condition associated to an overload candidate specifies a set of states argument data terms should have for a valid function or field application. Essentially, state conditions represent precondition of function or field overload candidates; a state condition specifies a class of desired data terms, similar to accepting states in a standard automaton. Preconditions of overload candidates are accounted for in construction of the DTA such that its states provide enough information to determine which combination of preconditions a data term satisfies. Separate DTAs are constructed for each precondition and merged into a single DTA as a cartesian product of individual automata. The cartesian product of automata provides an joint state space that can cupture which combination of preconditions a data term satisfies. Through out this section, we assume a fixed DTA $\mathcal{A} = (Q, \delta, N)$ and a function $Q_{\mathcal{A}}$ from data terms to the set of states Q.

Since functions and fields can be overloaded, function and field names can be reused among different forms of a data type or even among different data types. We assume that the compiler gives unique names to overload candidates, e.q. by adding an index.

A field map assignes names to positions in tuples, so that we can use field notations, as is common in programming languages.

Definition 5.0.1. $\mathcal{F}^{\textbf{field}}$ *is a mapping from field names to pairs of form* (F, i) *where* F *is a state condition defined on* Q*, and* $i \geq 1$*.*

$\mathcal{F}^{\textbf{field}}(f) = (F, i)$ *means that field* f *can be taken on a data term* d *such that* $F(\ Q_{\mathcal{A}}(d)\ )$ *holds and that it refers to field at position* i *in* d*.*

Example 5.0.2. *Continuing the* **Prop** *data type defined in 1.0.1 and the DTA* $\mathcal{A}_{\text{Prop}}$ *defined in Example 4.0.8, we construct* $\mathcal{F}^{\textbf{field}}$ *as follows:*

$$\begin{aligned}
\mathcal{F}^{\textbf{field}}(c) &= (F_{q_{\text{Prop1}}}, 1) \\
\mathcal{F}^{\textbf{field}}(sub) &= (F_{q_{\text{Prop2}}}, 1) \\
\mathcal{F}^{\textbf{field}}(sub1) &= (F_{q_{\text{Prop3}}}, 1) \\
\mathcal{F}^{\textbf{field}}(sub2) &= (F_{q_{\text{Prop3}}}, 2) \\
\mathcal{F}^{\textbf{field}}(sub) &= (F_{q_{\text{Prop4}}}, 1)
\end{aligned}$$

Note that state conditions as defined on the states of $\mathcal{A}_{\text{Prop}}$*.* $\mathcal{A}_{\text{Prop}}$ *designates the states* $q_{\text{Prop1}}, \ldots, q_{\text{Prop4}}$ *for distinct froms of* **Prop***. Using these states,* $\mathcal{F}^{\textbf{field}}$ *ensures that fields are taken on a data term of* **Prop** *with an appropriate form.*

Note that $\mathcal{F}^{\textbf{field}}$ *defines two overloads of 'sub' field: one for propositional formulas with negation as top-most operation represented by* q_{Prop2}*, and an other for formulas with disjunction and conjunction with unrestricted arity as top-most operation represented by* q_{Prop4}*.*

Definition 5.0.3. $\mathcal{F}^{\textbf{func}}$ *is mapping from function names to tuples of form* $(F_1, \ldots, F_n)$*,* $(n \geq 0)$*, where* F_i *are state conditions.*

$\mathcal{F}(f) = (F_1, \ldots, F_n)$ *means that* f *is a function with* n *arity that can be applied on data terms* $d_1, \ldots, d_n$ *such that* $F_i(Q_{\mathcal{A}}(d_i))$ *holds for all* $0 \leq i \leq n$*.*

Unlike fields which are entered into the field map before type checking, functions are entered into the function map once they are type checked.

Example 5.0.4. *Let* $\mathcal{A}_{\text{Prop}\times\text{ArrowFree}\times\text{NNF}}$ *be a product of three automata which are constructed for distingiushing different froms of* **Prop** *data type, propositional formulas without implication and equivalence operators, and formulas in negation normal form. We define* $\mathcal{F}^{\textbf{func}}$ *for function in Example 1.0.2 as follows:*

$$\mathcal{F}^{\mathbf{func}}(\text{make\_arrowfree}) = (F \begin{array}{l}(q_{\text{Prop1}}, q_{\text{ArrowFree}}, q_{\text{NNF}}), (q_{\text{Prop2}}, q_{\text{ArrowFree}}, q_{\text{NNF}}),\\ (q_{\text{Prop2}}, q_{\text{ArrowFree}}, q_{\text{def}}), (q_{\text{Prop2}}, q_{\text{def}}, q_{\text{def}}),\\ (q_{\text{Prop3}}, q_{\text{def}}, q_{\text{def}}), (q_{\text{Prop4}}, q_{\text{ArrowFree}}, q_{\text{NNF}}),\\ (q_{\text{Prop4}}, q_{\text{ArrowFree}}, q_{\text{def}}), (q_{\text{Prop4}}, q_{\text{def}}, q_{\text{def}}),\end{array})$$

$$\mathcal{F}^{\text{func}}(\text{make\_nnf\_pos}) = (F \begin{array}{l}(q_{\text{Prop1}}, q_{\text{ArrowFree}}, q_{\text{NNF}}),\\ (q_{\text{Prop2}}, q_{\text{ArrowFree}}, q_{\text{NNF}}), (q_{\text{Prop2}}, q_{\text{ArrowFree}}, q_{\text{def}}),\\ (q_{\text{Prop3}}, q_{\text{ArrowFree}}, q_{\text{NNF}}), (q_{\text{Prop3}}, q_{\text{ArrowFree}}, q_{\text{def}}),\end{array})$$

Note that the definition of the make_nnf_neg *is the same as the definition of* make_nnf_pos.

Function and field application can be ambiguous if there exist more that one mapping for the same indentifier. All possible overload candidates are insearted into an intermediate representation as special branch statements where exactly one of the branches should hold in every possible execution. In enery execution case, we choose an overload with the most specific precondition among the other candidates whose precondition is also satisfied by the argument date trems. The most specific precondition is determined based on an order on the set of states Q. The order of two state is decided based on the reachability of the one state from the other.

Definition 5.0.5. *Given q_1, q_2 in Q, $q \preceq q_2$ iff q_1 is reachable from q_2 w.r.t δ. Reachabilityi between states means that there exists a finite sequence of transitions that start at q_2 and end at q_1.*

The most specific precondition satisfied by a data term d corresponds to a state q such that for any other state q' of the term d, $q \preceq q'$.

Chapter 6

Intermediate Representation

We introduce a format for representation of intermediate code. For our type system, intermediate code cannot be generated in the standard way. Standard compilers create the intermediate represention immediately after type checking. In the intermediate representation all operations have become unambiguous. This is not possible for our type system, because in some cases, the type depends on loop invariants, which can be checked only by inspecting the complete loop. Consider for example:

```
x = 0
while ...
   print(x)    // Special overload for even numbers.
   x = ( ?succ, ( ?succ, x )).
```

In order to accept `print(x)`, one must know that all operations on `x` in the loop preserve evennes. It could also be the case that there are different overloads for even and arbitrary numbers. In that case, both options have to be inserted into the intermediate representation, and the final choice can be made only after the loop has been analyzed.

Most intermediate representations are based on flow graphs, which implies that they don't contain any nested statements. All control is handled by **goto**s and conditional branches. Initially, we believed that flow graphs are not suitable for our purpose but actually they might be. In the example above, if there is more than

one print operator, all overload candidates have to be inserted into the intermediate representation, in some parallel fashion. One needs a kind of branch statement and a kind of merge. The branch is not a usual branch, because there must be one option that is unambiguously the most specific one, which can be determined at compile time. The merge after this branch is not a usual merge, because in case the branches contain assignments, the merge must take some kind of intersection of the possible values, which is hard to define.

Each function body is translated separately into intermediate representation, and checked. In this thesis, we will not describe the programming language itself, or the translation procedure, but we describe the type checking procedure in Section 8.

Definition 6.0.1. *We assume a countably infinite set $\mathcal{V}$ of local variables.*

Local variables are used for the parameters of a function, for variables declared by the user in the original program, and for represnting variables resulting from the flattening of statements.

Definition 6.0.2. *We define* flatterms*:*

- *If v is a variable, then v is a flatterm.*
- *If c is a constant of a primitive type, then c is a flatterm.*
- *If $\mathcal{A}$ is a DTA, and F is a state condition of $\mathcal{A}$, then $\mathbf{some}(\mathcal{A}, F)$ is a flatterm.*
- *If v is a variable, then the following are flatterms:*

$$\mathbf{len}(v),\ \mathbf{not}(v),\ \mathbf{neg}(v).$$

- *If v_1 and v_2 are variables, then the following are flatterms:*

$$\mathbf{eq}(v_1, v_2),\ \mathbf{ne}(v_1, v_2),\ \mathbf{lt}(v_1, v_2),\ \mathbf{gt}(v_1, v_2),\ \mathbf{le}(v_1, v_2),\ \mathbf{ge}(v_1, v_2),$$

$$\mathbf{add}(v_1, v_2),\ \mathbf{sub}(v_1, v_2).$$

- *If v is a variable, $\mathcal{A}$ is a DTA, and F is a state condition, then* $\mathbf{check}(v, \mathcal{A}, F)$ *is a flatterm.*

- *if v and w are variables, then* $\mathbf{ind}(v, w)$ *is a flatterm.*

- *If v is a variable, and f is an identifier, then* $\mathbf{fld}_f(v)$ *is a flatterm.*

- *If $v_1, \ldots, v_n$ are variables with $n \geq 0$, then $(v_1, \ldots, v_n)$ and $[v_1, \ldots, v_n]$ are flatterms.*

- *If $v_1, \ldots, v_n$, are variables, f is an identifier, then $f(v_1, \ldots, v_n)$ is a flatterm.*

The meanings are as follows: If v is an array, then $\mathbf{len}(v)$ equals its size. The intuitive meaning of $\mathbf{ind}(v, w)$ is $v[w]$. The intuitive meaning of $\mathbf{fld}_f(v)$ is $v.f$. Flatterm $\mathbf{some}(\mathcal{A}, F)$ non-deterministically creates a data tree t, s.t. $F(\, Q_{\mathcal{A}}(t)\,)$, while $\mathbf{check}(t, \mathcal{A}, F)$ returns a boolean representing truth of $F(\, Q_{\mathcal{A}}(t)\,)$.

Definition 6.0.3. *We assume a countable set* Π *of program points.*

It is not very important how program point are represented. Since statements defined below are trees, one can represent them as paths.

Definition 6.0.4. *We recursively define* statements. *For a given statement, we assume that every substatement has a unique program point. A statement has one of the following forms:*

- *A scalar assignment of $v \leftarrow t$, in which v is a variable and t is a flatterm.*

- *A array assignment of $v[w] := t$, in which v and w are variables, and t is a flatterm. Note that $v[w]$ is not a subterm.*

- *If $s_1, \ldots, s_n$ are statements, then* $\mathbf{block}(s_1, \ldots, s_n)$ *is also a statement.*

- *If $s_1, \ldots, s_n$ are statements, then* $\mathbf{repeat}(s_1, \ldots, s_n)$ *is also a statement.*

- *If $s_1, \ldots, s_n$ are statements, then* $\mathbf{branch}(s_1, \ldots, s_n)$ *is also a statement.*

- *If v is a variable, and s is a statement, then* **istrue** v *and* **isfalse** v *are also statements.*
- **nop** *is a statement.*
- *If v is a variable, then* **return** v *is a statement.*
- v.push_back(w) *with v, w variables is a statement.*
- v.pop_back() *is a statement.*
- *A field assignment $w_1.f := w_2$, where f is a scalar field name, and w_1, w_2 are variables.*
- *If $\overline{v}$ is a sequence of variables, and for each i with $1 \leq i \leq n$, $\mathcal{A}_i$ is a sequence of automata, F_i is a sequence of state conditions, s.t. $\|v\| = \|\mathcal{A}_i\| = \|F_i\|$, each s_i is a statement, then*

$$\textbf{resolve}(\overline{v}, (\mathcal{A}_1, F_1)/s_1, \ldots, (\mathcal{A}_n, F_n)/s_n)\,)$$

  is also a statement.
- *If v is a variable, then* **erase** v *is a statement.*

Both **repeat** and **branch** are non-deterministic. The **repeat** statement non-deterministically skips $s_1, \ldots, s_n$ or starts at s_1. Similarly, **branch** non-deterministically executes one of the s_i. We made chose for non-determinism because analyzing non-deterministic code is not harder than deterministic code, due to the fact that overapproximation will introduce non-determinism anyway. In order to model deterministic code, one has to put a **istrue** or **isfalse** at the beginning of every branch. Both statements quietly fail when the condition is not met.

In the **resolve** statement, variables $\overline{v}$ are used for type checking. Overload $(\mathcal{A}_i, F_i)/s_i$ is viable, if for each variable in $\overline{v}$, the value, when run with the corresponding automaton in $\mathcal{A}_i$, reaches a state that satisfies the state corresponding condition.

Definition 6.0.5. *A* special statement *has one of the following forms:*

- *An* abort *statement has form* **abort** S*, where* S *is an error message.*
- *An* error *statement has form* **error** S*, where* S *is the error message.*

The difference between **abort** and **error** is that **error** is checked during type checking, while **abort** is checked at run time.

Definition 6.0.6. *A* state σ *is a finite mapping from variables to data terms.*

We now define a function $\mathbf{eval}_\sigma(t)$ that evaluates a flatterm t in state σ. Because the semantics of flatterms is non-deterministic, $\mathbf{eval}_\sigma(t)$ defines a set of possible values.

Definition 6.0.7. *Let* σ *be valuation. Let* t *be a flatterm. We define the function* $\mathbf{eval}_\sigma(t)$ *as follows:*

The definition of $\mathbf{eval}_\sigma$ can easily be made recursive, but since flatterms are non-recursive, we didn't do it.

Definition 6.0.8. *For a given statement* s*, we assume two functions* $\Phi_{\mathbf{in}}$ *and* $\Phi_{\mathbf{out}}$ *that map program points to set of states. Both functions are defined for the program points occurring in* s*. They are defined together as the least fixed point satisfying the following conditions:*

- *If* π *contains a* **nop**, *then* $\Phi_{\mathbf{in}}(\pi) \subseteq \Phi_{\mathbf{out}}(\pi)$.
- *If* π *contains* $\mathbf{block}(s_1, \ldots, s_n)$, *then let* $\pi_1, \ldots, \pi_n$ *be the positions of* $s_1, \ldots, s_n$. *We have* $\Phi_{\mathbf{in}}(\pi) \subseteq \Phi_{\mathbf{in}}(\pi_1)$. *For every* i *with* $1 \leq i \leq n-1$, *we have* $\Phi_{\mathbf{out}}(\pi_i) \subseteq \Phi_{\mathbf{in}}(\pi_{i+1})$, *and* $\Phi_{\mathbf{out}}(\pi_n) \subseteq \Phi_{\mathbf{out}}(\pi)$.
- *If* π *contains* $\mathbf{repeat}(s_1, \ldots, s_n)$, *then let* $\pi_1, \ldots, \pi_n$ *be the positions of* $s_1, \ldots, s_n$. *We have* $\Phi_{\mathbf{in}}(\pi) \subseteq \Phi_{\mathbf{in}}(\pi_1)$. *For every* i *with* $1 \leq i \leq n-1$, *we have* $\Phi_{\mathbf{out}}(\pi_i) \subseteq \Phi_{\mathbf{in}}(\pi_{i+1})$. *In addition we have* $\Phi_{\mathbf{in}}(\pi) \subseteq \Phi_{\mathbf{out}}(\pi)$, *and* $\Phi_{\mathbf{out}}(\pi_n) \subseteq \Phi_{\mathbf{in}}(\pi)$.
- *If* π *contains* $\mathbf{branch}(s_1, \ldots, s_n)$, *then let* $\pi_1, \ldots, \pi_n$ *be the positions of* $s_1, \ldots, s_n$. *For every* i, *we have* $\Phi_{\mathbf{in}}(\pi) \subseteq \Phi_{\mathbf{in}}(\pi_i)$, *and* $\Phi_{\mathbf{out}}(\pi_i) \subseteq \Phi_{\mathbf{out}}(\pi)$.

- *If* π *contains* **istrue** v, *then* $\Phi_{\mathbf{in}}(\pi) \cap \{\sigma \mid \sigma(v) = \mathbf{t}\} \subseteq \Phi_{\mathbf{out}}(\pi)$.

- **isfalse** *is analogous to* **istrue**.

- *If* π *contains a statement of form* $v \leftarrow t$ *with* t *a flatterm, then*

$$\{\sigma \cup \{v := d\}\} \mid \sigma \in \Phi_{\mathbf{in}}(\pi) \text{ and } d \in \mathbf{eval}_\sigma(t)\} \subseteq \Phi_{\mathbf{out}}(\pi).$$

  function $\mathbf{eval}_\sigma(t)$ *was defined in Definition 6.0.7.*

- *If* π *contains* $\mathbf{resolve}(c_1/s_1, \ldots, c_n/s_n)$, *then let* $\pi_1, \ldots, \pi_n$ *be the positions of* $s_1, \ldots, s_n$. *Similary, let* $\rho_1, \ldots, \rho_n$ *be the positions of* $c_1, \ldots, c_n$. *We have* $\Phi_{\mathbf{in}}(\pi) \subseteq \Phi_{\mathbf{in}}(\pi_i)$ *and* $\Phi_{\mathbf{in}}(\pi) \subseteq \Phi(\rho_i)$.

  We also have

$$\bigcap \Phi_{\mathbf{out}}(\pi_i) \subseteq \Phi_{\mathbf{out}}(\pi).$$

The last condition is a bit tricky. It is possible that some of the s_i use temporary variables that other s_i do not use. In that case, the resulting state will always be unique, and not in the intersection. This problem can be solved by erasing temporary variables. In fact, this is the reason that **erase** was created.

We do not store any scope information for local variables in statements. This implies that local variables in principle remaing visible, once initialized. This may lead to space inefficiency when statements are naively translated into C or C^{++}, which allow more restricted scopes. This problem can be solved by doing a stack based liveness analysis before translation.

Definition 6.0.9. *We define* $\mathbf{live}_{\mathbf{in}}(\pi)$ *and* $\mathbf{live}_{\mathbf{out}}(\pi)$ *as the least fixed points of the propagation rules below. The intuitive meaning is:* $\mathbf{live}(\pi)_{\mathbf{in/out}}$ *are the variables whose value may be later used in a computation that passes through* π.

- *If* π *contains a statement of form* $v \leftarrow t$, *then let* V *be the set of variables occurring in flatterm* t.

$$\Phi_{\mathbf{in}}(\pi) = (\Phi_{\mathbf{out}}(\pi) \backslash \{v\}) \cup V.$$

- *If* π *contains a statement of form* $\textbf{isfalse}\, v,\ \textbf{istrue}\, v,\ \textbf{return}\, v$ *then*

$$\textbf{live}_{\textbf{in}}(\pi) = \textbf{live}_{\textbf{out}}(\pi) \cup \{v\}.$$

- *If* π *contains a statement of form* $v.\text{push\_back}(w)$, *then*

$$\textbf{live}_{\textbf{in}}(\pi) = \textbf{live}_{\textbf{out}}(\pi) \cup \{w, v\}.$$

- *If* π *contains* **nop**, *then* $\textbf{live}_{\textbf{out}}(\pi) \subseteq \textbf{live}_{\textbf{in}}(\pi)$.

- *If* π *contains* $\textbf{branch}(s_1, \ldots, s_n)$, *then let* $\pi_1, \ldots, \pi_n$ *be the positions of* $s_1, \ldots, s_n$. *For every* i, *we have* $\Phi_{\textbf{in}}(\pi_i) \subseteq \Phi_{\textbf{in}}(\pi)$, *and* $\Phi_{\textbf{out}}(\pi) \subseteq \Phi_{\textbf{out}}(\pi_i)$.

A variable v *is* live *at some program point* π *if* $v \in \textbf{live}_{\textbf{in}}(\pi)$.

Intead of defining $\textbf{live}(\pi)$ as a set of variables, one can also define $\textbf{use}(\pi)$ as a set of pairs (v, π'), indicating that π' is a point where the value of v is potentially used. The definition is only slightly more complicated than Definition 6.0.9 above.

In order to simplify analysis, one can require that no assignment assigns to a live variable. More precisely, if π contains an assignment of form $v \leftarrow t$, then one can require that $v \notin \textbf{live}_{\textbf{in}}(\pi)$. In order to meet this requirement any violating assignment can be replaced by two initializations as follows $w \leftarrow t$ becomes $w' \leftarrow t;\ w \leftarrow w'$.

Chapter 7

Range States

We approximate set computations by means of approximating terms. If we do this carefully, the sets will still contain enough information to be useful in type checking, and at the same time be finite in number.

Definition 7.0.1. *We will use the following* abstraction automaton Q : **PROPVARS**$\times$ **TAG**(k).

Our type checking algorithm treats arrays as multisets, which then will be mapped to states of the precondition automata. We want to be able to handle loops that walk through an array, and change the values in such a way that they satisfy different preconditions. In order to do this, the array has to be partitioned in a part before the index, at the index, and after the index. The different parts may satisfy different preconditions. In addition, we must be able to test when the index stands at the beginning of an array, or has reached the end of the array. In order to be able to do this, one needs to keep track of the relative order between variables that are a potential index, as well as 0, and the sizes of arrays. We do this by remembering differences. If two indices are equal, their difference is zero. If the first index is bigger than the second, the difference is positive.

Definition 7.0.2. *A* difference *has one of the following forms:* $-2^*, -1, 0, 1, 2^*$. *A* difference set *is a set of differences. We write* $\top$ *for the full difference set* $\{-2^*, -1, 0, 1, 2^*\}$.

Figure 7-1: Unary $-$

$-$	-2^*	-1	0	1	2^*
	2^*	1	0	-1	-2^*

The intended meaning of -2^* is '-2 or less'. The intended meaning of 2^* is '2 or more'. Difference sets have two levels of uncertainty: The first is in the semantics of -2^* and 2^*, which represent a set of possible difference values. The second is the the fact that a set may contain more than one element. This ugly asymmetry could be removed by replacing -2^* by $-2, -3, -4, \ldots$, but we want the difference sets to be finitely representable. We don't know a better alternative to Definition 7.0.2.

Definition 7.0.3. *We define two operations on difference sets: The first is* negation*: If D is a difference set, then*

$$-D = \{-d \mid d \in D\},$$

using $-$ on differences as defined in Figure 7-1.

If D_1 and D_2 are difference sets, then $D_1 + D_2$ is defined as

$$\bigcup \{\, d_1 \oplus d_2 \mid d_1 \in D_1 \text{ and } d_2 \in D_2 \,\},$$

where $\oplus$ is defined in Figure 7-2. Note that the entries in the table are sets, so that it is possible to take their union.

Definition 7.0.4. *Let I be a finite set of objects that potentially are an index. A* difference matrix Δ over I *is a mapping of $I \times I$ to difference sets. We write $\Delta.\mathbf{at}(i_1, i_2)$ for $\Delta(i_1, i_2)$.*

Possible indices are variables that can have type **u64**, as well as 0 and expressions of $\mathbf{len}(s)$, where s is a variable that holds an array term. Note that the elements of I are symbolic: I consists of expressions, not of their values.

We will now define a few operations on difference matrices. We define them imperatively, instead of declaratively. In other words, we define them as operations that

Figure 7-2: Operator $\oplus$

$\oplus$	-2^*	-1	0	1	2^*
-2^*	-2^*	-2^*	-2^*	$-2^*, -1$	$\top$
-1	-2^*	-2^*	-1	0	$1, 2^*$
0	-2^*	-1	0	1	2^*
1	$-2^*, -1$	0	1	2^*	2^*
2^*	$\top$	$1, 2^*$	2^*	2^*	2^*

The entries in the table are always sets. Remember that $\top$ denotes the complete set $\{-2^*, -1, 0, 1, 2^*\}$.

modify an existing difference matrix, instead of functions. The reason for this choice is the following: In Definition 7.0.5, we will define the effect of a statement. Due to overapproximation, and the essential non-deterministic nature of high-level computation, the effect of a statemenet is non-deterministic. After having tried several ways to define the possible successor states, we concluded that a description using non-deterministic computation is the simplest. The alternative, using a Prolog-style declarative definition, is possible, but longer and harder to grasp. The imperative definitions that we will give, fit better into the non-deterministic computation, than declarative definitions.

Definition 7.0.5. *Let Δ be a difference matrix over an index set I.*

- *If $i \notin I$, then $\Delta.\mathbf{extend}(i)$ modifies Δ into a difference matrix Δ' over $I \cup \{i\}$, s.t. whenever both $i_1, i_2 \neq i$, then $\Delta'.\mathbf{at}(i_1, i_2) = \Delta.\mathbf{at}(i_1, i_2)$. If either of $i_1, i_2 = i$, then $\Delta'.\mathbf{at}(i_1, i_2) = \top$.*

- *$\Delta.\mathbf{assign}(i_1, i_2, d)$ changes Δ into a new difference matrix Δ', s.t. $\Delta'.\mathbf{at}(i_1, i_2) = d$, and $\Delta'.\mathbf{at}(i_1', i_2') = \Delta.\mathbf{at}(i_1', i_2')$, whenever $i_1' \neq i_1$ or $i_2' \neq i_2$.*

- *If $i \in I$, then $\Delta.\mathbf{erase}(i)$ is obtained from Δ by making $\Delta.\mathbf{at}(i, i_2)$ and $\Delta.\mathbf{at}(i_1, i)$ undefined.*

- *$\Delta.\mathbf{contains}(i)$ returns true if $\Delta.\mathbf{at}(i)$ is defined.*

- *$\Delta.\mathbf{isconsistent}()$ returns true if there are no $i_1, i_2 \in I$, s.t. $\Delta.\mathbf{at}(i_1, i_2) = \emptyset$.*

Note that I denotes the set of expressions that can be used as index in principle. A concrete difference matrix does not need to contain values for all $i \in I$.

Definition 7.0.6. *Let Δ be a difference matrix. We define a procedure* $\mathbf{close}(\Delta)$ *that closes Δ under reflexivity, transitivity and antisymmetry:* $\mathbf{clos}(\Delta)$ *is obtained by applying the following operations, until no further change is obtained:*

- *For every i, s.t.* $\Delta.\mathbf{contains}(i)$:

$$\begin{aligned}&\Delta.\mathbf{assign}(i, i, \{0\})\\ &\Delta.\mathbf{assign}(0, i,\ \Delta.\mathbf{at}(0, i) \cap \{0, 1, 2^*\})\\ &\Delta.\mathbf{assign}(i, \mathbf{len}(v), \Delta.\mathbf{at}(i, \mathbf{len}(v)) \cap \{0, 1, 2^*\}\ )\end{aligned}$$

- *For every i_1, i_2, s.t.* $\Delta.\mathbf{contains}(i_1)$ *and* $\Delta.\mathbf{contains}(i_2)$:

$$\Delta.\mathbf{assign}(i_1, i_2,\ \Delta.\mathbf{at}(i_1, i_2) \cap -\Delta.\mathbf{at}(i_2, i_1)\ ).$$

- *For every i_1, i_2, i_3, s.t.* $\Delta.\mathbf{contains}(i_1)$, $\Delta.\mathbf{contains}(i_2)$, *and* $\Delta.\mathbf{contains}(i_3)$:

$$\Delta.\mathbf{assign}(i_1, i_3),\ \Delta.\mathbf{at}(i_1, i_3) \cap (\Delta.\mathbf{at}(i_1, i_2) + \Delta.\mathbf{at}(i_2, i_3))\ ).$$

We now define state sequences and a couple of operations on them. We define them in imperative fashion, it will result in a clearer description of the effect of an operation, because instead of a large nested functional expression, we can give a sequence of operations that modify the state. In addition, we can make the computation non-deterministic, which removes the need to define the effect as relation. The operations on state sequences are not yet non-deterministic, but they can fail.

Definition 7.0.7. *Let $\mathcal{A} = (Q, \delta, N)$ be a DTA. A* state sequence *over $\mathcal{A}$ is a (possibly empty) sequence of states of $\mathcal{A}$. We define the following operations on state sequences:*

- $\overline{q}.\mathbf{push\_back}(q')$ *replaces* $\overline{q} = (q_1, \ldots, q_n)$ *by* $(q_1, \ldots, q_n, q')$, *and* $\overline{q}.\mathbf{push\_front}(q')$ *replaces* $\overline{q} = (q_1, \ldots, q_n)$ *by* $(q', q_1, \ldots, q_n)$.

- *If* $\overline{q}$ *has form* $(q_1, \ldots, q_n)$ *with* $n > 0$, *then* $\overline{q}$.**pop_back**() *replaces* $\overline{q}$ *by* $(q_1, \ldots, q_{n-1})$, *and* $\overline{q}$.**pop_front**() *replaces* $\overline{q}$ *by* $(q_2, \ldots, q_n)$.

- *If* $\overline{q}$ *has form* $(q_1, \ldots, q_n)$ *with* $n > 0$, *then* $\overline{q}$.**back**() *equals* q_n, *and* $\overline{q}$.**front**() *equals* q_1.

Definition 7.0.8. *Let* $\mathcal{A} = (Q, \delta, N)$ *be a DTA.*

- *Let* q *be a state of* $\mathcal{A}$. *Operation* q.**insert**(q') *replaces* q *by* $\delta(q; q')$.

- *Operation* q.**extract**() *non-deterministically does the following: Find* q', q_1, *s.t.* $\delta(q', q_1) = q$. *Replace* q *by* q' *and return* q_1.

The second operation is one of the sources of non-determinism in the effect of an operation, the other being uncertainty about the relative order of indices.

Definition 7.0.9. *Let* $\mathcal{A}$ *be a an abstraction automaton. An* range state *is a pair of form* (M, Δ), *s.t.*

- M *is a finite partial function, which*

  - *maps single variables to* Q.
  - *maps pairs* (v, i) *consisting of two variables* v, i *to triples* $(q_1, \overline{q}, q_2)$.

  Both components can be partial.

- Δ *is a difference matrix over a set* I *consisting of variables,* 0, *and expressions of form* **len**(v), *with* v *a variable.*

The intended meaning of $q_1, \overline{q}, q_2$ is that they represent the multisets of values in $v[0 \cdots i\rangle$, $v[i]$, and $v\langle i \cdots \rangle$, respectively.

For each program point, we will construct a set of range states that are possible at that point, using Definition 6.0.8. As usual, it cannot be avoided that there will be abstract states attached to a program point that are in fact unreachable.
For simplicity, we will always assume that all assignments to variables (not to a member) are initializations.

We explain the relation between assignment and comparison. Since our computational model is non-deterministic, assignment can be viewed as a special case of comparison, where nothing is known about the assigned variable. More precisely, the assignment $i \leftarrow 0$ can be viewed as two statements: **(1)**. Initialize i with a random value. **(2)** Verify that i equals 0, failing if it is not. Since computation is non-deterministic, i will be equal to 0 in a succesful computation. Our decision is treat assignment in the same way as comparison as much as possible, and to deal with known equalities in a postprocessing state. This has the advantage that common parts of assignment and comparison can be treated by a single procedure.

In order to define the effect of a statement on a range state, we use an imperative, non-deterministic notation. We give statements that change the range state. We tried a declarative approach in earlier versions (inductively defining the effect relation $\vdash$ by means of Horn clauses), but it turned out less readable. Using imperative language is somewhat natural, because in the end we are modeling an imperative language. We use the following methods of M :

- $M.\mathbf{contains}(v),\ M.\mathbf{contains}(v,i)$. True if M contains a value for variable v, or for the pair (v,i).

- $M.\mathbf{at}(v),\ M.\mathbf{at}(v,i)$. Returns the value for the variable v or the pair (v,i).

- $M.\mathbf{assign}(v,d),\ M.\mathbf{assign}(v,i,d)$. Sets the value of variable v, or the pair (v,i) to d.

- $M.\mathbf{erase}(v),\ M.\mathbf{erase}(v,i)$. Remove the value for variable v, or the pair (v,d).

- The statement $i \leftarrow 0$ transforms (M,Δ) as follows:

$$\begin{aligned}&\Delta.\mathbf{extend}(i)\\&\Delta.\mathbf{assign}(i,0,\{0\})\\&\mathbf{close}(\Delta)\\&M.\mathbf{assign}(i,Q_{\mathcal{A}}(0))\end{aligned}$$

- A statement of form $i \leftarrow \mathbf{len}(v)$ transforms (M, Δ) as follows:

$$\begin{aligned}&\Delta.\mathbf{extend}(i)\\&\Delta.\mathbf{assign}(i, \mathbf{len}(v), \{0\})\\&\mathbf{close}(\Delta)\end{aligned}$$

- A statement of form $i \leftarrow j$, transforms (M, Δ) as follows:

$$\begin{aligned}&\Delta.\mathbf{extend}(i)\\&\Delta.\mathbf{assign}(i, j, \{0\})\\&\mathbf{close}(\Delta)\end{aligned}$$

- If the statement at π has form $j \leftarrow i + 1$, then
  - Let $\Delta' = \mathbf{clos}(\ (\Delta + \{j\})(j, i) := 1\ )$.
  - If $M.\mathbf{at}(f[i])$ is not defined, execute $v' \leftarrow v.f[i]$ using a dummy variable v'. After that,

$$\begin{aligned}&M.\mathbf{assign}(v.f[\cdots j\rangle,\ \delta(\ \{\ M.\mathbf{at}(v.f[i])\ \} \cup M.\mathbf{at}(v.f[\cdots i\rangle)\ )\ )\\&M.\mathbf{assign}(v.f[j\cdots\rangle], M.\mathbf{at}(\langle i\cdots\rangle)\ )\end{aligned}$$

- If the statement has form $i \leftarrow j - 1$, then add $\Delta(\alpha, \beta)/\{-1\}$ to σ.

- A statement of form $w \leftarrow v[i]$ transforms (M, Δ) as follows:

$$\begin{aligned}&(q_1, \overline{q}, q_2) = M.\mathbf{at}(v, i)\\&\text{if } \|\overline{q}\| = 0 \text{ then}\\&\quad \overline{q}.\mathbf{push\_back}(q_2.\mathbf{extract}())\\&M.\mathbf{assign}(w, \overline{q}.\mathbf{front}())\\&M.\mathbf{assign}(v, i, (q_1, \overline{q}, q_2))\end{aligned}$$

- A π has form $v.f[i] := w$, then if either $M.\mathbf{at}(\ v.f[0\cdots i]\ )$ or $M.\mathbf{at}(\ [i\cdots\rangle\ )$ is defined, execute $v' \leftarrow v.f[i]$ using a dummy variable v'.

After that,

$$
\begin{aligned}
&M.\mathbf{assign}(\, v.f[i], w\,)\\
&M.\mathbf{assign}(\, v.f[0\cdots\rangle,\, \delta(\,\{M.\mathbf{at}(\, v.f[i]\,)\} \cup M.\mathbf{at}(\, v.f[0\ldots i\rangle\,))\,)\\
&M.\mathbf{assign}(\, v.f[0\cdots\rangle\, \delta(\,\{M.\mathbf{at}(\, v.f[i]\,)\} \cup M.\mathbf{at}(\, v.f\langle i\ldots\rangle\,))\,)
\end{aligned}
$$

- A statement of form $v.\mathbf{push\_back}(w)$ transforms (M, Δ) as shown below:
  - First add w to the multiset of values of v :

$$M.\mathbf{assign}(v, \delta(M.\mathbf{at}(v); M.\mathbf{at}(w))\,).$$

  - For every i, s.t. $M.\mathbf{at}(v, i)$ is defined:

$$
\begin{aligned}
&(q_1, \overline{q}, q_2) = M.\mathbf{at}(v, i)\\
&q_2 = \delta(q_2; M.\mathbf{at}(w))\\
&M.\mathbf{assign}(v, i, (q_1, \overline{q}, q_2)\,)
\end{aligned}
$$

  - We also have to register the fact that the size of v increases by 1. In order to do this, create a new variable z, register the fact that $z = \mathbf{len}(v) + 1$. After that, erase $\mathbf{len}(v)$. Reinsert $\mathbf{len}(v)$, this time using $\mathbf{len}(v) = z$.

$$
\begin{aligned}
&\Delta.\mathbf{extend}(z)\\
&\Delta.\mathbf{assign}(z, \mathbf{len}(v), \{1\})\\
&\mathbf{close}(\Delta)\\
&\Delta.\mathbf{erase}(\mathbf{len}(v))\\
&\Delta.\mathbf{extend}(\mathbf{len}(v))\\
&\Delta.\mathbf{assign}(z, \mathbf{len}(v), \{0\})\\
&\mathbf{close}(\Delta)\\
&\Delta.\mathbf{erase}(z)
\end{aligned}
$$

    This procedure could be shortened by adding more primitive operations on difference matrices, like for example $\Delta.\mathbf{rename}(z, \mathbf{len}(v))$.

- A statement of form $v \leftarrow w.f$ transforms (M, Δ) as follows:

  $(q_1, \ldots, q_n) \rightarrow q$ in δ. Look up $\mathcal{F}(f) = p/i$. Each state $\sigma' = \sigma \cup \{\alpha/q_i\}$ is a possible successor.

- A statement of form $v \leftarrow \mathbf{lt}(w_1, w_2)$ non-deterministically changes (M, Δ) in the following two ways:

  –

  $\Delta.\mathbf{assign}(w_1, w_2, \Delta.\mathbf{at}(w_1, w_2) \cap \{-2^*, -1\})$
  $\mathbf{close}(\Delta)$
  if $\neg\Delta.\mathbf{isconsistent}()$)
  fail
  $M.\mathbf{assign}(v, \mathbf{t})$

  –

  $\Delta.\mathbf{assign}(w_1, w_2, \Delta.\mathbf{at}(w_1, w_2) \cap \{0, 1, 2^*\})$
  $\mathbf{close}(\Delta)$
  if $\neg\Delta.\mathbf{isconsistent}()$)
  fail
  $M.\mathbf{assign}(v, \mathbf{f})$

  The other comparison operators are analogeous. (M, Δ) either has one or two successor states, dependent on whether assuming $w_1 < w_2$ or $w_1 \geq w_2$ is consistent with Δ.

- A statement of form $v \leftarrow F(w)$, with F is a unary state condition, changes (M, Δ) as follows:

  if $F(w)$
  $M.\mathbf{assign}(v, \mathbf{t})$
  else
  $M.\mathbf{assign}(v, \mathbf{f})$

- A statement of form $v \leftarrow F(w_1, w_2)$ with F a binary state condition has similar effect as a unary state conditions.

We define a postprocessing procedure that must be called after every operation on a range state. It checks equality using Δ, and it removes variables (or pairs of variables) from M that are not live any more. As for the definition of the effect of a statement, we use an imperative, non-deterministic description. The procedure never branches, but it may fail.

Definition 7.0.10.

- *For every variable i, s.t. $\Delta.\mathbf{at}(i,0) = \{0\}$, for every variable v, s.t. $M.\mathbf{contains}(v)$ and $M.\mathbf{contains}(v,i)$, do*

$$
\begin{aligned}
&(q_1, \overline{q}, q_2) = M.\mathbf{at}(v,i)\\
&q = q_2\\
&\textit{for } q' \in \overline{q} \textit{ do}\\
&\quad q = \delta(q'; q)\\
&\textit{if } q \neq q' \textit{ then}\\
&\quad \mathbf{fail}
\end{aligned}
$$

- *Every variable i, s.t. $\Delta.\mathbf{at}(i, \mathbf{len}(v))$ is treated analogously.*

- *For every variable v, s.t. $M.\mathbf{contains}(v)$, but v does not occur in $\mathbf{live_{out}}(\pi)$, call $M.\mathbf{erase}(v)$. For every pair of variables v, i, s.t. $M.\mathbf{contains}(v,i)$, and either v or i does not occur in $\mathbf{live_{out}}(\pi)$, call $M.\mathbf{erase}(v,i)$.*

The definition above cannot stay in this form.

Chapter 8

Overload Resolution

As in the previous section, we assume a fixed DTA $\mathcal{A} = (Q, \delta, N)$ throughout the entire section. We handle overload resolution by cutting out non-viable overloads from the flow graph. For this purpose, one can use the semantics defined in Definition 6.0.8, or the rules defined in Section 7.

Definition 8.0.1. *Let d be a sequence of data terms, let $\mathcal{A}$ be a sequence of DTA's, let F be a sequence of state conditions, s.t. $\|d\| = \|\mathcal{A}\| = \|F\|$. We say that d* satisfies *$(\mathcal{A}, \mathcal{F})$ if for every i with $1 \leq i \leq \|\mathcal{A}\|$, we have $F_i(\, Q_{i,\mathcal{A}}(d_i)\,)$.*

Definition 8.0.2. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be sequences of DTA's, let F_1 and F_2 be sequences of state conditions, s.t. $\|\mathcal{A}_1\| = \|\mathcal{A}_2\| = \|F_1\| = \|F_2\|$. We say that $(\mathcal{A}_1, F_1)$ implies $(\mathcal{A}_2, F_2)$ if for every data term d, for every i with $1 \leq i \leq \|\mathcal{A}_1\|$,*

$$F_{1,i}(Q_{1,i,\mathcal{A}}(d)) \text{ implies } F_{2,i}(Q_{2,i,\mathcal{A}}(d)).$$

We write $(\mathcal{A}_1, F_1) \sqsubseteq (\mathcal{A}_2, F_2)$ if $(\mathcal{A}_1, F_1)$ implies $(\mathcal{A}_2, F_2)$. We write $(\mathcal{A}_1, F_1) \sqsubset (\mathcal{A}_2, F_2)$ if $(\mathcal{A}_1, F_1) \sqsubseteq (\mathcal{A}_2, F_2)$ and not $(\mathcal{A}_2, F_2) \sqsubseteq (\mathcal{A}_1, F_1)$.

Definition 8.0.3. *Let s be a statement.* Typechecking and overload resolution *works as follows:*

1. *If s contains a* **resolve** *statement without any options left, then report failure for the given* **resolve**.

2. *Construct* $\Phi_{\mathbf{in}}$ *and* $\Phi_{\mathbf{out}}$ *for every program point in* s.

3. *For every* π, *s.t.* π *contains a statement of form*

$$\mathbf{resolve}(\overline{v}, (\mathcal{A}_1, F_1)/s_1, \ldots, (\mathcal{A}_n, F_n)/s_n)\,)$$

for every state $\sigma \in \Phi_{\mathbf{in}}(\pi)$, *let* $d = (\sigma(v_1), \ldots, \sigma(v_n))$. *Remove the* $(\mathcal{A}_i, F_i)/s_i$ *for which* d *does not satisfy* $(\mathcal{A}_i, F_i)$ *from the* **resolve** *statement.*

If this results in changes, restart at Step 1.

4. *At this point, we know that every* **resolve** *statement has overloads, but there may be too many. For every* π, *s.t.* π *contains a statement of form*

$$\mathbf{resolve}(\overline{v}, (\mathcal{A}_1, F_1)/s_1, \ldots, (\mathcal{A}_n, F_n)/s_n)\,)$$

with $n \geq 2$, *check that there exists an* i *with* $1 \leq i \leq n$, *s.t.* $i' \neq i$ *implies* $(\mathcal{A}'_i, F'_i) \sqsubset (\mathcal{A}_i, F_i)$. *If no such* i *exists, then report that the* **resolve** *statement is ambiguous.*

Chapter 9

Conclusion

We presented a context aware type system and a type-checking procedure that could be used for implementing an experimental imperative programming language working on tree-like data structures which can have different forms based on the way they are constructed. The type system can determine forms of data trees from a program context and ensure the correctness of field accesses and function applications. The from of a data tree is inferred from program context information since the from depends on runtime modifications of data trees. During type checking, the program context is approximated as a function (called Φ) that maps each program point to a set of possible program states. Indexing information is approximated with special array range states and relative order of indices. This range interpretation allows us to reason about the state of an array at some index without storing the states of individual elements. Once the program approximation is computed, the type-checking algorithm checks correctness of function and field applications, and removes overloads whose preconditions are not met. Function and field applications are valid if there is a unique, most specific overload whose precondition is satisfied by the program state just before it. Otherwise, our type-checking algorithm concludes that the program is ill-typed.

At this moment, we do not have an implementation of the type system. We have no empirical data about its feasibility. Future work will involve developing a working prototype based on the proposed design. This would allow for practical validation of

the system and provide insights into real-world challenges. In particular, it will be important to assess the scalability and performance of the system under real-world examples. Moreover, in this project, it was not discussed how tree automata are obtained from real type conditions. Future work needs to explore different ways in which tree automata are generated from preconditions and prove that such automata exists.

At present, the $\Phi_{\mathbf{in}}$ and $\Phi_{\mathbf{out}}$ sets are constructed as sets of independent states. Since a single state can be represented as a conjunction of atoms of form $\sigma(v) = t$, a set of possible states can be viewed as a single formula in disjunctive normal form. Using conjunctive normal form would result in a much more compact representation, but in the process, information about dependencies between variables will be lost. In most cases, this is unproblematic, but it matters in some cases, e.g. after one has checked that two selector variables are equal, or when one has stored the result of a comparison in a boolean variable. Future work needs to focus on finding a compact representation that keeps just enough information about dependencies. Perhaps one could use a decision tree based on a few binary predicates.

Appendix A

An Example Mixing Construction and Destruction

Consider the following flow graph $\mathcal{G}$ that represents a function. The function that takes an even number m as an argument and after arbitrary number of modification of the value of m returns a still even m. Here, the goal of type checking is to prove m is even natural number at the return statement.

```
p1   repeat
p2       α ← some(bool)
p3       β ← m.sel
p4       γ0 ←?succ
p5       γ1 ← eq(β, γ0)
p6       branch
p7           block
p8               isfalse α
p9               isfalse γ1
p10              n ← (?succ, m)
p11              m ← (?succ, n)
p12          block
p13              istrue α
p14              istrue γ1
p15              resolve n ← m.pred
p16              resolve m ← n.pred
p17  resolve return m
```

Here, a data type of natural numbers **Nat** along with two adjectives **Even** and **Odd** are defined as follows:

```
typedef Nat = {
    ?zero => ;
    ?succ => pred : Nat;
};
typedef Even = (?zero) || (?succ, Odd);
typedef Odd = (?succ, Even);
```

First, one needs to define a deterministic tree automata that can classify data terms into usefull groups. Since the funtion precondition and postcondition involves even and odd numbers, the automaton should distinguish even and odd numbers. Moreover, the function operates differently on the zero even number and non-zero

even numbers, thus the automaton should also distinguish these two forms of even numbers. Taking into account these requirements, we construct a deterministic tree automaton $\mathcal{A}_{\text{EvenOdd}} = (Q, \delta, 2)$. The transition function δ is defined as follows:

$$\begin{aligned}
\delta(?\text{zero}) &= q_{\text{zero}} \\
\delta(?\text{succ}) &= q_{\text{succ}} \\
\delta(\top) &= q_{\text{True}} \\
\delta(\bot) &= q_{\text{False}} \\
\delta(q_{\text{zero}}) &= q_{\text{Even1}} \\
\delta(q_{\text{succ}}, q_{\text{Odd}}) &= q_{\text{Even2}} \\
\delta(q_{\text{succ}}, q_{\text{Even1}}) &= q_{\text{Odd}} \\
\delta(q_{\text{succ}}, q_{\text{Even2}}) &= q_{\text{Odd}}
\end{aligned}$$

Note that δ is made total by mapping all other data terms and multisets to a default state q_{def}.

Next, we define a field map for the fields of **Nat**. Type definition of **Nat** specifies only one field named pred that belongs to non-zero natural numbers.

$$\mathcal{F}^{\textbf{field}}(\text{pred}) = (F_{q_{\text{Even2}},\ q_{\text{Odd}},\ 1})$$

Appendix B

An Example with Repeated Field Inspection

Example B.0.1. *In the flow graph below, we assume that both x_1 and x_2 are natural numbers. The program checks that x_1 and x_2 have the same selector. After that, it checks that the selector of x_1 equals* ?zero. *We assume that function f has precondition*

(?zero).

```
repeat
    α1 ← n1.op
    α2 ← n2.op
    β ← eq(α1, α2)
    branch
        block
            isfalse β
            return false
        block
            z ← ?zero
            β ← eq(α1, z)
            branch
                block
                    istrue β
                    return true
                block
                    isfalse β
                    resolve n1 ← n1.pred
                    resolve n2 ← n2.pred
```

Appendix C

An Example with Property Conversion in an Array

Consider a function with input parameter x. Initially, we have $x \models A^{\forall}$ and the return statement requires $x \models B^{\forall}$.

block

$i \leftarrow 0$

$\alpha \leftarrow \mathbf{len}(x)$

$\beta \leftarrow \mathbf{lt}(i, \alpha)$

$\gamma \leftarrow \mathbf{not}(\beta)$

repeat

isfalse γ

resolve $\delta \leftarrow x.f[i]$

resolve $\eta \leftarrow F(\delta)$

resolve $x.f[i] := \eta$

$j \leftarrow 1$

$i' \leftarrow \mathbf{add}(i, j)$

$i \leftarrow i'$

$\alpha \leftarrow \mathbf{len}(x)$

$\beta \leftarrow \mathbf{lt}(i, \alpha)$

$\gamma \leftarrow \mathbf{not}(\beta)$

resolve return x

Assume that function $F(\ )$ requires adjective A and returns adjective B. Both are adjectives defined on a type T. Assume that x initially satisfies $(A^{\forall})$, we need to show x satisfies $(B^{\forall})$ at the return statement.

The generating automata for properties $A^{\forall}$ and $B^{\forall}$ are:

$$\mathcal{A}_{A^{\forall}} = \begin{cases} \emptyset & \rightarrow & q_{A^{\forall}} \\ \{q_A\} \cup q_{A^{\forall}} & \rightarrow & q_{A^{\forall}} \\ \{q_{\overline{A}}\} \cup q_{A^{\forall}} & \rightarrow & q_{\overline{A}^{\exists}} \\ \{q_A\} \cup q_{\overline{A}^{\exists}} & \rightarrow & q_{\overline{A}^{\exists}} \\ \{q_{\overline{A}}\} \cup q_{\overline{A}^{\exists}} & \rightarrow & q_{\overline{A}^{\exists}} \end{cases} \quad \text{and} \quad \mathcal{A}_{B^{\forall}} = \begin{cases} \emptyset & \rightarrow & q_{B^{\forall}} \\ \{q_B\} \cup q_{B^{\forall}} & \rightarrow & q_{B^{\forall}} \\ \{q_{\overline{B}}\} \cup q_{B^{\forall}} & \rightarrow & q_{\overline{B}^{\exists}} \\ \{q_B\} \cup q_{\overline{B}^{\exists}} & \rightarrow & q_{\overline{B}^{\exists}} \\ \{q_{\overline{B}}\} \cup q_{\overline{B}^{\exists}} & \rightarrow & q_{\overline{B}^{\exists}} \end{cases}$$

Bibliography

[1] Alexander Aiken, Dexter Kozen, Moshe Vardi, and Ed Wimmers. The complexity of set constraints. In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *Computer Science Logic*, pages 1–17, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[2] Alexander Aiken and Brian Murphy. Implementing regular tree expressions. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, page 427–447, Berlin, Heidelberg, 1991. Springer-Verlag.

[3] Alexander Aiken and Brian Murphy. Static type inference in a dynamically typed language. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, page 279–290, New York, NY, USA, 1991. Association for Computing Machinery.

[4] Alexander Aiken and Edward Wimmers. Solving systems of set constraints. In *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, 1992.

[5] Alexander Aiken and Edward Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 31–41, New York, NY, USA, 1993. Association for Computing Machinery.

[6] Alexander Aiken, Edward Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 163–173, New York, NY, USA, 1994. Association for Computing Machinery.

[7] Witold Charatonik and Leszek Pacholski. Set constraints with projections are in NEXPTIME . In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 642–653, Los Alamitos, CA, USA, November 1994. IEEE Computer Society.

[8] Witold Charatonik and Andreas Podelski. Set constraints with intersection. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 362–372, 1997.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[10] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, page 170–181, New York, NY, USA, 1995. Association for Computing Machinery.

[11] Robert Floyd. *Assigning Meanings to Programs*, volume 14. Springer Netherlands, Dordrecht, 1967.

[12] Nevin Heintze. Set-based analysis of ml programs. *SIGPLAN Lisp Pointers*, VII(3):306–317, July 1994.

[13] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 197–209, New York, NY, USA, 1989. Association for Computing Machinery.

[14] Nevin Heintze and Joxan Jaffar. A decision procedure for a class of set constraints. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.

[15] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, page 244–256, New York, NY, USA, 1979. Association for Computing Machinery.

[16] Hans de Nivelle. Tools for compiler construction. `http://compiler-tools.eu/`. Accessed 10-04-2025.

[17] John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morrel, editor, *Information Processing, Proceedings of IFIP Congress 1968, Edinburgh, UK, 5-10 August 1968, Volume 1 - Mathematics, Software*, pages 456–461, 1968.