Implementing Set-Theoretic Types

MICKAËL LAURENT*, Charles University, Czech Republic
KIM NGUYỄN*, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France

Set-theoretic types provide a rich type algebra that supports unrestricted unions, intersections, and negations, together with a decidable type constraint-solving algorithm known as tallying. These types are particularly well suited for typing dynamic languages, where functions often exhibit both generic and overloaded behavior. However, the complexity of their implementation has hindered their widespread adoption. In this paper, we introduce a modular representation for set-theoretic types and revisit the algorithms for subtyping and tallying. We compare our approach with the historical CDuce implementation and evaluate the performance impact of some optimizations and design choices.

CCS Concepts: • **Theory of computation** → *Type structures*; • **Software and its engineering** → *Polymorphism*; *Data types and structures.*

Additional Key Words and Phrases: set-theoretic types, semantic subtyping, tallying, implementation

1 Introduction

For over a decade, programmers have added static type systems to dynamic, originally untyped languages such as JavaScript and Python. These efforts aim to combine the flexibility of dynamic typing with the reliability of static type checking. Examples of such type systems are TypeScript and Flow for JavaScript or Mypy, Pyre and Pyright for Python. To account for the variety of programming patterns expressible in untyped languages, these type systems often feature several (if not all) of the typing constructs that are part of the programming language literature, namely:

- parametric polymorphism (often referred to as *generics*)
- overloading or ad-hoc polymorphism, together with subtyping
- occurrence typing [27] (the refinement of a type following a dynamic type test)
- arbitrary union types (e.g. to express heterogenous collections or optional results)
- intersection types (in particular for functions or records)
- gradual typing [26], the ability to mix typed and untyped code

Mixing together *all* of these, in a principled way, is a daunting task. Most, if not all the previously mentioned systems do not guaranty type safety and mix the various typing constructs in an ad-hoc way. Yet, for some of their use cases (documentation, test generation, code completion in IDE, …) they provide excellent performance, and are able to infer *some* type information relatively quickly.

A theoretical formalism handling these features does exist: *set-theoretic types*. They were first introduced in the context of XML programming [18, 21] and later extended with parametric polymorphism [13], gradual typing [10] and type narrowing [11]. While set-theoretic types have started to be used as a theoretical foundation for type systems (e.g. Etylizer [25] for Erlang, Elixir [8]), they have not seen the wide adoption one could have hoped for. We believe that one of the main issues is the disconnect between the mathematically elegant theoretical foundation and the design and implementation issues a practical implementation faces.

The goal of this work is to report on SSTT, the simple set-theoretic type library, a reference implementation of set-theoretic types. We cover the implementation and data-structure of the basic type algebra, show how to extend it to cover practical (but often ignored) data types, and present

*Both authors contributed equally to the paper

Authors' Contact Information: Mickaël Laurent, mickael.laurent@matfyz.cuni.cz, Charles University, Prague, Czech Republic; Kim Nguyễn, kn@lmf.cnrs.fr, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France.

implementation techniques of high-level operators, namely subtyping and tallying (computing a set of substitutions that solve a subtyping constraint).

1.1 Overview

As a practical introduction to set-theoretic types, we use the following example from [13].

```
val map  : ∀α, β.(α → β) → [α*] → [β*]
val even : ∀γ.(int → bool) ∧ ((γ \ int) → (γ \ int))
```

Here, the `map` function has its familiar type, with the twist that list types are *regular expression types*. These are merely syntactic sugar for recursive types. For instance, the type $[\alpha*]$ is the recursive type defined by the equation $X = \texttt{nil} \vee \alpha \times X$, which is a *union* of `nil` — the *singleton* type of the constant representing the empty list — and a product type of the type variable α (the polymorphic type of the head of the list) and X standing for the list type itself (the tail of the list). The `even` function is an *intersection* of two arrow types, meaning that `even` is an overloaded function. When this function is applied to a value of type `int`, it returns a value of type `bool`. Otherwise, when applied to values that are not integers, captured by the *set difference* $\gamma \setminus \texttt{int}$, it returns a value of that type[1]. Now, let's imagine what would be needed to type the (partial) application `map even`. In a Hindley-Milner style type system, we would need to:

(1) find a type substitution which unifies the type of the domain of `map` with the type of `even`
(2) apply this substitution to the codomain of `map` ($[\alpha*] \to [\beta*]$) to deduce the type of the whole expression.

However, in the context of set-theoretic types, *syntactic* unification is not powerful enough. Indeed, it would fail here, since it confronts an arrow type ($\alpha \to \beta$) against an intersection type (the type of `even`). One of the key feature of set-theoretic types is the associated notion of *semantic subtyping*. With this notion, the behavior of set-theoretic connectives w.r.t. type constructors (e.g. idempotence, distributivity) is derived from set theory. To reconcile parametric polymorphism and subtyping-based polymorphism, [13] introduces the *tallying* operation. In a nutshell, tallying consists in finding type substitutions for type variables that make a set of type inequations hold. Thus, typing the application `map even` consists in finding type substitutions such that

$$(\alpha \to \beta) \to [\alpha*] \to [\beta*] \leq ((\texttt{int} \to \texttt{bool}) \wedge ((\gamma \setminus \texttt{int}) \to (\gamma \setminus \texttt{int}))) \to \delta$$

where $\leq$ denotes subtyping and δ is a freshly introduced type variable that represents the type we are interested in, that is, the type of `map even`. This single problem contains the constraints that:

- the type of `even` must be a subtype of $\alpha \to \beta$ (contravariance on the domain of `map`)
- δ must be supertype of $[\alpha*] \to [\beta*]$ (covariance on the codomain of `map`)

The result of this tallying problem is a set of four substitutions $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$, and the type of the application is therefore $\delta\sigma_1 \wedge \delta\sigma_2 \wedge \delta\sigma_3 \wedge \delta\sigma_4$, which becomes the type of the overloaded function:

$$\begin{array}{c}([(\texttt{int} \vee \gamma_1)*] \to [(\texttt{bool} \vee (\gamma_1 \setminus \texttt{int}))*]) \ \wedge \ (\texttt{nil} \to \texttt{nil}) \ \wedge \\ ([(\gamma_3 \setminus \texttt{int})*] \to [(\gamma_3 \setminus \texttt{int})*]) \ \wedge \ ([\texttt{int}*] \to [\texttt{bool}*])\end{array}$$

This short example highlights most of the challenges one faces when trying to implement set-theoretic types. While some of them are documented in the literature, we propose in this work a systematic presentation of all the relevant aspects of the implementation of set-theoretic types, together with novel implementation techniques and insights discovered while implementing the SSTT library. Each technique is evaluated on a realistic use of set-theoretic types.

[1] To implement such behavior, the underlying programming language supports a dynamic type case construct.

The rest of the paper is structured as follows. In Section 2, we recall basic definitions about set-theoretic types. Then, Section 3 introduces *Binary Decision Trees* (BDTs), a data-structure which is used pervasively in the implementation of types presented in Section 4. The subtyping and tallying algorithms are described in Section 5. Section 6 shows how several useful extensions can be encoded in the base type algebra. Section 7 evaluates and compares our implementation performance in different settings. Finally, Section 8 presents related and future work.

2 Set-theoretic types

2.1 Definitions

The types we implement in this paper are the set-theoretic types. For more detail, we refer the reader to the survey paper [7] which synthesises the relevant results from the literature.

Definition 2.1 (Set-theoretic types). The set $\mathcal{T}$ of *set-theoretic types* is the set of regular and contractive terms coinductively defined by the following grammar:

$$\textbf{Types} \quad t \quad ::= \quad b \mid \alpha \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

where $b \in \mathcal{B}$ is a base type (in particular, it includes the constants $c \in \mathcal{C}$ of the language we want to type) and $\alpha \in \mathcal{V}$ is a type variable. The notation $t_1 \setminus t_2$ is a syntactic sugar for $t_1 \wedge \neg t_2$. When writing a term, we use the following precedence (by decreasing priority): $\neg$, $\setminus$, $\wedge$, $\vee$, $\times$, $\to$.

As they are defined coinductively, types can be infinite trees, provided that they satisfy the constraints of regularity and contractivity explained below to ensure decidability of the subtyping relation. This yields a definition of equirecursive types that does not require explicit binders for recursion. A term is said *regular* if it only has a finite number of distinct subterms, and *contractive* if every infinite branch goes through an infinite number of arrows and products ($\to$ and $\times$).

The type $\mathbb{0}$ is a special type that is not inhabited by any value, and is the subtype of all types. Conversely, the type $\mathbb{1}$ is the supertype of all types. The $\times$ constructor is used to type pairs, and the $\to$ constructor is used to type functions. These are the most common type constructors, but set-theoretic types can be extended with other constructors as we will see throughout this paper.

Set-theoretic types are equipped with a decidable subtyping relation $\leq$ (referred to as *semantic subtyping*, cf. Appendix A for more details). For this presentation, it suffices to consider that each type can be interpreted as a set of values that have that type, and that subtyping is set containment. Type connectives (union, intersection, negation) are interpreted as the corresponding set-theoretic operators. Although the interpretation of type variables is more complex, the following property is sufficient to grasp the behavior of subtyping in presence of type variables:

Proposition 2.2 (Subtyping, [16]). *Let t and s be two types. The type s is a subtype of type t if, for every type substitution σ, we have $s\sigma \leq t\sigma$.*

We note $\simeq$ the semantic equivalence: $t_1 \simeq t_2$ if and only if $t_1 \leq t_2$ and $t_2 \leq t_1$. Thanks to negation, the subtyping problem is equivalent to checking the emptiness of a type: $s \leq t \iff s \setminus t \leq \mathbb{0}$.

As presented in the introduction, subtyping is a building block for the definition of tallying:

Definition 2.3 (Tallying, [13]). Let $S = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ be a finite set of pairs of types and Δ a set of type variables. The solution of the tallying problem for S and Δ is:

$$\mathsf{tally}(S, \Delta) = \{\sigma \mid \mathsf{dom}(\sigma) \cap \Delta = \varnothing \ \wedge \ \forall (s, t) \in S.\ s\sigma \leq t\sigma\}$$

The set S contains the subtyping constraints, and the set Δ is the set of variables that the tallying is not allowed to instantiate. An important property of the tallying operation is that it is complete: it computes a finite set of substitutions which exactly characterize all the possible solutions.

2.2 Types in disjunctive normal forms

As hinted to in the previous section, deciding subtyping is equivalent to checking the emptiness of a type. A convenient way to express the subtyping algorithm is to put the type in *disjunctive normal form* (DNF). Given a type t, its DNF as defined in [16] is a syntactic term of the following form:

$$\mathsf{DNF}(t) = \begin{array}{lll} & \bigvee \bigwedge_i \alpha_i^{\mathrm{b}} \wedge \bigwedge_j \neg\beta_j^{\mathrm{b}} \wedge \bigwedge_k b_k \wedge \bigwedge_l \neg b_l \\ \vee & \bigvee \bigwedge_i \alpha_i^{\mathrm{p}} \wedge \bigwedge_j \neg\beta_j^{\mathrm{p}} \wedge \bigwedge_k (t_k^1 \times t_k^2) \wedge \bigwedge_l \neg(s_l^1 \times s_l^2) \\ \vee & \bigvee \bigwedge_i \alpha_i^{\mathrm{a}} \wedge \bigwedge_j \neg\beta_j^{\mathrm{a}} \wedge \bigwedge_k (t_k^1 \to t_k^2) \wedge \bigwedge_l \neg(s_l^1 \to s_l^2) \end{array} \quad (*)$$

In this formula, each row is itself a DNF of positive and negative type variables and positive and negative instances of a particular type constructor: a basic type (we assume for now a single kind of basic type), a product, or an arrow. The subtyping algorithm can then iterate through the elements of this DNF to test the emptiness of the type. Notice that as usual, the DNF of a type may be exponential in the size of the original type expression. The complexity of subtyping is therefore in EXPTIME (which was shown by [20] or, alternatively, can be shown by reducing the problem to testing the inclusion of regular tree languages given by non-deterministic tree automata). Keeping types explicitly as DNF would be particularly impractical, we therefore need to devise a generic data-structure to represent DNF compactly.

3 Binary Decision Trees

In this section, we describe *Binary Decision Trees* (BDTs), a data structure for representing Boolean combinations (union, intersection, and negation) of *atoms*. The nature of these atoms is varied, as shown in the previous section. They can be type variables, basic types, or type constructors. We develop the meta-theory of BDT and operations by abstracting over atoms. In addition to the standard Boolean operations on BDTs (Section 3.2), we define a semantic simplification operation (Section 3.3).

3.1 Structure of a BDT

Let $\mathcal{A}$ be a set of atoms, ranged over by a. We assume we have a total order $\preceq$ over atoms, as well as an interpretation $[\![a]\!]$ of an atom a as a type. Let $\mathcal{L}$ be a set of leaves, ranged over by l. We assume we have an interpretation $[\![l]\!]$ of a leaf l as a type, a bottom leaf element $\bot$ whose interpretation $[\![\bot]\!]$ is the type $\mathbb{0}$, and a top leaf element $\top$ whose interpretation $[\![\top]\!]$ is the type $\mathbb{1}$. In addition, the set-theoretic operations $\wedge$, $\vee$, and $\neg$ must be defined on leaves, in accordance with the interpretation $[\![.]\!]$ ($[\![l_1 \wedge l_2]\!] \simeq [\![l_1]\!] \wedge [\![l_2]\!]$, $[\![l_1 \vee l_2]\!] \simeq [\![l_1]\!] \vee [\![l_2]\!]$, $[\![\neg l]\!] \simeq \neg[\![l]\!]$).

Definition 3.1. The set of Binary Decision Trees $\mathrm{BDT}(\mathcal{A}, \mathcal{L})$ over atoms $\mathcal{A}$ and leaves $\mathcal{L}$ are the finite trees produced by the following grammar (where $l \in \mathcal{L}$ and $a \in \mathcal{A}$):

$$\textbf{(Ordered) Binary Decision Trees} \quad B \quad ::= \quad \mathsf{L}(l) \mid \mathsf{N}(a?B\!:\!B)$$

with the property that for any non-root node labeled a whose parent is labeled a', we have $a \not\preceq a'$.

For concision, we use the notation $\bot$ for $\mathsf{L}(\bot)$, $\top$ for $\mathsf{L}(\top)$, and $\mathsf{N}(a)$ for the BDT node $\mathsf{N}(a?\top\!:\!\bot)$. The auxiliary definitions below allow us to extract the set of atoms or the set of leaves of a BDT:

Definition 3.2. For any BDT B, we define $\mathsf{leaves}(B)$ and $\mathsf{atoms}(B)$ as follows:

$$\begin{array}{lcllcl} \mathsf{leaves}(\mathsf{L}(l)) & = & \{l\} & \mathsf{leaves}(\mathsf{N}(a?B^+\!:\!B^-)) & = & \mathsf{leaves}(B^+) \cup \mathsf{leaves}(B^-) \\ \mathsf{atoms}(\mathsf{L}(l)) & = & \varnothing & \mathsf{atoms}(\mathsf{N}(a?B^+\!:\!B^-)) & = & \mathsf{atoms}(B^+) \cup \mathsf{atoms}(B^-) \cup \{a\} \end{array}$$

The meaning of a BDT, that is, the type that it represents, is given by the way of an interpretation function from BDTs to types.

Definition 3.3. The *interpretation* $[\![B]\!]$ of a binary decision tree B is inductively defined as follows:

$$[\![\mathsf{L}(l)]\!] \quad = \quad [\![l]\!] \qquad\qquad [\![\mathsf{N}(a?B^+:B^-)]\!] \quad = \quad ([\![a]\!] \wedge [\![B^+]\!]) \vee (\neg[\![a]\!] \wedge [\![B^-]\!])$$

Note that the interpretation of a BDT B is a Boolean combination of atoms and leaves. An example of a BDT and the associated interpretation can be found in Figure 1. This Boolean combination can be expressed as a DNF (cf. Section 2.2):

Definition 3.4. The DNF $\mathsf{dnf}(B)$ of a BDT B is a set of triples (A_p, A_n, l), where each triple represents a clause intersecting the positive atoms A_p, the negative atoms A_n, and the leaf l:

$$\begin{array}{rcl} \mathsf{dnf}(\mathsf{L}(l)) & = & \{(\varnothing, \varnothing, l)\} \\ \mathsf{dnf}(\mathsf{N}(a?B^+:B^-)) & = & \{(A_p \cup \{a\}, A_n, l) \mid (A_p, A_n, l) \in \mathsf{dnf}(B^+)\} \cup \\ & & \{(A_p, A_n \cup \{a\}, l) \mid (A_p, A_n, l) \in \mathsf{dnf}(B^-)\} \end{array}$$

To ensure a BDT does not represent a trivially empty type, we define the following property:

Definition 3.5 (Reduced BDT). We say that a BDT is reduced, if and only if all non leaf subtrees of the BDT are of the form $\mathsf{N}(a?B^+:B^-)$ with $B^+ \neq B^-$ (where $\neq$ denotes syntactic inequality).

Note that it is always possible to create BDTs that are reduced. To do so, we assume that $\mathsf{N}(\_?\_:\_)$ behaves as a *smart constructor* which performs the following simplification: $\mathsf{N}(a?B:B) \rightsquigarrow B$. Henceforth, we assume all BDTs to be reduced.

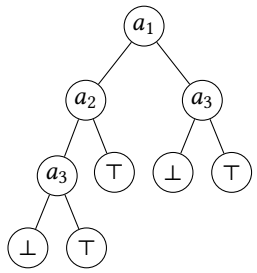

Fig. 1. BDT whose interpretation is $[\![a_1]\!] \wedge ([\![a_2]\!] \wedge \neg[\![a_3]\!] \vee \neg[\![a_2]\!]) \vee \neg[\![a_1]\!] \wedge \neg[\![a_3]\!]$

The knowledgeable reader may recognize our BDTs as variants of Reduced Ordered Binary Decision Diagrams (ROBDD) [3]. There are however two key distinctions between the two formalisms. The first one is that for BDTs, leaf nodes may be arbitrary elements of a lattice (that can be mapped to the lattice of types). The second one is that we do not require BDTs to be DAGs, that is, we do not enforce that structurally equal subtrees are shared in memory. While it is possible to enforce this property (e.g. by performing hash-consing at construction time), it has two drawbacks. The first one is that, in typical use, it is seldom the case that identical (and sufficiently large) subtrees are built to offset the cost of hash-consing. The second is that in our setting, we may have semantically equivalent subtrees which are not syntactically equivalent and thus will not be compacted by hash-consing. We set these considerations aside for now, and will revisit them in Section 7.

3.2 Set-theoretic operations

Implementing set-theoretic operations on BDTs is straightforward. For negation, it suffices to recursively traverse the BDT and negate the leaves:

Definition 3.6. The negation of a BDT can be computed by negating its leaves:

$$\neg\mathsf{L}(l) \quad = \quad \mathsf{L}(\neg l) \qquad\qquad \neg\mathsf{N}(a?B^+:B^-) \quad = \quad \mathsf{N}(a?\neg B^+:\neg B^-)$$

For binary set-theoretic operations, their definitions can be given in a generic way by propagating the operations down the leaves, following the $\preceq$ order over atoms.

Definition 3.7. Let $\circledast \in \{\wedge, \vee, \setminus\}$ be a binary set-theoretic operation. Let B_1 and B_2 be two BDTs. $B_1 \circledast B_2$ can be computed inductively as follows:

$$\begin{aligned}
\mathsf{L}(l_1) \circledast \mathsf{L}(l_2) &= \mathsf{L}(l_1 \circledast l_2) \\
\mathsf{N}(a?B^+ : B^-) \circledast \mathsf{L}(l) &= \mathsf{N}(a?B^+ \circledast \mathsf{L}(l) : B^- \circledast \mathsf{L}(l)) \\
\mathsf{L}(l) \circledast \mathsf{N}(a?B^+ : B^-) &= \mathsf{N}(a?\mathsf{L}(l) \circledast B^+ : \mathsf{L}(l) \circledast B^-)
\end{aligned}$$

$$\mathsf{N}(a_1?B_1^+ : B_1^-) \circledast \mathsf{N}(a_2?B_2^+ : B_2^-) = \begin{cases} \mathsf{N}(a_1?B_1^+ \circledast B_2^+ : B_1^- \circledast B_2^-) & \text{if } a_1 = a_2 \\ \mathsf{N}(a_1?B_1^+ \circledast \mathsf{N}(a_2?B_2^+ : B_2^-) : B_1^- \circledast \mathsf{N}(a_2?B_2^+ : B_2^-)) & \text{if } a_1 \preceq a_2 \\ \mathsf{N}(a_2?\mathsf{N}(a_1?B_1^+ : B_1^-) \circledast B_2^+ : \mathsf{N}(a_1?B_1^+ : B_1^-) \circledast B_2^-) & \text{if } a_2 \preceq a_1 \end{cases}$$

Finally, we define a map operation on BDTs (it will be used later to implement type substitutions).

Definition 3.8. Let $f : \mathcal{A} \cup \mathcal{L} \to \mathrm{BDT}(\mathcal{A}, \mathcal{L})$ be a function from atoms and leaves to BDTs. We define the extension $\bar{f} : \mathrm{BDT}(\mathcal{A}, \mathcal{L}) \to \mathrm{BDT}(\mathcal{A}, \mathcal{L})$ of f inductively, as follows:

$$\bar{f}(\mathsf{L}(l)) = f(l) \qquad \bar{f}(\mathsf{N}(a?B^+ : B^-)) = (f(a) \wedge \bar{f}(B^+)) \vee (\neg f(a) \wedge \bar{f}(B^-))$$

3.3 Semantic simplification

Unlike BDDs traditionally used for representing Boolean formulas whose atoms are propositional variables, our BDTs may have more complex atoms (e.g. arrows, products). Two atoms may be semantically equivalent without being syntactically equivalent, or more generally, they may be correlated by the subtyping relation. These subtyping relations between the atoms is not captured by the total order $\preceq$ we use for ordering the nodes of our BDTs, as this total order is purely syntactic. Hence, we define a simplification operation that removes nodes that are (semantically) redundant from a BDT. Note that this simplification operation does not try to reorder nodes or to change the atom they are labeled with. As such, it does not guarantee minimality of the resulting BDT, as a smaller semantically equivalent BDT that uses different atoms or a different order may exist.

We define our BDT semantic simplification operation inductively as follows:

$$\mathsf{simpl}(t, \mathsf{L}(l)) = \mathsf{L}(l) \qquad \mathsf{simpl}(t, \mathsf{N}(a?B^+ : B^-)) = \begin{cases} B^{+\prime} & \text{if } t \wedge [\![B^{+\prime}]\!] \simeq t \wedge [\![B']\!] \\ B^{-\prime} & \text{if } t \wedge [\![B^{-\prime}]\!] \simeq t \wedge [\![B']\!] \\ B' & \text{otherwise} \end{cases}$$

where $B^{+\prime} = \mathsf{simpl}(t \wedge a, B^+)$, $B^{-\prime} = \mathsf{simpl}(t \wedge \neg a, B^-)$, and $B' = \mathsf{N}(a?B^{+\prime} : B^{-\prime})$.
The simplified form $\mathsf{simpl}(B)$ of a BDT B is then defined as $\mathsf{simpl}(B) = \mathsf{simpl}(\mathbb{1}, B)$.

In essence, the type t in $\mathsf{simpl}(t, B)$ represents the context of B, that is, the conjunction of positive and negative atoms traversed from the root to B. For each node $B = \mathsf{N}(a?B^+ : B^-)$, if we note t its context, the simplification procedure tests whether $t \wedge [\![B]\!] \simeq t \wedge [\![B^+]\!]$. If that is the case, it means that the atom and right subtree are redundant and that B can be replaced by (the simplified version of) B^+. Otherwise, the same test is done for $t \wedge [\![B]\!] \simeq t \wedge [\![B^-]\!]$, which allows one to replace B by (the simplified version of) its right subtree. If both tests failed, the node is kept and its subtrees are recursively simplified.

Note that simplifying a BDT may be expensive as it requires deciding several semantic equivalences, each encoded as two subtyping tests. The simplification procedure is correct:

Proposition 3.9 (Correctness). *For any BDT B, we have $[\![\mathsf{simpl}(B)]\!] \simeq [\![B]\!]$.*

More interestingly, while the simplification procedure does not ensure minimality, it enjoys two desirable properties. First, the negation of a simplified type is itself simplified:

Proposition 3.10. *For any BDT B, we have $\neg\mathsf{simpl}(B) = \mathsf{simpl}(\neg B)$ (where $=$ is syntactic equality).*

Second, the simplification is sufficient to always reduce the empty type to the trivial BDT:

Proposition 3.11. *For any BDT B, we have $[\![B]\!] \simeq \mathbb{0} \Leftrightarrow \mathsf{simpl}(B) = \bot$.*

This guarantees that, if a BDT is simplified, then checking its semantic emptiness is equivalent to checking its syntactic emptiness. Both Propositions 3.10 and 3.11 will be exploited by our representation of set-theoretic types to improve performance (cf. Section 4.8).

4 Internal representation of types

In this section, we propose a structure to represent set-theoretic types using BDTs (cf. Section 3). This structure allows building types using different operations:

- type constructors and set-theoretic connectives ($\wedge$, $\vee$, $\neg$),
- application of a type substitution $\{\alpha_i \rightsquigarrow t_i\}_{i \in I}$ on a type t,
- construction of recursive types from a set of equations $\{\alpha_i = t_i\}_{i \in I}$.

For simplicity, we only consider the constant, arrow, and product type constructors. Our structure is modular, so adding other type constructors is straightforward and will be discussed in Section 6.

4.1 Overall structure

Set-theoretic types are defined coinductively and, thus, may be infinite trees. However, the regularity property ensures that the number of distinct subtrees is finite: we can thus represent a type as a graph that may contain cycles. We build this graph such that each subtree whose parent is a type constructor ($\times$ or $\rightarrow$) is represented by a node. A node thus defines the top-level structure of a subtree (i.e. the Boolean combination of constructors and type variables composing it), and references other nodes in the graph to describe the subtrees appearing inside a type constructor. The contractivity property of types ensures no such node definition consists in an infinite union or intersection. This graph structure is illustrated in Figure 2.

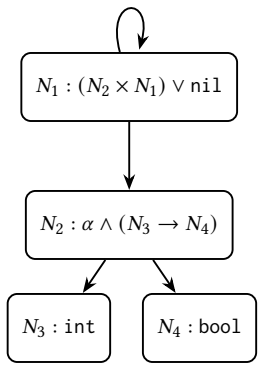

Fig. 2. Graph for the type $\mu X.\ ((\alpha \wedge (\mathtt{int} \rightarrow \mathtt{bool})) \times X) \vee \mathtt{nil}$

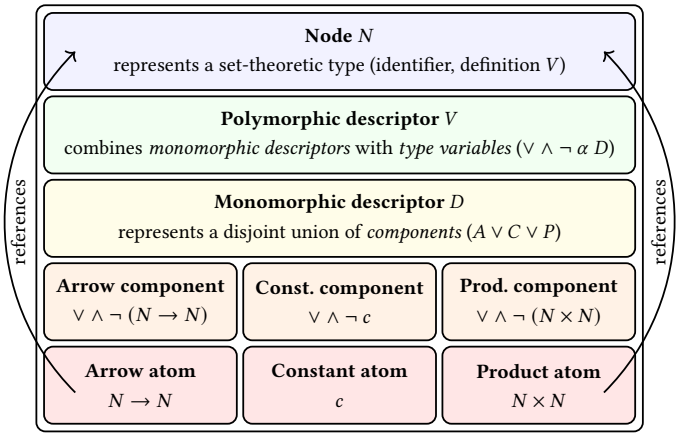

Fig. 3. Structure of a node (lower layers are used by upper layers)

Figure 3 illustrates the internal structure of each node of the graph. Each node can be decomposed into several layers. At the bottom, we have the different kinds of *atoms* (i.e. type constructors): products $\times$, arrows $\rightarrow$, and constants. Atoms of each kind are used inside *components* that represent a Boolean combination of atoms. Those different kinds of components are all disjoint (products, arrows and constants have disjoint interpretations): their disjoint union is represented by a *monomorphic*

descriptor. Finally, a *polymorphic descriptor* expresses a Boolean combination of type variables and monomorphic descriptors.

In the following, nodes are ranged over by N. We assume we have a total order $\preceq$ on nodes (it can be any total order, unrelated to the subtyping relation over types).

Definition 4.1. A *node substitution* ρ is a function from nodes to nodes which is the identity everywhere except for a finite set of nodes, called its domain and denoted by $\mathsf{dom}(\rho)$.

4.2 Atoms

An atom a represents an instance of a type constructor. For an atom a of any kind (constant, product, arrow), we define $\mathsf{deps}(a)$ and the application $a\rho$ of a node substitution ρ to a.

Arrow. An arrow atom is a pair (N_1, N_2), written $N_1 \to N_2$ for clarity. Intuitively, it represents the type $t_1 \to t_2$ where t_1 (resp. t_2) is the type associated with N_1 (resp. N_2).

$$\mathsf{deps}(N_1 \to N_2) \quad = \quad \{N_1, N_2\} \qquad\qquad (N_1 \to N_2)\rho \quad = \quad \rho(N_1) \to \rho(N_2)$$

Product. A product atom is a pair (N_1, N_2), written $N_1 \times N_2$ for clarity. Intuitively, it represents the type $t_1 \times t_2$ where t_1 (resp. t_2) is the type associated with N_1 (resp. N_2).

$$\mathsf{deps}(N_1 \times N_2) \quad = \quad \{N_1, N_2\} \qquad\qquad (N_1 \times N_2)\rho \quad = \quad \rho(N_1) \times \rho(N_2)$$

Constant. A constant atom is a label $c \in C$ (e.g. `true`, `false`, `nil`).

$$\mathsf{deps}(c) \quad = \quad \varnothing \qquad\qquad c\rho \quad = \quad c$$

We also extend the total order $\preceq$ over atoms (we can use the lexicographic order for arrows and products, and an arbitrary order over constants).

4.3 Components

A component represents a Boolean combination of atoms of a given kind. For simplicity, all kinds of components will be represented using BDTs. In practice, though, simple components can be represented using a more efficient, tailored representation (e.g. finite/cofinite sets for constants).

Definition 4.2. The BDTs we use for components use *Boolean leaves*, defined as follows:

$$\textbf{Boolean Leaves} \quad b \quad ::= \quad \bot \mid \top$$

We define the interpretation of a Boolean leaf as follows: $[\![\bot]\!] = \mathbb{0}$, $[\![\top]\!] = \mathbb{1}$. Set-theoretic operations ($\wedge$, $\vee$, $\neg$) on Boolean leaves are defined accordingly.

Definition 4.3. A component C is a BDT whose leaves are Boolean leaves, and whose atoms are one kind of atoms defined above (arrows, products, constants).

Components inherit the operations defined on BDTs (cf. Section 3): $\wedge$, $\vee$, $\neg$, $\mathsf{atoms}(C)$, $[\![C]\!]$, $\mathsf{dnf}(C)$. For each component C and node substitution ρ, we define $\mathsf{deps}(C)$ and $C\rho$ as follows:

$$\mathsf{deps}(C) \quad = \quad \bigcup_{a \in \mathsf{atoms}(C)} \mathsf{deps}(a) \qquad\qquad C\rho \quad = \quad \bar{f}(C)$$

where $\bar{f}$ is the extension on BDTs (cf. Definition 3.8) of the following function f:

$$f(b) \quad = \quad \mathtt{L}(b) \qquad\qquad f(a) \quad = \quad \mathtt{N}(a\rho)$$

We also extend the total order $\preceq$ over components (we can choose any syntactic total order over the structure of the underlying BDT).

4.4 Monomorphic descriptors

A monomorphic descriptor D represents a disjoint union of components.

Definition 4.4. A *monomorphic descriptor* is a record $\{\mathtt{const}; \mathtt{prod}; \mathtt{arrow}\}$, where `const` is a component for constants, `prod` is a component for products, and `arrow` is a component for arrows.

As these components have disjoint interpretations, all the set-theoretic operations ($\neg$, $\vee$, $\wedge$) as well as node substitutions can be performed component-wise. The set of dependencies $\mathsf{deps}(D)$ of a descriptor D is the union of the dependencies of its components. We extend the total order $\preceq$ over monomorphic descriptors (we can use the lexicographic order over the different components).

4.5 Polymorphic descriptors

Definition 4.5. A polymorphic descriptor V is a BDT whose leaves are monomorphic descriptors, and whose atoms are type variables (we fix an arbitrary total order $\preceq$ over type variables).

Polymorphic descriptors inherit the operations defined on BDTs (cf. Section 3): $\wedge$, $\vee$, $\neg$, $\mathsf{atoms}(V)$, $\mathsf{leaves}(V)$, $[\![V]\!]$, $\mathsf{dnf}(C)$. We define $\mathsf{deps}(V)$ and $V\rho$ (where ρ is a node substitution) as follows:

$$\mathsf{deps}(V) \quad = \quad \bigcup_{D \in \mathsf{leaves}(V)} \mathsf{deps}(D) \qquad\qquad V\rho \quad = \quad \bar{f}(V)$$

where $\bar{f}$ is the extension on BDTs (cf. Definition 3.8) of the following function f:

$$f(D) \quad = \quad \mathsf{L}(D\rho) \qquad\qquad f(\alpha) \quad = \quad \mathsf{N}(\alpha)$$

Definition 4.6. A *definition mapping* σ is a mapping from type variables to polymorphic descriptors. Its domain is denoted by $\mathsf{dom}(\sigma)$.

The application of a mapping σ on a polymorphic descriptor V is defined as $V\sigma = \bar{f}(V)$, where $\bar{f}$ is the extension on BDTs (cf. Definition 3.8) of the following function f:

$$f(D) \quad = \quad \mathsf{L}(D) \qquad\qquad f(\alpha) \quad = \quad \begin{cases} \sigma(\alpha) & \text{if } \alpha \in \mathsf{dom}(\sigma) \\ \mathsf{N}(\alpha) & \text{otherwise} \end{cases}$$

We extend the total order $\preceq$ over polymorphic descriptors (we can choose any syntactic total order over the structure of the underlying BDT).

4.6 Nodes

Definition 4.7. A node is represented by a record $\{\mathsf{id}; \mathsf{def}\}$ where id is a unique identifier (e.g. an integer), and def is a polymorphic descriptor.

The total order $\preceq$ on nodes only depends on their id field. We write $\mathsf{node}(d)$ to denote a node with the definition d and a fresh identifier. Set-theoretic operations on nodes are defined as follows:

$$\neg N = \mathsf{node}(\neg(N.\mathsf{def})) \quad N_1 \wedge N_2 = \mathsf{node}(N_1.\mathsf{def} \wedge N_2.\mathsf{def}) \quad N_1 \vee N_2 = \mathsf{node}(N_1.\mathsf{def} \vee N_2.\mathsf{def})$$

The set of top-level type variables of a node N is defined as $\mathsf{top\_vars}(N) = \mathsf{atoms}(N.\mathsf{def})$. The set of dependencies of a node N, $\mathsf{deps}(N)$, is the smallest set of nodes S such that $N \in S$, and $\forall N' \in S.\ \mathsf{deps}(N'.\mathsf{def}) \subseteq S$. The set of type variables of a node N is defined as $\mathsf{vars}(N) = \bigcup_{N' \in \mathsf{deps}(N)} \mathsf{top\_vars}(N')$.

Definition 4.8. A *type substitution* ϕ is a function from type variables to nodes which is the identity everywhere except for a finite set of type variables, denoted by $\mathsf{dom}(\phi)$.

The application $N\phi$ of a type substitution ϕ on a node N is the node $\rho(N)$ where:

- ρ is a node substitution $\{N_i \rightsquigarrow N_i'\}_{i \in I}$ where $\{N_i\}_{i \in I} = \mathsf{deps}(N)$, and
- for every $i \in I$, $N_i' = \mathsf{node}(((N_i.\mathsf{def})\rho)\sigma)$, and

- σ is the definition mapping $\{\alpha \rightsquigarrow \phi(\alpha).\mathsf{def}\}_{\alpha \in \mathsf{dom}(\phi)}$

Intuitively, the connected component of the node is copied (ρ associates each node to its copy), and the definition mapping σ is applied to the definition each copied node.

Note that this definition is recursive: the definition of the node N'_i depends on ρ, which itself involves N'_i. In SSTT, the field $N.\mathsf{def}$ is mutable, making it possible to generate a fresh node and only set its definition later (it also allows us to simplify the definition of a node after it is created, as it will be discussed in Section 4.8). Alternatively, these equations can be implemented by storing node definitions in a separate data structure (for instance a dictionary mapping node identifiers to their definitions), or in a lazy language (e.g. Haskell) using recursive definitions.

4.7 Construction of recursive types

Nodes can be constructed from atoms and combined using set-theoretic operations. However, we do not have a way to construct recursive types yet. We propose here a method for building a recursive type from a set of equations.

Let N be a node and E be a set of equations, each equation being a pair (α, N_α) corresponding to the equation $\alpha = N_\alpha$. Let $S = \bigcup_{(\alpha,N_\alpha)\in E} \mathsf{deps}(N_\alpha)$. Using a topological sort algorithm, we order S into an ordered set $\{N_i\}_{i\in 1..n}$ such that $\forall i \in 1..n.\ \forall \alpha \in \mathsf{top\_vars}(N_i).\ \forall j \in i..n.\ (\alpha, N_j) \notin E$.

In other words, our order must guarantee that for every binding (α, N_α) of our set of equations E, N_α is smaller than all the nodes that depend on α at top-level. If no such ordering exist, then it means that the set of equations is not contractive (it contains dependency cycles that do not pass under a type constructor): there may be no solution or infinitely many solutions that satisfy the set of equations, and thus we reject it.

Otherwise, for every $i \in 1..n$, we define a node $N'_i = \mathsf{node}(((N_i.\mathsf{def})\rho)\sigma_i)$, where:

- ρ is the node substitution $\{N_j \rightsquigarrow N'_j\}_{j\in 1..n}$, and
- σ_i is the definition mapping $\{\alpha \rightsquigarrow N'_j.\mathsf{def} \mid j \in 1..(i-1), \alpha \in \mathcal{V} \text{ s.t. } (\alpha, N_j) \in E\}$

As for type substitutions, this definition is recursive and should be implemented using similar techniques. The function $\mathsf{build}(N, E)$ returns a node $N\phi$ where $\phi = \{\alpha \rightsquigarrow N'_i \mid (\alpha, N_i) \in E\}$. If E is not contractive, $\mathsf{build}(N, E)$ is not defined.

4.8 Node simplification

Depending on how they are constructed, our nodes may contain redundant atoms in their internal representation. For instance, consider the following sequence of operations: (*i*) the type `bool` $\to$ `bool` is constructed, (*ii*) it is intersected with `true` $\to$ `false`, and (*iii*) it is intersected with `false` $\to$ `true`. The type we obtain is semantically equivalent to (`true` $\to$ `false`) $\wedge$ (`false` $\to$ `true`), but its internal representation will still contain the atom `bool` $\to$ `bool`. Redundancy can also accumulate after applying multiple successive substitutions on a type, which typically happens when implementing a type inference algorithm based on tallying such as [11].

The BDT simplification procedure simpl(.) defined in Section 3.3 can be used to avoid the accumulation of redundant atoms. In SSTT, we implemented a systematic simplification of types: a node is automatically simplified after being constructed by a $\wedge$, $\vee$, a substitution, or a build operation. According to Proposition 3.10, a type does not need to be simplified again after a $\neg$ operation. Keeping node definitions in a simplified form allows us to decide emptiness just by checking if the definition of the node is syntactically the empty BDT (cf. Proposition 3.11). This means that, though the subtyping algorithm (Section 5) is used to simplify the definition of a node initially, subsequent emptiness checks are constant time. The time overhead induced by a systematic simplification of types is measured and discussed in Section 7.

5 Type operations

In the rest of this paper, we assimilate a type t with a node N that represents it. Set-theoretic operations and substitutions on types correspond to the associated operations on nodes.

5.1 Subtyping

As already explained, the subtyping algorithm is really an emptiness test. Following [16], to test a type t for emptiness one only has to:

- Compute $\mathsf{DNF}(t)$ as in equation (*)
- Disregard the top-level variables $\alpha_i^{\{b,p,a\}}$ and $\beta_j^{\{b,p,a\}}$
- Test that all combinations of basic types, product types and arrow types are empty

The reason for the second condition is that, since the type must be empty *for all* possible substitutions, in particular for those mapping positive variables to $\mathbb{1}$ and negative variables to $\mathbb{0}$. We detail a first version of the emptiness test in Algorithm 1.

Algorithm 1 (Subtyping, non-optimized).

```
1 let rec is_empty N Σ =
2   if N ∈ Σ then true
3   else
4     let Σ' = Σ ∪ {N} in
5     let V = N.def in
6     ∀D ∈ leaves(V),
7       is_empty_const D.const Σ' ∧
8       is_empty_prod D.prod Σ' ∧
9       is_empty_arrow D.arrow Σ'
```

In this code[2], N is the node of the type we are testing and Σ is a set of nodes. Each layer (in the sense of Figure 3) handles a particular aspect of the emptiness test. Recursive types are handled at the level of nodes, by remembering which node has already been visited. If a node is encountered during its own traversal, we return that it is empty, since a recursive type whose emptiness depends only on itself, e.g. $\mu X.(\mathbb{1}, X)$[3], is empty. The first time a node is visited, it is recorded (l. 4), and its polymorphic descriptor V (a BDT whose atoms are type variables) is inspected. At that point, it is sufficient to inspect all the monomorphic descriptors at the leaves of the BDT. Each such descriptor D is a disjoint union of BDTs of constants, products, and arrows, for which the emptiness test may call `is_empty` again on nested nodes.

Constant component. Since constants are disjoint one from the other, testing the emptiness of a BDT of constants is done in constant time, by checking if it is structurally equal to the leaf $\bot$.

Product component. Testing the emptiness of products is done by the `is_empty_prod` function:

$$\texttt{let is\_empty\_prod } p\ \Sigma = \forall (\{P_1^i \times P_2^i\}_{i\in I}, A_n, \top) \in \mathsf{dnf}(p).\ \Psi_{\mathsf{prod}}(\textstyle\bigwedge_{i\in I} P_1^i \times \bigwedge_{i\in I} P_2^i, A_n, \Sigma)$$

This function enumerates each conjunction of the DNF of p whose leaf node is $\top$. For each one, the positive part of the intersection is transformed into a single product, by pushing the intersection below the product constructor. It then remains to test whether the negative part of the intersection is enough to negate this single product, which is tested by the Ψ_{prod} function:

$$\begin{aligned}
\Psi_{\mathsf{prod}}(P_1 \times P_2, \varnothing, \Sigma) &= (\texttt{is_empty } P_1\ \Sigma) \vee (\texttt{is_empty } P_2\ \Sigma)\\
\Psi_{\mathsf{prod}}(P_1 \times P_2, \{N_1 \times N_2\} \cup S, \Sigma) &= (\texttt{is_empty } P_1\ \Sigma) \vee (\texttt{is_empty } P_2\ \Sigma) \vee\\
&\quad (\ \Psi_{\mathsf{prod}}((P_1 \setminus N_1) \times P_2, S, \Sigma) \wedge\ \Psi_{\mathsf{prod}}(P_1 \times (P_2 \setminus N_2), S, \Sigma))
\end{aligned}$$

Note that in the worst case, the Ψ_{prod} function may perform an exponential number of calls to `is_empty`, each Boolean connective introducing a potential backtracking point. Crucially, even

[2] We use an OCaml like syntax to present algorithm.
[3] The model of semantic subtyping only allows for finite values.

though new types are generated (e.g. when pushing the intersection below the products or when computing type differences), only a finite number of such new types are created (see [18], [20]) which ensures the termination of the algorithm (the number of all nodes collected in Σ is finite).

Arrow component. Testing the emptiness of arrows works similarly to the case of products. The only difference is that intersection and arrow constructors do not commute, and that a positive intersection of arrows is never empty. We can apply these principles to test emptiness as follows.

$$\texttt{let is\_empty\_arrow } a\ \Sigma = \forall\,(A_p, A_n, \top) \in \mathsf{dnf}(a).\ \exists\,(N_1 \to N_2) \in A_n.\ \Psi_{\text{arrow}}(N_1, \neg N_2, A_p, \Sigma)$$

For an intersection of positive arrows A_p and negative arrows A_n, there must be a single negative arrow $N_1 \to N_2$, which negates the whole positive intersection. For that to be true, it suffices that at least one arrow in A_p is completely negated by the selected negative arrow (which makes the positive intersection empty). This is done by the auxiliary Ψ_{arrow} function defined as:

$$\begin{aligned}
\Psi_{\text{arrow}}(N_1, T_2, \varnothing, \Sigma) &= (\texttt{is_empty}\ N_1\ \Sigma) \vee (\texttt{is_empty}\ T_2\ \Sigma)\\
\Psi_{\text{arrow}}(N_1, T_2, \{P_1 \to P_2\} \cup A_p, \Sigma) &= (\texttt{is_empty}\ N_1\ \Sigma) \vee (\texttt{is_empty}\ T_2\ \Sigma) \vee\\
&\quad (\Psi_{\text{arrow}}((N_1 \setminus P_1), T_2, A_p, \Sigma) \wedge \Psi_{\text{arrow}}(N_1, (P_2 \wedge T_2), A_p, \Sigma))
\end{aligned}$$

This function is similar to the one used for products, but takes a single negative arrow $N_1 \to N_2$ and check that it is a super type of the positive part (and thus, that their difference is empty). Notice that T_2 is initially the negation of the co-domain N_2.

Recursive types and caching. To explain how recursive types are handled, consider the example:

$$X = (Y \times Y) \qquad Y = (\texttt{nil} \times Y) \vee (Z \times Z) \vee (\texttt{nil} \to \texttt{nil}) \qquad Z = Y \times \texttt{nil}$$

where types are written as a set of recursive equations between nodes (and some basic types). If one uses the algorithm of Figure 1 to test the emptiness of X, the result is `false`. A relevant subset of the recursive calls is the following:

```
1  is_empty X ∅
2       is_empty_prod Y × Y {X}
3           is_empty Y {X}
4              is_empty_prod (nil × Y) ∨ (Z × Z) {X, Y}
5                  (is_empty_const nil {X, Y} ⇝ false
6                  ∨ is_empty Y {X, Y} ⇝ true) (*recursive occurrence*)
7                ∧ is_empty_pair Z × Z {X, Y, Z} ⇝ true (*depends on Y*)
8            ∧ is_empty_arrow (nil → nil) {X, Y} ⇝ false
9         ∨ is_empty Y {X}
```

To test the emptiness of X, we must test its definition, since X is not in Σ (l. 2). Its definition is the product type $Y \times Y$, which is empty if either projection is empty. Those are tested at l. 3 and l. 9. This illustrates an inefficiency of the algorithm. The set Σ is used to handle recursion, and not as a cache, which means that several occurrences of the same node may be traversed several times.

While it is tempting to replace sigma with a mutable map, one must take special care in doing so. Indeed, in the above example, it would be recorded that:

- at first, Y is empty (initial recursive call, l. 3)
- during its traversal, Z (which depends on Y) is found to be empty
- ultimately, Y contains $\texttt{nil} \to \texttt{nil}$ and therefore is *not* empty

It is not sufficient here to update Σ to record that Y is non-empty. One must also *invalidate* the results stored for all nodes which used on the *wrong* initial assumption that Y was empty. This improvement is described in Algorithm 2. The algorithm maintains three data-structures:

Algorithm 2 (Subtyping, optimized).

```
1  let memo = H.create ()
2  let stack = ref []
3  let mem_stack = H.create ()
4  let rec is_empty N =
5    if N ∈ memo then begin
6     if N ∈ mem_stack then
7      H.add mem_stack N
8      (!stack::H.find mem_stack N);
9     H.find memo N
10   end else begin
11    stack := N :: !stack;
12    H.add mem_stack N [];
13    H.add memo N true;
14    let V = N.def in
15    let b = ∀D ∈ leaves(V),
16      is_empty_const D.const ∧
17      is_empty_prod D.prod ∧
18      is_empty_arrow D.arrow in
19    if not b then
20     invalidate N memo mem_stack;
21    H.add memo N b;
22    H.remove mem_stack N;
23    stack := List.tl !stack;
24    b end
```

- the hash table `memo` (l. 1)
- a reference to a persistent list, used as a stack (`stack`, l. 2)
- a hash table mapping nodes to stacks (`mem_stack`, l. 3)

While `memo` plays the same purpose as in the previous algorithm, the other two data-structures are used to track dependencies between nodes. Assuming a node N is not in `memo`:

- push it on `stack` (l. 11)
- create an entry in `mem_stack` (l. 12)
- map it to `true` in `memo` as usual (l. 13)

We then explore the descriptor of the node recursively. If a node N is in `memo`, then:

- if an entry exists for N in `mem_stack`, record the current stack (l. 6–8)
- return the memoized value for N (l. 9)

Lastly, after returning from a recursive call, if the node turns out to be non-empty then:

- for each recorded stack for N, remove from `memo` any node X appearing on that stack, until N is reached (done by function `invalidate`, l. 20)
- update `memo` with the result for N (l. 21)
- remove N from `mem_stack` (l. 22)
- pop N from the stack (l. 23)
- return the result (l. 24)

Whenever we encounter a node N a second time during a recursive traversal, either:

- it is still in an "unknown" state (i.e. it has a binding in `mem_stack`), and every node that has been put on the stack during its inspection depends on the (possibly wrong) assumption that it is empty, we must remember them to remove them from `memo` later on;
- or it is in a definitive state (it has no binding in `mem_stack`), we can therefore return its emptiness state from the `memo` table

5.2 Tallying

We revisit the tallying algorithm formalized by [13] which does not perform well in practice. At its heart, the tallying algorithm recursively traverses a type t and generates a set of sets of constraints on the variables of t which, when satisfied, make the type empty. For instance, for the "`map even`" example given in the introduction, the tallying algorithm produces a set of four sets of constraints, each yielding a type substitution. We first introduce *normalized constraint sets*.

Definition 5.1 (Normalized constraint set). A normalized constraint set C is a set of triples $\{(s_i, \alpha_i, t_i)\}_{i\in I}$ where all α_i are distinct. The set $\{\alpha_i\}_{i\in I}$ is called the *domain* of C and is written $\mathrm{dom}(C)$. The empty constraint set is noted $\top$. Lastly, we define the constraint associated with a variable α in a constraint C by:

$$C(\alpha) = (s,t) \text{ if } \alpha \in \mathrm{dom}(C) \qquad C(\alpha) = (\mathbb{0},\mathbb{1}) \text{ otherwise}$$

We now define operations on normalized constraint sets. These are parametrized by a set Δ of type variables the tallying algorithm is allowed to instantiate.

Definition 5.2 (Intersection of normalized constraint sets). Let C_1 and C_2 be two normalized constraint sets and Δ a set of type variables. We define the *intersection* $C_1 \sqcap^\Delta C_2$ as follows. Let $C = \{(s_1 \vee s_2, \alpha, t_1 \wedge t_2) \mid , (s_i, \alpha, t_i) \in C_i, \text{ for } i = 1..2\}$.

$$\begin{array}{lcll} C_1 \sqcap^\Delta C_2 & = & C & \text{if } \forall (s,\alpha,t) \in C, \mathsf{vars}(s) \cup \mathsf{vars}(t) \subseteq \Delta \Rightarrow s \leq t \\ C_1 \sqcap^\Delta C_2 & = & \bullet & \text{otherwise} \end{array}$$

Intuitively, the intersection of two constraint sets is only defined if it does not create a trivially unsatisfiable constraint such as $(\texttt{int}, \alpha, \texttt{bool})$, which can never be satisfied.

Definition 5.3 (Subsumption of constraint sets). Let C_1 and C_2 be two normalized constraint sets. We say that C_1 subsumes C_2, written $C_1 \sqsubseteq C_2$, if and only if:

$$\forall (s_2, \alpha, t_2) \in C_2, \exists (s_1, \alpha, t_1) \in C_1, \text{ such that } s_2 \leq s_1 \text{ and } t_1 \leq t_2$$

In other words, a set of constraints C_1 subsumes a set of constraints C_2 if C_1 gives better bounds (higher lower bounds and lower upper bounds) for all variables of C_2. We are now equipped to define sets of normalized constraint sets. Sets of normalized constraint sets are ranged over by $\mathscr{C}$.

Definition 5.4 (Simplification of sets of constraint sets). The simplification of a set $\mathscr{C}$ of constraint sets, written $\mathsf{csimp}(\mathscr{C})$, is defined as follows:

$$\begin{array}{lcll} \mathsf{csimp}(\mathscr{C}) & = & \mathsf{csimp}'(\mathscr{C}, \varnothing) & \\ \mathsf{csimp}'(\varnothing, \mathscr{D}) & = & \mathscr{D} & \\ \mathsf{csimp}'(\{C\} \cup \mathscr{C}, \mathscr{D}) & = & \mathsf{csimp}'(\mathscr{C}, \mathscr{D}) & \text{if } \exists C' \in \mathscr{D}, C' \sqsubseteq C \\ \mathsf{csimp}'(\{C\} \cup \mathscr{C}, \mathscr{D}) & = & \mathsf{csimp}'(\mathscr{C}, \{C\} \cup \mathscr{D} \setminus \mathscr{D}') & \text{where } \mathscr{D}' = \{C' \mid C' \in \mathscr{D}, C \sqsubseteq C'\} \end{array}$$

Simplified sets of constraint sets ensures that their elements are pairwise non-subsumable, each element denotes a different, incomparable substitution. Maintaining simplified sets is a key ingredient to tame the complexity of the tallying operation.

Definition 5.5 (Union and intersection of sets of constraint sets). Given two sets of normalized constraint sets $\mathscr{C}_1$ and $\mathscr{C}_2$ and a set of type variables Δ, we define their intersection and union as:

$$\begin{array}{lcll} \mathscr{C}_1 \sqcap^\Delta \mathscr{C}_2 & = & \mathsf{csimp}(\{C_1 \sqcap^\Delta C_2 \mid C_1 \in \mathscr{C}_1, C_2 \in \mathscr{C}_2, C_1 \sqcap_\Delta C_2 \neq \bullet\}) & \text{(intersection)} \\ \mathscr{C}_2 \sqcup \mathscr{C}_2 & = & \mathsf{csimp}(\mathscr{C}_1 \cup \mathscr{C}_2) & \text{(union)} \end{array}$$

Algorithm 3 (Tallying).

```
1 let tally Δ S =
2   let 𝒞 = ⨅^Δ_{(N1,N2)∈S} normalize Δ (N1\N2) in
3   let 𝒞' = ⨆_{C∈𝒞} propagate Δ ⊤ C ∅ in
4   { solve C  | C ∈ 𝒞'}
```

We can now review the tallying algorithm, Algorithm 3, which consists of three steps. The function `normalize` transforms a subtyping constraint $N_1 \leq N_2$ into a set of normalized constraint sets $\mathscr{C}$. Like in the subtyping algorithm, the initial subtyping constraint is encoded as an emptiness constraint for the type $N_1 \setminus N_2$. Each normalized constraint set is then processed through the `propagate` function, which eliminates unsatisfiable constraint sets and recursively enriches satisfiable ones with new induced constraints: for every constraint (s, α, t) produced by `normalize`, if either s or t contains type variables that are not in Δ, then $s \leq t$ may require new constraints. These new constraints are added by `propagate` to

form a final set of constraints set $\mathscr{C}'$. Each constraint set in $\mathscr{C}'$ can then be turned into a set of recursive type equation and solved, yielding a substitution. The code of `normalize` and `propagate` is given in Figure 4 (the code of `solve` is straightforward).

Normalize. The implementation of `normalize` Δ N follows the same structure as the emptiness test. It uses a hash table M to map already visited nodes to their set of constraint sets (l. 3). It plays a similar role as Σ in the subtyping algorithm. Whenever the emptiness test (which computes a Boolean) uses an "and" (resp. an "or"), the normalize function uses $\sqcap$ (resp. $\sqcup$). The first time a type is visited (l. 5), the satisfiable constraint is added to the memoization table M. Each conjunction of the DNF of the type is then explored to generate individual sets of constraint sets which are intersected (l. 6). The node is removed from the table M and the resulting constraints returned.

The emptiness of a conjunction of the form $\bigwedge_{\alpha\in A_p} \wedge \bigwedge_{\alpha\in A_n} \neg\alpha \wedge D$ is expressed by the function `normalize_summand`. The function chooses the smallest (according to the total order $\preceq$ on type variables) top-level variable that is not in Δ, and extract it from the summand, yielding a single constraint on that variable denoting the emptiness of the summand (l. 10–12). If there are no top-level variable, the function recursively explores the underlying constructors, with a function that mimics the emptiness test on components (and recursively calls `normalize`).

```
1  let M = H.create ()
2  let rec normalize Δ N =
3    if N ∈ M then H.find M N
4    else begin
5      H.add M N {⊤};
6      let 𝒞 = ⨅^Δ_{(A_p,A_n,D)∈dnf(N.def)} normalize_summand Δ A_p A_n D in
7      H.remove M N; 𝒞 end
8  and normalize_summand Δ A_p A_n D =
9    let A = (A_p ∪ A_n) \ Δ in
10   if A ≠ ∅ then
11    let α = min A in (* Minimal type variable according to ≼ *)
12    if α ∈ A_p then {{(𝟘, α, ¬(A_p \ {α} ∧ A_n ∧ D))}} else {{(A_p ∧ A_n \ {α} ∧ D, α, 𝟙)}}
13   else (norm_const Δ D.const) ⊓^Δ (norm_prod Δ D.prod) ⊓^Δ ...
14 let rec propagate Δ C_0 C_1 Σ =
15   if C_1 = ⊤ then {C_0} else
16     let (N_1, α, N_2) :: C'_1 = C_1 in
17     let N = N_1 \ N_2 in
18     if N ∈ Σ then propagate Δ (C_0 ⊓^Δ {(N_1, α, N_2)}) C'_1 Σ
19     else
20       let 𝒞 = {C_0 ⊓ C_1} ⊓^Δ (normalize Δ N) in
21       ⨆_{C∈𝒞} propagate Δ ⊤ C (Σ ∪ {N})
```

Fig. 4. The `normalize` and `propagate` functions

Propagate. The function `propagate` $\Delta\ C_0\ C_1\ M\ \Sigma$ iteratively enriches the constraint set $C_0 \sqcap C_1$ with the constraints induced by C_1 (C_0 contains the constraints already treated) until it reaches a fixpoint (Σ stores the induced constraints already visited). Note the interplay between `normalize` and `propagate`. When `normalize` generates a single constraint for a top-level variable, `propagate` calls `normalize` again on the bounds of that constraint.

Solve. The function `solve` C represents the constraint set C as a principal type substitution. It makes use of the `build` function (Section 4) to solve a set of equations. Each equation is generated by turning a subtyping constraint $N_1 \leq \alpha \leq N_2$ into an equation $\alpha = (N_1 \vee \alpha') \wedge N_2$, for a fresh α'.

6 Extensions

In Section 4 and Section 5, we described three core components: arrows, products, and constants. These components are the most commonly used in set-theoretic type formalizations, but a practical implementation of set-theoretic types also need other core components to model common programming structures (Section 6.1). Additionally, these core components can be used as building blocks and combined to define new components via encodings (Section 6.2).

6.1 Core extensions

Intervals. The component for constants cannot be used to represent all the individual integers, as the type `int` would then need to be defined as an infinite (or very large) union. Therefore, SSTT features a component representing intervals as sorted lists of disjoint intervals.

Tuples. Products can be generalized to cover tuples of any arity. For that, we can implement a generic tuple component parametrized by an arity n. It is straightforward to extend the subtyping and tallying algorithms accordingly (e.g. for $n = 3$, a DNF of tuples (t_1, t_2, t_3) is empty if and only if the corresponding DNF of pairs $t_1 \times (t_2 \times t_3)$ is empty).

Records. The theory of set-theoretic types features records that can be open or closed, and fields that may be optionally absent. Record atoms are simply maps from field names to optional types, as well as a Boolean indicating if the record is closed (its list of fields is exact) or opened (it may have other, unknown fields). In order to represent absent fields, we define optional types as the pairs (t, a) where a is either $\top$ (the value may be absent) or $\bot$ (the value is not absent). An optional type (t, a) is empty if and only if t is empty and $a = \bot$. Then, subtyping for records can be encoded as subtyping for tuples: the record atoms composing a DNF can be turned into tuple atoms of optional types by fixing an order over fields. For instance, the DNF $\{\ell_1 : t_1\} \vee \{\ell_2 :? t_2 \ ..\}$ can be encoded as $((t_1, \bot), (\mathbb{0}, \top), (\mathbb{0}, \top)) \vee ((\mathbb{1}, \top), (t_2, \top), (\mathbb{1}, \top))$ (the first component of the tuples represents the field ℓ_1, the second component represents ℓ_2, and the last component represents the other fields). Note that while our record can be opened (extensible via sub-typing) they do not feature row polymorphism, which requires substantial modifications of the tallying procedure [15].

Tagged types. A tagged type $T(t)$ intuitively represents the values $\{T(v) \mid v \in t\}$, where $T \in \mathtt{Tags}$ is a unary constructor of our language disjoint from other values (in particular, two types $T_1(t)$ and $T_2(t)$ with $T_1 \neq T_2$ are disjoint). Tagged types have several uses: (i) they can be used to represent boxed values, and in conjunction with unions, to implement algebraic data types where every constructor is disjoint from the others, (ii) they can be used as a building block to extend our type algebra with new type constructors in a modular way using encodings (cf. Section 6.2), and (iii) in addition to the representation of boxed values, they can be generalized to represent opaque data types with an invariant or covariant parameter, as described in [2]. Extending the subtyping and tallying algorithms is straightforward, following the inductive relations given in this last work.

6.2 Encodings

In addition to core extensions, new type constructors can be added by encoding them as combinations of core components. This approach has two advantages: (*i*) it allows the programmer to extend the type algebra in a modular way, without having to modify the core components of the library, and (*ii*) it does not require any addition to the subtyping and tallying algorithms. Designing such an extension requires to answer these three questions:

(1) how to encode our new type constructors using the existing core components?
(2) how to recognize the encoding in a type?
(3) how to extract the parameters of our type constructors from their encoding?

As long as we are only interested in subtyping and tallying, only the first point matters. However, as soon as we want to be able to inspect our types (e.g. for pretty-printing, cf. Appendix B), it becomes necessary to recognize when a subtree of our type comes from an encoding, and how to destruct this encoding. In order to make our encodings easily recognizable in a type, each extension should be disjoint from the other values. For instance, encoding the list type $[\alpha*]$ as the type $X = \texttt{nil} \vee \alpha \times X$ as we did in Section 1 is ambiguous, as it becomes unclear whether $\texttt{int} \times \texttt{nil}$ should be printed as a pair or as a list. We solve this problem by using tagged type: each constructor of an extension is tagged by a distinct tag that ensures its disjointness w.r.t other types (e.g. list and pairs are kept apart) and can be used by the pretty-printer to defer printing to a specialized function, provided by the extension.

Extension	Constructor	Encoding
Characters	Single character `'a'`	$\texttt{Chr}(i_a)$ where i_a is the ASCII encoding of `'a'`
	Char interval `'a'-'z'`	$\texttt{Chr}(i_a \,..\, i_z)$
	Any (supertype)	`Chr(int)`
Strings	Single string `"abc"`	$\texttt{Str}(c_{\text{abc}})$ where c_{abc} is a constant representing `"abc"`
	Any (supertype)	`Str(cst)` where `cst` is the supertype of constants
Lists (regexp)	Empty list `[]`	$\texttt{Lst}(c_{\text{nil}})$ where c_{nil} is a constant representing `[]`
	Any (supertype)	$\mu X.\, \texttt{Lst}(\mathbb{1} \times X) \vee \texttt{Lst}(c_{\text{nil}})$
	t followed by l	$\texttt{Lst}(t \times l)$
	$t*$ followed by l	$\mu X.\, \texttt{Lst}(t \times X) \vee l$

Fig. 5. Example of extensions and their encodings

As an example, Figure 5 proposes some extensions and the corresponding encodings. The list encoding allows us to support regular expression types purely as an extension. Although they are isomorphic to recursive products, we can provide a printing function which decompiles the type into a regular expression using using Brzozowski Algebraic Method [4]. Other extensions have also been implemented through encodings, such as Booleans, floating-point numeric types, and opaque data types supporting multiple parameters of different variances.

Gradual types. An important contribution of [23] is that gradual types (that may contain a dynamic component « ? ») can be fully represented by a pair of static types (its lower and upper materializations). A type system implementor can simply represent types as pair as SSTT types without any modification. This is the approach taken by Elixir [8].

7 Evaluation

In this section, we evaluate the performance of our implementation (submitted as an anonymized artifact). in particular the semantic simplification of BDTs, and the subtyping and tallying algorithms.

SSTT is implemented in OCaml. The library (excluding tests) amounts to 6900 loc, of which 3700 are the core type algebra and subtyping (Sections 3, 4 and 5.1). The code for the tallying is 470 loc, the rest being extensions (Section 6) and the pretty-printer (Appendix B). We measured the performance[4] of SSTT for two categories of operations: type construction ($\wedge$, $\vee$, $\neg$, and the function build), and constraint solving (via the tallying algorithm, which itself uses subtyping). We generated numerous sets of constraints by running a type system implementation [2] (an improved version of [11, 12]) on a corpus, and exporting the tallying instances it solves as a JSON file. This has been done for three corpuses (some examples are given in Appendix C):

A. Hindley-Milner (150 loc, 30 functions, $\approx$ 2000 tallying instances) Generic functions in a Hindley-Milner style. Most functions are recursive and involve pattern-matching over list or balanced binary tree types, and their type is inferred. The tallying instances generated involve multiple type variables (due to parametric polymorphism and recursion), but few unions or intersections (unions are only used to encode ADTs with disjoint constructors).

B. Overloaded (220 loc, 68 functions, $\approx$ 8000 tallying instances) Non-recursive and non-generic overloaded functions. For each, a type capturing its overloaded behavior is inferred using type narrowing techniques. The most overloaded function is an intersection of 26 arrows. The tallying instances generated involve many intersections and unions but fewer type variables.

C. HM+Overloaded (345 loc, 83 functions, $\approx$ 3500 tallying instances) Recursive, generic and overloaded functions. This corpus involves parametric polymorphism, ad-hoc polymorphism, and encodings to model mutable data structures (references, arrays, etc.). The tallying instances generated involve unions, intersections, and multiple type variables.

Our first observation is that none of these corpuses can be type-checked in a reasonable time if types are not semantically simplified regularly as described in Section 4.8: inferred types (in particular arrows) get too large during type inference, to a point where some tallying instances do not terminate after a minute. Implementing hash-consing for all the components and layers of the node structure does not help, though it allows sharing about 50% of the nodes, thus reducing the size of some BDTs by making more atoms syntactically equivalent. Consequently, our benchmark files containing the tallying instances have been generated by running the type-checker implementation in a setting where types are systematically simplified.

Corpus	Measure	None	SS	HC	SS+HC	None*	SS*	CDuce
A. Hindley-Milner	Type building	0.04	0.12	0.10	0.37	0.04	0.16	N/A
	Solving	0.14	0.16	0.28	0.44	0.22	7.17	N/A
	Total	0.18	0.28	0.38	0.81	0.26	7.33	N/A
B. Overloaded	Type building	0.02	0.07	0.09	0.28	0.02	0.08	0.07
	Solving	0.06	0.08	0.25	0.40	0.06	0.10	0.18
	Total	0.08	0.15	0.34	0.68	0.08	0.18	0.25
C. HM+Overloaded	Type building	0.02	0.06	0.13	0.34	0.02	0.14	N/A
	Solving	0.10	0.11	0.24	0.44	0.96	1.14	N/A
	Total	0.12	0.17	0.37	0.77	0.98	1.29	N/A

Fig. 6. Time performance (in s) for different simplification and optimization settings

Figure 6 shows the time performance for building the types and solving the constraints for each of these benchmark files, under different settings: *(SS)* systematic semantic simplification of BDTs,

[4]All experiments were done on a Windows 11 laptop running Ubuntu 24 on WSL 2, Intel i5-12500H CPU, 32 GB ram.

(HC) hash-consing, *(SS+HC)* combination of the two, and *(None)* none of the two. An asterisk *(*)* is added when the non-optimized version of subtyping (Algorithm 1) is used instead of the hash-based one (Algorithm 2). All configurations implement the tallying optimizations described in Section 5.2 (without it, none of the benchmarks finish after a minute).

This table clearly outlines that semantic simplification is essential. Even for instances that can be handled by all strategies (and recall that only the SS strategy was able to generate those instances) its overhead in type construction is smaller than for hash-consing, while its performance for constraint solving is close to the *None* strategy. Hash-consing on the other hand negatively impacts performance of constraint solving. Although it allows more sharing of nodes, all tests used less than 250 MB of peak ram usage, so memory consumption is not an issue.

The positive impact of having an optimized cache for subtyping is also visible. For benchmarks with complex tallying instances (in particular corpuses A and C), it greatly reduces the constraint solving time, as shown by the columns *(None)* and *(None*)*. The effect is even more visible when semantic simplification is enabled, in the columns *(SS)* and *(SS*)*, as the simplification of a BDT generates numerous similar subtyping instances that will share the same cache and be optimized away by Algorithm 2 (this is why we also observe a positive impact on type building performance).

The column *(CDuce)* indicates the time performance of the CDuce implementation of set-theoretic types (cf. Section 8), which does not perform global hash-consing nor semantic simplification of BDTs, but does perform local hash-consing when building a type from equations to avoid duplicate nodes. On the corpus B, our implementation with semantic simplification is 66% faster than the CDuce implementation and generates simpler solutions. Our implementation with no type simplification is 212% faster than CDuce. Note that the comparison is not available for the other corpuses, as CDuce does not support tagged types (Section 6) which are used by the other corpuses to encode algebraic and mutable data types.

8 Related work and future work

The overall architecture of descriptors we use to represent set-theoretic types follows the one described in [5], though the latter does not feature type variables. In this paper, we describe this architecture in more details and focus on some specific points that we think are important and non-trivial: the handling of type variables, type substitutions, the simplification of BDTs, and the construction of recursive types. The algorithms used for subtyping and tallying have been first described by [18] and [13] respectively. This paper presents an alternative formulation of the tallying algorithm that enables optimizations and a novel implementation of the subtyping algorithm which makes better use of cached results.

Set-theoretic types in the CDuce language. The CDuce language (implemented in OCaml) has been the first to implement monomorphic higher-order set-theoretic types (the earlier XDuce [21] only featured first-order, top-level functions and no arrow type). It was later extended with type variables and tallying to handle parametric polymorphism [13]. This implementation supports arrows, pairs, records, integer intervals, characters and XML types. Although CDuce has been until recently the only complete open-source implementation of set-theoretic types, its architecture makes it difficult to reuse for other purposes. First, several XML-only features (document types, XML namespaces) needlessly complicate the type algebra. Second, since its design is very monolithic (the type algebra and subtyping algorithm are in a single file containing mutually recursive definitions), extending it with other construct or experimenting different data-structure or caching strategies is not trivial.

Set-theoretic types in mainstream languages. Two examples of set-theoretic type systems for mainstream languages are Etylizer [25] (for Erlang), and Elixir, [8]. Both provide an implementation of set-theoretic types that supports arrows and a generalized form of records for typing maps. Both

use BDDs to represent the different components; however, they also differ from our approach on several aspects. For instance, a node in the Etylizer implementation is represented by an identifier, and the corresponding definition must be looked up from a separate mapping. This choice of design seems motivated by the fact that Erlang does not allow the definition of mutable data structures, making it difficult to build recursive types or to simplify types if their definition is stored in the node itself. To our knowledge, these two implementations do not perform systematic simplification nor hash-consing of the types.

Lazy BDDs. The CDuce and Elixir use lazy BDDs for representing arrow components. As explained in [5], lazy BDDs avoid a potential explosion in size when repeatedly applying unions, and this until an intersection, difference or negation is performed. When it happens, the lazy union branches of the BDD must be materialized, the hope being that the intersection or difference still yields a simpler BDD by canceling some lazy union branches. Our semantic simplification (Section 3.3) can be adapted to work on lazy BDDs, but doing so increases its complexity and voids Property 3.10 that allows to quickly compute simplified negations. Overall, even if both approaches aim at keeping type representations small, they have different trade-offs: our systematic simplification tends to increase the building time of BDDs but simplifies future manipulations. Lazy BDDs have linear time unions but an increased complexity for some operations (such as negations).

Syntactic and algebraic approaches. While there are clear *theoretical* differences between semantic subtyping and syntax based ones, the separation lies elsewhere when it comes to implementation. Either the type system enjoys only *disjoint* union types, as is the case of MLsub [17]. In such works, subtyping remains polynomial, since types are equivalent to deterministic automata, and subtyping amounts to inclusion of their language, which is polynomial. Or some form of non-disjoint union occurs, as is the case in the early work of Aiken [1] or more recently MLStruct [24]. In both cases, either an exponential blow-up occurs when computing type operations (e.g. negation) or when deciding subtyping, by traversing the potentially exponential DNF of the type as we do. Both works mention in passing that memoizing intermediate results in a mutable table helps in practice, but do not give any detail. We believe our techniques could be applied to these works.

Data-flow analysis. The knowledgeable reader will remark that the subtyping algorithm could be cast in the framework of data-flow analysis. It is, after all, the computation of a fixed point of some monotonic predicate over a graph (the type, whose vertices are the nodes). For instance, Kildall's algorithm [22] or one of its variations could be used to compute the emptiness predicate. The major difference with our approach is that, to the best of our knowledge, all variants of Kildall's algorithm require the full graph (and often pre-process it). In our case, this would entail calling the $\Psi_{\mathsf{prod,arrow}}$ functions eagerly, generating a potentially exponential type up-front to test its emptiness.

Conclusion and future work. We proposed a modular data structure to represent recursive set-theoretic types supporting multiple type constructors, unions, intersections, negations, and type substitutions. This representation relies on Binary Decision Trees (BDTs), equipped with a semantic simplification procedure which eliminates redundancy that accumulates when manipulating types, and we provide practical improvement to the traditional algorithms for subtyping and tallying. The performance of SSTT is compared against CDuce, the historical implementation of set-theoretic types, showing a significative improvement (212% faster, and still 66% faster when systematically simplifying types). While SSTT is already usable, we plan to extend it to support row polymorphism for record [6], as well as extensions relevant to dynamic languages such as Python (such as object types).

References

[1] Alexander Aiken and Brian R. Murphy. 1991. Implementing regular tree expressions. In *Functional Programming Languages and Computer Architecture*, John Hughes (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 427–447.

[2] Anonymized. 2025. Anonymized article (under review). (2025).

[3] Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. doi:10.1109/TC.1986.1676819

[4] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. doi:10.1145/321239.321249

[5] Giuseppe Castagna. 2020. Covariance and Contravariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *Logical Methods in Computer Science* 16, 1 (2020), 15:1–15:58. doi:10.23638/LMCS-16(1:15)2020

[6] Giuseppe Castagna. 2023. Typing Records, Maps, and Structs. *Proc. ACM Program. Lang.* 7, ICFP (sep 2023). The video of the presentation given at ICFP 23 is available here.

[7] Giuseppe Castagna. 2024. *Programming with Union, Intersection, and Negation Types.* Springer International Publishing, Cham, 309–378. doi:10.1007/978-3-031-34518-0_12

[8] Giuseppe Castagna, Guillaume Duboc, and José Valim. 2024. The Design Principles of the Elixir Type System. *The Art, Science, and Engineering of Programming* 8, 2 (2024). Video of the presentation I gave at the Erlang Symposium 2023: https://youtu.be/VYmo867YF6g. Also available, Guillaume Duboc's presentation at ElixirConf EU 2023: https://www.youtube.com/watch?v=gJJH7a2J9O8.

[9] Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 198–208. doi:10.1145/1069774.1069793

[10] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: a New Perspective. *Proc. ACM Program. Lang.* 3, Article 16, POPL '19: 46th ACM Symposium on Principles of Programming Languages (jan 2019).

[11] Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. 2024. Polymorphic Type Inference for Dynamic Languages. *Proc. ACM Program. Lang.* 8, POPL, Article 40 (Jan. 2024), 32 pages. doi:10.1145/3632882

[12] Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. 2024. *Prototype: Polymorphic Type Inference for Dynamic Languages.* https://doi.org/10.5281/zenodo.10155221

[13] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '15)*. 289–302. doi:10.1145/2676726.2676991

[14] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. 2016. Set-Theoretic Types for Polymorphic Variants. *SIGPLAN Not.* 51, 9 (sep 2016), 378–391. doi:10.1145/3022670.2951928

[15] Giuseppe Castagna and Loïc Peyrot. 2025. Polymorphic Records for Dynamic Languages. *Proc. ACM Program. Lang.* 9, OOPSLA1 (4 2025).

[16] Giuseppe Castagna and Zhiwu Xu. 2011. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 94–106. doi:10.1145/2034773.2034788

[17] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*, Giuseppe Castagna and Andrew D. Gordon (Eds.). Association for Computing Machinery, New York, NY, USA, 60–72. doi:10.1145/3009837.3009882

[18] Alain Frisch. 2004. *Théorie, conception et réalisation d'un langage de programmation adapté à XML.* Ph. D. Dissertation. Université Paris Diderot.

[19] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM* 55, 4 (Sept. 2008), 19:1–19:64. http://doi.acm.org/10.1145/1391289.1391293

[20] Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2015. A logical approach to deciding semantic subtyping. *ACM Transactions on Programming Languages and Systems* 38, 1 (2015), 3. doi:10.1145/2812805

[21] Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.* 3, 2 (May 2003), 117–148. doi:10.1145/767193.767195

[22] Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. doi:10.1145/512927.512945

[23] Victor Lanvin. 2021. *A semantic foundation for gradual set-theoretic types.* Theses. Université Paris Cité. https://theses.hal.science/tel-03853222

[24] Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (oct 2022), 30 pages. doi:10.1145/3563304

[25] Albert Schimpf, Stefan Wehr, and Annette Bieniusa. 2023. Set-theoretic Types for Erlang. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (Copenhagen, Denmark) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. doi:10.1145/3587216.3587220

[26] Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming*.

[27] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. *ACM SIGPLAN Notices* 43, 1 (2008), 395–406.

A Semantic Subtyping

This appendix summarizes the interpretation of set-theoretic types formalized in [20] and how it can be used to define semantic subtyping.

A.1 Type interpretation

In order to define subtyping over these types, the idea is to interpret each ground type (i.e., a type that does not contain type variables) as a set of values of our language. Then, subtyping can be defined as set containment over the interpretation of types. Intuitively, each ground type is associated to the set of values having this type: for instance, the base type `true` is interpreted as the singleton containing the constant `true`, while the type `bool` = `true` ∨ `false` is interpreted as the set {`true`, `false`}.

However, this idea becomes subtler when dealing with arrow types. Although an arrow type intuitively corresponds to a function (i.e., a λ-abstraction), interpreting an arrow type as a set of λ-abstractions is problematic as it yields a circular reasoning: determining if a λ-abstraction is in the interpretation of a type requires to define a type system, which in turns needs the subtyping relation that we are trying to build. In order to break this circularity, the interpretation of types is not defined over values of our language but over a domain $\mathcal{D}$ defined below. Note that this does not necessarily invalidate the "types as set of values" intuition, as it is discussed in Castagna and Frisch [9, Section 2.7].

In addition, we need to define an interpretation for all types, and not only ground ones (the interpretation domain $\mathcal{D}$ should account for type variables). A simple model was proposed by [20]. We succinctly present it in this section. The reader may refer to [7, Section 3.3] for more details.

Definition A.1 (Interpretation domain [20]). The *interpretation domain* $\mathcal{D}$ is the set of finite terms d produced inductively by the following grammar

$$d ::= c^L \mid (d, d)^L \mid \{(d, \partial), \ldots, (d, \partial)\}^L$$
$$\partial ::= d \mid \Omega$$

where c ranges over the set C of constants, L ranges over finite sets of type variables, and where Ω is such that $\Omega \notin \mathcal{D}$.

The elements of $\mathcal{D}$ correspond, intuitively, to (denotations of) the results of the evaluation of expressions, labeled by finite sets of type variables. In particular, in a higher-order language, the results of computations can be functions which, in this model, are represented by sets of finite relations of the form $\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^L$, where Ω (which is not in $\mathcal{D}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input. This is implemented by using in the second projection the meta-variable ∂ which ranges over $\mathcal{D}_\Omega = \mathcal{D} \cup \{\Omega\}$ (we reserve d to range over $\mathcal{D}$, thus excluding Ω). This constant Ω is used to ensure that $\mathbb{1} \to \mathbb{1}$ is not a supertype of all function types: if we used d instead of ∂, then every well-typed function could be subsumed to $\mathbb{1} \to \mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, independently of its argument as long as this argument is typeable (see Section 4.2 of [19] for details). The restriction to *finite* relations corresponds to the intuition that the denotational semantics of a function is given by the set of its finite approximations, where finiteness is a restriction necessary (for cardinality reasons) to give the semantics to higher-order functions. Finally, the sets of type variables that label the elements of the domain are used to interpret type variables: we interpret a type variable α by the set of all elements that are labeled by α, that is the set $\{d \mid \alpha \in \mathsf{tags}(d)\}$ (where we define $\mathsf{tags}(c^L) = \mathsf{tags}((d, d')^L) = \mathsf{tags}(\{(d_1, \partial_1), \ldots, (d_n, \partial_n)\}^L) = L$).

We now define the interpretation $[\![t]\!]$ of a type t (this notation should not be mistaken with the interpretation $[\![B]\!]$ of a BDD as a type, defined in Section 3). The interpretation $[\![t]\!]$ should satisfy

the following equalities, where $\mathcal{P}_{\text{fin}}$ denotes the restriction of the powerset to finite subsets and $\mathbb{B}$ denotes the function that assigns to each base type the set of constants of that type, so that for every constant c we have $c \in \mathbb{B}(\boldsymbol{b}_c)$ (we use $\boldsymbol{b}_c$ to denote the base type of the constant c):

$$\begin{array}{lll} [\![\mathbb{0}]\!] = \emptyset & [\![\alpha]\!] = \{d \mid \alpha \in \mathsf{tags}(d)\} & [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \\ [\![b]\!] = \mathbb{B}(b) & [\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] & [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] = \{R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, \partial) \in R.\, d \in [\![t_1]\!] \implies \partial \in [\![t_2]\!]\} & & \end{array}$$

Note that, even though we included $\mathbb{1}$ and the intersection $\wedge$ in the syntax of our types (Definition 2.1), those two can be defined from the other constructors: $\mathbb{1} = \neg\mathbb{0}$ and $t_1 \wedge t_2 = \neg(\neg t_1 \vee \neg t_2)$ (De Morgan's law). It is easy to see that, with these definitions, we have $[\![\mathbb{1}]\!] = \mathcal{D}$ and $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$. Thus, it is not necessary to define an interpretation for them.

We cannot take the equations above directly as an inductive definition of $[\![\cdot]\!]$ because types are not defined inductively but coinductively. Notice however that the contractivity condition of Definition 2.1 ensures that the binary relation $\triangleright \subseteq \mathcal{T} \times \mathcal{T}$ defined by $t_1 \vee t_2 \triangleright t_i$, $t_1 \wedge t_2 \triangleright t_i$, $\neg t \triangleright t$ is Noetherian. This gives an induction principle[5] on $\mathcal{T}$ that we use combined with structural induction on $\mathcal{D}$ to give the following definition, which validates the equalities above.

Definition A.2 (Set-theoretic interpretation of types). We define a binary predicate $(d : t)$ ("the element d belongs to the type t"), where $d \in \mathcal{D}$ and $t \in \mathcal{T}$, by induction on the pair (d, t) ordered lexicographically. The predicate is defined as follows:

$$\begin{array}{rl} (c : b) & = c \in \mathbb{B}(b) \\ (d : \alpha) & = \alpha \in \mathsf{tags}(d) \\ ((d_1, d_2) : t_1 \times t_2) & = (d_1 : t_1) \text{ and } (d_2 : t_2) \\ (\{(d_1, \partial_1), ..., (d_n, \partial_n)\} : t_1 \to t_2) & = \forall i \in [1..n].\ \text{if } (d_i : t_1) \text{ then } (\partial_i : t_2) \\ (d : t_1 \vee t_2) & = (d : t_1) \text{ or } (d : t_2) \\ (d : \neg t) & = \text{not } (d : t) \\ (\partial : t) & = \text{false} \qquad \text{otherwise} \end{array}$$

We define the *set-theoretic interpretation* $[\![.]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ as $[\![t]\!] = \{d \in \mathcal{D} \mid (d : t)\}$.

A.2 Semantic subtyping

Now that we have a set-theoretic interpretation of types, we can define the subtyping preorder and its associated equivalence relation as follows.

Definition A.3 (Subtyping relation). We define the *subtyping* relation $\leq$ and the *subtyping equivalence* relation $\simeq$ as $t_1 \leq t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ and $t_1 \simeq t_2 \stackrel{\text{def}}{\iff} (t_1 \leq t_2)$ and $(t_2 \leq t_1)$.

This subtyping relation is decidable and is sometimes referred to as *semantic subtyping*, as it is not defined on the syntax of the type but on its interpretation.

With this set-theoretic definition of subtyping, usual properties of sets are inherited by subtyping, for instance:

$$\begin{array}{lll} t_1 \vee t_2 \simeq t_2 \vee t_1 & t_1 \wedge t_2 \simeq t_2 \wedge t_1 & \text{(commutativity)} \\ t \vee t \simeq t & t \wedge t \simeq t & \text{(idempotence)} \\ \neg(\neg t) \simeq t & & \text{(double complement)} \\ t \vee (s_1 \wedge s_2) \simeq (t \vee s_1) \wedge (t \vee s_2) & t \wedge (s_1 \vee s_2) \simeq (t \wedge s_1) \vee (t \wedge s_2) & \text{(distributivity)} \end{array}$$

[5] In a nutshell, we can do proofs and give definitions by induction on the structure of unions and negations—and, thus, intersections—but arrows, products, and base types are the base cases for the induction.

For any two type substitutions ϕ_1 and ϕ_2, we write $\phi_1 \simeq \phi_2$ the pointwise subtyping equivalence of ϕ_1 and ϕ_2. An important property of the interpretation above is that subtyping is preserved by type substitutions:

$$\forall t_1, t_2, \phi.\ t_1 \leq t_2 \Rightarrow t_1\phi \leq t_2\phi$$

However, a naive definition of $\mathsf{vars}(t)$ is not preserved by subtyping equivalence: for instance, we have $\mathbb{1} \simeq \alpha \vee \neg\alpha$, while a purely syntactic definition of $\mathsf{vars}(t)$ would yield $\mathsf{vars}(\mathbb{1}) = \emptyset$ and $\mathsf{vars}(\alpha \vee \neg\alpha) = \{\alpha\}$. In order to avoid this, we define $\mathsf{vars}(t)$ as being the set of *meaningful type variables* in t. This notion has been introduced by Castagna et al. [14], where it was noted as $\mathtt{mvar}(t)$, and is defined below.

Definition A.4 (Type variables). The set of type variables of a type t, noted $\mathsf{vars}(t)$, is the following set of type variables:

$$\mathsf{vars}(t) \stackrel{\text{def}}{=} \{\alpha \in \mathcal{V} \mid t\{\alpha \rightsquigarrow \mathbb{0}\} \not\simeq t\}$$

With this definition, the set of variables of a type is preserved by subtyping equivalence: $\forall t_1, t_2.\ t_1 \simeq t_2 \Rightarrow \mathsf{vars}(t_1) = \mathsf{vars}(t_2)$.

B Pretty printing of types

Section 4 defines an internal representation for types that captures their semantic properties. This semantic representation makes it possible to implement set-theoretic connectives ($\wedge$, $\vee$, $\neg$) efficiently, going from an algebraic representation of a type into a more semantic representation. However, when printing a set-theoretic type, we need to go the other direction: going from our internal representation back into an algebraic one. We propose in this section a quick description of the challenges and techniques we implemented to convert types into an algebraic form.

B.1 Printing of descriptors

For each component of a descriptor (arrow, product, constant), we can easily extract a DNF of atoms directly from their BDT representation. However finding a concise algebraic representation for unions of components requires some more work. The naive approach consisting in systematically printing the union of all components may generate overly complicated notations: for instance, the type $\mathbb{1}$ is printed as `AnyArrow | AnyPair | AnyConst`, and the type $t \stackrel{\text{def}}{=} \neg(\texttt{int} \times \texttt{int})$ is printed as `AnyArrow | (AnyPair \ (int,int)) | AnyConst`. Instead, we propose the following algorithm for printing a descriptor D in a more concise way:

(1) Each component of D is converted into an algebraic representation, and these algebraic representations are regrouped inside a union. We simplify this union by eliminating empty clauses. This yields an algebraic representation P_1 for D. For instance, for the descriptor of t, this yields $P_1 \stackrel{\text{def}}{=}$ `AnyArrow | (AnyPair \ (int,int)) | AnyConst`.
(2) Independently, the complement of each component of D is converted into an algebraic representation, and these algebraic representations are regrouped inside a union. For instance, for the descriptor of t, this yields `EmptyArrow | (int,int) | EmptyConst`. Again, we simplify this union by eliminating empty clauses, yielding `(int,int)`. We obtain an algebraic representation N for the negation of D.
(3) We negate the algebraic representation N and perform trivial simplifications (such as elimination of double negations), yielding a representation P_2 for D. In our example, we obtain $P_2 \stackrel{\text{def}}{=} \neg$`(int,int)`.
(4) We compare the length of P_1 and P_2 (using any relevant metric, for instance the number of nodes in their AST), and return the smaller one (in our case, P_2).

B.2 Printing of (recursive) nodes

Our types are represented by graphs of nodes that may contain cycles (in the case of recursive types). Thus, when printing a descriptor, it is not enough to recursively call the printer on each node referenced by its atoms, as this may result in an infinite inlining of recursive types. Instead, we follow the following strategy for printing a node N:

(1) For each node in the connected component $\text{deps}(N)$ of N, we compute an algebraic form for the top-level structure of its definition (referenced nodes are kept as symbolic references). For instance, when printing the type $\mu X.\ ((\alpha \wedge (\texttt{int} \to \texttt{bool})) \times X) \vee \texttt{nil}$ represented in Figure 2, this leaves us with:
`N1 where N1=(N2,N1)|nil and N2='a&(N3->N4) and N3=int and N4=bool`
(2) Then, we try to inline the definition of each node (`N1,N2,N3,N4` in our example), but only when doing so reduces the length of the whole algebraic representation. This way, recursive references will not be inlined. In our example, the nodes `N2, N3, N4` are inlined, yielding the algebraic expression `N1 where N1=('a&(int->bool),N1)|nil`.

C Examples of benchmarks

The corpuses have been type-checked using [2], a type system based on the techniques of inference and type narrowing introduced in [11].

C.1 A. Hindley-Milner corpus

This corpus has been type-checked in a configuration where the type inference algorithm does not try to infer overloaded types for functions, but single arrow types (as in HM systems). Here is an excerpt (the type inferred for each function is added as a comment):

```
let fixpoint = fun f ->
  let delta = fun x ->
     f ( fun  v -> ( x x v ))
   in delta delta
(* (('a -> 'b) -> x1) -> x1 where x1 = 'c & ('a -> 'b) *)

let length_stub length lst =
  if lst is [] then 0 else (length (tl lst))+1
(* ('a & [ any* ] -> int) -> [] | any::('a & [ any* ]) -> int *)

let length = fixpoint length_stub
(* [ any* ] -> int *)
```

Type-checking this excerpt generates 74 tallying instances, for instance:

```
"vars": [ "'a", "'b", "'c", "'d", "'e" ],
"mono": [],
"constr": [
    [
    "(('a -> 'b) -> 'c & ('a -> 'b)) -> 'c & ('a -> 'b)",
    "(('d & lst(x1) -> int) -> lst(tuple0 | (any, 'd & lst(x1))) -> int) -> 'e
        where x1 = tuple0 | (any, lst(x1))"
    ]
]
```

C.2 B. Overloaded corpus

This corpus has been type-checked in a configuration where the type inference algorithm tries to infer overloaded types for functions (except when the domain of the function is explicitly given). Here is an excerpt (the type inferred for each function is added as a comment):

```
let succ x =
  match x with
  | 0 -> 1 | 1 -> 2 | 2 -> 3 | 3 -> 4 | 4 -> 5
  | 5 -> 6 | 6 -> 7 | 7 -> 8 | 8 -> 9 | 9 -> 10
  | 10 -> 11 | 11 -> 12 | 12 -> 13 | 13 -> 14 | 14 -> 15
  | 15 -> 16 | 16 -> 17 | 17 -> 18 | 18 -> 19 | 19 -> 20
  | 20 -> 21 | 21 -> 22 | 22 -> 23 | 23 -> 24 | 24 -> 25
  | _ -> 0
  end
(* (0 -> 1) & (1 -> 2) & ... & (24 -> 25) & (~(0..24) -> 0) *)
```

```
let test (y:(5..15)) = succ (succ (succ (succ (succ y))))
(* (5..15) -> (10..20) *)
```

Type-checking this excerpt generates 2941 tallying instances (most are trivial and are triggered by simplification heuristics), for instance:

```
"vars": [ "'a" ],
"mono": [],
"constr": [
    [
    "(1 -> 2) & (2 -> 3) & (3 -> 4) & (4 -> 5)
    & (5 -> 6) & (6 -> 7) & (7 -> 8) & (8 -> 9) & (9 -> 10)
    & (10 -> 11) & (11 -> 12) & (12 -> 13) & (13 -> 14) & (14 -> 15)
    & (15 -> 16) & (16 -> 17) & (17 -> 18) & (18 -> 19) & (19 -> 20)
    & (20 -> 21) & (21 -> 22) & (22 -> 23) & (23 -> 24) & (24 -> 25)
    & (~(0..24) -> 0) & (0 -> 1)",
    "(5..15) -> 'a"
    ]
]
```

C.3 C. HM+Overloaded corpus

This corpus has been type-checked in a configuration where the type inference algorithm performs type narrowing, and where mutable data structures (arrays, references) are encoded as opaque data types. Here is an excerpt (the type inferred for each function is added as a comment):

```
let filter_imp (f:('a -> bool) & ('b -> false)) (arr:array('a|'b)) =
  let res = array () in
  let mut i = 0 in
  while i < (len arr) do
    let e = arr[i] in
    if f e do push res e end ;
    i := i + 1
  end ;
  return res
(* ('b -> false)&('a -> bool) -> array('b|'a) -> array('a\'b | 'c) *)
```

Type-checking this excerpt generates 70 tallying instances, for instance:

```
"vars": [ "'a", "'b", "'c" ],
"mono": [ "'a", "'b" ],
"constr": [
    [
    "('b -> bool(false)) & ('a -> bool(true | false))", "'a \\ 'b -> 'c"
    ]
]
```