Fast Verified BCD Subtyping

Jan Bessai[1(✉)], Jakob Rehof[1], and Boris Düdder[2]

[1] Technische Universität Dortmund,
Otto-Hahn-Straße 12, 44227 Dortmund, Germany
{jan.bessai,jakob.rehof}@tu-dortmund.de
[2] University of Copenhagen, Universitetsparken 5, 2100 Copenhagen, Denmark
boris.d@di.ku.dk

Abstract. A decision procedure for the Barendregt-Coppo-Dezani subtype relation on intersection types ("BCD subtyping") is presented and formally verified in Coq. Types are extended with unary, covariant, distributing, preordered type constructors and binary products. A quadratic upper bound on the algorithm runtime is established. The formalization can be compiled to executable OCaml or Haskell code using the extraction mechanism of Coq.

Keywords: Intersection types · Subtyping · Coq · BCD

1 Introduction

The subtyping relation of Barendregt, Coppo, and Dezani [4] is a natural, semantically motivated notion of subtyping on intersection types. The relation, often referred to as BCD subtyping for short, is important within the theory of intersection types, and many variants of the intersection type system are associated with theories of subtyping which are contained in the theory of BCD subtyping (see [3] for an overview). The present paper concerns the formal verification, by means of theorem proving, of an efficient (quadratic time) decision procedure for the BCD subtyping relation: Given two intersection types A and B, does $A \leq B$ hold (where $\leq$ denotes the BCD subtyping relation)? Decidability of the subtyping relation is probably most easily established by first performing a pre-processing step called normalization following Hindley [14]. However, since this step may cause exponential blow-up in type size, it only gives rise to a computationally suboptimal algorithm. In fact, a quadratic time algorithm time is known for deciding BCD subtyping [10], and quadratic time is in all likelihood asymptotically optimal for the problem. But efficient algorithms for the problem tend to get complicated, in part due to the necessity of organizing rather intricate case analyses under recursive descent over type expressions. When the relation is further extended with type constants for applications, correctness of efficient implementations becomes even more of an issue of interest (see Sect. 2 for further discussion). From the perspective of formal verification it is a topic of

T. Margaria et al. (Eds.): Steffen Festschrift, LNCS 11200, pp. 356–371, 2019.
https://doi.org/10.1007/978-3-030-22348-9_21

more general interest to see how far we can get towards verification of correctness of algorithms under given complexity bounds.

In this paper a decision procedure for the BCD subtype relation on intersection types is presented and formally verified in Coq. Types are extended with unary, covariant, distributing, preordered type constructors and binary products. A quadratic upper bound on the algorithm runtime is established. The formalization can be compiled to executable OCaml or Haskell code using the extraction mechanism of Coq. The accompanying Coq proofs for this paper are available online[1].

This paper is organized as follows: Related work on decision procedures for the BCD subtype rules is discussed in Sect. 2. This discussion is also used to pinpoint novel contributions made here. Section 3 contains the definition of intersection types and the subtype relation on them. The decision procedure is presented and proven correct in Sect. 4. An upper bound on its runtime is proven in Sect. 5. Finally, Sect. 6 provides some concluding remarks and ideas for future work.

2 Related Work and Contribution

Pierce [22] provides an algorithm for deciding the BCD subtype relation under a set of additional constraints on type variables. No asymptotic runtime bound is established. Damm [8,9] reduces the problem of deciding subtyping for intersection types extended with recursive types and a union operator to regular tree expressions, resulting in a non-deterministic exponential time algorithm. Kurata and Takahashi [18] provide an algorithm for intersection types without additional extensions. It needs a pre-computation step to normalize (see [14]) types, which requires exponential runtime. Rehof and Urzyczyn [23] first established an algorithm with an $\mathbb{O}(n^4)$ upper bound on its runtime. Their memoization-based formalization is manual. Practical experience from the (CL)S-Framework [5] has shown that implementing the required memoization techniques is possible but error prone. Also within the context of (CL)S, the algorithm implementation was experimentally extended with distributing covariant n-ary type-constructors. Subsequently, Statman [24] presented a rewriting-based $\mathbb{O}(n^5)$ algorithm for which no implementation or computer supported verification exists yet. His presentation includes some insights about factorizations of intersection types, which were then picked up in [10], where a simpler $\mathbb{O}(n^2)$ algorithm is sketched. This algorithm uses a preprocessing step, which can in contrast to [18] be performed in linear time. Additionally, the insights of [24] led to the development of the first theorem prover verified algorithm by Bessai et al. [6]. The algorithm is based on purely mathematical principles (ideals and filters) and can be extracted to OCaml and Haskell. Throughout the formalization, high-level mathematical concepts are accessed via the Coq tactics language, which hides algorithmic aspects and makes reasoning about runtime (except for guaranteed termination) difficult. Hence, while practical experiments hint at an $\mathbb{O}(n^4)$ upper bound, this result has never been formally established. The development in [6] sparked at least two

[1] https://github.com/JanBessai/SubtypeMachine.

more formalization efforts. Honsell et al. [16] extend the type system with union types, basing their work on a fork of the original formalization. Laurent [20] extends intersection types with n-ary co- and contra-variant constructors and translates the subtype rules into a syntax-driven sequent-calculus. The translation is proven correct in Coq, but its algorithmic aspects remain unstudied. Similarly, Dunfield [11] earlier proposed an extension with union types presentable in sequent-calculus form. Bi et al. [7] go back to the algorithm presented in [22] and propose an extension with records and coercions. They do not study runtime complexity. A theorem proven Coq formalization and a manual translation of their syntax directed rules into a Haskell implementation are provided. The formalization effort led to the discovery of a mistake in the original manual proof in [22].

The contribution of this paper is to describe a Coq-formalized subtype decision procedure for intersection types extended with unary, co-variant, distributing, preordered type constructors and binary products. An $\mathbb{O}(n^2)$ upper bound on the runtime is formally established. The decision procedure does not require any preprocessing steps on the input types, no translation into another calculus, and no imperative programming language features such as memoization. Extraction of the Coq formalization into Haskell and OCaml is possible. Besides the improvements over prior attempts, the formalization can serve as an example application for a technique recently presented by Larchey-Wendling and Monin [19]. The goal of this technique is to decompose termination proofs from fixpoint definitions and thereby make reasoning more compositional. The quest for compositional proof methods is an old topic, which can also be found in earlier work by Steffen and Cleaveland [25] in the context of model-checking. This contribution can be seen as a very detailed study of an improved algorithm for a particular problem, as well as a larger example scenario of applying a proof compositionality technique in a theorem prover.

3 Types and Subtyping

Intersection types A, B are formed over the following syntax:

$$\mathbb{T} \ni A, B ::= \omega \mid c(A) \mid (A \times B) \mid (A \to B) \mid (A \cap B)$$

where $c \in \mathbb{C}$ is a type constructor drawn from a countable set $\mathbb{C}$. Intuitively, ω is a universal type and supertype of every other type. Constants are encoded by type constructors and may appear nested inside type expressions. This allows for types such as $\texttt{List}(A)$, $\texttt{List}(\texttt{List}(A))$. Instead of assigning an arity to each constructor, all constructors take exactly one argument. This restriction avoids lots of index sets, adds uniformity to proofs and does not affect the runtime complexity of subtyping. Atomic types, often modeled by constructors without arguments, can still be represented using ω as argument: types such as `bool` or `int` are formally written as $\texttt{int}(\omega)$ and $\texttt{bool}(\omega)$. Multiple arguments can be passed to a type constructor by wrapping them into the product type $(A \times B)$, e.g. $\texttt{Graph}(N \times E)$ for a graph with nodes of type N and edges of type E.

Function types are written as $(A \to B)$ and the presence of products allows to choose between curried $(A \to (B \to C))$ and un-curried $((A \times B) \to C)$ representations. Finally, the intersection type operator $(A \cap B)$ encodes the greatest lower bound of two types. In contrast to the pair $(A \times B)$, which is used to assign two types to two components, the intersection $(A \cap B)$ is used to assign two types to a single component. In the rest of this paper, superfluous parentheses in types are omitted by following the convention that arrows and intersections associate to the right, products associate to the left, and intersections bind stronger than products, which bind stronger than arrows.

The subtype relation $A \leq B$ is the least relation closed under the rules:

$$\frac{c \leq_{\mathbb{C}} d \qquad A \leq B}{c(A) \leq d(B)}\ (\textsc{CAx}) \qquad \frac{}{c(A) \cap c(B) \leq c(A \cap B)}\ (\textsc{CDist})$$

$$\frac{}{A \leq \omega}\ (\omega) \qquad \frac{}{\omega \leq \omega \to \omega}\ (\to \omega)$$

$$\frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}\ (\textsc{Sub}) \qquad \frac{}{(A \to B_1) \cap (A \to B_2) \leq A \to B_1 \cap B_2}\ (\textsc{Dist})$$

$$\frac{A_1 \leq B_1 \qquad A_2 \leq B_2}{A_1 \times A_2 \leq B_1 \times B_2}\ (\textsc{ProdSub})$$

$$\frac{}{(A_1 \times A_2) \cap (B_1 \times B_2) \leq A_1 \cap B_1 \times A_2 \cap B_2}\ (\textsc{ProdDist})$$

$$\frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \cap B_2}\ (\text{GLB}) \qquad \frac{}{B_1 \cap B_2 \leq B_1}\ (\text{LUB}_1) \qquad \frac{}{B_1 \cap B_2 \leq B_2}\ (\text{LUB}_2)$$

$$\frac{A \leq B \qquad B \leq C}{A \leq C}\ (\textsc{Trans}) \qquad \frac{}{A \leq A}\ (\textsc{Refl})$$

Rules (CAx), (CDist), (ProdSub) and (ProdDist) are extensions, while the other rules are standard [10,24] and equivalent [2] to the rules originally presented by Barendregt, Coppo and Dezani [4]. In rule (CAx) constructors are compared using an externally defined relation $c \leq_{\mathbb{C}} d$, which can be instantiated according to application specific use-cases and has to be transitive and reflexive. A potential application are nominal comparisons between type-constructors according to a class-table: to model a Java-like type-system `ArrayList` $\leq_{\mathbb{C}}$ `List` can be added, which would allow using instances of `ArrayList` whenever `List` is required. Rule (CDist) allows to distribute intersections over constructors. If something is simultaneously a list of A and a list of B, it is a list of things which are simultaneously A and B: $\texttt{List}(A) \cap \texttt{List}(B) \leq \texttt{List}(A \cap B)$. Note, that the converse direction is derivable from (CAx) and (GLB). Rules (ProdSub) and (ProdDist) are analogous to the constructor rules, allowing to compare products and again to distribute intersections. The chosen extensions are conservative over the original BCD-system, which only supports atoms: choosing

$c \leq_{\mathbb{C}} d$ iff $c = d$ and encoding atom a as $a(\omega)$ collapses rule (CDist) into an instance of (LUB_1), and (CAx) into (Refl).

There are several functions on types which are useful throughout the entire formalization. The arity of a type is the arity of its outermost operation or constructor:

$$\text{arity}(A) = \begin{cases} \mathbb{1} & \text{if } A = \omega \\ \mathbb{T} & \text{if } A = c(A') \\ \mathbb{T} \times \mathbb{T} & \text{otherwise} \end{cases}$$

where $\mathbb{1}$ is the (meta-logical) unit set $\{\emptyset\}$, and $\mathbb{T} \times \mathbb{T}$ is the Cartesian product of the set of types with itself (not to be confused with the product type operator). For $(A, B) \in \mathbb{T} \times \mathbb{T}$, the first and second projections are defined as $(A, B).1 = A$ and $(A, B).2 = B$. The size, depth, length and breadth of a type are useful measures for runtime-complexity and termination of algorithms:

$$\text{size}(A) = \begin{cases} 1 & \text{if } A = \omega \\ 1 + \text{size}(B) & \text{if } A = c(B) \\ 1 + \text{size}(B) + \text{size}(C) & \text{if } A = B \to C, A = B \times C, \text{ or } A = B \cap C \end{cases}$$

$$\text{depth}(A) = \begin{cases} 1 & \text{if } A = \omega \\ 1 + \text{depth}(B) & \text{if } A = c(B) \\ 1 + \max\{\text{depth}(B), \text{depth}(C)\} & \text{if } A = B \to C, \text{ or } A = B \times C \\ \max\{\text{depth}(B), \text{depth}(C)\} & \text{if } A = B \cap C \end{cases}$$

$$\text{length}(A) = \begin{cases} 1 + \text{length}(A_2) & \text{if } A = A_1 \to A_2 \\ \text{length}(A_1) + \text{length}(A_2) & \text{if } A = A_1 \cap A_2 \\ 1 & \text{otherwise} \end{cases}$$

$$\text{breadth}(A) = \begin{cases} \text{breadth}(A_1) + \text{breadth}(A_2) & \text{if } A = A_1 \cap A_2 \\ 1 & \text{otherwise} \end{cases}$$

From now on size(A) will be abbreviated as $\|A\|$. Following the definition of Ω in [4], a type can be identified as subtype-equal to ω ($A \leq \omega$ and $\omega \leq A$) in $\mathbb{O}(\text{length}(A))$ by:

$$\text{isOmega}(A) = \begin{cases} \text{true} & \text{if } A = \omega \\ \text{isOmega}(B) & \text{if } A = A' \to B \\ \text{false} & \text{otherwise} \end{cases}$$

The intersection of a list of n types is computed in n steps by

$$\text{intersect}(\Delta) = \begin{cases} \omega & \text{if } \Delta = [::] \\ A & \text{if } \Delta = [::A] \\ A \cap \text{intersect}(\Delta') & \text{if } \Delta = [::A \,\&\, \Delta'] \end{cases}$$

where $[::]$ is the empty list, $[::A]$ is a list with one element A and $[::A \;\&\; \Delta']$ is the list constructed by inserting A before list Δ'. In the following, $\bigcap_{A_i \in \Delta} M$ will serve as a shorthand notation for intersect(map($\lambda A_i.M, \Delta$)), where map is defined as usual by map($\lambda A_i.M, [::]$) = $[::]$ and map($\lambda A_i.M, [::A \;\&\; \Delta]$) = $[::M[A_i := A] \;\&\; \text{map}(\lambda A_i.M, \Delta)]$.

4 Decision Procedure

While elegant and concise, the axiomatic relational presentation of $\leq$ is inherently non-algorithmic and therefore ill-suited for the construction of executable decision programs. This becomes obvious, considering the cut-type B in the transitivity rule (Trans) and cycles which can arise, e.g. by instantiating A to ω in rule (ω) and proceeding with rule $(\to \omega)$. This motivates the rest of the paper. Following the approach in [19], the decision procedure is designed in three phases. First, the algorithm is defined by means of its relational semantics. Then, the semantical relation is shown to be functional. Finally, a termination certificate is designed to turn the relational semantics into an executable denotational equivalent. Soundess of the functional interpretation with respect to the relational description is enforced by its type. Correctness of the algorithm is proven by showing that the relation is equivalent to the subtype relation. This way, the termination proof is effectively separated from the correctness proof. Also, bounds on the runtime are established by bounding the number of transitive steps in the semantical relation. An additional benefit is to establish a mental model of a subtype machine executing instructions in a step-wise fashion. Each of these steps can be understood and reasoned about independently, while the usual presentation of abstract pseudo code does not allow this kind of mental specification debugging. The instruction set $\mathcal{I}$ of the subtype machine contains two instructions: [`subty` A `of` B] and [`tgt_for_srcs_gte` A `in` Δ] for types A, B, and a list Δ which contains pairs $(A_n, B_n) \in \mathbb{T} \times \mathbb{T}$. The first instruction advises the machine to check if A is a subtype of B, while the second instruction collects all types B_n in Δ, for which A is a subtype of the corresponding A_n. Outputs $\mathcal{O}$ are [`Return` b] and [`check_tgt` Δ] for a boolean value b and a list of collected types Δ. The functions defined in the last section are taken as meta-operations usable during the machine specification. All of these functions could have been specified together with the relational machine semantics, but since none of them have interesting termination or runtime behavior, this would have been unnecessarily complicated. Another meta-function, cast, is needed before defining the semantical relation:

$$\text{cast}_B(A) = \begin{cases} [::\omega] & \text{if } B = \omega \\ [::(\omega, \omega)] & \text{if } B = B_1 \to B_2 \text{ and isOmega}(B_2) \\ \text{cast}'_B(A, [::]) & \text{otherwise} \end{cases}$$

$$\mathrm{cast}'_B(A, \Delta) = \begin{cases} [::A' \;\&\; \Delta] & \text{if } A = c(A'), B = d(B') \text{ and } c \leq_{\mathbb{C}} d \\ [::(A_1, A_2) \;\&\; \Delta] & \text{if } A = A_1 \to A_2 \text{ and } B = B_1 \to B_2 \\ [::(A_1, A_2) \;\&\; \Delta] & \text{if } A = A_1 \times A_2 \text{ and } B = B_1 \times B_2 \\ \mathrm{cast}'_B(A_1, \mathrm{cast}'_B(A_2, \Delta)) & \text{if } A = A_1 \cap A_2 \\ \Delta & \text{otherwise} \end{cases}$$

The range of function cast_B is $\mathrm{arity}(B)$ and collects all relevant components of A for recursive comparison with B. If B is subtype equal to ω, comparison can proceed with ω components, otherwise cast' loops over all parts of A, filtering those which are irrelevant. Note that cast' collects components in an accumulator argument Δ rather than using list concatenation. This way the runtime of cast' is linear in $\mathrm{breadth}(A)$ (all cases except for intersection are in $\mathbb{O}(1)$ and the intersection is in $\mathbb{O}(\mathrm{breadth}(A))$). List concatenation in the intersection case would have caused quadratic runtime if cast' were to be implemented using functional programming and immutable lists with concatenation in $\mathbb{O}(n)$. Overall the runtime of cast is in $\mathbb{O}(\mathrm{breadth}(A) + \mathrm{length}(B))$ and the output of cast will be a list of size less or equal to $\mathrm{breadth}(A)$. Now the subtype machine execution semantics $\rightsquigarrow$ is defined to be the least relation closed under the following rules, where annotations in boxes indicate runtime bounds and can be ignored until Sect. 5:

$$\frac{}{[\ \texttt{subty}\ A\ \texttt{of}\ \omega] \rightsquigarrow [\ \texttt{Return true}]\ \boxed{\mathbb{O}(1)}}\ (\textsc{step}_\omega)$$

$$\frac{[\ \texttt{subty}\ \bigcap_{A_i \in \mathrm{cast}_{c(B)}(A)} A_i\ \texttt{of}\ B] \rightsquigarrow [\ \texttt{Return}\ b]\ \boxed{n}}{\begin{array}{c}[\ \texttt{subty}\ A\ \texttt{of}\ c(B)] \rightsquigarrow [\ \texttt{Return}\ \mathrm{cast}_{c(B)}(A) \neq [::] \wedge b] \\ \boxed{(\mathbb{O}(\mathrm{breadth}(A) + \mathrm{length}(c(B))) + \mathbb{O}(\mathrm{breadth}(A)) + \mathbb{O}(1)) + n}\end{array}}\ (\textsc{step}_{\textsc{Ctor}})$$

$$\frac{\begin{array}{c}[\ \texttt{tgt\_for\_srcs\_gte}\ B_1\ \texttt{in}\ \mathrm{cast}_{B_1 \to B_2}(A)] \rightsquigarrow [\ \texttt{check\_tgt}\ \Delta]\ \boxed{m} \\ [\ \texttt{subty}\ \bigcap_{A_i \in \Delta} A_i\ \texttt{of}\ B_2] \rightsquigarrow [\ \texttt{Return}\ b]\ \boxed{n}\end{array}}{\begin{array}{c}[\ \texttt{subty}\ A\ \texttt{of}\ B_1 \to B_2] \rightsquigarrow [\ \texttt{Return}\ \mathrm{isOmega}(B_2) \vee b] \\ \boxed{\begin{array}{c}(\mathbb{O}(\mathrm{breadth}(A) + \mathrm{length}(B_1 \to B_2)) + \mathbb{O}(\mathrm{breadth}(A)) + \mathbb{O}(\mathrm{length}(B_2)) + \\ \mathbb{O}(1)) + m + n\end{array}}\end{array}}\ (\textsc{step}_{\to})$$

$$\frac{\begin{array}{c}[\ \texttt{subty}\ B\ \texttt{of}\ A.1] \rightsquigarrow [\ \texttt{Return}\ b]\ \boxed{m} \\ [\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ \Delta] \rightsquigarrow [\ \texttt{check\_tgt}\ \Delta']\ \boxed{n}\end{array}}{\begin{array}{c}[\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ [::A\ \&\ \Delta]] \rightsquigarrow \\ [\ \texttt{check\_tgt}\ \text{if } b \text{ then } [::A.2\ \&\ \Delta'] \text{ else } \Delta']\ \boxed{\mathbb{O}(1) + m + n}\end{array}}\ (\textsc{step}_{\textsc{ChooseTgt}})$$

$$\frac{}{[\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ [::]] \rightsquigarrow [\ \texttt{check\_tgt}\ [::]]\ \boxed{\mathbb{O}(1)}}\ (\textsc{step}_{\textsc{DoneTgt}})$$

$$\frac{\begin{array}{c}[\ \texttt{subty}\ \bigcap_{A_i \in \text{cast}_{B_1 \times B_2}(A)} A_i.1\ \texttt{of}\ B_1] \rightsquigarrow [\ \texttt{Return}\ b_1] \quad \boxed{m} \\ [\ \texttt{subty}\ \bigcap_{A_i \in \text{cast}_{B_1 \times B_2}(A)} A_i.2\ \texttt{of}\ B_2] \rightsquigarrow [\ \texttt{Return}\ b_2] \quad \boxed{n}\end{array}}{[\ \texttt{subty}\ A\ \texttt{of}\ B_1 \times B_2] \rightsquigarrow [\ \texttt{Return}\ \text{cast}_{B_1 \times B_2}(A) \neq [::] \wedge b_1 \wedge b_2]}\ (\text{STEP}_\times)$$

$$\boxed{(\mathbb{O}(\text{breadth}(A) + \text{length}(B_1 \times B_2)) + 2 \cdot \mathbb{O}(\text{breadth}(A)) + \mathbb{O}(1)) + m + n}$$

$$\frac{\begin{array}{c}[\ \texttt{subty}\ A\ \texttt{of}\ B_1] \rightsquigarrow [\ \texttt{Return}\ b_1] \quad \boxed{m} \\ [\ \texttt{subty}\ A\ \texttt{of}\ B_2] \rightsquigarrow [\ \texttt{Return}\ b_2] \quad \boxed{n}\end{array}}{[\ \texttt{subty}\ A\ \texttt{of}\ B_1 \cap B_2] \rightsquigarrow [\ \texttt{Return}\ b_1 \wedge b_2] \quad \boxed{\mathbb{O}(1) + m + n}}\ (\text{STEP}_\cap)$$

Rules (STEP_ω) and ($\text{STEP}_\cap$) are immediate implementations of the (GLB) and (ω) subtype rules. Similarly, ($\text{STEP}_{\text{CTOR}}$) and ($\text{STEP}_\times$) can be thought of as implementations combinations of (CAx) with (CDist), and (ProdSub) with (ProdDist). In both cases cast projects A to relevant components, which are intersected (distribution axioms) and recursively compared. If no relevant components are present, A cannot be a subtype. This is in contrast to the rule for arrows ($\text{STEP}_\rightarrow$), which allows B_2 to be ω in which case no restrictions have to be imposed on A because of the subtype rule ($\rightarrow \omega$). The contra-variant nature of arrow sources requires ($\text{STEP}_\rightarrow$) to additionally filter relevant components using ($\text{STEP}_{\text{ChooseTgt}}$) and ($\text{STEP}_{\text{DoneTgt}}$). This can be illustrated by showing $(A_1 \rightarrow B_1) \cap (A_2 \rightarrow B_2) \leq (A_1 \cap A_2) \rightarrow (B_1 \cap B_2)) \not\leq A_1 \rightarrow B_1 \cap B_2$ for $A_1 \not\leq A_2$.

The next part of the formalization, Lemma 1, is to prove that $\rightsquigarrow$ is functional.

Lemma 1 (Functionality). *For all instructions $i \in \mathcal{I}$ and outputs $o_1, o_2 \in \mathcal{O}$, if $i \rightsquigarrow o_1$ and $i \rightsquigarrow o_2$ then $o_1 = o_2$.*

Proof. Induction on the proof of $i \rightsquigarrow o_1$ followed by case analysis on $i \rightsquigarrow o_2$. □

The final step in obtaining an algorithm is to design a termination certificate for each instruction. The certificate Dom is inductively defined by the rules:

$$\frac{}{[\ \texttt{subty}\ A\ \texttt{of}\ \omega] \in \text{Dom}} \qquad \frac{\boxed{1}\ [\ \texttt{subty}\ \bigcap_{A_i \in \text{cast}_{c(B)}(A)} A_i\ \texttt{of}\ B] \in \text{Dom}}{[\ \texttt{subty}\ A\ \texttt{of}\ c(B)] \in \text{Dom}}$$

$$\frac{\begin{array}{c}\boxed{1}\ [\ \texttt{tgt\_for\_srcs\_gte}\ B_1\ \texttt{in}\ \text{cast}_{B_1 \rightarrow B_2} A] \in \text{Dom} \\ \boxed{2}\ \text{for all } \Delta,\ \text{if } [\ \texttt{tgt\_for\_srcs\_gte}\ B_1\ \texttt{in}\ \text{cast}_{B_1 \rightarrow B_2} A] \rightsquigarrow [\ \texttt{check\_tgt}\ \Delta] \\ \text{then } [\ \texttt{subty}\ \bigcap_{A_i \in \Delta} A_i\ \texttt{of}\ B_2] \in \text{Dom}\end{array}}{[\ \texttt{subty}\ A\ \texttt{of}\ B_1 \rightarrow B_2] \in \text{Dom}}$$

$$\frac{\begin{array}{c}\boxed{1}\ [\ \texttt{subty}\ B\ \texttt{of}\ A.1] \in \text{Dom} \\ \boxed{2}\ [\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ \Delta] \in \text{Dom}\end{array}}{[\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ [::A\ \&\ \Delta]] \in \text{Dom}}$$

$$\frac{}{[\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ [::]] \in \mathrm{Dom}}$$

$$\frac{\boxed{1}\ [\ \texttt{subty}\ \bigcap_{A_i \in \mathrm{cast}_{B_1 \times B_2}(A)} A_i.1\ \texttt{of}\ B_1] \in \mathrm{Dom} \quad \boxed{2}\ [\ \texttt{subty}\ \bigcap_{A_i \in \mathrm{cast}_{B_1 \times B_2}(A)} A_i.2\ \texttt{of}\ B_2] \in \mathrm{Dom}}{[\ \texttt{subty}\ A\ \texttt{of}\ B_1 \times B_2] \in \mathrm{Dom}}$$

$$\frac{\boxed{1}\ [\ \texttt{subty}\ A\ \texttt{of}\ B_1] \in \mathrm{Dom} \quad \boxed{2}\ [\ \texttt{subty}\ A\ \texttt{of}\ B_2] \in \mathrm{Dom}}{[\ \texttt{subty}\ A\ \texttt{of}\ B_1 \cap B_2] \in \mathrm{Dom}}$$

Analyzing Dom yields a new termination certificate for each recursive call needed by the subtype machine. In Coq, Dom is defined as an inductive datatype, which ensures that termination certificates for recursive calls are structurally smaller in each step. In the following text for a proof $p : i \in \mathrm{Dom}$ function inv_k is used to obtain premise $\boxed{\mathrm{i}}$ of the proof.

The denotational interpretation is defined in Fig. 1 and exactly follows each step of the subtype relation. The range restriction $\{o \in \mathcal{O} \mid i \rightsquigarrow o\}$ ensures soundness wrt. the relational semantics. On paper it has to be checked manually, while in Coq a Σ-type is used to attach proofs of $i \rightsquigarrow p$ to each function result. Completeness follows from Lemmas 1 and 2, which ensures that every possible instruction gives rise to an instance of the termination certificate.

Lemma 2 (Totality). *For all instructions $i \in \mathcal{I}$, $i \in \mathrm{Dom}$.*

Proof. Induction on B to obtain $[\ \texttt{subty}\ \omega\ \texttt{of}\ B] \in \mathrm{Dom}$.
Then for $[\ \texttt{subty}\ A\ \texttt{of}\ B]$ induction on the maximal depth of A and B followed by induction on the structure of B. Either cast decreases the depth of compared components, or it returns an empty list or a list only containing ω or (ω, ω). In the first case, an induction hypothesis can be used and in the second case $[\ \texttt{subty}\ \omega\ \texttt{of}\ B] \in \mathrm{Dom}$ can be used. Using the prior result, for $[\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ \Delta]$ simple induction on the length of list Δ is sufficient. □

When extracting the above specification from Coq to OCaml or Haskell, the termination certificate $i \in \mathrm{Dom}$ and the soundness proof $i \rightsquigarrow o$ are automatically erased, because these languages do not require termination certificates or proofs. Also uses of cast are surrounded by type-casts in the target language (`Obj.magic`, `unsafeCoerce`) since neither OCaml nor Haskell can natively express the type dependency between the input and output of cast which is encoded by function arity. In all other aspects, the extracted code exactly follows the specification up to syntax, which is why we elide it here and refer to the online sources.

It remains to show, that the machine specification is correct wrt. the BCD subtype relation. First, some properties of $\leq$ are established in Lemma 3.

Lemma 3 (Properties of the BCD-Relation).

1. $\text{intersect}(\text{map}(\lambda A_i.M, [::A_1 \;\&\; [::A_2 \;\&\; \ldots [::A_n \;\&\; \Delta]]])) \leq$
   $\text{intersect}(\text{map}(\lambda A_i.M, [::A_1 \;\&\; [::A_2 \;\&\; \ldots [::A_n \;\&\; [::]]]])) \cap$
   $\text{intersect}(\text{map}(\lambda A_i.M, \Delta))$
2. $\text{isOmega}(B)$ *implies* $A \leq B$
3. $\text{cast}_{c(B)}(A) \neq [::]$ *implies* $A \leq c(\bigcap_{A_i \in \text{cast}_{c(B)}(A)} A_i)$
4. $A \leq \bigcap_{A_i \in \text{cast}_{B_1 \to B_2}(A)} (A_i.1 \to A_i.2)$
5. $A \leq \bigcap_{A_i \in \text{cast}_{B_1 \times B_2}(A)} (A_i.1 \times A_i.2)$
6. $(A_1 \to B_1) \cap (A_2 \to B_2) \leq (A_1 \cap A_2) \to (B_1 \cap B_2)$

$$\text{subtypes}(i \in \mathcal{I}) : i \in \text{Dom} \to \{p \in \mathcal{O} \mid i \rightsquigarrow p\}$$

$$\text{subtypes}(i)(p) = \begin{cases}
[\ \texttt{Return}\ \mathit{true}] \text{ if } i = [\ \texttt{subty}\ A\ \texttt{of}\ \omega] \\
[\ \texttt{Return}\ \text{cast}_{c(B)}(A) \neq [::] \wedge b] \\
\quad \text{if } i = [\ \texttt{subty}\ A\ \texttt{of}\ c(B)] \text{ and} \\
\quad \text{for } A' := \bigcap_{\text{cast}_{c(B)}(A)} A_i \\
\quad \text{subtypes}([\ \texttt{subty}\ A'\ \texttt{of}\ B])(\text{inv}_1(p)) = [\ \texttt{Return}\ b] \\
[\ \texttt{Return}\ \text{isOmega}(B_2) \vee b] \\
\quad \text{if } i = [\ \texttt{subty}\ A\ \texttt{of}\ B_1 \to B_2] \text{ and} \\
\quad \text{for } A_1 := \text{cast}_{B_1 \to B_2}(A) \\
\quad \text{subtypes}([\ \texttt{tgt_for_srcs_gte}\ B_1\ \texttt{in}\ A_1])(\text{inv}_1(p)) = \\
\qquad [\ \texttt{check_tgt}\ \Delta] \text{ and} \\
\quad \text{for } A_2 := \bigcap_{A_i \in \Delta} A_i \\
\quad \text{subtypes}([\ \texttt{subty}\ A_2\ \texttt{of}\ B_2])(\text{inv}_2(p)) = [\ \texttt{Return}\ b] \\
[\ \texttt{Return}\ \text{cast}_{B_1 \times B_2}(A) \neq [::] \wedge b_1 \wedge b_2] \\
\quad \text{if } i = [\ \texttt{subty}\ A\ \texttt{of}\ B_1 \times B_2] \text{ and} \\
\quad \text{for } A_1 := \bigcap_{A_i \in \text{cast}_{B_1 \times B_2}(A)} A_i.1 \\
\quad \text{subtypes}([\ \texttt{subty}\ A_1\ \texttt{of}\ B_1])(\text{inv}_1(p)) = [\ \texttt{Return}\ b_1] \text{ and} \\
\quad \text{for } A_2 := \bigcap_{A_i \in \text{cast}_{B_1 \times B_2}(A)} A_i.2 \\
\quad \text{subtypes}([\ \texttt{subty}\ A_2\ \texttt{of}\ B_2])(\text{inv}_2(p)) = [\ \texttt{Return}\ b_2] \\
[\ \texttt{Return}\ b_1 \wedge b_2] \\
\quad \text{if } i = [\ \texttt{subty}\ A\ \texttt{of}\ B_1 \cap B_2] \text{ and} \\
\quad \text{subtypes}([\ \texttt{subty}\ A\ \texttt{of}\ B_1])(\text{inv}_1(p)) = [\ \texttt{Return}\ b_1] \text{ and} \\
\quad \text{subtypes}([\ \texttt{subty}\ A\ \texttt{of}\ B_2])(\text{inv}_2(p)) = [\ \texttt{Return}\ b_2] \\
[\ \texttt{check_tgt}\ \ \text{if } b \text{ then } [::A.2 \;\&\; \Delta'] \text{ else } \Delta'] \\
\quad \text{if } i = [\ \texttt{tgt_for_srcs_gte}\ B\ \texttt{in}\ [::A \;\&\; \Delta]] \text{ and} \\
\quad \text{subtypes}([\ \texttt{subty}\ B\ \texttt{of}\ A.1])(\text{inv}_1(p)) = [\ \texttt{Return}\ b] \text{ and} \\
\quad \text{subtypes}([\ \texttt{tgt_for_srcs_gte}\ B\ \texttt{in}\ \Delta])(\text{inv}_2(p)) = \\
\qquad [\ \texttt{check_tgt}\ \Delta'] \\
[\ \texttt{check_tgt}\ [::]] \text{ if } i = [\ \texttt{tgt_for_srcs_gte}\ B\ \texttt{in}\ [::]]
\end{cases}$$

Fig. 1. Interpreter for the subtype machine

7. $\bigcap_{A_i \in \Delta}(A_i.1 \times A_i.2) \leq \quad (\bigcap_{A_i \in \Delta} A_i.1) \times (\bigcap_{A_i \in \Delta} A_i.2)$ *if* $\Delta \neq [::]$

Proof. Easy induction and case analysis. □

Now soundness, which is the easier part of the correctness proof, can be shown.

Lemma 4 (Soundness). *Relation* $\rightsquigarrow$ *is sound wrt. relation* $\leq$*, i.e.* [`subty` A `of` B] $\rightsquigarrow$ [`Return true`] *implies* $A \leq B$.

Proof. First induction on the depth maximum of the depths of types A and B. Then induction on the derivation of [`subty` A `of` B] $\rightsquigarrow$ [`Return true`]. The induction hypotheses generated by the second induction are strong enough to solve the cases when B is ω, $c(B)$, $B_1 \times B_2$, and $B_1 \cap B_2$ with the help of Lemma 3. For rule (Step$_{\rightarrow}$) with $B = B_1 \rightarrow B_2$, the outer induction hypothesis is required to allow for the covariant position of B_1. The proof requires using a transitive step with an intersection of arrows selected by executing [`tgt_for_srcs_gte` $B1$ `in` $\text{cast}_{B_1 \rightarrow B_2} A$] as the center element. It then succeeds using Lemmas 3.4, 3.6, the induction hypotheses and an extra case-analysis for isOmega(B_2), and $\text{cast}_{B_1 \rightarrow B_2} A = [::]$. □

The converse direction, completeness, is more difficult to prove. It follows from Lemma 5, the sub-cases of which should be proven in the order they are presented.

Lemma 5 (Properties of the subtype machine).

1. isOmega(B) *implies* [`subty` A `of` B] $\rightsquigarrow$ [`Return true`]
2. [`tgt_for_srcs_gte` $B1$ `in` Δ] $\rightsquigarrow$ [`check_tgt` Δ'] *implies* $\Delta' \sqsubseteq \text{map}(\lambda A_i.A_i.2, \Delta)$.
3. [`tgt_for_srcs_gte` B_1 `in` $\text{cast}_{B_1 \rightarrow B_2} A$] $\rightsquigarrow$ [`check_tgt` Δ] *and* isOmega(A) *implies* isOmega(A_i) *for all* A_i *in* Δ
4. [`subty` A `of` B] $\rightsquigarrow$ [`Return true`] *and* isOmega(A) *implies* isOmega(B)
5. $\Delta_2 \sqsubseteq \Delta_1$ *and* [`tgt_for_srcs_gte` B_1 `in` Δ_1] $\rightsquigarrow$ [`check_tgt` Δ_1'] *and* [`tgt_for_srcs_gte` B_1 `in` Δ_2] $\rightsquigarrow$ [`check_tgt` Δ_2'] *implies* $\Delta_2' \sqsubseteq \Delta_1'$
6. $\Delta \sqsubseteq \Delta'$ *and* [`subty` $\bigcap_{A_i \in \Delta} A_i$ `of` A] $\rightsquigarrow$ [`Return true`] *implies* [`subty` $\bigcap_{B_i \in \Delta'} B_i$ `of` A] $\rightsquigarrow$ [`Return true`]
7. $\Delta_1 = [::A_1$ & $[::A_2$ & $\ldots [::A_n$ & $[::]]]]$ *for* $n \geq 0$ *and* [`subty` A `of` $\bigcap_{A_i \in \Delta_1} A_i$] $\rightsquigarrow$ [`Return` b_1] *and* [`subty` A `of` $\bigcap_{A_i \in \Delta_2} A_i$] $\rightsquigarrow$ [`Return` b_2] *implies* [`subty` A `of` $\bigcap_{A_i \in \Delta_3} A_i$] $\rightsquigarrow$ [`Return` $b_1 \wedge b_2$] *for* $\Delta_3 = [::A_1$ & $[::A_2$ & $\ldots [::A_n$ & $\Delta_2]]]$
8. [`subty` A `of` B] $\rightsquigarrow$ [`Return true`] *implies* [`subty` $\bigcap_{A_i \in \text{cast}_{c(C)} A} A_i$ `of` $\bigcap_{B_i \in \text{cast}_{c(C)} B} B_i$] $\rightsquigarrow$ [`Return true`]
9. [`tgt_for_srcs_gte` B `in` $[::(\omega, A)]$] $\rightsquigarrow$ [`check_tgt` Δ] *implies* $\Delta = [::A]$
10. [`subty` A `of` A] $\rightsquigarrow$ [`Return true`]

11. $\Delta_1 = [::A_1 \,\&\, [::A_2 \,\&\, \ldots [::A_n \,\&\, [::]]]]$ *for* $n \geq 0$ *and*
$\Delta'_1 = [::A'_1 \,\&\, [::A'_2 \,\&\, \ldots [::A'_m \,\&\, [::]]]]$ *for* $m \geq 0$ *and*
$[\ \texttt{tgt\_for\_srcs\_gte}\ A\ \texttt{in}\ \Delta_1] \rightsquigarrow [\ \texttt{check\_tgt}\ \Delta'_1]$ *and*
$[\ \texttt{tgt\_for\_srcs\_gte}\ A\ \texttt{in}\ \Delta_2] \rightsquigarrow [\ \texttt{check\_tgt}\ \Delta'_2]$ *implies*
$[\ \texttt{tgt\_for\_srcs\_gte}\ A\ \texttt{in}\ \Delta_3] \rightsquigarrow [\ \texttt{check\_tgt}\ \Delta'_3]$ *for*
$\Delta_3 = [::A_1 \,\&\, [::A_2 \,\&\, \ldots [::A_n \,\&\, \Delta_2]]]$ *and*
$\Delta'_3 = [::A'_1 \,\&\, [::A'_2 \,\&\, \ldots [::A'_m \,\&\, \Delta'_2]]]$
12. $[\ \texttt{subty}\ A\ \texttt{of}\ B_1 \cap B_2] \rightsquigarrow [\ \texttt{Return true}]$ *implies* $[\ \texttt{subty}\ A\ \texttt{of}\ B_1] \rightsquigarrow [\ \texttt{Return true}]$ *and* $[\ \texttt{subty}\ A\ \texttt{of}\ B_2] \rightsquigarrow [\ \texttt{Return true}]$
13. $C = c(C')$ *or* $C = C_1 \times C_2$ *and* $\text{cast}_C B \neq [::]$ *and* $[\ \texttt{subty}\ A\ \texttt{of}\ B] \rightsquigarrow [\ \texttt{Return true}]$ *implies* $\text{cast}_C A \neq [::]$
14. $[\ \texttt{subty}\ A\ \texttt{of}\ B] \rightsquigarrow [\ \texttt{Return true}]$ *and*
$[\ \texttt{subty}\ B\ \texttt{of}\ C] \rightsquigarrow [\ \texttt{Return true}]$ *implies*
$[\ \texttt{subty}\ A\ \texttt{of}\ C] \rightsquigarrow [\ \texttt{Return true}]$
15. $[\ \texttt{subty}\ a(A_1) \cap a(A_2)\ \texttt{of}\ a(A_1 \cap A_2)] \rightsquigarrow [\ \texttt{Return true}]$
16. $[\ \texttt{subty}\ A \cap A\ \texttt{of}\ A] \rightsquigarrow [\ \texttt{Return true}]$

where list Δ *is a sublist of* Δ', $\Delta \sqsubseteq \Delta'$, *if* $\Delta' = [::A_1 \,\&\, [::A_2 \,\&\, \ldots [::A_n \,\&\, [::B_1 \,\&\, [::B_2 \,\&\, \ldots [::B_m \,\&\, \Delta'']]]]]]$ *and* $\Delta = [::B_1 \,\&\, [::B_2 \,\&\, \ldots [::B_m]]]$ *for some* $n, m \geq 0$.

Proof. Mostly straightforward induction, case-analysis, and applying the previously proven facts. The weakening property expressed in 6 is inspired by [20] and is crucial for the proof of the reflexivity property 10 in the case for $A = A_1 \cap A_2$. Just as in sequent calculus, the proof of the transitivity property 5 is complicated. Similar to the proof of Lemma 4 it needs nested induction on the maximal depth of types A and C, and then on the structure of C. The cases for constructors, products, and targets of arrows need an additional nested induction on the left proof. The case for collecting sources needs an additional induction on the casted type A. □

Lemma 6 (Completeness). *Relation* $\rightsquigarrow$ *is complete wrt. relation* $\leq$, *i.e.* $A \leq B$ *implies* $[\ \texttt{subty}\ A\ \texttt{of}\ B] \rightsquigarrow [\ \texttt{Return true}]$.

Proof. By induction on the proof of $A \leq B$. The cases are either immediate, or, like (Trans) follow from Lemma 5 and the induction hypothesis. □

Logical properties of the semantic relation also hold for the interpreter, allowing to proof that subtypes of Fig. 1 really is a decision procedure for the BCD subtype relation.

Theorem 1 (Correctness). *Function* subtypes *of Fig. 1 is a correct decision procedure for BCD subtyping, i.e.: for all types* A *and* B, *there exists a proof* $p : [\ \texttt{subty}\ A\ \texttt{of}\ B] \in \text{Dom}$ *and* $A \leq B$ *if and only if* $\text{subtypes}([\ \texttt{subty}\ A\ \texttt{of}\ B])(p) = [\ \texttt{Return true}]$.

Proof. The termination certificate p always exists because of Lemma 2. Elements r in the image of $\text{subtypes}([\ \texttt{subty}\ A\ \texttt{of}\ B])(p)$ are those which satisfy

[`subty` A `of` B] $\rightsquigarrow r$. This implies $A \leq B$ by Lemma 4. Lemma 6 allows to deduce [`subty` A `of` B] $\rightsquigarrow r$ from $A \leq B$. Lemma 1 implies that r is equal to subtypes([`subty` A `of` B])(p). □

5 Quadratic Runtime

The termination certificate Dom ensures that function subtypes does not loop and can also be used to put an upper bound on the function runtime. To this end, Dom is indexed with the size of its proof. Formally:

$$p \in \mathrm{Dom}_{1+n_1+n_2} \text{ iff } p \in \mathrm{Dom} \text{ and for } k = 1, 2 : \\ \mathrm{inv}_k(p) \in \mathrm{Dom}_{n_k} \text{ or } \mathrm{inv}_k(p) \text{ does not exist and } n_k = 0.$$

It is easy to check by induction, that every termination certificate $p : i \in \mathrm{Dom}$ is also valid for Dom_n and the opposite holds by definition. For a termination certificate $p : i \in \mathrm{Dom}_n$, subtypes can perform no more than n recursive steps: all recursive calls are started with the result of inv_k and every certificate is only used once. The next lemma establishes a bound on n for any instruction i.

Lemma 7 (Domain size). *For any i, n: if $p : i \in \mathrm{Dom}_n$ exists, then $n \leq \mathrm{cost}(i)$, where*

$$\mathrm{cost}(i) = \begin{cases} 2 \cdot \|A\| \cdot \|B\| \text{ if } i = [\ \texttt{subty}\ A\ \texttt{of}\ B] \\ 1 + \|B\| \cdot \sum_{i=1}^{k} (1 + 2 \cdot \|A_i.1\|) \\ \quad \text{if } i = [\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ [::A_1\ \&\ [::A_2\ \&\ [::\ldots\ \&\ [::A_k]]]]] \end{cases}$$

Proof. By induction and case-analysis on casted types, their size is less or equal to the size before casting. The only exception is the case $\mathrm{cast}_{B_1 \to B_2} A$ if isOmega($B2$) is true and the size of A is less than 3. Now the lemma follows by induction on p, taking care of the special case by observing that its minimal value for cost is 3, which is enough to bound the proof-tree. □

Bounding the size of Dom is the most fine-grained analysis possible in a formally verified way. In the present formalization this only limits the number of recursive steps, but not the amount of time spent on "primitive" operations such as cast. In future work, these could be removed and their stepwise execution made part of the instruction set. For now, the cost-annotations on the (Step)-rules allow for a more detailed, albeit manual, analysis. Every rule has overhead and recursive costs. Overhead costs are stated in $\mathbb{O}$-Notation and recursive costs are given as variables m and n. If present, the recursive costs can grow up to $\mathbb{O}(\|A\| \cdot \|B\|)$ by the prior argument about Dom. This always dominates over the overhead costs, which are constant or bound by the sum of the breadth and length of the types, since $\mathrm{breadth}(A) \leq \|A\|$ and $\mathrm{length}(B) \leq \|B\|$. Hence, $\mathbb{O}(\|A\| \cdot \|B\|)$ is an upper bound on the runtime of the presented algorithm.

6 Conclusion and Future Work

A procedure to decide the BCD subtyping relation of intersection types has been presented and formalized. The BCD relation is at the core of countless extended subtype systems [7,8,11,16]. Advances in formalized procedures for its decision may one day help to deal with the current undecidability issues in the type systems of modern programming languages [12,17]. The formalization is based on the Coq theorem-prover and makes use of a technique newly introduced by Larchey-Wendling and Monin [19]. The asymptotic complexity of the algorithm is in $\mathbb{O}(n^2)$ and thereby on a par with the currently best known result presented by Dudenhefner et al. in 2017 [10]. In contrast to the former algorithm it does not require preprocessing types. This avoids redundant work if types are large and requests fails early, which is often the case during proof search. It makes it a candidate for future integration into the (CL)S framework [5]. The formalization can be extracted to purely functional executable OCaml or Haskell code, which closely matches its specification. The key properties of soundness and completeness wrt. the BCD subtype relation could be proven without referring to any termination arguments. Additionally, a formalized proof for a bound on the number of recursive calls was enabled by the technique. Proving asymptotic bounds with theorem provers is an active field of study [1,13,15]. The relatively easy established bound on recursive calls might be interesting for its further development. In reverse, the $\mathbb{O}(n^2)$ runtime result could be proven in Coq using one of the aforementioned results. All current attempts without adding another heavy weight framework were hindered by the tediousness of solving inequations performing rewrite steps (especially for associativity) in the theorem prover by hand. Here, clearly more automation would help. The currently experimental early stage project of making to the existing automation compatible with the mathematical components library [21], which was employed in the proof, has great potential benefit.

Acknowledgments. The authors would like to thank Olivier Laurent, as well as Andrej Dudenhefner, Tristan Schäfer, Anna Vasileva, and Jan Winkels for the prior work, and patient as well as enlightening discussions without which the results in this paper would have been impossible.

References

1. Avigad, J., Donnelly, K.: Formalizing O notation in Isabelle/HOL. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 357–371. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25984-8_27
2. van Bakel, S.: Complete restrictions of the intersection type discipline. Theor. Comput. Sci. **102**(1), 135–163 (1992). https://doi.org/10.1016/0304-3975(92)90297-S
3. Barendregt, H.P., Dekkers, W., Statman, R.: Lambda Calculus with Types. Perspectives in logic. Cambridge University Press (2013). http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types

4. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment. J. Symb. Log. **48**(4), 931–940 (1983). https://doi.org/10.2307/2273659
5. Bessai, J., Dudenhefner, A., Düdder, B., Martens, M., Rehof, J.: Combinatory logic synthesizer. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8802, pp. 26–40. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45234-9_3
6. Bessai, J., Dudenhefner, A., Düdder, B., Rehof, J.: Extracting a formally verified Subtyping algorithm for intersection types from ideals and filters. Types (2016)
7. Bi, X., Oliveira, B.C.d.S., Schrijvers, T.: The essence of nested composition. In: 32nd European Conference on Object-Oriented Programming, ECOOP 2018, Amsterdam, The Netherlands, 16–21 July 2018, pp. 22:1–22:33 (2018). https://doi.org/10.4230/LIPIcs.ECOOP.2018.22
8. Damm, F.M.: Subtyping with union types, intersection types and recursive types. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 687–706. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57887-0_121
9. Damm, F.M.: Subtyping with union types, intersection types and recursive types II. Ph.D. thesis, INRIA (1994)
10. Dudenhefner, A., Martens, M., Rehof, J.: The algebraic intersection type unification problem. Log. Methods Comput. Sci. **13**(3) (2017). https://doi.org/10.23638/LMCS-13(3:9)2017
11. Dunfield, J.: A unified system of type refinements. Ph.D. thesis, Carnegie Mellon University (2007)
12. Grigore, R.: Java generics are turing complete. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 73–85 (2017). http://dl.acm.org/citation.cfm?id=3009871
13. Guéneau, A., Charguéraud, A., Pottier, F.: A fistful of dollars: formalizing asymptotic complexity claims via deductive program verification. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 533–560. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_19
14. Hindley, J.R.: The simple semantics for Coppo-Dezani-Sallé types. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 212–226. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_15
15. Hoffmann, J., Das, A., Weng, S.: Towards automatic resource bound analysis for OCaml. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 359–373 (2017). http://dl.acm.org/citation.cfm?id=3009842
16. Honsell, F., Liquori, L., Stolze, C., Scagnetto, I.: The Delta-framework. CoRR abs/1808.04193 (2018). http://arxiv.org/abs/1808.04193
17. Kennedy, A., Pierce, B.C.: On decidability of nominal subtyping with variance. In: International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD), January 2007
18. Kurata, T., Takahashi, M.: Decidable properties of intersection type systems. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 297–311. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0014060
19. Larchey-Wendling, D., Monin, J.F.: Simulating induction-recursion for partial algorithms. In: TYPES (2018)
20. Laurent, O.: Intersection subtyping with constructors. In: Pagani, M. (ed.) Proceedings of the Ninth Workshop on Intersection Types and Related Systems (2018)

21. Magaud, N.: Transferring arithmetic decision procedures (on Z) to alternative representations. In: CoqPL 2017: The Third International Workshop on Coq for Programming Languages (2017)
22. Pierce, B.C.: A decision procedure for the subtype relation on intersection types with bounded variables. Citeseer (1989)
23. Rehof, J., Urzyczyn, P.: Finite combinatory logic with intersection types. In: Ong, L. (ed.) TLCA 2011. LNCS, vol. 6690, pp. 169–183. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21691-6_15
24. Statman, R.: A finite model property for intersection types. In: Proceedings Seventh Workshop on Intersection Types and Related Systems, ITRS 2014, Vienna, Austria, 18 July 2014, pp. 1–9 (2014). https://doi.org/10.4204/EPTCS.177.1
25. Steffen, B., Cleaveland, R.: When is "partial" adequate? A logic-based proof technique using partial specifications. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS 1990), Philadelphia, Pennsylvania, USA, 4–7 June 1990, pp. 440–449 (1990). https://doi.org/10.1109/LICS.1990.113768