Type-Theoretic Galois Connections*

Pierre-Évariste Dagand
Sorbonne Universités
UPMC Univ Paris 06
CNRS—Inria—LIP6 UMR 7606
pierre-evariste.dagand@lip6.fr

Nicolas Tabareau
Inria
nicolas.tabareau@inria.fr

Éric Tanter
PLEIAD lab
Computer Science Dept (DCC)
University of Chile
etanter@dcc.uchile.cl

We observe that one approach to address the compilation of source programs working with rich(er) types to target programs with weak(er) types is to formally relate both source and target types via a *type-theoretic partial Galois connection* that accounts for the semantic correspondence between both. The partiality of the connection reflects the potential for failure when attempting to derive a corresponding value at a source type given a value at a target type.

We explore the applicability of type-theoretic Galois connections in the specific setting of *dependent interoperability*: trading static guarantees for runtime checks, a dependent interoperability framework provides a mechanism by which simply-typed values can safely be coerced to dependent types and, conversely, dependently-typed programs can defensively be exported to a simply-typed application.

In this setting, a traditional monotone type-theoretic Galois connections enforces a translation of dependent types to runtime checks that is both sound and complete with respect to the invariants encoded by dependent types. We also study a variant, dubbed *anticonnection*, which lets us induce weaker sound conditions that can amount to more efficient runtime checks.

Using our Coq framework, users can specify domain-specific partial connections between data structures. The library takes care of the (heavy) lifting that leads to interoperable programs. It thus becomes possible to internalize and hand-tune the extraction of dependently-typed programs to interoperable OCaml programs within Coq itself.

Methodology We motivate our formalism, *type-theoretic (partial) Galois connections*, through the familiar relation between the dependently-typed $\mathtt{Vec}\ \mathbb{N}\ \mathtt{n}$ and its simply-typed counterpart $\mathtt{List}\ \mathbb{N}$. Our objective is to define a pair of (partial) functions mediating between vectors and lists, allowing us to switch safely between both representations. We also want to precisely characterize the relationship between both functions.

We notice that there is a total embedding

$$\mathtt{forget}\ \mathtt{n} : \mathtt{Vec}\ \mathbb{N}\ \mathtt{n} \to \mathtt{List}\ \mathbb{N}$$

from vectors of size n to lists. The challenge is thus to define a (necessarily partial) function mapping lists to vectors. To do so, we adopt a two-step process. First, we exploit the standard notion of *type equivalence* to relate the type $\mathtt{Vec}\ \mathbb{N}\ \mathtt{n}$ to its image by $\mathtt{forget}$:

$$\mathtt{im}\ \mathtt{forget}\ \mathtt{n} := \{\, \mathtt{l} : \mathtt{List}\ \mathbb{N}\ \&\ \exists\, \mathtt{v} : \mathtt{Vec}\ \mathbb{N}\ \mathtt{n},\ \mathtt{forget}\ \mathtt{n}\ \mathtt{v} = \mathtt{l} \,\}$$

that explicitly segregates the computational content of vectors—"a list" —from its logical content—"whose length is equal to n". Indeed, the property $\exists\, \mathtt{v} : \mathtt{Vec}\ \mathbb{N}\ \mathtt{n},\ \mathtt{forget}\ \mathtt{n}\ \mathtt{v} = \mathtt{l}$ is logically equivalent to the property $\mathtt{length}\ \mathtt{l} = \mathtt{n}$.

Because $\mathtt{forget}$ is an embedding, the restriction of $\mathtt{forget}$ to its image establishes a type equivalence (denoted $\simeq$) between an inductive family and a subset type:

$$\mathtt{Vec}\ \mathbb{N}\ \mathtt{n} \simeq \mathtt{im}\ \mathtt{forget}\ \mathtt{n} \qquad \text{for all } n \in \mathbb{N}$$

Second, we must relate the subset type $\mathtt{im}\ \mathtt{forget}\ \mathtt{n}$ with the simple type $\mathtt{List}\ \mathbb{N}$. Once again, there is a total embedding from the dependent type to the simple one: it is in fact the first projection of the Σ-type! Conversely, going from simple types to subset types is a potentially partial operation. We model partiality using the standard monadic framework with a monadic composition $\circ_K$, an identity $\mathtt{creturn}$, a lifting $\mathtt{lift}$ from pure to monadic operations and a flat partial order—*i.e.* $\bot \leq \mathtt{a}$ and $\mathtt{Some}\ \mathtt{a} \leq \mathtt{Some}\ \mathtt{b}$ *iff* $\mathtt{a} = \mathtt{b}$.

The back-translation from lists to $\mathtt{im}\ \mathtt{forget}\ \mathtt{n}$ is thus a partial function, denoted with the arrow $\rightharpoonup$:

$$\mathtt{make}\ \mathtt{n} : \mathtt{List}\ \mathbb{N} \rightharpoonup \{\mathtt{l} : \mathtt{List}\ \mathbb{N}\ \&\ \exists\, \mathtt{v} : \mathtt{Vec}\ \mathbb{N}\ \mathtt{n},\ \mathtt{forget}\ \mathtt{v} = \mathtt{l}\}$$

that may fail, *esp.* if the length of the input list is different from n:

$$\begin{aligned} \forall\, \mathtt{n}, \forall\, \mathtt{l} : \mathtt{List}\ \mathbb{N}, \quad & ((\mathtt{lift}\ \pi_1) \circ_K (\mathtt{make}\ \mathtt{n}))\ \mathtt{l} = \mathtt{Some}\ \mathtt{l} \\ & \vee\ ((\mathtt{lift}\ \pi_1) \circ_K (\mathtt{make}\ \mathtt{n}))\ \mathtt{l} = \mathtt{Fail} \end{aligned}$$

and must amount to an identity function when the input list is of the right length:

$$\forall\, \mathtt{n}, \forall\, \mathtt{l} : \mathtt{im}\ \mathtt{forget}\ \mathtt{n},\ \mathtt{Some}\ \mathtt{l} = \mathtt{make}\ \mathtt{n}\ (\pi_1\ \mathtt{l})$$

Translated in the monadic language, these two specifications become:

$$\begin{aligned} &\forall\, \mathtt{n}, (\mathtt{lift}\ \pi_1) \circ_K (\mathtt{make}\ \mathtt{n}) \leq \mathtt{creturn} \\ &\forall\, \mathtt{n}, \qquad\qquad\qquad\quad \mathtt{creturn} \leq (\mathtt{make}\ \mathtt{n}) \circ_K (\mathtt{lift}\ \pi_1) \end{aligned}$$

whereby we recognize a type-theoretic and partial version of a monotone Galois connection—or *partial connection* for short, conventionally written $\mathtt{A} \lesssim_K \mathtt{B}$.

Note that having $\mathtt{creturn} \leq \mathtt{f}$ means that $\mathtt{f}$ is total and is, in fact, the identity function; hence $\mathtt{A} \lesssim_K \mathtt{B}$ denotes a

*This work was partially funded by the CoqHoTT ERC Grant 637339, by FONDECYT Project 1150017 and the Émergence(s) program of Paris.

PriSC'18, January 08–13, 2018, Los Angeles, CA, USA
2018.

$$
\begin{array}{ccc}
\text{Connection} & \text{Type equivalence} & \text{Anticonnection} \\
\texttt{Connection A B} & \texttt{Equiv A B} & \texttt{Anticonnection A B} \\
A \lesssim B & A \simeq B & A \approx B \\
\downarrow & \swarrow \quad \searrow & \downarrow \\
\text{Partial connection} & & \text{Partial anticonnection} \\
\texttt{ConnectionK A B} & & \texttt{AnticonnectionK A B} \\
A \lesssim_K B & & A \approx_K B \\
\downarrow & & \downarrow \\
\text{First-order dependent connection} & & \text{First-order dependent anticonnection} \\
\texttt{DepConnection B C} & & \texttt{DepAnticonnection B C} \\
B \lesssim^{\square}_K C & & B \approx^{\square}_K C
\end{array}
$$

Figure 1. Bird-eye view of the dependent interoperability framework

Galois insertion. Here, this means that one direction (from subset type to simple type) never fails: given an index `n`, and a list enriched with the property that it corresponds to a vector of size `n`, projecting out the list and then attempting to reestablish that it corresponds to a vector of size `n` never fails (and gets back to the same value). This explains the directedness of partial connections: we have, for all index `n`, `im forget n` $\lesssim_K$ `List` $\mathbb{N}$.

This machinery enables us to account for first-order connections between dependent and simple datatypes (`Vec` $\mathbb{N}$ `n` $\lesssim_K$ `List` $\mathbb{N}$) by composition of a partial connection (`im forget n` $\lesssim_K$ `List` $\mathbb{N}$) and a type equivalence (`Vec` $\mathbb{N}$ `n` $\simeq$ `im forget n`). We can then abstract the index and establish a *dependent connection* between an indexed type and a simple type (`Vec` $\mathbb{N}$ $\lesssim^{\square}_K$ `List` $\mathbb{N}$).

To account for higher-order transformations, we must generalize partial connections to relate any *pointed partial order* through a pair of *monotone* functions subject to a similar adjunction property. We thus obtain a type-theoretic version of monotone Galois connections, or *connection* for short, written `A` $\lesssim$ `B`.

We also develop a symmetric variant of monotone Galois connections (where both composite functions are below the identity), which we dub *anticonnections*. An anticonnection, written `A` $\approx$ `B`, gives the freedom to arbitrarily abort a coercion—for example, if it turns out to be too costly to run. Similarly, a *partial anticonnection*, written `A` $\approx_K$ `B` allows us to relate a subset type that represents only a strict subset of its corresponding simple type—the translation from subset to simple types may thus be partial.

The overall view of the framework and its conceptual dependencies is summarized in Figure 1.

Relation to prior work. This presentation reports on an extension and refinement of an article that appeared in the proceedings of ICFP 2016 [2], accepted for publication at JFP [3]. The main novelties wrt the ICFP version are: (i) We have clarified the overall conceptual framework by identifying the structuring role of a type-theoretic form of Galois connections. (ii) We have been led to explore the classical monotone presentation of Galois connections, which provides tighter relations between programs, and to provide a separate treatment of anticonnections, which we had implicitly adopted in our earlier work. In turn, this leads to a rationalized presentation of *checkable* properties as the counterpart to *decidable* properties; (iii) We have made significant efforts to simplify our type-theoretic structures. In particular, defining partial orders in `HProp`, hence stating that ordering proofs are irrelevant, led to dramatic simplifications in the definitions of type connections, by removing the need for coherence conditions between sections and retractions.

Perspectives. Interestingly, the monotone framework seems to capture a notion of *type precision*, akin to that used in recent accounts of gradual typing [4, 6] to characterize the amount of static information that a type carries. This suggests that building a form of gradual typing on top of the dependent interoperability framework could be possible; and more generally, that monotone partial Galois insertions could serve as a type-theoretic foundation for various forms of gradual typing.

Directly relevant to the secure compilation research agenda is the extension of our approach to effects. This is particularly relevant when considering the interoperable extraction to OCaml, which features exceptions, memory cells and input/outputs. The extensive body of literature on type isomorphisms [1] may provide some useful guiding principles. Recently, Levy [5] provides a blueprint for adapting the notion of type isomorphism to a wide range of side-effects by segregating types between value types and computation types. It would be interesting to understand how this distinction fares in our dependently-typed setting.

Coq Formalization. The full Coq formalization, with documentation, is available at http://coqhott.github.io/DICoq/.

References

[1] Roberto Di Cosmo. 2005. A short survey of isomorphisms of types. *Mathematical Structures in Computer Science* 15, 5 (2005), 825–838.

[2] Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. 2016. Partial Type Equivalences for Verified Dependent Interoperability. In *Proceedings of the 21st ACM SIGPLAN Conference on Functional Programming (ICFP 2016)*. ACM Press, Nara, Japan, 298–310.

[3] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of Dependent Interoperability. *Journal of Functional Programming* (2018). To appear.

[4] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM Press, St Petersburg, FL, USA, 429–442.

[5] Paul Blain Levy. 2017. Contextual Isomorphisms. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 400–414.

[6] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. [n. d.]. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. 274–293.