ITRS 2020 - Abstracts

Ugo de' Liguoro (ed.)

Turin, March 2020

Foreword

This volume contains the abstracts of the talks accepted for presentation at the 10th International Conference on Intersection Types and Related Systems, ITRS 2020, in Turin, satellite event of TYPES 2020. Unfortunately, the circumstances have been not favorable; being planned on the 6th March 2020 in Turin, the workshop has not been held, because of the sanitary emergency caused by the spreading of the SARS-CoV-2 virus infection.

The ITRS workshop series aims to bring together researchers working on both the theory and practical applications of systems based on intersection types and related approaches. Topics for submitted papers include, but are not limited to:

- Formal properties of systems with intersection types.
- Results for related systems, such as union types, refinement types, or singleton types.
- Applications to lambda calculus, pi-calculus and similar systems.
- Applications for programming languages, program analysis, and program verification.
- Applications for other areas, such as database query languages and program extraction from proofs.
- Related approaches using behavioural/intensional types and/or denotational semantics to characterize computational properties.
- Quantitative refinements of intersection types.

The abstracts contained in this document were accepted for presentation to the Turin edition of the workshop, by the program committee formed by:

Ugo de' Liguoro (Turin University)

Jeremy Siek (Indiana University Bloomington)

Andrej Dudenhefner (Saarland University)

Antonio Bucciarelli (Université Paris Diderot)

Daniel de Carvalho (Innopolis University)

Kazushige Terui (Kyoto University)

Silvia Ghilezan (University of Novi Sad)

Ugo de' Liguoro

Turin, 1 March 2020

Contents

Kripke Semantics for Intersection Formulas

Andrej Dudenhefner
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
dudenhefner@ps.uni-saarland.de

Paweł Urzyczyn
Institute of Informatics, University of Warsaw, Warsaw, Poland
urzy@mimuw.edu.pl

Introduction Intersection types are broadly used in type assignment systems but are seldom understood as logical formulas. Exceptions include [4, 8], and the line of research on *intersection logic* [5, 6, 3], initiated by S. Ronchi Della Rocca and L. Roversi. The underlying *semantics* accompanying those efforts is proof-theoretical. However, if we think of intersection formulas as independent of term-assignment, we naturally ask for mathematical semantics of formulas that could be defined and investigated without any direct reference to λ-terms.

Here, we outline the first (to the authors' knowledge) attempt to define a sound and complete possible-world (Kripke) semantics for intersection logic. The approach develops from the idea of proof-search, or type inhabitation algorithm, understood as a game.

Adaptation of previous methods [7] to intersection logic is difficult for two reasons. First, for proof construction one must consider *parallel* inhabitation problems, where a single proof has to satisfy multiple constraints. This complicates proof syntax (we use matrices of formulas) and model definition. Second, intersection formulas may be non-uniform, e.g. $p \cap (q \to p)$, exhibiting "functional" *and* "atomic" behavior. To accommodate both, our models must satisfy a global condition of being "monotone".

Formulas and matrices *Formulas* (ranged over by σ, τ, ρ) are Barendregt–Coppo–Dezani (BCD) intersection types [1], i.e. $\sigma, \tau ::= p \mid \omega \mid \sigma \to \tau \mid \sigma \cap \tau$, where *atoms* are ranged over by p, q. The preorder $\leq$ is intersection type subtyping [1, Def. 2.3]. The intersection type constructor $(\cap)$ is assumed commutative, associative, and idempotent. We write $\tau \subseteq \sigma$ if $\sigma = \tau \cap \rho$. A formula σ is *functional* if $\sigma = \bigcap_{i \in I}(\sigma_i \to \tau_i)$, and we define $\mathrm{lhs}(\sigma) = \bigcap_{i \in I} \sigma_i$ and $\mathrm{rhs}(\sigma) = \bigcap_{i \in I} \tau_i$.

A *column* (ranged over by γ, δ, ν) of *height* m is a vector of m formulas. A column is called *functional* if all its coordinates are functional, otherwise it is *atomic*.

An $m \times n$-*matrix* (ranged over by Γ) of formulas σ_{ij}, where $i = 1 \ldots m$ and $j = 1 \ldots n$, is written $[\sigma_{ij}]_{j=1\ldots n}^{i=1\ldots m}$. If Γ is an $m \times n$-matrix, and γ is a column of height m, then Γ, γ stands for an $m \times (n+1)$-matrix obtained by adding γ as the $(n+1)$-st column.

Let $f : \{1, \ldots, m_2\} \to \{1, \ldots, m_1\}$ be onto. If $\gamma = (\sigma_1, \ldots, \sigma_{m_2})$, then $f(\gamma)$ is a column of height m_1 whose k-th coordinate is $\bigcap_{f(i)=k} \sigma_i$. And if $\delta = (\tau_1, \ldots, \tau_{m_1})$, then $f^{-1}(\delta)$ is a column of height m_2 whose j-th coordinate is $\tau_{f(j)}$. The notation f^{-1} extends to matrices columnwise.

We write $\Gamma_1 \sqsubseteq_f \Gamma_2$ if there exist columns $\gamma_1, \ldots, \gamma_k$ such that Γ_2 is $f^{-1}(\Gamma_1), \gamma_1, \ldots, \gamma_k$ up to column permutation. The relations $\leq, \subseteq$ and functions lhs, rhs extend to columns coordinatewise. If $\gamma \leq \delta$ (resp. $\delta \subseteq \gamma$), for some column γ of Γ, then we write $\Gamma \leq \delta$ (resp. $\delta \subseteq \Gamma$). If $\gamma = (\sigma_1, \ldots, \sigma_m)$ and $\delta = (\tau_1, \ldots, \tau_m)$, then $\gamma \Rightarrow \delta$ denotes the column $(\sigma_1 \to \tau_1, \ldots, \sigma_m \to \tau_m)$.

Submitted to:
ITRS 2020

Sequent calculus *Judgments* (cf. *molecules* of [6]) take the form $\Gamma \vdash \gamma$, where Γ is a matrix and γ is a column of the same height. The following sequent calculus is sound and complete (Proposition 1) for BCD inhabitation [1, Def. 2.5].

$$\frac{(\Gamma \leq \gamma)}{\Gamma \vdash \gamma}\,(\mathrm{A}) \qquad \frac{}{\Gamma \vdash (\omega,\dots,\omega)}\,(\Omega) \qquad \frac{\Gamma \vdash \gamma \quad \Gamma,\gamma \vdash \delta}{\Gamma \vdash \delta}\,(\mathrm{Cut})$$

$$\frac{\Gamma,\mathrm{rhs}(\gamma) \vdash \delta \quad \Gamma \vdash \mathrm{lhs}(\gamma) \quad (\gamma \subseteq \Gamma)}{\Gamma \vdash \delta}\,(\mathrm{L}) \qquad \frac{f^{-1}(\Gamma),\gamma \vdash \delta}{\Gamma \vdash f(\gamma \Rightarrow \delta)}\,(\mathrm{R})$$

For $\Gamma = [\sigma_{ij}]^{i=1\dots m}_{j=1\dots n}$ and $\gamma = (\tau_1,\dots,\tau_m)$, we write $\Gamma \vdash_{\mathrm{BCD}} \gamma$ if there exists a λ-term M such that $\{x_1 : \sigma_{i1},\dots,x_n : \sigma_{in}\} \vdash_{\mathrm{BCD}} M : \tau_i$, for $i = 1\dots m$.

Proposition 1. *We have* $\Gamma \vdash \gamma$ *iff* $\Gamma \vdash_{\mathrm{BCD}} \gamma$.

Kripke-style semantics We define a *Kripke model* $\mathcal{M} = \langle \mathcal{C},\leq,\mathcal{G},\mathcal{H}\rangle$, where: $\mathcal{C}$ is a nonempty set of *states*; $\leq$ is a partial order on $\mathcal{C}$; $\mathcal{G}$ is a function that assigns an atomic matrix Γ^C to every state $C \in \mathcal{C}$; $\mathcal{H}$ is a function that assigns a surjection f to every pair $C \leq D$, written $C \leq_f D$. Additionally, for every $C,D,E \in \mathcal{C}$ we require: if $C \leq_f D$, then $\Gamma^C \sqsubseteq_f \Gamma^D$; if $C \leq_f D \leq_g E$, then $C \leq_{g\circ f} E$; and $C \leq_{\mathrm{id}} C$.

Forcing: Let Γ^C and δ be of height m. The state C *forces* δ, written $C \Vdash \delta$, if either δ is equivalent to $(\omega,\dots,\omega)$ under subtyping, or one of the following holds:

- The column δ is atomic and $\Gamma^C \leq \delta$.
- The column δ is functional, and for all $D \in \mathcal{C}$ such that $C \leq_f D$ and for every column ν such that $\nu \subseteq f^{-1}(\delta)$ and $D \Vdash \mathrm{lhs}(\nu)$ we have $D \Vdash \mathrm{rhs}(\nu)$.

A model is *monotone*[1] if for every state C and γ such that $\Gamma^C \leq \gamma$, we have $C \Vdash \gamma$.

We write $C \Vdash \Gamma$ if C forces all columns in Γ. Finally, the notation $\Gamma \Vdash \gamma$ means that:

For every monotone model $\mathcal{M}$ *and every state* C*, if* $C \Vdash \Gamma$*, then* $C \Vdash \gamma$.

Example 2. *Let* $\delta = ((p \to \omega \to p) \cap (\omega \to p \to p))$ *be a column of height 1 and* $\mathcal{C} = \{1,2\}$*, where* $\Gamma^1 = ()$ *is a* 1×0*-matrix,* $\Gamma^2 = \binom{p\ \omega}{\omega\ p}$ *is a* 2×2*-matrix, and* $1 \leq_f 2$*, where* $f : \{1,2\} \to \{1\}$ *is such that* $f(1) = f(2) = 1$*. For* $\nu = \binom{p\to\omega\to p}{\omega\to p\to p}$*, we have that* $1 \leq_f 2$*,* $\nu \subseteq f^{-1}(\delta)$*, and* $2 \Vdash \mathrm{lhs}(\nu) = \binom{p}{\omega}$*. However, we have that* $2 \nVdash \mathrm{rhs}(\nu) = \binom{\omega\to p}{p\to p}$*, because* $2 \Vdash \mathrm{lhs}(\mathrm{rhs}(\nu)) = \binom{\omega}{p}$*, but* $2 \nVdash \mathrm{rhs}(\mathrm{rhs}(\nu)) = \binom{p}{p}$*.*

Overall, we have that $1 \nVdash \delta$*, providing a countermodel. Therefore,* $\emptyset \nVdash \delta$*. Besides,* $\emptyset \nvdash_{\mathrm{BCD}} \delta$*.*

Game playing The *game* is played by two competitors, who influence a game *position*, which is a judgment $\Gamma \vdash \delta$. The existential player, $\exists$ros, tries to prove δ from Γ, i.e. to reach a position in which $\delta = (\omega,\dots,\omega)$ or $\Gamma \leq \delta$. The universal player, $\forall$phrodite, attempts to refute his claims. Similarly to [7, Section 2], possible game moves essentially correspond to rules (L) and (R). Therefore, a winning $\exists$ros' strategy leads to a cut-free sequent calculus proof. Complementarily, from a winning $\forall$phrodite's strategy a Kripke countermodel can be constructed.

Theorem 3 (Soundness and Completeness)**.** *We have* $\emptyset \vdash_{\mathrm{BCD}} \sigma$ *iff* $\emptyset \Vdash (\sigma)$*.*

Proof Sketch. Soundness is shown by Proposition 1 followed by structural induction with respect to proofs. Completeness is shown by constructing a countermodel from a winning strategy of $\forall$phrodite using methods of [7, Section 2]. $\square$

[1] Model monotonicity is naturally satisfied if we restrict attention to formulas that have uniform structure, i.e. refine some simple type in the sense of [2], such as in Example 2.

References

[1] Henk Barendregt, Mario Coppo & Mariangiola Dezani-Ciancaglini (1983): *A Filter Lambda Model and the Completeness of Type Assignment. J. Symb. Log.* 48(4), pp. 931–940.

[2] Michael Kohlhase & Frank Pfenning (1993): *Unification in a Lambda-Calculus with Intersection Types.* In Dale Miller, editor: *Logic Programming, Proceedings of the 1993 International Symposium, Vancouver, British Columbia, Canada, October 26-29, 1993*, MIT Press, pp. 488–505.

[3] Elaine Pimentel, Simona Ronchi Della Rocca & Luca Roversi (2012): *Intersection Types from a Proof-theoretic Perspective. Fundam. Inform.* 121(1-4), pp. 253–274.

[4] Garrel Pottinger (1980): *A Type Assingment for the Strongly Normalizable λ-terms.* In J.P. Seldin & J.R. Hindley, editors: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp. 561–577.

[5] Simona Ronchi Della Rocca & Luca Roversi (2001): *Intersection Logic.* In Laurent Fribourg, editor: *Computer Science Logic, 15th International Workshop, CSL 2001.*, *Lecture Notes in Computer Science* 2142, Springer, pp. 414–428.

[6] Simona Ronchi Della Rocca, Alexis Saurin, Yiorgos Stavrinos & Anastasia Veneti (2010): *Intersection Logic in sequent calculus style.* In Elaine Pimentel, Betti Venneri & Joe B. Wells, editors: *Proceedings Fifth Workshop on Intersection Types and Related Systems, ITRS 2010, Edinburgh, U.K., 9th July 2010.*, *EPTCS* 45, pp. 16–30.

[7] Paweł Urzyczyn (2016): *Intuitionistic Games: Determinacy, Completeness, and Normalization. Studia Logica* 104(5), pp. 957–1001.

[8] Betti Venneri (1994): *Intersection Types as Logical Formulae. Journal of Logic and Computation* 4(2), pp. 109–124.

Revisiting the Bang Calculus

Antonio Bucciarelli
IRIF, CNRS and Université de Paris, France

Delia Kesner
IRIF, CNRS and Université de Paris, France
Institut Universitaire de France, France

Alejandro Ríos
Universidad de Buenos Aires, Argentina

Andrés Viso
Universidad de Buenos Aires, Argentina
Universidad Nacional de Quilmes, Argentina

Extended abstract[1]

Call-by-Push-Value. The Call-by-Push-Value (CBPV) paradigm, introduced by P.B. Levy [34, 35], distinguishes between values and computations under the slogan *"a value is, a computation does"*. It subsumes the λ-calculus by adding some primitives that allow to capture both the Call-by-Name (CBN) and Call-by-Value (CBV) semantics. CBN is a lazy strategy that consumes arguments without any preliminary evaluation, potentially duplicating work, while CBV is greedy, always computing arguments disregarding whether they are used or not, which may prevent a normalising term from terminating, *e.g.* $(\lambda x.I)\,\Omega$, where $I = \lambda x.x$ and $\Omega = (\lambda x.xx)\,(\lambda x.xx)$.

Essentially, CBPV introduces unary primitives `thunk` and `force`. The former freezes the execution of a term (*i.e.* it is not allowed to compute under a `thunk`) while the latter fires again a frozen term. Informally, $\mathtt{force}\,(\mathtt{thunk}\,t)$ is semantically equivalent to t. Resorting to the paradigm slogan, `thunk` turns a computation into a value, while `force` does the opposite. Thus, CBN and CBV are captured by conveniently labelling a λ-term using `force` and `thunk` to pause/resume the evaluation of a subterm depending on whether it is an argument (CBN) or a function (CBV). In doing so, CBPV provides a unique formalism capturing two distinct λ-calculi strategies, thus allowing to study operational and denotational semantics of CBN and CBV in a unified framework.

Bang calculus. T. Ehrhard [23] introduces a typed calculus, that can be seen as a variation of CBPV, to establish a relation between this paradigm and Linear Logic (LL). A simplified version of this formalism is later dubbed Bang calculus [24], showing in particular how CBPV captures the CBN and CBV semantics of λ-calculus via Girard's translations of intuitionistic logic into LL. The Bang calculus is essentially an extension of λ-calculus with two new constructors, namely *bang* (!) and *dereliction* (der), together with the reduction rule $\mathrm{der}\,(!t) \mapsto t$. There are two notions of reduction for the Bang calculus, depending on whether it is allowed to reduce under a bang constructor or not. They are called *strong* and *weak reduction* respectively. Indeed, it is weak reduction that makes bang/dereliction play the role of the primitives `thunk`/`force`. Hence, these modalities are essential to capture the essence behind the CBN–CBV duality. A similar approach appears in [38], studying (simply typed) CBN and CBV translations into a fragment of IS4, recast as a very simple λ-calculus equipped with an indeterminate lax monoidal comonad.

Non-Idempotent Types. Intersection types, pioneered by [15, 16], can be seen as a syntactical tool to denote programs. They are invariant under the equality generated by the evaluation rules, and type all and only all normalising terms. They are originally defined as *idempotent* types, so that the equation

[1] The full version of this work is currently submitted to an international conference.

Submitted to:
ITRS 2020

$\sigma \cap \sigma = \sigma$ holds, thus preventing any use of the intersection constructor to count resources. On the other hand, *non-idempotent* types, pioneered by [25], are inspired from LL, they can be seen as a syntactical formulation of its relational model [27, 11]. This connection suggests a *quantitative* typing tool, being able to specify properties related to the consumption of resources, a remarkable investigation pioneered by the seminal de Carvalho's PhD thesis [17] (see also [19]). Non-idempotent types have also been used to provide characterisations of complexity classes [8]. Several papers explore the qualitative and quantitative aspects of non-idempotent types for different higher order languages, as for example Call-by-Name, Call-by-Need and Call-by-Value λ-calculi, as well as extensions to Classical Logic. Some references are [13, 22, 4, 3, 33]. Other relational models were directly defined in the more general context of LL, rather than in the λ-calculus [18, 30, 21, 20].

An interesting recent research topic concerns the use of non-idempotent types to provide *bounds* of reduction lengths. More precisely, the size of type derivations has often been used as an *upper bound* to the length of different evaluation strategies [36, 22, 31, 13, 32, 33]. A key notion behind these works is that when t evaluates to t', then the size of the type derivation of t' is smaller than the one of t, thus the size of type derivations provides an upper bound for the *length* of the reduction to a normal form as well as for the *size* of this normal form.

A crucial point to obtain *exact bounds*, instead of upper bounds, is to consider only *minimal* type derivations, as the ones in [17, 9, 21]. Another approach was taken in [1], which uses an appropriate notion of *tightness* to implement minimality, a technical tool adapted to Call-by-Value [28, 3] and Call-by-Need [4].

Contributions and Related Works

This work presents a reformulation of the untyped Bang calculus, and proposes a quantitative study of it by means of non-idempotent types.

The Untyped Reduction. The Bang calculus in [23] suffers from the absence of *commutative conversions* [37, 14], making some redexes to be syntactically blocked when open terms are considered. A consequence of this approach is that there are some normal forms that are semantically equivalent to non-terminating programs, a situation which is clearly unsound. This is repaired in [24] by adding commutative conversions specified by means of σ-reduction rules, which are crucial to unveil hidden (value) redexes. However, this approach presents a major drawback since the resulting combined reduction relation is not confluent.

Our revisited Bang calculus, called $\lambda!$, fixes these two problems at the same time. Indeed, the syntax is enriched with explicit substitutions, and σ-equivalence is integrated in the primary reduction system by using the *distance* paradigm [5], without any need to unveil hidden redexes by means of an independent relation. This approach restores *confluence*.

The Untyped CBN and CBV Encodings. CBN and CBV (untyped) translations are extensively studied in [29]. The authors establish two encodings *cbn* and *cbv*, from untyped λ-terms into untyped terms of the Bang calculus, such that when t reduces to u in CBN (resp. CBV), $cbn(t)$ reduces to $cbn(u)$ (resp. $cbv(t)$ reduces to $cbv(u)$) in the Bang calculus. However, CBV normal forms in λ-calculus are not necessarily translated to normal forms in the Bang calculus.

Our revisited notion of reduction naturally encodes (weak) CBN as well as (open) CBV. These two notions are dual: weak CBN forbids reduction inside arguments, which are translated to bang terms, while open CBV forbids reduction under λ-abstractions, also translated to bang terms. More precisely, we simply extend to explicit substitutions the original CBN translation from λ-calculus to the Bang calculus, which preserves normal forms, but we subtly reformulate the CBV one. In contrast to [29], our

CBV translation does preserve normal forms.

The Typed System. We propose a type system for the $\lambda!$-calculus, called $\mathscr{U}$, based on non-idempotent intersection types. System $\mathscr{U}$ is able to fully *characterise* normalisation, in the sense that a term t is $\mathscr{U}$-typable if and only if t is normalising. More interestingly, we show that system $\mathscr{U}$ has also a quantitative flavour, in the sense that the length of any reduction sequence from t to normal form *plus* the size of this normal form is *bounded* by the size of the type derivation of t. We show that system $\mathscr{U}$ also captures the non-idempotent intersection type system for CBN given in [25, 17], and extended with explicit substitutions as in [32], as well as a new type system $\mathscr{V}$ that we define for CBV, as defined in [6]. System $\mathscr{V}$ characterises termination of open CBV, in the sense that t is typable in $\mathscr{V}$ if and only if t is terminating in open CBV. This can be seen as another (collateral) contribution of this work. Moreover, the CBV embedding in [29] is not complete with respect to their type system for CBV. System $\mathscr{V}$ recovers completeness (left as an open question in [29]). Finally, an alternative CBV encoding of typed terms is proposed. This encoding is not only sound and complete, but now enjoys preservation of normal-forms.

A Refinement of the Type System Based on Tightness. A major observation concerning β-reduction in λ-calculus (and therefore in the Bang calculus) is that the size of normal forms can be exponentially bigger than the number of steps to these normal forms. This means that bounding the sum of these two integers *at the same* time is too rough, not very relevant from a quantitative point of view. Following ideas in [17, 9, 1], we go beyond upper bounds. Indeed, another major contribution of this work is the refinement of the non-idempotent type system $\mathscr{U}$ to another type system $\mathscr{E}$, equipped with constants and counters, together with an appropriate notion of *tightness* (*i.e.* minimality). This new formulation fully exploits the quantitative aspect of the system, in such a way that *upper bounds* provided by system $\mathscr{U}$ are refined now into *independent exact bounds* for time and space. More precisely, given a tight type derivation Φ with counters (b,e,s) for a term t, we can show that t is normalisable in $(b+e)$-steps and its normal form has *size* s. The opposite direction also holds. Therefore, exact measures concerning the *dynamic* behaviour of t, are extracted from a *static* (tight) typing property of t.

Acknowledgements. This is an extended abstract of [12], to appear in FLOPS 2020. This work has been partially funded by the projects ECOS-Sud PA17C01 and LIA INFINIS/IRP SINFIN.

References

[1] Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds. *PACMPL*, 2 (ICFP):94:1–94:30, 2018.

[2] Beniamino Accattoli and Giulio Guerrieri. Open call-by-value. *Asian Programming Languages and Systems Symposium* (APLAS), Hanoi, Vietnam, LNCS 10017, pages 206–226, 2016.

[3] Beniamino Accattoli and Giulio Guerrieri. Types of fireballs. *Asian Programming Languages and Systems Symposium* (APLAS), Wellington, New Zealand, LNCS 11275, pages 45–66. Springer, 2018.

[4] Beniamino Accattoli, Giulio Guerrieri, and Maico Leberle. Types by need. *Programming Languages and Systems European Symposium on Programming* (ESOP), Prague, Czech Republic, LNCS 11423, pages 410–439. Springer, 2019.

[5] Beniamino Accattoli and Delia Kesner. The structural λ-calculus. *Int. Conf. on Computer Science Logic* (CSL), Brno, Czech Republic, LNCS 6247, pages 381–395. Springer, 2010.

[6] Beniamino Accattoli and Luca Paolini. Call-by-value solvability, revisited. *Int. Symposium on Functional and Logic Programming* (FLOPS), Kobe, Japan, LNCS 7294, pages 4–16. Springer, 2012.

[7] Hendrik P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.

[8] Erika De Benedetti and Simona Ronchi Della Rocca. A type assignment for λ-calculus complete both for FPTIME and strong normalization. *Information and Computation*, 248:195–214, 2016.

[9] Alexis Bernadet and Stéphane Lengrand. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 9(4), 2013.

[10] Marc Bezem, Jan Willem Klop, and Vincent van Oostrom. *Term Rewriting Systems (TeReSe)*. Cambridge University Press, 2003.

[11] Antonio Bucciarelli and Thomas Ehrhard. On phase semantics and denotational semantics: the exponentials. *Annals of Pure Applied Logic*, 109(3):205–241, 2001.

[12] Antonio Bucciarelli, Delia Kesner, Alejandro Ríos and Andrés Viso. The Bang Calculus Revisited. *International Symposium on Functional and Logic Programming* (FLOPS), Akita, Japan, LNCS, Springer, 2020.

[13] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017.

[14] Alberto Carraro and Giulio Guerrieri. A semantical and operational account of call-by-value solvability. *Int. Conf. on Foundations of Software Science and Computation Structures* (FOSSACS), Grenoble, France, LNCS 8412, pages 103–118. Springer, 2014.

[15] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for λ-terms. *Archive for Mathematical Logic*, 19(1):139–156, 1978.

[16] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[17] Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. PhD thesis, Université Aix-Marseille II, 2007.

[18] Daniel de Carvalho. The relational model is injective for multiplicative exponential linear logic. *Int. Conf. on Computer Science Logic* (CSL), Marseille, France, *LIPIcs* 62, pages 41:1–41:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[19] Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018.

[20] Daniel de Carvalho and Lorenzo Tortora de Falco. A semantic account of strong normalization in linear logic. *Information and Computation*, 248:104–129, 2016.

[21] Daniel de Carvalho, Michele Pagani, and Lorenzo Tortora de Falco. A semantic measure of the execution time in linear logic. *Theoretical Computer Science*, 412(20):1884–1902, 2011.

[22] Thomas Ehrhard. Collapsing non-idempotent intersection types. *Int. Conf. on Computer Science Logic* (CSL), Fontainebleau, France, *LIPIcs* 16, pages 259–273. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[23] Thomas Ehrhard. Call-by-push-value from a linear logic point of view. *European Symposium on Programming* (ESOP), Eindhoven, The Netherlands, LNCS 9632, pages 202–228. Springer, 2016.

[24] Thomas Ehrhard and Giulio Guerrieri. The bang calculus: an untyped lambda-calculus generalizing call-by-name and call-by-value. *Int. Symposium on Principles and Practice of Declarative Programming* (PPDP), Edinburgh, United Kingdom, pages 174–187. ACM, 2016.

[25] Philippa Gardner. Discovering needed reductions using type theory. *Int. Conf. on Theoretical Aspects of Computer Software* (TACS), Sendai, Japan, LNCS 789, pages 555–574. Springer, 1994.

[26] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[27] Jean-Yves Girard. Normal functors, power series and λ-calculus. *Annals of Pure Applied Logic*, 37(2):129–177, 1988.

[28] Giulio Guerrieri. Towards a semantic measure of the execution time in call-by-value lambda-calculus. *Joint Int. Workshops on Developments in Computational Models and Intersection Types and Related Systems* (DCM/ITRS), Oxford, UK, *EPTCS* 283, pages 57–72, 2018.

[29] Giulio Guerrieri and Giulio Manzonetto. The bang calculus and the two girard's translations. *Joint Int. Workshops on Linearity & Trends in Linear Logic and Applications* (Linearity-TLLA), Oxford, UK, EPTCS, pages 15–30, 2019.

[30] Giulio Guerrieri, Luc Pellissier, and Lorenzo Tortora de Falco. Computing connected proof(-structure)s from their taylor expansion. *Int. Conf. on Formal Structures for Computation and Deduction* (FSCD), Porto, Portugal, *LIPIcs* 52, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[31] Delia Kesner. Reasoning about call-by-need by means of types. *Int. Conf. on Foundations of Software Science and Computation Structures* (FOSSACS), Eindhoven, The Netherlands, LNCS 9634, pages 424–441. Springer, 2016.

[32] Delia Kesner and Daniel Ventura. Quantitative types for the linear substitution calculus. *Int. Conf. on Theoretical Computer Science* (IFIP/TCS), Rome, Italy, LNCS 8705, pages 296–310. Springer, 2014.

[33] Delia Kesner and Pierre Vial. Types as resources for classical natural deduction. *Int. Conf. on Formal Structures for Computation and Deduction* (FSCD), Oxford, UK, *LIPIcs* 84, pages 24:1–24:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[34] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

[35] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.

[36] Michele Pagani and Simona Ronchi Della Rocca. Solvability in resource lambda-calculus. *Int. Conf. on Foundations of Software Science and Computational Structures* (FOSSACS), Paphos, Cyprus, LNCS 6014, pages 358–373. Springer, 2010.

[37] Laurent Regnier. Une équivalence sur les lambda-termes. *TCS*, 2(126):281–292, 1994.

[38] José Espírito Santo, Luís Pinto, and Tarmo Uustalu. Modal embeddings and calling paradigms. *Int. Conf. on Formal Structures for Computation and Deduction* (FSCD), Dortmund, Germany, *LIPIcs* 131, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

Observability by Means of Typability and Inhabitation

Antonio Bucciarelli
IRIF, CNRS and Université de Paris, France

Delia Kesner
IRIF, CNRS and Université de Paris, France
Institut Universitaire de France, France

Simona Ronchi Della Rocca
Dipartimento di Informatica, Università di Torino, Italy

Extended abstract

In these last years there has been a growing interest in *pattern λ-calculi* [13, 10, 7, 11, 9, 12] which are used to model the pattern-matching primitives of functional programming languages (*e.g.* OCAML, ML, Haskell) and proof assistants (*e.g.* Coq, Isabelle). These calculi are extensions of λ-calculus: abstractions are written as $\lambda \mathtt{p.t}$, where $\mathtt{p}$ is a *pattern* specifying the expected structure of the argument. In this work we restrict our attention to *pair* patterns, which are expressive enough to illustrate the challenging notion of solvability/observability in the framework of pattern λ-calculi. More precisely, we consider a pattern *calculus* called p-calculus, introduced in [2], with *explicit pattern-matching* and reduction rules *at a distance* [1]. The p-calculus is inspired from the $\Lambda_{\mathtt{p}}$-calculus in [4]. The use of explicit pattern-matching becomes very appropriate to implement different *evaluation strategies*, thus giving rise to different *languages* with pattern-matching [7, 8, 3].

We aim to study *observability* of the p-calculus, which corresponds to *solvability* of λ-calculus. Let us first recall this last notion: a closed λ-term $\mathtt{t}$ is solvable if there is $n \geq 0$ and there are terms $\mathtt{u}_1, \ldots, \mathtt{u}_n$ such that $\mathtt{t}\mathtt{u}_1 \ldots \mathtt{u}_n$ reduces to the identity. Closed solvable terms represent meaningful programs: if $\mathtt{t}$ is closed and solvable, then $\mathtt{t}$ can produce any desired result when applied to a suitable sequence of arguments. The relation between solvability and meaningfulness is also evident in the semantics: it is sound to equate all unsolvable terms, as in Scott's original model D_∞ [14]. This notion can be easily extended to open terms, through the notion of *head-context*, which does the job of both closing the term and then applying it to an appropriate sequence of arguments. Thus a term $\mathtt{t}$ is solvable if there is a head-context $\mathtt{H}$ such that, when $\mathtt{H}$ is filled by $\mathtt{t}$, then $\mathtt{H}[\mathtt{t}]$ is closed and reduces to the identity.

In order to extend the notion of solvability to the p-calculus, it is clear that pairs have to be taken into account. A relevant question is whether a pair should be considered as meaningful in any case. At least two choices are possible: a *lazy* semantics considering any pair to be meaningful, or a *strict* one requiring both of its components to be meaningful. In the operational semantics we supply for the p-calculus the constant $\mathtt{fail}$ is different from $\langle \mathtt{fail}, \mathtt{fail} \rangle$: if a term reduces to $\mathtt{fail}$ we do not have any information about its result, but if it reduces to $\langle \mathtt{fail}, \mathtt{fail} \rangle$ we know at least that it represents a pair. In fact, being a pair is already an observable property, which in particular is sufficient to unblock an explicit matching, independently from the observability of its components. As a consequence, a term $\mathtt{t}$ is *defined* to be *observable* iff there exists a head-context $\mathtt{H}$ such that $\mathtt{H}[\mathtt{t}]$ is closed and reduces to a pair. Thus for example, the term $\langle \mathtt{t}, \mathtt{t} \rangle$ is always observable, also in case $\mathtt{t}$ is not observable. Observability turns out to be conservative with respect to the notion of solvability for the λ-calculus.

We characterize observability for the p-calculus through two different and complementary notions related to a type assignment system with non-idempotent intersection types that we call $\mathscr{P}$. The first one

Submitted to:
ITRS 2020

is *typability*, concerning the possibility to construct a typing derivation for a given term, and the second one is *inhabitation*, concerning the possibility to construct a term from a given typing. More precisely, we first supply a notion of *canonical form* such that reducing a term to some canonical form is a *necessary* but *not a sufficient* condition for being observable. In fact, canonical forms may contain blocking explicit matchings, so that we need to guess whether or not there exists a substitution being able to *simultaneously* unblock all these blocked forms. In contrast, if a λ-term is in canonical form, it is always possible to find suitable arguments to feed it in order to produce any desired term (in fact, the identity). Hence typable λ-terms are observable. This is somehow an accidental property, and for more general calculi, like the p-calculus, the fact that a term t has a type does not guarantee that suitable arguments to be applied to t in order to produce the desired observable result do exist. Our type system $\mathscr{P}$ *characterizes* canonical forms: a term $\mathtt{t}$ has a canonical form if and only if it is typable in system $\mathscr{P}$. Types are of the shape $\mathtt{A}_1 \to \mathtt{A}_2 \to \ldots \to \mathtt{A}_n \to \sigma$, for $n \geq 0$, where the $\mathtt{A}_i$'s are multisets of types and σ is a type. The use of multisets to represent the non-idempotent intersection is standard, namely $[\sigma_1, \ldots, \sigma_m]$ is just a notation for $\sigma_1 \cap \ldots \cap \sigma_m$. By using type system $\mathscr{P}$ we can supply the following *characterization* of observability: a closed term $\mathtt{t}$ is observable if and only if $\mathtt{t}$ is typable in system $\mathscr{P}$, let say with a type of the shape $\mathtt{A}_1 \to \mathtt{A}_2 \to \ldots \to \mathtt{A}_n \to \sigma$ (where σ is a product type), and for all $1 \leq i \leq n$ there is a term $\mathtt{t}_i$ such that every type in $\mathtt{A}_i$ is inhabited by $\mathtt{t}_i$. In fact, if $\mathtt{u}_i$ inhabits all the types in $\mathtt{A}_i$, then $\mathtt{t}\mathtt{u}_1 \ldots \mathtt{u}_n$, resulting from plugging $\mathtt{t}$ into the head context $\Box \mathtt{u}_1 \ldots \mathtt{u}_n$, reduces to a pair. The extension of this notion to open terms is obtained by suitably adapting the notion of head context.

Clearly, the property of being observable is undecidable, exactly as the solvability property for λ-calculus. More precisely, the property of having canonical form is undecidable, since the λ-terms that are typable in system $\mathscr{P}$, characterizing terms having canonical form, are exactly the solvable ones. But our characterization of observability through the inhabitation property of $\mathscr{P}$ does not add a further level of undecidability: in fact we prove that inhabitation for system $\mathscr{P}$ is *decidable*,in constrast to the idempotent case where the problem is known to be undecidable [15]. The inhabitation algorithm presented here is a non trivial extension of the one given in [5, 6] for the λ-calculus, the difficulty of the extension being due to the explicit pattern matching and to the type information of patterns.

This work simplifies a previous study of observability for pattern calculi in [4], not only from the operational semantics of the pattern calculus, but also from the typing system point of view.

References

[1] B. Accattoli and D. Kesner. The structural *lambda*-calculus. In A. Dawar and H. Veith, editors, *Proceedings of 24th EACSL Conference on Computer Science Logic*, volume 6247 of *LNCS*, pages 381–395. Springer, Aug. 2010.

[2] S. Alves, D. Kesner, and D. Ventura. A quantitative understanding of pattern matching. *arXiv:1912.01914*, 2020.

[3] T. Balabonski. On the implementation of dynamic patterns. In E. Bonelli, editor, *HOR*, volume 49 of *EPTCS*, pages 16–30, 2010.

[4] A. Bucciarelli, D. Kesner, and S. R. D. Rocca. Observability for pair pattern calculi. In T. Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 123–137. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[5] A. Bucciarelli, D. Kesner, and S. Ronchi Della Rocca. The inhabitation problem for non-idempotent intersection types. In *TCS*, volume 8705 of *LNCS*, pages 341–354. Springer, 2014.

[6] A. Bucciarelli, D. Kesner, and S. Ronchi Della Rocca. Inhabitation for non-idempotent intersection types. *Logical Methods in Computer Science*, 14(3), 2018.

[7] S. Cerrito and D. Kesner. Pattern matching as cut elimination. *Theoretical Computer Science*, 323(1-3):71–127, 2004.

[8] H. Cirstea, G. Faure, and C. Kirchner. A rho-calculus of explicit constraint application. *Higher-Order and Symbolic Computation*, 20(1-2):37–72, 2007.

[9] B. Jay and D. Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009.

[10] W. Kahl. Basic pattern matching calculi: A fresh view on matching failure. In Y. Kameyama and P. Stuckey, editors, *FLOPS*, volume 2998 of *LNCS*, pages 276–290. Springer, 2004.

[11] J.-W. Klop, V. van Oostrom, and R. de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, 398(1-3):16–31, 2008.

[12] B. Petit. A polymorphic type system for the lambda-calculus with constructors. In *TLCA*, volume 5608 of *lncs*, pages 234–248. Springer, 2009.

[13] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

[14] D. S. Scott. Outline of a mathematical theory of computation. In *Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176. Departement of Elctrical Engeneering, Princeton Univ., 1970.

[15] P. Urzyczyn. The emptiness problem for intersection types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999.

Quantitative Types for the Atomic λ-calculus

Loïc Peyrot and Delia Kesner

Université de Paris (IRIF, CNRS)
`lpeyrot@irif.fr, kesner@irif.fr`

Deep inference is a logical formalism developed inside the framework of the calculus of structures. It exhibits *proofs with contexts*, which avoids some syntactic bureaucracies [6]. The atomic λ-calculus was designed as a Curry-Howard interpretation of deep-inference for intuistionistic logic [7]. It is an extension of λ-calculus with sharing mechanisms resembling explicit substitutions. When duplication of shared terms is needed, it is carried out *atomically*: constructor by constructor. These two features, sharing and atomic duplication, are at the core of *fully lazy sharing* [1]. Whereas this mechanism is usually specified by means of sharing graphs or other graphical formalisms, atomic λ-calculus is to our knowledge the first *algebraic* representation of the λ-calculus which can express it in a natural way. As expected, the atomic λ-calculus can be typed inside the deep inference formalism, but also inside the sequent calculus. Typed atomic λ-calculus enjoys the desired properties of subject reduction, confluence, and strong normalisation [8].

Still, the calculus has not been extensively studied until today. It is also strongly *syntactical*: there is still no known model for it. We approach this calculus throughout non-idempotent intersection types [5]. We hope that their semantical flavour [4] will help us uncover some of the calculus' aspects.

We define a quantitative type system characterising strong normalisation, *i.e* we use non-idempotent types instead of the more standard idempotent ones. Our type system follows the sequent calculus style. We prove the *correctness* of our type system: every typed term strongly normalises, as well as completeness: every strongly normalisable term is typable. Our proof of correctness is an alternative to the original one using candidates of reducibility [8].

Usually, proofs of termination using a quantitative discipline are straightforward (a survey can be found in [3]). During the substitution of a variable x by a term u, typing derivations for the types of u are dispatched in place of the axioms rules of x, but none is duplicated. Therefore, the size of the typing derivation can only decrease during reduction. But within the atomic λ-calculus, the duplication of the constructors induces an introduction of *new* resources (typically variables). For instance, to duplicate an application uv, we have to substitute every variable x bound to uv by an application of two new variables, say y and z where y is bound to u and z to v. By doing that, we duplicate the types of u and v to be able to type the new variables.

However, we are able to give a characterisation of termination through a subtle transformation of terms, that we call *expansion*. The idea of an expansion is to anticipate the creation of the duplicates. Then, we show that the size of the typing derivation of the expansion of a term decreases along any reduction step.

Our main contribution is an original type system for atomic λ-calculus, which is the first quantitative one, to our knowledge. Because we have to use a mechanism to reveal the resource-consuming nature of reduction in a quantitative typing discipline, we believe that a quantitative sequent calculus type system is not the most appropriate for atomic λ-calculus. It strengthens the idea that we should be looking for a way to describe intersection types directly in deep inference. This has already been studied but never led to a result on the full atomic λ-calculus [9]. We believe that our work will contribute to the quest of such typing system.

References

[1] Thibaut Balabonski. A unified approach to fully lazy sharing. *ACM SIGPLAN Notices*, 47:469–480, 01 2012.

[2] Alexis Bernardet and Stéphane Graham-Lengrand. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science*, 9(4):17–42, 2013.

[3] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25:433–464, 08 2017.

[4] Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. PhD thesis, Université Aix-Marseille II, 2007.

[5] Philippa Gardner. Discovering needed reductions using type theory. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 555–574, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[6] Alessio Guglielmi, Tom Gundersen, and Michel Parigot. A proof calculus which reduces syntactic bureaucracy. *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010*, 6, 01 2010.

[7] Tom Gundersen, Willem Heijltjes, and Michel Parigot. Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 311–320, Washington, DC, USA, 2013. IEEE Computer Society.

[8] Tom Gundersen, Willem Heijltjes, and Michel Parigot. A proof of strong normalisation of the typed atomic lambda-calculus. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 340–354, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[9] Joseph W. N. Paulus and Heijltjes Willem. Deep-inference intersection types, 2018.

A Type Checker for a Logical Framework with Union and Intersection Types

Claude Stolze
IRIF, Université de Paris
stolze@irif.fr

Luigi Liquori
Inria, Université Côte d'Azur
Luigi.Liquori@inria.fr

1 Introduction

We present the syntax, semantics, and typing rules of Bull [14], a prototype theorem prover based on the Δ-framework, *i.e.* a fully-typed lambda-calculus decorated with union and intersection types, as described in [15, 9]. Bull also implements the subtyping algorithm of [10] for the Type Theory Ξ of [2]. Bull has a command-line interface where the user can declare axioms, terms, and perform computations and some basic terminal-style features like error pretty-printing, subexpressions highlighting, and file loading. Moreover, it can typecheck a proof or normalize it. These terms can be incomplete, therefore the typechecking algorithm uses unification to try to construct the missing subterms. Bull uses the syntax of Berardi's Pure Type Systems [3] to improve the compactness and the modularity of the kernel. Abstract and concrete syntax are mostly aligned and similar to the concrete syntax of Coq. Bull uses a *higher-order unification algorithm* for terms, while typechecking and partial type inference are done by a *bidirectional refinement algorithm*, similar to the one found in [1]. The refinement can be split into two parts: the essence refinement and the typing refinement. The bidirectional refinement algorithm aims to have partial type inference, and to give as much information as possible to the unifier. For instance, if we want to find a $?y$ such that $\vdash_\Sigma \langle \lambda x{:}\sigma.x, \lambda x{:}\tau.?y\rangle : (\sigma \to \sigma) \cap (\tau \to \tau)$, we can infer that $x{:}\tau \vdash ?y : \tau$ and that $\wr ?y \wr =_\beta x$. Binders are implemented using commonly-used de Bruijn indices. We have defined a concrete language syntax that will allow user to write Δ-terms. We have defined the reduction rules and an evaluator. We have implemented from scratch a refiner which does partial typechecking and type reconstruction. We have experimented Bull with classical examples of the intersection and union literature, such as the ones formalized by Pfenning with his Refinement Types in LF [12].

Syntax of terms. The abstract syntax for the language is sketched below. The main differences with the Δ-framework [9] are the additions of a placeholder and meta-variables, used by the refiner. We also added a **let** operator and changed the syntax of the strong sum so it looks more like the concrete syntax used in the implementation. A meta-variable $?x[\Delta_1;\ldots;\Delta_n]$ uses *suspended substitutions*, inspired by [1] and intuitively explained as follows: if we want to unify $(\lambda x{:}\sigma.?y)\,c_1$ with c_1, we could unify $?y$ with c_1 or with x, the latter being the preferred solution. However, if we normalize $(\lambda x{:}\sigma.?y)\,c_1$, we should record the fact that c_1 can be substituted by any occurrence of x appearing the term to be replaced by $?y$; the term is actually noted $(\lambda x{:}\sigma.?y[x])\,c_1$ and reduces to $?y[c_1]$, noting that c_1 has replaced x. Finally, following the Cervesato-Pfenning jargon [5], applications are in *spine form*, *i.e.* the arguments of a function are

Submitted to:
ITRS 2020

stored together in a list, exposing the head of the term separately.

Δ,σ	$::=$	$s \mid c \mid x \mid \_$	Sorts, constant, variables and placeholders
	$\mid$	$?x[\Delta;\ldots;\Delta]$	Meta-variable
	$\mid$	$\textbf{let}\ x{:}\sigma := \Delta\ \textbf{in}\ \Delta$	Local definition
	$\mid$	$\Pi x{:}\sigma.\Delta \mid \lambda x{:}\sigma.\Delta$	Dependent product and λ-abstraction
	$\mid$	$\Delta S \mid \sigma \cap \sigma \mid \sigma \cup \sigma$	Application, intersection and union
	$\mid$	$\langle \Delta, \Delta \rangle \mid \mathsf{pr}_1\,\Delta \mid \mathsf{pr}_2\,\Delta$	Strong pair, left and right projection
	$\mid$	$\textbf{smatch}\ \Delta\ \textbf{return}\ \sigma\ \textbf{with}\,[x{:}\sigma \Rightarrow \Delta \mid x{:}\sigma \Rightarrow \Delta]$	Strong sum
	$\mid$	$\mathsf{in}_1\,\sigma\,\Delta \mid \mathsf{in}_2\,\sigma\,\Delta \mid \textbf{coe}\,\sigma\,\Delta$	Left/right injection and coercions
S	$::=$	$() \mid (S;\Delta)$	Spines

Concrete syntax is mostly aligned with the abstract one with the intention to mimic Coq.

Environments. There are four kinds of environments, namely 1) the *global environment* (noted Σ). The global environment holds constants which are fully typechecked. 2) the *local environment* (noted Γ). It is used for the first step of typechecking, and looks like a standard environment. 3) the *essence environment* (noted Ψ). It is used for the second step of typechecking, and holds the essence of the local variables. 4) the *meta-environment* (noted Φ). It is used for unification, and records meta-variables and their instantiation whenever the unification algorithm has found a solution.

$$
\begin{array}{lcl}
\Sigma & ::= & \cdot \mid \Sigma, c{:}\varsigma@\sigma \mid \Sigma, c := M@\Delta : \varsigma@\sigma \\
\Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \mid \Gamma, x := \Delta : \sigma \\
\Psi & ::= & \cdot \mid \Psi, x \mid \Psi, x := M \\
\Phi & ::= & \cdot \mid \Phi, \textbf{sort}(?x) \mid \Phi, ?x := s \mid \Phi, (\Gamma \vdash ?x : \sigma) \mid \Phi, (\Gamma \vdash ?x := \Delta : \sigma) \mid \Phi, \Psi \vdash ?x \mid \Phi, \Psi \vdash ?x := M
\end{array}
$$

Evaluator. The evaluator follows the applicative order strategy, which recursively normalizes all sub-terms from left to right; in addition to the standard $\beta\delta\eta\zeta$-reduction rules, we have rules for projections pr_i and injections in_i.

Subtyping. It is the one extracted by Coq in [10]. The algorithm has been mechanically proved correct in Coq by extending the certification of the algorithm for intersection types of Bessai [4], and it represent, at the moment, the only mechanically certified (by Coq) part.

Unification. The Bull higher-order unification algorithm is inspired by the Reed [13] and Ziliani-Sozeau [16] papers. The unification algorithm takes as input a meta-environment Φ, a global environment Σ, a local environment Γ, the two terms to unify Δ_1 and Δ_2, and either fails or returns the updated meta-environment Φ, namely $\Phi;\Sigma;\Gamma \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \stackrel{\mathscr{U}}{\leadsto} \Phi$.

Refiner. Our typechecker is also a *refiner*: intuitively, a refiner takes as input an incomplete term, and possibly an incomplete type, and tries to infer as much information as possible in order to reconstruct a well-typed term. The Bull refiner is inspired by the work on the Matita ITP [1]. It is defined using *bi-directionality*, in the style of Harper-Licata [8]. The bi-directional technique is a mix of typechecking and type reconstruction, in order to trigger the unification algorithm as soon as possible. Moreover, it gives more precise error messages than standard type reconstruction. There are five kind of judgments,

$$
\begin{array}{lll}
\Phi_1;\Sigma;\Gamma \vdash \Delta_1 \stackrel{\Uparrow}{\leadsto} \Delta_2 : \sigma;\Phi_2 & \Phi_1;\Sigma;\Gamma \vdash \sigma_1 \stackrel{\mathscr{F}}{\leadsto} \sigma_2 : \tau;\Phi_2 & \\
\Phi_1;\Sigma;\Gamma \vdash \Delta_1 : \sigma \stackrel{\Downarrow}{\leadsto} \Delta_2;\Phi_2 & \Phi_1;\Sigma;\Psi \vdash \Delta \stackrel{\mathscr{E}\Uparrow}{\leadsto} M;\Phi_2 & \Phi_1;\Sigma;\Psi \vdash M@\Delta \stackrel{\mathscr{E}\Downarrow}{\leadsto} \Phi
\end{array}
$$

Read-Eval-Print-Loop. The REPL reads a command which is given by the parser as a list of atomic commands. These commands are similar to the vernacular Coq commands and are quite intuitive. Here is the list of the REPL commands, along with their description:

`Help.`	`show this list of commands`
`Load "file".`	`for loading a script file`
`Axiom term : type.`	`define a constant or an axiom`
`Definition name [: type] := term.`	`define a term`
`Print name.`	`print the definition of name`
`Printall.`	`print all the signature (axioms and definitions)`
`Compute name.`	`normalize name and print the result`
`Quit.`	`quit`

Future work. The current version of Bull [14] (ver. 1.0, December 2019) is still a work-in-progress: we plan to implement the following features: *i*) *Inductive types à la* Paulin-Mohring [11] (reasonably feasible); *ii*) *Mixing subtyping and unification*, taking inspiration by the work of Dudenhefner, Martens, and Rehof [7]; *iii*) *Relevant arrow*, as defined in [9], it could be useful to add more expressivity to our system. Relevant implication allows for a natural introduction of subtyping, in that $A \supset_r B$ morally means $A \leqslant B$. Relevant implication amounts to a notion of "proof-reuse"; *iv*) conceiving a *Tactic language* should be feasible in the medium term.

Acknowledgements. This project could not be have be done without the many useful discussions with Ugo de'Liguoro, Daniel Dougherty, Furio Honsell, and Ivan Scagnetto.

References

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen & Enrico Tassi (2012): *A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions*. Logical Methods in Computer Science 8(1).

[2] Franco Barbanera, Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (1995): *Intersection and union types: syntax and semantics*. Information and Computation 119(2), pp. 202–230.

[3] Stefano Berardi (1988): *Towards a mathematical analysis of the Coquand–Huet calculus of constructions and the other systems in Barendregt's cube*. Ph.D. thesis, Dipartimento Matematica, Universita di Torino.

[4] Jan Bessai (2016): *Extracting a formally verified Subtyping Algorithm for Intersection Types from Ideals and Filters*. Talk at TYPES, slides.

[5] Iliano Cervesato & Frank Pfenning (2003): *A linear spine calculus*. Journal of Logic and Computation 13(5), pp. 639–688.

[6] Daniel J. Dougherty, Ugo de'Liguoro, Luigi Liquori & Claude Stolze (2016): *A Realizability Interpretation for Intersection and Union Types*. In: Asian Symposium on Programming Languages and Systems (APLAS), Lecture Notes in Computer Science 10017, Springer-Verlag, pp. 187–205.

[7] Andrej Dudenhefner, Moritz Martens & Jakob Rehof (2016): *The Intersection Type Unification Problem*. In: Formal Structures for Computation and Deduction (FSCD), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, pp. 19:1–19:16.

[8] Robert Harper & Daniel R. Licata (2007): *Mechanizing Metatheory in a Logical Framework*. Journal of Functional Programming 17(4–5), pp. 613–673.

[9] Furio Honsell, Luigi Liquori, Claude Stolze & Ivan Scagnetto (2018): *The Delta-Framework*. In: Foundations of Software Technology and Theoretical Computer Science (FSTTCS), pp. 37:1–37:21.

[10] Luigi Liquori & Claude Stolze (2017): *A Decidable Subtyping Logic for Intersection and Union Types*. In: Topics In Theoretical Computer Science (TTCS), Lecture Notes in Computer Science 10608, Springer-Verlag, pp. 74–90.

[11] Christine Paulin-Mohring (1993): *Inductive definitions in the system coq rules and properties*. In: *Typed Lambda Calculi and Applications (TLCA)*, Springer, pp. 328–345.

[12] Frank Pfenning (1993): *Refinement types for logical frameworks*. In: *TYPES*, pp. 285–299.

[13] Jason Reed (2009): *Higher-order constraint simplification in dependent type theory*. In: *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, ACM, pp. 49–56.

[14] Claude Stolze (2019): *Bull*. `https://github.com/cstolze/Bull`.

[15] Claude Stolze, Luigi Liquori, Furio Honsell & Ivan Scagnetto (2017): *Towards a Logical Framework with Intersection and Union Types*. In: *Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*, pp. 1–9.

[16] Beta Ziliani & Matthieu Sozeau (2015): *A unification algorithm for Coq featuring universe polymorphism and overloading*. In: *ACM SIGPLAN Notices*, 50(9), ACM, pp. 179–191.

Intersection Types and Positive Almost Sure Termination

Ugo Dal Lago* Claudia Faggian† Simona Ronchi Della Rocca‡

Intersection types are well known not only to *guarantee*, but also to *characterize* certain normalization properties through typability. This include head, weak head, and strong normalization, and induces a compositional methodology for verifying the properties above, given that typing can be assigned to terms in a syntax-directed way. This is not in contrast with undecidability of the underlying decision problem: type inference itself remains an undecidabile problem.

The picture above have been generalized from termination to complexity properties [6, 3], exploiting in a crucial way a non-idempotent form of intersection, in which any type τ is fundamentally different from $\tau \cap \tau$. This allows to somehow *count* the number of copies each subterm can be subject to, eventually allowing to give a precise bound [1, 2] on the number of reduction steps necessary to reach the normal form.

Intersection types have recently been generalized to *probabilistic* lambda-calculi [7, 5] by the first author with Flavien Breuvart [4], obtaining a characterization of *almost sure termination* (AST in the following), namely the condition in which non-termination can possibly happen, but only with null probability. AST not being recursively enumerable [8], there is no hope to get a type system whose type derivations can be effectively enumerated and in which typability corresponds to AST. As a consequence, AST can be checked by providing *infinitely many* type derivations for the term at hand, each certifying that it normalizes with probability at least $1 - \varepsilon$, this of course for arbitrary small ε. Completeness is achieved for both CBN and CBV evaluation, and in two different flavours, namely by way of *oracle* intersection types and *monadic* intersection types. In both cases, however, intersection is idempotent.

Almost sure termination is however not the only possible notion of termination in a probabilistic setting. In particular, it is well possible that a probabilistic process terminates with probability 1, but the expected number of steps to termination is infinite. As an example, the symmetric random walk on the natural numbers is well-known to be AST, but the journey from 1 to 0 takes, on average, infinite times. Requiring the latter value to be *finite* corresponds to the so-called *positive* almost sure termination constraint (PAST in the following), a more restrictive criterion which however is itself *not* recursively enumerable.

In this talk, we show that injecting non-idempotency in monadic intersection types allows us to characterize both AST *and PAST* within the same intersection type system, this way matching the recursion-theoretic nature of the two notions above. This is done in presence of CBN evaluation, and requires not only dropping idempotency, but also the use of *scaling*.

References

[1] B. Accattoli, S. Graham-Lengrand, and D. Kesner. Tight typings and split bounds. *PACMPL*, 2(ICFP):94:1–94:30, 2018.

[2] B. Accattoli, G. Guerrieri, and M. Leberle. Types by need. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019,*, pages 410–439, 2019.

[3] A. Bernadet and S. Lengrand. Complexity of strongly normalising λ-terms via non-idempotent intersection types. In *Proc. of FOSSACS 2011*, pages 88–107, 2011.

*University of Bologna and INRIA

†CNRS

‡University of Torino

[4] F. Breuvart and U. Dal Lago. On intersection types and probabilistic lambda calculi. In *Proc. of PPDP 2018*, pages 8:1–8:13, 2018.

[5] U. Dal Lago and M. Zorzi. Probabilistic operational semantics for the lambda calculus. *RAIRO - Theor. Inf. and Applic.*, 46(3):413–450, 2012.

[6] D. de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018.

[7] C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Proc. of LICS 1989*, pages 186–195, 1989.

[8] B. L. Kaminski, J. Katoen, and C. Matheja. On the hardness of analyzing probabilistic programs. *Acta Inf.*, 56(3):255–285, 2019.

Exploiting the power of intersection types in Java (Extended Abstract)

Mariangiola Dezani-Ciancaglini, Paola Giannini, Betti Venneri
Dipartimento di Informatica, Università di Torino, Italy
DiSIT, Università del Piemonte Orientale, Alessandria, Italy
Dipartimento di Statistica, Informatica, Applicazioni, Università di Firenze, Italy

Intersection types have been introduced originally for the λ-calculus to increase the set of terms having meaningful types [1], Part III. The typability power of intersection types is essentially due to the possibility of giving intersection types to the arguments of functions. In a programming language this corresponds to allowing the types of input parameters and output results to be intersection types. In an object-oriented language also the types of fields should be intersection types. In [3], Büchi and Weck proposed to extend Java 1 by allowing intersection types (called *compound types*) as parameter types, variable types, return types of methods, and cast operators. They justify intersection types by means of an interesting example. (Here we use a simplified version of this example.) Java 8 has intersection types, but their use in writing code is limited to type casts and bounds of generic type variables. Java 8 allows a generic type variable bound by an intersection type as a parameter type, a variable type and a return type of a method. In some cases intersection types can be simulated by the use of generics bounded by intersections. Consider the following example where the type of field intField in line 10 and the return type of method sum in line 15 are meant to be intersection types:

```
1  interface IInt { int opI(int x, int y); }
2
3  interface IDouble { double opD(double x, double y); }
4
5  class IntIDouble implements IInt, IDouble {
6    public double opD(double x, double y) {return x + y;}
7    public int opI(int x, int y) {return x + y;}
8  }
9  class UseIntersection<T1 extends IInt&IDouble> {
10   T1 intField;
11   UseIntersection(T1 intField){
12      super();
13      this.intField=intField;
14    }
15   <T2 extends IInt&IDouble> T2 sum(){
16     return (T2) new UseIntersection(new IntIDouble()).intField;
17   }
18 }
```

In Java 8 λ-expressions are *poly expressions*, i.e. they can have various types according to the context requirements. More specifically, the contexts must prescribe *target types* for λ-expressions: indeed, Java code does not compile when λ-expressions come without target types. A target type can be either a *functional interface* (i.e., an interface with a single abstract method) or an intersection of interfaces that induces a functional interface. Notably the abstract method header must have a type derivable for the λ-expression. The target type cannot be a generic type variable. This ban is needed since the variable could be instantiated by a class or by an interface with an abstract method header which cannot be used to type the λ-expression. As a result only the λ-expressions which are casted to intersection types in the

Submitted to:
ITRS 2020

user code have target types which are intersection types. If we now define the class UseIntersection with the method sum as follows

```
<T2 extends IInt&IDouble> T2 sum(){
  return (T2) new UseIntersection((x,y)->x+y).intField;
```

the code is well typed in Java. So the λ-expression (x,y)−>x+y implements both interfaces. However, this λ-expression cannot be returned by method sum, since λ-expressions in Java may be typed only by *functional interfaces*, i.e., intersections of interfaces with a single abstract method. In our example IInt &IDouble specifies two abstract methods. However, even if we remove the method opD from the interface IDouble, obtaining the functional type IInt &IDouble, the generic variable T2 is still not a functional type. This is needed for safety, since extending IInt &IDouble more methods could be added and/or a class could be obtained. Line 12 of the following code produces a compilation error in Java, since even though (x, y) −> x + y has type IInt &IDouble, still the interface T2 is not a functional interface, so cannot be the target type of a λ-expression.

```
1  interface IInt { int opI(int x, int y); }
2
3  interface IDouble { }
4
5  class UseIntersection<T1 extends IInt & IDouble> {
6    T1 intField;
7    UseIntersection(T1 intField){
8      super();
9      this.intField=intField;
10   }
11   <T2 extends IInt & IDouble> T2 q() {
12     return (T2) (x, y) -> x + y; // ERROR
13   }
14 }
```

In [4] we proposed to go back to [3] and allow intersection types as parameter types of constructors and methods and as return types of methods and allow target type for a λ-expression to have an arbitrary number of abstract methods, proviso that all the method headers have types derivable for the λ-expression. This proposal is formalised trough the calculus FJP&λ (Featherweight Java with polymorphic intersection types and λ-expressions). As expected FJP&λ enjoys subject reduction and progress.

In the current paper we show the meaningfulness of FJP&λ, by providing examples in which the use of intersection enhance the expressive power of the language. For instance, the Interface Segregation Principle prescribes to keep Interfaces as tiny as possible, in order to avoid interface pollution. On the other hand, in other cases we need to deal with objects implementing several Interfaces. The introduction of intersection types in Java has been a crucial step in this direction, allowing to segregate different methods in separated interfaces, while combining interfaces in the type of objects when needed. However, we cannot explicitly assign an intersection type to a variable. In such cases, a partial solution could be to use the var-mechanism, the new feature introduced in Java 10 for the definition of variables whose type is omitted and is inferred by the local type inference. Unfortunately, this solution does not work in the crucial case of λ-expressions, that need to have explicit target types. By discussing significant use cases, we will show that Java restrictions on intersection types result in critical drawbacks for programming with functions and, more in general, for a clean code design. Finally, concerning the feasibility of our proposal, we will discuss the compiling of FJP&λ typed programs into FJ&λ. FJ&λ is a core calculus that extends Featherweight Java with interfaces, supporting multiple inheritance in a restricted form, λ-expressions, and intersection types, see [2]. FJ&λ was introduced to give a faithful formalisation of the static and dynamic semantics of Java 8.

References

[1] Henk Barendregt, Wil Dekkers & Richard Statman (2013): *Lambda Calculus with Types*. Perspectives in Logic, Cambridge.

[2] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini & Betti Venneri (2018): *Java & Lambda: a Featherweight Story*. *Logical Methods in Computer Science* 14(3). Available at `https://arxiv.org/pdf/1801.05052.pdf`.

[3] Martin Büchi & Wolfgang Weck (1998): *Compound Types for Java*. In Bjørn N. Freeman-Benson & Craig Chambers, editors: *OOPSLA*, ACM, pp. 362–373.

[4] Mariangiola Dezani-Ciancaglini, Paola Giannini & Betti Venneri (2018): *Intersection Types in Java: back to the future*. In Tiziana Margaria, Susanne Graf & Kim G Larsen, editors: *Models, Mindsets, Meta: The What, the How, and the Why Not?*, *LNCS* 11200, Springer, pp. 68–86.