Branching Types*

J. B. Wells[1] and Christian Haack[1]

Heriot-Watt University
http://www.cee.hw.ac.uk/ultra/

Abstract. Although systems with intersection types have many unique capabilities, there has never been a fully satisfactory explicitly typed system with intersection types. We introduce λ^{B} with *branching types* and types which are quantified over *type selectors* to provide an explicitly typed system with the same expressiveness as a system with intersection types. Typing derivations in λ^{B} effectively squash together what would be separate parallel derivations in earlier systems with intersection types.

1 Introduction

1.1 Background and Motivation

Intersection Types. Intersection types were independently invented near the end of the 1970s by Coppo and Dezani [3] and Pottinger [15]. Intersection types provide type polymorphism by listing type instances, differing from the more widely used $\forall$-quantified types [8, 16], which provide type polymorphism by giving a type scheme that can be instantiated into various type instances. The original motivation was for analyzing and/or synthesizing λ-models as well as in analyzing normalization properties, but over the last twenty years the scope of research on intersection types has broadened.

Intersection types have many unique advantages over $\forall$-quantified types. They can characterize the behavior of λ-terms more precisely, and can be used to express exactly the results of many program analyses [13, 1, 25, 26]. Type polymorphism with intersection types is also more flexible. For example, Urzyczyn [20] proved the λ-term

$$(\lambda x.z(x(\lambda fu.fu))(x(\lambda vg.gv)))(\lambda y.yyy)$$

to be untypable in the system F_ω, considered to be the most powerful type system with $\forall$-quantifiers measured by the set of pure λ-terms it can type. In contrast, this λ-term is typable with intersection types satisfying the *rank-3* restriction [12]. Better results for automated type inference (ATI) have also been obtained for intersection types. ATI for type systems with $\forall$-quantifiers that are more powerful than the very-restricted Hindley/Milner system is a murky area,

* This work was partly supported by NSF grants CCR 9113196, 9417382, 9988529, and EIA 9806745, EPSRC grants GR/L 36963 and GR/R 41545/01, and Sun Microsystems equipment grant EDUD-7826-990410-US.

and it has been proven for many such type systems that ATI algorithms can not be both complete and terminating [11, 23, 24, 20]. In contrast, ATI algorithms have been proven complete and terminating for the rank-k restriction for every finite k for several systems with intersection types [12, 10].

We use intersection types in typed intermediate languages (TILs) used in compilers. Using a TIL increases reliability of compilation and can support useful type-directed program transformations. We use intersection types because they support both more accurate analyses (as mentioned above) and interesting type/flow-directed transformations [5, 19, 7, 6] that would be very difficult using $\forall$-quantified types. When using a TIL, it is important to regularly check that the intermediate program representation is in fact well typed. Provided this is done, the correctness of any analyses encoded in the types is maintained across transformations. Thus, it is important for a TIL to be *explicitly* typed, i.e., to have type information attached to internal nodes of the program representation. This is necessary both for efficiency and because program transformations can yield results outside the domain of ATI algorithms. Unfortunately, intersection types raise troublesome issues for having an explicitly typed representation. This is the main motivation for this paper.

The Trouble with the Intersection-Introduction Rule. The important feature of a system with intersection types is this rule:

$$\frac{E \vdash M : \sigma; \quad E \vdash M : \tau}{E \vdash M : \sigma \wedge \tau} \ (\wedge\text{-intro})$$

The proof terms are the same for both premises and the conclusion! No syntax is introduced. A system with this rule does not fit into the proofs-as-terms (PAT, a.k.a. propositions-as-types and Curry/Howard) correspondence, because it has proof terms that do not encode deductions. Unfortunately, this is inadequate for many needs, and there is an immediate dilemma in how to make a type-annotated variant of the system. The usual strategy fails immediately, e.g.:

$$\frac{E \vdash (\lambda x{:}\sigma.\, x) : (\sigma \to \sigma); \quad E \vdash (\lambda x{:}\tau.\, x) : (\tau \to \tau)}{E \vdash (\lambda x{:}\boxed{???}\,.\, x) : (\sigma \to \sigma) \wedge (\tau \to \tau)}$$

Where $\boxed{???}$ appears, what should be written? This trouble is related to the fact that the $\wedge$ type constructor is not a truth-functional propositional connective.

Earlier Approaches. In the language Forsythe [17], Reynolds annotates the binding of $(\lambda x.M)$ with a list of types, e.g., $(\lambda x{:}\sigma_1|\cdots|\sigma_n.\, M)$. If the abstraction body M is typable with a fixed type τ for each type σ_i for x, then the abstraction gets the type $(\sigma_1 \to \tau) \wedge \cdots \wedge (\sigma_n \to \tau)$. However, this approach can not handle dependencies between types of nested variable bindings, e.g., this approach can not give $K = (\lambda x.\lambda y.x)$ the type $\tau_K = (\sigma \to (\sigma \to \sigma)) \wedge (\tau \to (\tau \to \tau))$.

Pierce [14] improves on Reynolds's approach by using a **for** construct which gives a type variable a finite set of types to range over, e.g., K can be annotated as (**for** $\alpha \in \{\sigma, \tau\}.\lambda x{:}\alpha.\ \lambda y{:}\alpha.\ x$) with the type τ_K. However, this approach can not represent some typings, e.g., it can not give the term $M_f = \lambda x.\lambda y.\lambda z.(xy, xz)$ the type $(((\alpha \to \delta) \wedge (\beta \to \epsilon)) \to \alpha \to \beta \to (\delta \times \epsilon)) \wedge ((\gamma \to \gamma) \to \gamma \to \gamma \to (\gamma \times \gamma))$. Pierce's approach could be extended to handle more complex dependencies if simultaneous type variable bindings were added, e.g., M_f could be annotated as:

$$\begin{array}{l}\textbf{for}\ \{[\theta \mapsto \alpha, \kappa \mapsto \beta, \eta \mapsto \delta, \nu \mapsto \epsilon], [\theta \mapsto \gamma, \kappa \mapsto \gamma, \eta \mapsto \gamma, \nu \mapsto \gamma]\}.\\ \quad \lambda x : (\theta \to \eta) \wedge (\kappa \to \nu)\, .\, \lambda y : \theta\, .\, \lambda z : \kappa\, .\, (xy, xz)\end{array}$$

Even this extension of Pierce's approach would still not meet our needs. First, this approach needs intersection types to be associative, commutative, and idempotent (ACI). Recent research suggests that non-ACI intersection types are needed to faithfully encode flow analyses [1]. Second, this approach arranges the type information inconveniently because it must be found from enclosing type variable bindings by a tree-walking process. This is bad for flow-based transformations, which reference arbitrary subterms from distant locations. Third, reasoning about typed terms in this approach is not compositional. It is not possible to look at an arbitrary subterm independently and determine its type.

The approach of λ^{CIL} [25, 26] is essentially to write the typing derivations as terms, e.g., K can be "annotated" as $\bigwedge((\lambda x{:}\sigma.\ \lambda y{:}\sigma.\ x), (\lambda x{:}\tau.\ \lambda y{:}\tau.\ x))$ in order to have the type τ_K. Here $\bigwedge(M, N)$ is a *virtual* tuple where the type erasure of M and N must be the same. In λ^{CIL}, subterms of an untyped term can have many disjoint representatives in a corresponding typed term. This makes it tedious and time-consuming to implement common program transformations, because *parallel contexts* must be used whenever subterms are transformed.

Venneri succeeded in completely removing the ($\wedge$-intro) rule from a type system with intersection types, but this was for combinatory logic rather than the λ-calculus [21, 4], and the approach seems unlikely to be transferable to the λ-calculus.

1.2 Contributions of this Paper

Our Approach: Branching Types. In this paper, we define and prove the basic properties of λ^{B}, a system with *branching types* which represent the effect of simultaneous derivations in the old style. Consider this untyped λ-term:

$$M_a = \lambda a.\lambda b.\lambda c.\ c\ (\lambda d.\ d\ a\ b)$$

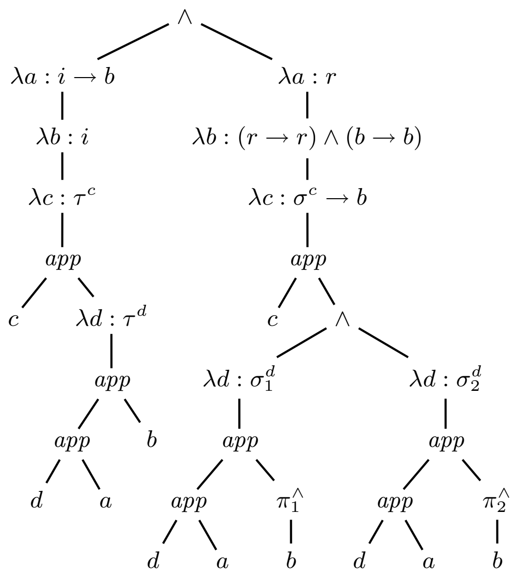

Fig. 1. A syntax tree in a λ^{CIL}-style system

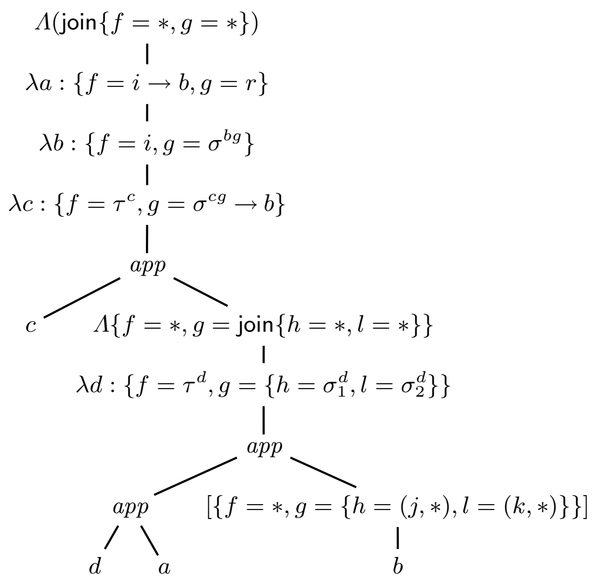

Fig. 2. A syntax tree in our system λ^{B}

In an intersection type system, M_a can have type $(\tau \wedge \sigma)$, where τ and σ are:

$$\begin{aligned}
\tau &= (i \to b) \to i \to \tau^c \to b \\
\tau^c &= (\tau^d \to b) \to b \\
\tau^d &= (i \to b) \to i \to b
\end{aligned}$$

$$\begin{aligned}
\sigma &= r \to ((r \to r) \wedge (b \to b)) \to (\sigma^c \to b) \to b \\
\sigma^c &= ((\sigma_1^d \to b) \wedge (\sigma_2^d \to b)) \\
\sigma_1^d &= r \to (r \to r) \to b \\
\sigma_2^d &= r \to (b \to b) \to b
\end{aligned}$$

In λ^{B}, correspondingly the term M_a can be annotated to have the type ρ where:

$$\begin{aligned}
\rho &= \forall(\mathsf{join}\{f = *, g = *\}).\, \{f = \tau, g = \sigma^g\}, \text{ where } \tau \text{ is as above} \\
\sigma^g &= r \to \sigma^{bg} \to (\sigma^{cg} \to b) \to b \\
\sigma^{bg} &= \forall(\mathsf{join}\{j = *, k = *\}).\, \{j = r \to r, k = b \to b\} \\
\sigma^{cg} &= \forall(\mathsf{join}\{h = *, l = *\}).\, \{h = \sigma_1^d \to b, l = \sigma_2^d \to b\}
\end{aligned}$$

Figure 1 shows the syntax tree of the corresponding explicitly typed term in a λ^{CIL}-style system and figure 2 shows the corresponding syntax tree in λ^{B}.

The λ^{CIL} tree can be seen to have 2 uses of ($\wedge$-intro), one at the root, and one inside the right child of the root. In the λ^{B} tree, the left branch of the λ^{CIL} tree is effectively named f, the right branch g, and the left and right subbranches of the inner ($\wedge$-intro) rule are named $g.h$ and $g.l$. Every type τ has a kind κ which indicates its branching shape. For example, the result type of the sole leaf occurrence of b has the kind $\{f = *, g = \{h = *, l = *\}\}$. This corresponds to the fact that in the λ^{CIL} derivation the leaf b is duplicated 3 times and occurs in the branches f, $g.h$, and $g.l$.

Each intersection type $\rho_1 \wedge \rho_2$ in this particular λ^{CIL} derivation has a corresponding type of the shape $\forall(\mathsf{join}\{f_1 = *, f_2 = *\}).\{f_1 = \rho_1', f_2 = \rho_2'\}$ in the λ^{B} derivation. A type of the shape $\forall P.\rho'$ has a *type selector parameter* P which is a pattern indicating what possible *type selector arguments* are valid to supply. Each parameter P has 2 kinds, its *inner kind* $\lfloor P \rfloor$ and its *outer kind* $\lceil P \rceil$. The kind of a type $\forall P.\rho'$ is $\lfloor P \rfloor$ and the kind of ρ' must be $\lceil P \rceil$.

One of the most important features of λ^{B} is its equivalences among types. The first two type equivalence rules are:

$$\forall *.\tau \;\simeq\; \tau \qquad\qquad \forall\{f_i = P_i\}^{i \in I}.\{f_i = \tau_i\}^{i \in I} \;\simeq\; \{f_i = \forall P_i.\tau_i\}^{i \in I}$$

To illustrate their use, we explain why the application of b to its type selector argument is well-typed in the example given above. The type of b at its binding site is written as $\{f = i, g = \sigma^{bg}\}$ and has kind $\{f = *, g = *\}$. Because the leaf occurrence of b must have kind $\{f = *, g = \{h = *, l = *\}\}$, the type at that location is expanded by the typing rules to be $\{f = i, g = \{h = \sigma^{bg}, l = \sigma^{bg}\}\}$. Then, using the additional names

$$\begin{aligned}
P_{jk} &= \mathsf{join}\{j = *, k = *\} \\
\sigma_2^{bg} &= \{j = r \to r, k = b \to b\}
\end{aligned}$$

so that $\sigma^{bg} = \forall P_{jk}.\sigma_2^{bg}$, the equivalences are applied to the type as follows to lift the occurrences of $\forall P$ to outermost position:

$$\begin{aligned} &\{f = i, g = \{h = \sigma^{bg}, l = \sigma^{bg}\}\} \\ \simeq\ &\{f = (\forall * .i), g = \forall\{h = P_{jk}, l = P_{jk}\}.\{h = \sigma_2^{bg}, l = \sigma_2^{bg}\}\} \\ \simeq\ &\forall\{f = *, g = \{h = P_{jk}, l = P_{jk}\}\}.\{f = i, g = \{h = \sigma_2^{bg}, l = \sigma_2^{bg}\}\} \end{aligned}$$

The final type is the type actually used in figure 2. The type selector parameter of the final type effectively says, "in the f branch, no type selector argument can be supplied and in the $g.h$ and $g.l$ branches, a choice between j and k can be supplied". This is in fact what the type selector argument $\{f = *, g = \{h = (j, *), l = (k, *)\}\}$ does; it supplies no choice in the f branch, a choice of j in the $g.h$ branch, and a choice of k in the $g.l$ branch. The $*$ in $(j, *)$ means that after the choice of j is supplied, no further choices are supplied. The use of type selector parameters and arguments in terms takes the place of the $\wedge$-introduction and $\wedge$-elimination rules of a system with intersection types.

The example just preceding of the type of b illustrated 2 of the 3 type equivalence rules. The third rule, which is particularly important because it allows using the usual typing rules for λ-calculus abstraction and application, is this:

$$\{f_i = \sigma_i\}^{i\in I} \to \{f_i = \tau_i\}^{i\in I} \quad \simeq \quad \{f_i = \sigma_i \to \tau_i\}^{i\in I}$$

We now give an example of where this rule was used in the earlier typing example. The binding type of c is a branching type, but the type of the leaf occurrence of c needs to be a function type in order to be applied to its argument. Fortunately, the type equivalence gives the following, providing exactly the type needed:

$$\{f = (\tau^d \to b) \to b, g = \sigma^{cg} \to b\} \simeq \{f = \tau^d \to b, g = \sigma^{cg}\} \to \{f = b, g = b\}$$

Another important feature of λ^{B} is its reduction rules which manipulate and simplify the type annotations as needed. As an example, consider the λ^{B} term $M = (\mathit{\Lambda} P_1.\mathit{\Lambda} P_2.\lambda x^\tau.x)[A]$, where:

$$\begin{aligned} P_1 &= \{f = \mathsf{join}\{j = *, k = *\}, g = *\}, \quad & P_2 &= \{f = *, g = \mathsf{join}\{h = *, l = *\}\} \\ \tau &= \{f = \{j = \alpha_1, k = \alpha_2\}, g = \{h = \beta_1, l = \beta_2\}\}, \quad & A &= \{f = *, g = (h, *)\} \end{aligned}$$

The term M first reduces to $\mathit{\Lambda} P_1.(\mathit{\Lambda} P_2.\lambda x^\tau.x)[A]$, by a $(\beta_{\mathit{\Lambda}})$-step where P_1 and A pass through each other without interacting. By another $(\beta_{\mathit{\Lambda}})$-step, it reduces to $\mathit{\Lambda} P_1.\mathit{\Lambda} P_2'.(\lambda x^\sigma.x)[A']$, where:

$$P_2' = \{f = *, g = *\},\ \sigma = \{f = \{j = \alpha_1, k = \alpha_2\}, g = \beta_1\}, \text{ and } A' = \{f = *, g = *\}$$

Finally, it reduces to $\mathit{\Lambda} P_1.\lambda x^\sigma.x$, removing the trivial P_2' and A' by a $(*_{\mathit{\Lambda}})$-step and a $(*_A)$-step.

Recent Related Work. Ronchi Della Rocca and Roversi have a system called Intersection Logic (IL) [18] which is similar to λ^{B}, but has nothing corresponding to our explicitly typed terms. IL has a meta-level operation corresponding to our equivalence for function types. IL has nothing corresponding to our other type equivalences, because IL does not group parallel occurrences of its equivalent of type selector parameters and arguments, but instead works with equivalence classes of derivations modulo permutations of what we call *individual* type selector parameters and arguments. We expect that the use of these equivalence classes will cause great difficulty with the proofs. Much of the proof burden for the IL system (corresponding to a large portion of this paper) is inside the 1-line proof of their lemma 4 in the calculation of $S(\Pi, \Pi')$ where S is not constructively specified. A proof-term-labelled version of IL is presented, but the proof terms are pure λ-terms and thus the proof terms do not represent entire derivations.

Capitani, Loreti, and Venneri have designed a similar system called HL (Hyperformulae Logic) [2]. HL is quite similar to IL, although it seems overall to have a less complicated presentation. HL has nothing corresponding to our equivalences on types. The set of properties proved for HL in [2] is not exactly the same as the set of properties proved for IL in [18], e.g., there is no attempt to directly prove any result related to reduction of HL proofs as there is for IL, although this could be obtained indirectly via their proofs of equivalence with traditional systems with intersection types. HL is reported in [18] to have a typed version of an untyped calculus like that in [9], but in fact there is no significant connection between [2] and [9] and there is no explicitly typed calculus associated with HL.

For both IL and HL, there are proofs of equivalence with earlier systems with intersection types. These proofs show that the proof-term-annotated versions of IL and HL can type the same sets of pure untyped λ-terms as a traditional system with intersection types. We expect that this could also be done for our system λ^{B}, but we have not yet bothered to complete such a proof because it is a theoretical point which does not directly affect our ability to use λ^{B} as a basis for representing analysis information in implementations.

Summary of Contributions (i.e., the Conclusion). In this paper, we present λ^{B}, the first typed calculus with the power of intersection types which is Church-style, i.e., typed terms do not have multiple disjoint subterms corresponding to single subterms in the corresponding untyped term. We prove for λ^{B} subject reduction, and soundness and completeness of typed reduction w.r.t. β-reduction on the corresponding untyped λ-terms. The main benefit of λ^{B} will be to make it easier to use technology (both theories and software) already developed for the λ-calculus on typed terms in a type system having the power and flexibility of intersection types. Due to the experimental performance measurements reported in [7], we do not expect a substantial size benefit in practice from λ^{B} over λ^{CIL}. In the area of logic, λ^{B} terms may be useful as typed realizers of the so-called *strong conjunction*, but we are not currently planning on investigating this ourselves.

2 Some Notational Conventions

We use expressions of the form $\{f_i = E_i\}^I$, where I is a finite index set, the f_i's are drawn from a fixed set of labels and the E_i are expressions that may depend on i. Such an expression stands for the set $\{\, (f_i, E_i) \mid i \in I \,\}$.

Let an expression E be called *partially defined* whenever E is built using partial functions. Such an expression E defines an object iff all of its subexpressions are also defined; this will depend on the valuation for E's free variables. Given a binary relation $\mathcal{R}$ and partially defined expressions E_1 and E_2, let $(E_1\ \mathcal{R}^\perp\ E_2)$ hold iff either both E_1 and E_2 are undefined or E_1 denotes an object x_1 and E_2 an object x_2 such that $(x_1\ \mathcal{R}\ x_2)$.

3 Types and Kinds

3.1 Kinds

Let Label be a countably infinite set of labels. Let f and g range over Label. The set Kind of *kinds* is defined by the following pseudo-grammar:

$$\kappa \in \mathsf{Kind} ::= * \ \mid\ \{f_i = \kappa_i\}^I$$

We define a partial order on kinds, inductively by the following rules:

$$\frac{}{* \leq \kappa} \qquad\qquad \frac{(\kappa_i \leq \kappa'_i) \text{ for all } i \text{ in } I}{\{f_i = \kappa_i\}^I \leq \{f_i = \kappa'_i\}^I}$$

Lemma 1. *The relation $\leq$ is a partial order on the set of kinds.* $\square$

3.2 Types

The sets $\mathsf{Parameter}$ of *type selector parameters* and $\mathsf{IndParameter}$ of *individual type selector parameters* are defined by the following pseudo-grammar:

$$\begin{aligned} P \in \mathsf{Parameter} &::= \bar{P} \ \mid\ \{f_i = P_i\}^I \\ \bar{P} \in \mathsf{IndParameter} &::= * \ \mid\ \mathsf{join}\{f_i = \bar{P}_i\}^I \end{aligned}$$

Given a parameter P, let P's *inner kind* $\lfloor P \rfloor$ and *outer kind* $\lceil P \rceil$ be defined as follows:

$$\begin{aligned} &\lfloor * \rfloor = *, \quad \lfloor \mathsf{join}\{f_i = \bar{P}_i\}^I \rfloor = *, && \lfloor \{f_i = P_i\}^I \rfloor = \{f_i = \lfloor P_i \rfloor\}^I \\ &\lceil * \rceil = *, \quad \lceil \mathsf{join}\{f_i = \bar{P}_i\}^I \rceil = \{f_i = \lceil \bar{P}_i \rceil\}^I, && \lceil \{f_i = P_i\}^I \rceil = \{f_i = \lceil P_i \rceil\}^I \end{aligned}$$

Let $\mathsf{TypeVar}$ be a countably infinite set of type variables. Let α and β range over $\mathsf{TypeVar}$. Let the set of types Type be given by the following pseudo-grammar:

$$\sigma, \tau \in \mathsf{Type} ::= \alpha \ \mid\ \sigma \to \tau \ \mid\ \{f_i = \tau_i\}^I \ \mid\ \forall P.\tau$$

The relation assigning kinds to types is given by these rules:

$$\frac{}{\alpha : *} \qquad \frac{\sigma : \kappa; \quad \tau : \kappa}{\sigma \to \tau : \kappa} \qquad \frac{\tau_i : \kappa_i \text{ for every } i \text{ in } I}{\{f_i = \tau_i\}^I : \{f_i = \kappa_i\}^I} \qquad \frac{\tau : \lceil P \rceil}{\forall P.\tau : \lfloor P \rfloor}$$

Note that every type has at most one kind. A type τ is called *well-formed* if there is a kind κ such that $(\tau : \kappa)$. A well-formed type is called *individual* if its kind is $*$. A well-formed type is called *branching* if its kind is not $*$. Individual types correspond to single types in the world of intersection types, whereas branching types correspond to collections of types where each type in the collection would occur in a separate derivation.

3.3 Type Equivalences

In order to be able to treat certain types as having essentially the same meaning, we define an equivalence relation on the set of types as follows. A binary relation $\mathcal{R} \subseteq \mathsf{Type} \times \mathsf{Type}$ is called *compatible* if it satisfies the following rules:

$$\begin{array}{l}
(\sigma \; \mathcal{R} \; \sigma') \;\Rightarrow\; ((\sigma \to \tau) \; \mathcal{R} \; (\sigma' \to \tau)) \\
(\tau \; \mathcal{R} \; \tau') \;\Rightarrow\; ((\sigma \to \tau) \; \mathcal{R} \; (\sigma \to \tau')) \\
((\tau_j \; \mathcal{R} \; \tau'_j) \wedge (j \notin I)) \;\Rightarrow\; (\{f_i = \tau_i\}^I \cup \{f_j = \tau_j\} \; \mathcal{R} \; \{f_i = \tau_i\}^I \cup \{f_j = \tau'_j\}) \\
(\tau \; \mathcal{R} \; \sigma) \;\Rightarrow\; ((\forall P.\tau) \; \mathcal{R} \; (\forall P.\sigma))
\end{array}$$

Let $\succ$ be the least compatible relation that contains all instances of the rules (1) through (3), below. Let $\succeq$ denote the reflexive and transitive closure of $\succ$, and $\simeq$ the least compatible equivalence relation that contains all instances of (1) through (3).

$$\forall *.\tau \; \mathcal{R} \; \tau \tag{1}$$

$$\forall \{f_i = P_i\}^I.\{f_i = \tau_i\}^I \; \mathcal{R} \; \{f_i = \forall P_i.\tau_i\}^I \tag{2}$$

$$\{f_i = \sigma_i\}^I \to \{f_i = \tau_i\}^I \; \mathcal{R} \; \{f_i = \sigma_i \to \tau_i\}^I \tag{3}$$

Lemma 2. *If $(\tau \simeq \sigma)$ and $(\tau : \kappa)$ then $(\sigma : \kappa)$.* □

Lemma 3 (Confluence).

1. *If $\tau_1 \succ \tau_2$ and $\tau_1 \succ \tau_3$, then there is a type τ_4 such that $\tau_2 \succ \tau_4$ and $\tau_3 \succ \tau_4$.*
2. *If $\tau_1 \succeq \tau_2$ and $\tau_1 \succeq \tau_3$, then there is a type τ_4 such that $\tau_2 \succeq \tau_4$ and $\tau_3 \succeq \tau_4$.*

□

Proof Sketch. An inspection of the reduction rules shows the following: Whenever a redex τ contains another redex σ, then σ still occurs in the contractum of τ. Moreover, σ doesn't get duplicated in the contraction step. For this reason, statement (1) holds. Statement (2) follows from (1) by a standard argument. □

Lemma 4 (Termination). *There is no infinite sequence $(\tau_n)_{n \in \mathbb{N}}$ such that $\tau_n \succ \tau_{n+1}$ for all n in $\mathbb{N}$.* □

Proof. Define a weight function $\|\cdot\|$ on types by

$$\|\tau\| = \begin{pmatrix} \text{(no. of occurrences of labels in } \tau) \\ + \text{ (no. of occurrences of } * \text{ in } \tau) \end{pmatrix}$$

An inspection of the reduction rules shows that $\tau \succ \sigma$ implies $\|\tau\| > \|\sigma\|$. Therefore, every reduction sequence is finite. □

3.4 Normal Types

A type τ is called *normal* if there is no type σ such that $\tau \succ \sigma$. We abbreviate the statement that τ is normal by $\mathsf{normal}(\tau)$. For any type τ, let $\mathsf{nf}(\tau)$ be the unique normal type σ such that $\tau \succeq \sigma$, which is proven to exist by Lemma 5 below.

Lemma 5.
For every type τ there is a unique normal type σ such that $\tau \succeq \sigma$. □

Lemma 6. $(\tau \simeq \sigma)$ *if and only if* $(\mathsf{nf}(\tau) = \mathsf{nf}(\sigma))$. □

An important property of normal types is that their top-level structure reflects the top-level structure of their kinds. In particular, if a type is normal and individual, then it is not of the form $\{f_i = \tau_i\}^I$. This is made precise in the following lemma:

Lemma 7. *If $(\tau : \{f_i = \kappa_i\}^I)$ and τ is normal, then there is a family $(\tau_i)_{i \in I}$ of normal types such that $(\tau = \{f_i = \tau_i\}^I)$ and $(\tau_i : \kappa_i)$ for all i in I.* □

4 Expansion and Selection for Types

4.1 Expansion

For the typing rules, we need to define some auxiliary operations on types. First, we define a partial function expand from $\mathsf{Type} \times \mathsf{Parameter}$ to Type. Applying expand to the pair (τ, P) adjusts the type τ of branching shape $\lfloor P \rfloor$ to the new branching shape $\lceil P \rceil$ (of which $\lfloor P \rfloor$ is an initial segment) by duplicating subterms of τ. The duplication is caused by the second of the defining equations below. The expansion operation is used in the typing rule that corresponds to the intersection introduction rule, in order to adjust the types of free variables to a new branching shape. The additional branches in the new branching shape $\lceil P \rceil$ correspond to different type derivations for the same term in an intersection type system.

$$\begin{aligned} \mathsf{expand}(\tau, *) &= \tau, && \text{if } \mathsf{normal}(\tau) \\ \mathsf{expand}(\tau, \mathsf{join}\{f_i = \bar{P}_i\}^I) &= \{f_i = \mathsf{expand}(\tau, \bar{P}_i)\}^I, && \text{if } \mathsf{normal}(\tau) \\ \mathsf{expand}(\{f_i = \tau_i\}^I, \{f_i = P_i\}^I) &= \{f_i = \mathsf{expand}(\tau_i, P_i)\}^I, && \text{if } \mathsf{normal}(\{f_i = \tau_i\}^I) \\ \mathsf{expand}(\tau, P) &= \mathsf{expand}(\mathsf{nf}(\tau), P), && \text{if } \neg\mathsf{normal}(\tau) \end{aligned}$$

Lemma 8.

1. *If* $(\lfloor P \rfloor \leq \kappa)$ *and* $(\tau : \kappa)$*, then* $\mathsf{expand}(\tau, P)$ *is defined.*
2. $(\tau : \lfloor P \rfloor)$ *if and only if* $\mathsf{expand}(\tau, P) : \lceil P \rceil$*.* □

Lemma 9.
If $(\sigma \rightarrow \tau : \kappa)$*, then* $(\mathsf{expand}(\sigma \rightarrow \tau, P) \simeq^{\perp} \mathsf{expand}(\sigma, P) \rightarrow \mathsf{expand}(\tau, P))$*.* □

4.2 Type Selector Arguments and Selection

The sets $\mathsf{Argument}$ of *type selector arguments* and $\mathsf{IndArgument}$ of *individual type selector arguments* are defined by the following pseudo-grammar:

$$\begin{aligned} A \in \mathsf{Argument} &::= \bar{A} \mid \{f_i = A_i\}^I \\ \bar{A} \in \mathsf{IndArgument} &::= * \mid f, \bar{A} \end{aligned}$$

We define two relations that assign kinds to arguments:

$$\frac{}{* : *} \qquad \frac{\bar{A} : *}{f, \bar{A} : *} \qquad \frac{A_i : \kappa_i \quad \text{for all } i \text{ in } I}{\{f_i = A_i\}^I : \{f_i = \kappa_i\}^I}$$

$$\frac{}{* \lhd \kappa} \qquad \frac{\bar{A} \lhd \kappa_j}{f_j, \bar{A} \lhd \{f_i = \kappa_i\}^I} \text{ if } j \in I \qquad \frac{A_i \lhd \kappa_i \quad \text{for all } i \text{ in } I}{\{f_i = A_i\}^I \lhd \{f_i = \kappa_i\}^I}$$

Note that for every argument A there is exactly one kind κ such that $(A : \kappa)$. On the other hand, there are many kinds κ such that $(A \lhd \kappa)$.

Lemma 10. *If* $(A \lhd \kappa)$ *and* $(\kappa \leq \kappa')$*, then* $(A \lhd \kappa')$*.* □

We define two partial functions $\mathsf{select}^{\mathsf{i}}$ and $\mathsf{select}^{\mathsf{b}}$. Both go from $\mathsf{Type} \times \mathsf{Argument}$ to Type. The two functions are similar. The main difference is that $\mathsf{select}^{\mathsf{i}}$ performs selections on individual (joined) types, whereas $\mathsf{select}^{\mathsf{b}}$ performs selections on branching types. We define the functions in two steps, first for the case where the function's first argument is a normal type:

$$\begin{aligned} \mathsf{select}^{\mathsf{i}}(\tau, *) &= \tau, \quad \text{if } \tau \text{ is individual} \\ \mathsf{select}^{\mathsf{i}}(\forall(\mathsf{join}\{f_i = \bar{P}_i\}^I).\{f_i = \tau_i\}^I, (f_j, \bar{A})) &= \mathsf{select}^{\mathsf{i}}(\forall \bar{P}_j.\tau_j, \bar{A}), \quad \text{if } (j \in I) \\ \mathsf{select}^{\mathsf{i}}(\{f_i = \tau_i\}^I, \{f_i = A_i\}^I) &= \{f_i = \mathsf{select}^{\mathsf{i}}(\tau_i, A_i)\}^I \end{aligned}$$

$$\begin{aligned} \mathsf{select}^{\mathsf{b}}(\tau, *) &= \tau \\ \mathsf{select}^{\mathsf{b}}(\{f_i = \tau_i\}^I, (f_j, \bar{A})) &= \mathsf{select}^{\mathsf{b}}(\tau_j, \bar{A}), \quad \text{if } (j \in I) \\ \mathsf{select}^{\mathsf{b}}(\{f_i = \tau_i\}^I, \{f_i = A_i\}^I) &= \{f_i = \mathsf{select}^{\mathsf{b}}(\tau_i, A_i)\}^I \end{aligned}$$

Now, for the case where the function's first argument is a type that isn't normal:

$$\begin{aligned} \mathsf{select}^{\mathsf{i}}(\tau, A) &= \mathsf{select}^{\mathsf{i}}(\mathsf{nf}(\tau), A) \\ \mathsf{select}^{\mathsf{b}}(\tau, A) &= \mathsf{select}^{\mathsf{b}}(\mathsf{nf}(\tau), A) \end{aligned}$$

Lemma 11.

1. *If* $(\tau : \kappa)$ *and* $\mathsf{select}^{\mathsf{i}}(\tau, A)$ *is defined, then* $(A : \kappa)$*.*
2. *If* $(\tau : \kappa)$ *and* $(\mathsf{select}^{\mathsf{i}}(\tau, A) = \tau')$*, then* $(\tau' : \kappa)$*.*
3. *If* $(\tau : \kappa)$ *and* $\mathsf{select}^{\mathsf{b}}(\tau, A)$ *is defined, then* $(A \triangleleft \kappa)$*.*

□

Lemma 12. *If* $(f \in \{\mathsf{select}^{\mathsf{i}}, \mathsf{select}^{\mathsf{b}}\})$ *and* $(\sigma \to \tau : \kappa)$*, then* $(f(\sigma \to \tau, A) \simeq^{\perp} f(\sigma, A) \to f(\tau, A))$*.* □

5 Terms and Typing Rules

Let $\mathsf{TermVar}$ be a countably infinite set of λ-term variables. Let x range over $\mathsf{TermVar}$. Let the set Term of explicitly typed λ-terms be given by the following pseudo-grammar:

$$M, N \in \mathsf{Term} ::= \Lambda P.M \;\mid\; M[A] \;\mid\; \lambda x^{\tau}.M \;\mid\; M\ N \;\mid\; x^{\tau}$$

The λx binds the variable x in the usual way.[1] We identify terms that are equal up to renaming of bound variables.

A *type environment* is defined to be a finite function from $\mathsf{TermVar}$ to Type. We use the metavariable E to range over type environments. We extend the definitions of expansion, kind assignment and type equality to type environments:

$$\begin{aligned}
\mathsf{expand}(E, P)(x) \quad &= \quad \mathsf{expand}(E(x), P) \\
E : \kappa &\overset{\text{def}}{\Longleftrightarrow} E(x) : \kappa \text{ for all } x \text{ in } \mathsf{dom}(E) \\
E \simeq E' &\overset{\text{def}}{\Longleftrightarrow} \begin{cases} \mathsf{dom}(E) = \mathsf{dom}(E') \text{ and} \\ (E(x) \simeq E'(x)) \text{ for all } x \text{ in } \mathsf{dom}(E) \end{cases}
\end{aligned}$$

Typing judgement are of the form:

$$E \vdash M : \tau \;\mathsf{at}\; \kappa$$

The valid typing judgement are those that can be proven using the typing rules in Figure 3.

Lemma 13.

1. *If* $(E \vdash M : \tau \;\mathsf{at}\; \kappa)$*, then* $(\tau : \kappa)$*.*
2. *If* $(E \vdash M : \tau \;\mathsf{at}\; \kappa)$*, then* $(E : \kappa)$*.*
3. *If* $(E \vdash M : \tau \;\mathsf{at}\; \kappa)$ *and* $E \simeq E'$*, then* $(E' \vdash M : \tau \;\mathsf{at}\; \kappa)$*.* □

6 Expansion, Selection and Substitution for Terms

In this section, we define auxiliary operations that are needed for the statements of the term reduction rules.

[1] In this language, ΛP does *not* bind any variables!

$$\text{(ax)} \qquad \frac{}{E \vdash x^{E(x)} : E(x) \ \mathsf{at} \ \kappa} \ \text{if } (E : \kappa)$$

$$(\rightarrow_{\mathsf{i}}) \qquad \frac{E[x \mapsto \sigma] \vdash M : \tau \ \mathsf{at} \ \kappa}{E \vdash \lambda x^{\sigma}.M : \sigma \rightarrow \tau \ \mathsf{at} \ \kappa}$$

$$(\rightarrow_{\mathsf{e}}) \qquad \frac{E \vdash M : \sigma \rightarrow \tau \ \mathsf{at} \ \kappa; \quad E \vdash N : \sigma \ \mathsf{at} \ \kappa}{E \vdash M \ N : \tau \ \mathsf{at} \ \kappa}$$

$$(\forall_{\mathsf{i}}) \qquad \frac{\mathsf{expand}(E, P) \vdash M : \tau \ \mathsf{at} \ \lceil P \rceil}{E \vdash \Lambda P.M : \forall P.\tau \ \mathsf{at} \ \lfloor P \rfloor}$$

$$(\forall_{\mathsf{e}}) \qquad \frac{E \vdash M : \tau \ \mathsf{at} \ \kappa}{E \vdash M[A] : \tau' \ \mathsf{at} \ \kappa} \ \text{if } (\mathsf{select}^{\mathsf{i}}(\tau, A) = \tau')$$

$$(\simeq) \qquad \frac{E \vdash M : \tau \ \mathsf{at} \ \kappa}{E \vdash M : \tau' \ \mathsf{at} \ \kappa} \ \text{if } (\tau \simeq \tau')$$

Fig. 3. Typing Rules

6.1 Expansion

In order to define substitution and β-reduction, we need to extend the expand operation to terms. This is necessary because the branching shape of type annotations changes when a term is substituted into a new context. First, expand is extended to parameters, arguments and kinds, by the following equations where X ranges over $\mathsf{Parameter} \cup \mathsf{Argument} \cup \mathsf{Kind}$:

$$\mathsf{expand}(X, *) = X$$
$$\mathsf{expand}(X, \mathsf{join}\{f_i = \bar{P}_i\}^I) = \{f_i = \mathsf{expand}(X, \bar{P}_i)\}^I$$
$$\mathsf{expand}(\{f_i = X_i\}^I, \{f_i = P_i\}^I) = \{f_i = \mathsf{expand}(X_i, P_i)\}^I$$

Now, the expand operation is inductively extended to terms:

$$\mathsf{expand}(\Lambda P'.M, P) = \Lambda(\mathsf{expand}(P', P)).\ \mathsf{expand}(M, P)$$
$$\mathsf{expand}(M[A], P) = (\mathsf{expand}(M, P))[\mathsf{expand}(A, P)]$$
$$\mathsf{expand}(\lambda x^{\tau}.M, P) = \lambda x^{\mathsf{expand}(\tau, P)}.\ \mathsf{expand}(M, P)$$
$$\mathsf{expand}(M \ N, P) = (\mathsf{expand}(M, P)) \ (\mathsf{expand}(N, P))$$
$$\mathsf{expand}(x^{\tau}, P) = x^{\mathsf{expand}(\tau, P)}$$

Lemma 14. *If* $(E \vdash M : \tau \ \mathsf{at} \ \kappa)$ *and* $(\lfloor P \rfloor \leq \kappa)$,
then $(\mathsf{expand}(E, P) \vdash \mathsf{expand}(M, P) : \mathsf{expand}(\tau, P) \ \mathsf{at} \ \mathsf{expand}(\kappa, P))$. □

Corollary 1. *If* $(E \vdash M : \tau \ \mathsf{at} \ \lfloor P \rfloor)$,
then $(\mathsf{expand}(E, P) \vdash \mathsf{expand}(M, P) : \mathsf{expand}(\tau, P) \ \mathsf{at} \ \lceil P \rceil)$. □

6.2 Substitution

A *substitution* is a finite function from $\mathsf{TermVar}$ to Term. We extend the operation expand to substitutions as follows:

$$\mathsf{expand}(s, P)(x) = \mathsf{expand}(s(x), P)$$

We now define the *application of a substitution* s to a term M. The definition is by induction on the structure of the term:

$$
\begin{aligned}
s(\Lambda P.M) &= \Lambda P.\ \mathsf{expand}(s, P)(M) \\
s(M[A]) &= (s(M))[A] \\
s(\lambda x^\tau.M) &= \lambda x^\tau.\ s(M), \quad \text{if } x \text{ does not occur freely in } \mathsf{ran}(s) \text{ and } x \notin \mathsf{dom}(s) \\
s(M\ N) &= s(M)\ s(N) \\
s(x^\tau) &= s(x), \quad \text{if } x \in \mathsf{dom}(s) \\
s(x^\tau) &= x^\tau, \quad \text{if } x \notin \mathsf{dom}(s)
\end{aligned}
$$

We write $M[x := N]$ for the term that results from applying the singleton substitution $\{(x, N)\}$ to M.

We define typing judgement for substitutions as follows:

$$(E' \vdash s : E \ \mathsf{at}\ \kappa) \overset{\text{def}}{\Longleftrightarrow} (E' \vdash s(x^{E(x)}) : E(x) \ \mathsf{at}\ \kappa) \text{ for all } x \text{ in } \mathsf{dom}(E)$$

Lemma 15. *If* $(E' \vdash s : E \ \mathsf{at}\ \lfloor P \rfloor)$,
then $(\mathsf{expand}(E', P) \vdash \mathsf{expand}(s, P) : \mathsf{expand}(E, P) \ \mathsf{at}\ \lceil P \rceil)$. □

Lemma 16. *If* $(E \vdash M : \tau \ \mathsf{at}\ \kappa)$ *and* $(E' \vdash s : E \ \mathsf{at}\ \kappa)$,
then $(E' \vdash s(M) : \tau \ \mathsf{at}\ \kappa)$. □

6.3 Selection

The preceding treatment of substitution prepares for the definition of β-reduction of terms of the form $((\lambda x^\tau.M)N)$. We also need to define reduction of terms of the form $(\Lambda P.M)[A]$. To this end, we extend the $\mathsf{select}^\mathsf{b}$ to parameters, arguments and kinds, by the following equations where X ranges over $\mathsf{Parameter} \cup \mathsf{Argument} \cup \mathsf{Kind}$:

$$
\begin{aligned}
\mathsf{select}^\mathsf{b}(X, *) &= X \\
\mathsf{select}^\mathsf{b}(\{f_i = X_i\}^I, (f_j, \bar{A})) &= \mathsf{select}^\mathsf{b}(X_j, \bar{A}), \quad \text{if } j \in I \\
\mathsf{select}^\mathsf{b}(\{f_i = X_i\}^I, \{f_i = A_i\}^I) &= \{f_i = \mathsf{select}^\mathsf{b}(X_i, A_i)\}^I
\end{aligned}
$$

Lemma 17. $(A \lhd \kappa)$ *if and only if* $\mathsf{select}^\mathsf{b}(\kappa, A)$ *is defined.* □

The $\mathsf{select^b}$ operation is extended to terms by induction on the structure of the term:

$$\begin{aligned}
\mathsf{select^b}(\Lambda P'.M, A) &= \Lambda(\mathsf{select^b}(P', A)).\ \mathsf{select^b}(M, A)\\
\mathsf{select^b}(M[A'], A) &= (\mathsf{select^b}(M, A))[\mathsf{select^b}(A', A)]\\
\mathsf{select^b}(\lambda x^\tau.M, A) &= \lambda x^{\mathsf{select^b}(\tau, A)}.\ \mathsf{select^b}(M, A)\\
\mathsf{select^b}(M\ N, A) &= (\mathsf{select^b}(M, A))\ (\mathsf{select^b}(N, A))\\
\mathsf{select^b}(x^\tau, A) &= x^{\mathsf{select^b}(\tau, A)}
\end{aligned}$$

The $\mathsf{select^b}$ operation is extended to type environments as follows:

$$\mathsf{select^b}(E, A)(x) = \mathsf{select^b}(E(x), A)$$

Lemma 18. *If* $(E \vdash M : \tau \ \mathsf{at}\ \kappa)$ *and* $(A \lhd \kappa)$,
then $(\mathsf{select^b}(E, A) \vdash \mathsf{select^b}(M, A) : \mathsf{select^b}(\tau, A)\ \mathsf{at}\ \mathsf{select^b}(\kappa, A))$. $\square$

7 Reduction Rules for Terms

We define a partial function match from $\mathsf{Parameter} \times \mathsf{Argument}$ to $\mathsf{Parameter} \times \mathsf{Argument} \times \mathsf{Argument}$.[2] This function is needed for the reduction rule (β_Λ) for type selection.

$$\frac{}{\mathsf{match}(*, \bar{A}) = (*, *, \bar{A})} \qquad\qquad \frac{}{\mathsf{match}(\bar{P}, *) = (\bar{P}, *, \lceil \bar{P} \rceil)}$$

$$\frac{\mathsf{match}(\bar{P}_j, \bar{A}) = (\bar{P}', \bar{A}^{\mathsf{s}}, A^{\mathsf{a}})}{\mathsf{match}(\mathsf{join}\{f_i = \bar{P}_i\}^I, (f_j, \bar{A})) = (\bar{P}', (f_j, \bar{A}^{\mathsf{s}}), A^{\mathsf{a}})}\ \text{if } j \in I$$

$$\frac{\mathsf{match}(P_i, A_i) = (P_i', A_i^{\mathsf{s}}, A_i^{\mathsf{a}}) \quad \text{for all } i \text{ in } I}{\mathsf{match}(\{f_i = P_i\}^I, \{f_i = A_i\}^I) = (\{f_i = P_i'\}^I, \{f_i = A_i^{\mathsf{s}}\}^I, \{f_i = A_i^{\mathsf{a}}\}^I)}$$

A parameter or argument is called *trivial* if it is also a kind.

Lemma 19.

1. *If P is a trivial parameter and $(\forall P.\tau : \kappa)$, then $(\forall P.\tau \simeq \tau)$.*
2. *If A is a trivial argument and $(\mathsf{select^i}(\tau, A) = \tau')$, then $(\tau' \simeq \tau)$.* $\square$

The reduction relation for terms is defined as the least compatible relation of terms that contains the following axioms:

$$\begin{array}{ll}
(\beta_\lambda) & ((\lambda x^\tau.M)N) \to (M[x := N])\\
(\beta_\Lambda) & (\Lambda P.M)[A] \to \Lambda P'.((\mathsf{select^b}(M, A^{\mathsf{s}}))[A^{\mathsf{a}}]),\\
& \text{if } \mathsf{match}(P, A) = (P', A^{\mathsf{s}}, A^{\mathsf{a}}) \text{ and neither } P \text{ nor } A \text{ is trivial}\\
(*_\Lambda) & (\Lambda P.M) \to M, \quad \text{if } P \text{ is trivial}\\
(*_A) & (M[A]) \to M, \quad \text{if } A \text{ is trivial}
\end{array}$$

[2] In the second rule, note that $\lceil P \rceil \in \mathsf{Argument}$ for all parameters P.

Theorem 1 (Subject Reduction). *If* $(M \to N)$ *and* $(E \vdash M : \tau \text{ at } \kappa)$*, then* $(E \vdash N : \tau \text{ at } \kappa)$. □

Proof. For (β_λ), one uses Lemma 16. For $(*_A)$ and $(*_\Lambda)$, one uses Lemma 19. For (β_Λ) one uses Lemma 18 and some other technical Lemmas, which we have omitted because of space constraints. □

8 Correspondence of Typed and Untyped Reduction

The set $\mathsf{UntypedTerm}$ of *untyped terms* is defined by the following pseudo-grammar:

$$M, N \in \mathsf{UntypedTerm} ::= x \;\mid\; \lambda x.M \;\mid\; M\ N$$

Substitution for untyped terms is defined as usual, and so is β-reduction:

$$(\beta) \qquad (\lambda x.M)N \to M[x := N]$$

Let $\to^*$ denote the reflexive and transitive closure of $\to$. We define a map $|\cdot|$ from Term to $\mathsf{UntypedTerm}$ that erases type-annotations:

$$\begin{array}{rcl} |\Lambda P.M| = |M| & \quad |\lambda x^\tau.M| = \lambda x.|M| & \quad \\ |M[A]| = |M| & \quad |MN| = |M|\ |N| & \quad |x^\tau| = x \end{array}$$

Lemma 20.

1. *If* $\mathsf{expand}(M, P)$ *is defined, then* $(|\mathsf{expand}(M, P)| = |M|)$.
2. *If* $M[x := N]$ *is defined, then* $|M[x := N]| = |M|[x := |N|]$.
3. *If* $\mathsf{select^b}(M, A)$ *is defined, then* $(|\mathsf{select^b}(M, A)| = |M|)$. □

Theorem 2 (Soundness of Reduction).
If $M, N \in \mathsf{Term}$ *and* $M \to N$*, then* $|M| \to^* |N|$. □

Lemma 21.

1. *Any sequence of* (β_Λ)*,* $(*_\Lambda)$ *and* $(*_A)$ *reductions is terminating.*
2. *If* M *reduces to* N *by a* (β_Λ)*,* $(*_\Lambda)$ *or* $(*_A)$ *reduction, then* $|M| = |N|$.
3. *If* $\mathsf{select^i}(\forall P.\tau, A)$ *is defined, then so is* $\mathsf{match}(P, A)$.
4. *A well-typed term that is free of* (β_Λ)*,* $(*_\Lambda)$ *or* $(*_A)$ *redices is of the form*

$$\Lambda P_1 \ldots \Lambda P_n.M[A_1] \ldots [A_m]$$

where $n, m \geq 0$*, the* P_i*'s and* A_i*'s are not trivial, and* M *is either a variable, a* λ*-abstraction or an application.* □

Theorem 3 (Completeness of Reduction).
If M *is well-typed and* $|M| \to |N|$*, then* $(M \to^* N)$. □

Proof Sketch. The proof uses the previous lemma. To simulate a β-reduction step of the type erasure of M, one first applies (β_Λ), $(*_\Lambda)$ and $(*_A)$-reductions until no more such reductions are possible. Because it is well-typed, the resulting term allows a β_λ-reduction that simulates the β-reduction of the type erasure. □

References

[1] T. Amtoft, F. Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*, pp. 26–40. Springer-Verlag, 2000.

[2] B. Capitani, M. Loreti, B. Venneri. Hyperformulae, parallel deductions and intersection types. *Electronic Notes in Theoretical Computer Science*, 50, 2001. Proceedings of ICALP 2001 workshop: Bohm's Theorem: Applications to Computer Science Theory (BOTH 2001), Crete, Greece, 2001-07-13.

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

[4] M. Dezani-Ciancaglini, S. Ghilezan, B. Venneri. The "relevance" of intersection and union types. *Notre Dame J. Formal Logic*, 38(2):246–269, Spring 1997.

[5] A. Dimock, R. Muller, F. Turbak, J. B. Wells. Strongly typed flow-directed representation transformations. In *Proc. 1997 Int'l Conf. Functional Programming*, pp. 11–24. ACM Press, 1997.

[6] A. Dimock, I. Westmacott, R. Muller, F. Turbak, J. B. Wells. Functioning without closure: Type-safe customized function representations for Standard ML. In *Proc. 2001 Int'l Conf. Functional Programming*, pp. 14–25. ACM Press, 2001.

[7] A. Dimock, I. Westmacott, R. Muller, F. Turbak, J. B. Wells, J. Considine. Program representation size in an intermediate language with intersection and union types. In *Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*, vol. 2071 of *LNCS*, pp. 27–52. Springer-Verlag, 2001.

[8] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse d'Etat, Université de Paris VII, 1972.

[9] A. J. Kfoury. A linearization of the lambda-calculus. *J. Logic Comput.*, 10(3), 2000. Special issue on Type Theory and Term Rewriting. Kamareddine and Klop (editors).

[10] A. J. Kfoury, H. G. Mairson, F. A. Turbak, J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming*, pp. 90–101. ACM Press, 1999.

[11] A. J. Kfoury, J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, pp. 196–207, 1994.

[12] A. J. Kfoury, J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 161–174, 1999.

[13] J. Palsberg, C. Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Funct. Programming*, 11(3):263–317, May 2001.

[14] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb. 1991.

[15] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In J. R. Hindley, J. P. Seldin, eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pp. 561–577. Academic Press, 1980.

[16] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, vol. 19 of *LNCS*, pp. 408–425, Paris, France, 1974. Springer-Verlag.

[17] J. C. Reynolds. Design of the programming language Forsythe. In P. O'Hearn, R. D. Tennent, eds., *Algol-like Languages*. Birkhauser, 1996.

[18] S. Ronchi Della Rocca, L. Roversi. Intersection logic. In *Computer Science Logic, CSL '01*. Springer-Verlag, 2001.

[19] F. Turbak, A. Dimock, R. Muller, J. B. Wells. Compiling with polymorphic and polyvariant flow types. In *Proc. First Int'l Workshop on Types in Compilation*, June 1997.

[20] P. Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$. *Math. Structures Comput. Sci.*, 7(4):329–358, 1997.

[21] B. Venneri. Intersection types as logical formulae. *J. Logic Comput.*, 4(2):109–124, Apr. 1994.

[22] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proc. 9th Ann. IEEE Symp. Logic in Comp. Sci.*, pp. 176–185, 1994. Superseded by [24].

[23] J. B. Wells. Typability is undecidable for F+eta. Tech. Rep. 96-022, Comp. Sci. Dept., Boston Univ., Mar. 1996.

[24] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999. Supersedes [22].

[25] J. B. Wells, A. Dimock, R. Muller, F. Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, pp. 757–771, 1997. Superseded by [26].

[26] J. B. Wells, A. Dimock, R. Muller, F. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programming*, 200X. To appear. Supersedes [25].