Singleton, Union and Intersection Types for Program Extraction

Susumu Hayashi *
Department of Applied Mathematics and Informatics
Ryukoku University
Seta, Ohtsu, Shiga, Japan
hayashi@whale.ryukoku.ac.jp

In memory of my mother, Yoshie Hayashi

Abstract

Two type theories, ATT and ATTT, are introduced. ATT is an impredicative type theory closely related to the polymorphic type theory of implicit typing of MacQueen et al. [MPS86]. ATTT is another version of ATT that extends the Girard-Reynolds second order lambda calculus. ATT has notions of intersection, union and singleton types. ATTT has a notion of refinement types as in the type system for ML by Freeman and Pfenning [FP91], plus intersection and union of refinement types and singleton refinement types. We will show how singleton, union and intersection types serve for development of programs without unnecessary codes via a variant of the Curry-Howard isomorphism. More exactly, they give a way to write types as specifications of programs without unnecessary codes which is inevitable in the usual Curry-Howard isomorphism.

1 Introduction

We introduce two constructive type theories: ATT (A Type Theory) and ATTT (ATT Typed more). ATT is an impredicative type theory closely related to the polymorphic type theory of implicit typing in [MPS86]. It is designed to develop programs without unnecessary codes via the Curry-Howard isomorphism. Constructive type theories are known as a basis for program development via constructive logic. But, when they were applied to the practice of formal program development of programs, they proved inadequate in practice. One defect of type theories is that developed programs *must*

*Partly supported by Grant No. 02249209 of the Japanese Ministry of Education.

often include unnecessary codes for computation (see [HN88, Hay90]). This comes from the identification of proofs and programs. Our main aim here is to remedy this defect by introducing singleton, union and intersection types. Union and intersection types hide entities from the scene and singleton type exposes them. By these, we can exclude purely logical parts of proofs from extracted programs.

We will introduce another type theory ATTT (A triple tee) as well. ATTT is a system based on the idea of "refinement" of types. ATTT is a conservative extension of the Girard-Reynolds second order lambda calculus, which we will refer as $\lambda 2$. Thus it can be used to develop typed programs in $\lambda 2$.

As a framework for program extraction, ATTT apparently has many superior points to ATT. But we dare present ATT as well. ATT and ATTT can be regarded representations of the same idea by two different typing disciplines, implicit typing and explicit typing, respectively. Furthermore, almost all literatures on union and intersection types use type systems similar to ATT rather than ATTT. ATT would be useful to compare them and ours. Last not but least, ATT is simpler to understand and present. It will help readers to understand ATTT.

The aim of this paper is to introduce ATT and ATTT and establish their fundamental mathematical properties. Applications of ATT and ATTT for the practice of program extraction will appear elsewhere.

2 A type theory ATT

Before giving a formal syntax of ATT, we will give an intuitive explanation of the syntax and the intended semantics. ATT has objects, types and kinds. These three syntactic categories are roughly defined by the following grammar:

$$\begin{array}{rcl} K & ::= & \mathit{Type}. \\ A & ::= & x \mid c \mid A \to A \mid A \wedge A \mid A \vee A \mid \\ & & \bigwedge x \in K.A \mid \bigvee x \in K.A \mid \\ & & \bigwedge x \in A.A \mid \bigvee x \in A.A \mid \\ & & \{M\}_A. \\ M & ::= & x \mid c \mid \lambda x.M \mid MM. \end{array}$$

We will use another constant, *Kind*, which does not appear in this grammar. In the grammar above, M, A and K denote objects, types and kinds, respectively. As it is a second order system, ATT has only one kind *Type*. So the kind level is inessential for ATT, but we include it for coherence with GTS-style presentations of type theories [Bar90], which will be used in this paper. (Although we follow GTS-presentation as far as possible, our notations and terminologies may sometimes diverge from the ones of GTS.)

The intended semantics of ATT assumes a fixed domain D of untyped partial λ-calculus. Objects are elements of the domain. Normally, types in programming languages are interpreted as sets with common structure. Types of ATT, however, are interpreted as *arbitrary* subsets of D. In particular, the empty set is a type, as ATT is designed to achieve a kind of Curry-Howard isomorphism. The kind *Type* is the set of all subsets of D. The constant *Kind* is the singleton set of *Type*.

Now we give a rigorous formulation of ATT. Although ATT is a second order system, we have to formulate it as higher order systems like CoC [CH88], since the formation of singleton types depends on the typing of objects. A singleton type $\{A\}_B$ is well formed only when B is a type of A.

Firstly, we define *preterms* (called pseudo terms in [Bar90]). Preterms are defined by the following grammar:

$$T \quad ::= \quad x \mid c \mid \lambda x.T \mid TT \mid T \to T \mid T \wedge T \mid T \vee T \mid \bigwedge x \in T.T \mid \bigvee x \in T.T \mid \{T\}_T.$$

Here, c and x stand for constants and variables, respectively. For program development, we have to extend this grammar by adding some basic types such as the type of natural numbers. To this end, we keep the definition of constants open. But we assume that at least *Type* and *Kind* are constants. These two constants are called *sorts*. In the following, s stands for a sort.

The symbols $\lambda, \bigwedge, \bigvee$ are "quantifiers." The notion of bound occurrences and free occurrences are defined as usual. The set of free variables of a preterm T will be written $FV(T)$. The operation of α-conversion is defined as usual and we will identify two α-convertible preterms. Substitution of preterms is defined as usual. When a preterm B is substituted for a variable x in a preterm A, we denote the resulting preterm by $A[x := B]$.

The reduction and conversion of preterms are defined as usual by β-reduction. When B is obtained from A by successive β-reductions, we will write $A \twoheadrightarrow_\beta B$. When A and B are convertible, we will write $A =_\beta B$.

A *judgment* of ATT is of the form $A \in B$, where A and B are preterms. A *context* is a sequence of judgements of the form $x \in A$, where x is a variable. We will denote a context by Γ. When a variable x is not declared in a context Γ, then we say x is Γ-fresh. A *sequent* (hypothetical judgement) of ATT is the form $\Gamma \vdash J$, where J is a judgment. Let Γ be $x_1 \in A_1, \ldots, x_n \in A_n$. Then $\Gamma[x := B]$ is $x_1 \in A_1[x := B], \ldots, x_n \in A_n[x := B]$. Similarly, if J is the judgment $A \in B$, then $J[x := C]$ is $A[x := C] \in B[x := C]$.

A *legal sequent* of ATT is a sequent derivable by the following axiom and inference rules:

Axiom

$$\langle\,\rangle \vdash \mathit{Type} \in \mathit{Kind}$$

Start rule

$$\frac{\Gamma \vdash A \in s}{\Gamma, x \in A \vdash x \in A}$$

where x is Γ-fresh.

Weakening rule

$$\frac{\Gamma \vdash A \in B \quad \Gamma \vdash C \in s}{\Gamma, x \in C \vdash A \in B}$$

where x is Γ-fresh.

Conversion rule

$$\frac{\Gamma \vdash A \in B \quad \Gamma \vdash B' \in s \quad B =_\beta B'}{\Gamma \vdash A \in B'}$$

Subject reduction

$$\frac{\Gamma \vdash A \in B \quad A \twoheadrightarrow_\beta A'}{\Gamma \vdash A' \in B}$$

Rules for function space

$\rightarrow$-formation

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type}}{\Gamma \vdash (A \rightarrow B) \in \mathit{Type}}$$

$\rightarrow$-elimination

$$\frac{\Gamma \vdash F \in (A \rightarrow B) \quad \Gamma \vdash C \in A}{\Gamma \vdash (FC) \in B}$$

$\rightarrow$-introduction

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type} \quad \Gamma, x \in A \vdash C \in B}{\Gamma \vdash (\lambda x.C) \in (A \rightarrow B)}$$

Rules for union

$\vee$-formation

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type}}{\Gamma \vdash A \vee B \in \mathit{Type}}$$

$\vee$-introduction

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type} \quad \Gamma \vdash C \in A}{\Gamma \vdash C \in A \vee B}$$

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type} \quad \Gamma \vdash C \in B}{\Gamma \vdash C \in A \vee B}$$

$\vee$-elimination

$$\frac{\Gamma \vdash C \in A \vee B \quad \Gamma, x \in A \vdash D \in E \quad \Gamma, x \in B \vdash D \in E}{\Gamma \vdash D[x := C] \in E[x := C]}$$

where E is not a sort.

$\bigvee$-formation

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{Type}}{\Gamma \vdash (\bigvee x \in A.B) \in \mathit{Type}}$$

$\bigvee$-introduction

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{Type} \quad \Gamma \vdash D \in A \quad \Gamma \vdash C \in B[x := D]}{\Gamma \vdash C \in (\bigvee x \in A.B)}$$

$\bigvee$-elimination

$$\frac{\Gamma \vdash C \in (\bigvee x \in A.B) \quad \Gamma, x \in A, y \in B \vdash D \in E}{\Gamma \vdash D[y := C] \in E[y := C]}$$

where x is not in $FV(D) \cup FV(E)$ and E is not a sort.

Rules for intersection

$\wedge$-formation

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type}}{\Gamma \vdash A \wedge B \in \mathit{Type}}$$

$\wedge$-elimination

$$\frac{\Gamma \vdash C \in A \wedge B}{\Gamma \vdash C \in A} \qquad \frac{\Gamma \vdash C \in A \wedge B}{\Gamma \vdash C \in B}$$

$\wedge$-introduction

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type} \quad \Gamma \vdash C \in A \quad \Gamma \vdash C \in B}{\Gamma \vdash C \in A \wedge B}$$

$\bigwedge$-formation

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{Type}}{\Gamma \vdash (\bigwedge x \in A.B) \in \mathit{Type}}$$

$\bigwedge$-elimination

$$\frac{\Gamma \vdash C \in (\bigwedge x \in A.B) \quad \Gamma \vdash D \in A}{\Gamma \vdash C \in B[x := D]}$$

$\bigwedge$-introduction

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{Type} \quad \Gamma, x \in A \vdash C \in B}{\Gamma \vdash C \in (\bigwedge x \in A.B)}$$

where x is not in $FV(C)$.

Rules for singleton

singleton-formation

$$\frac{\Gamma \vdash B \in \mathit{Type} \quad \Gamma \vdash A \in B}{\Gamma \vdash \{A\}_B \in \mathit{Type}}$$

singleton-introduction

$$\frac{\Gamma \vdash B \in \mathit{Type} \quad \Gamma \vdash A \in B}{\Gamma \vdash A \in \{A\}_B}$$

singleton-elimination

$$\frac{\Gamma \vdash P_1 \in \{A\}_C \wedge \{B\}_C \quad \Gamma \vdash P_2 \in D[x := A] \quad \Gamma, x \in C \vdash D \in \mathit{Type}}{\Gamma \vdash P_2 \in D[x := B]}$$

$$\frac{\Gamma \vdash A \in \{B\}_C}{\Gamma \vdash A \in C}$$

2.1 Some comments on the formal system of ATT

In this section, we will give some discussions on the formal system presented above.

2.1.1 The rule of subject reduction

ATT has the rule of subject reduction as a basic rule. The rule is an admissible rule for many type theories like CoC. Readers may think that the rule is unnecessary. But we have not succeeded to prove the rule for ATT without the rule. Furthermore, Pierce and Freeman [Pie91a] have observed an example which would be an counterexample for the rule in a type system closely related to ATT. Let's briefly sketch their example. Let a, b and c be types. Then $\lambda x.y.z.x((\lambda t.t)yz)((\lambda t.t)yz)$ has the type

$$((a \to a \to d) \wedge (b \to b \to d)) \to (c \to a \vee b) \to c \to d.$$

Trying to type these, we first deduce $(\lambda t.t)yz \in a \vee b$ under the obvious context for x, y and z. On the other hand, $xuu \in d$ is derivable under the same context plus $u \in a$ or $u \in b$. By applying $\vee$-elmination to these, $(\lambda t.t)yz$ substitutes u and we have the typing of the term. But, reduced forms $\lambda x.y.z.x(yz)((\lambda t.t)yz)$ or $\lambda x.y.z.x((\lambda t.t)yz)(yz)$ would not have the type, as there would not be such a term as xuu for these terms. Note that the fully reduced form $\lambda x.y.z.x(yz)(yz)$ has the type, again.

It has not been known if this is a real counterexample for Pierce's system, but it is very likely so. We conjecture that this is a counterexample also for ATT. In this paper, we do *not* need the rule of subject reduction at all. All results in this paper remain true for the system without the rule of subject reduction. But it would be useful for optimization and transformation of programs. So we added it as a basic rule.

2.1.2 The rules of singleton-elimination

We do not know if the first rule of singleton-elimination is the *only* right formulation. But this is the best formulation we know. Later, we will define a propositional equality by means of singleton and intersection types. Then this rule will turn into the equality rule used by Turner in his type theory NTT [Tur89]. Furthermore, we have been able to derive almost all properties which we need. Another possible rule of the singleton-elimination is

Strong rule of (first) singleton-elimination

$$\frac{\Gamma \vdash P_1 \in \{A\}_C \wedge \{B\}_C \quad \Gamma \vdash P_2 \in D[x := A] \quad \Gamma, x \in C \vdash D \in Type}{\Gamma \vdash P_2[x := B] \in D[x := B]}$$

This substitutes B for x in P_2 as well as x in D. This rule is sound w.r.t. the semantics of ATT given later. But, this rule is not sound w.r.t. the transformation from ATT to ibMPS. (see 2.5). Thus this would not be a correct rule from type theoretic point of view.

2.1.3 Distributive laws

The rules for intersection and union are not strong enough to derive the distributive laws for them. The following judgments are not derivable by the rules for intersection

and union:

$$a \in \mathit{Type}, b \in \mathit{Type}, c \in \mathit{Type}, x \in a \wedge (b \vee c) \vdash x \in (a \wedge b) \vee (a \wedge c)$$
$$a \in \mathit{Type}, b \in \mathit{Type}, c \in \mathit{Type}, x \in (a \vee b) \wedge (a \vee c) \vdash x \in a \vee (b \wedge c)$$

Note that the converses are derivable. Furthermore, the latter judgment can be derived from the former judgment. (This was pointed by Barbanera and Dezani-Ciancaglini.) Some formal systems for intersection and union types include the laws above as axioms, e.g. [BDC91],[Pie90],[Pie91b]. (These papers mentioned consider also distributive laws for union and intersection with arrow. Their unprovability seems a problem with η-rule rather than rules for union and intersection. So we do not consider them in this paper.) We may include these as basic rules in ATT, as the metamathematical properties of ATT in this paper remain true. But, these laws do not seem necessary for our aim. Hence we do not include them in ATT. Later, we will show these distributive laws are derivable in ATTT (see 3.5).

2.1.4 Type formation

The inference rules above are designed so that types are formed only through formation rules. The side condition "E is not a sort" for $\vee$-elimination and $\bigvee$-elimination prevents these rules forming new types unexpectedly. Dropping this side condition, E of $\vee$-elimination could be Type and then $D[x := C] \in \mathit{Type}$ would be derivable. Namely, a type $D[x := C]$ would be formed by $\vee$-elimination rule. The side condition was meant to avoid this. Note that the side condition is decidable.

However, the side condition may be superfluous. Types formed by these eliminations rules without the side condition might be formed by the formation rules. But, we do not know if this is alway true. At least, without the side condition, we do not know how to prove the results on classification of terms in 2.3. This is the reason why we included such a side condition. Even if the side conditon is superfluous, it would be better to keep the side condition to make track of formation of types clear.

2.2 Basic notions and lemmas

We will define some basic notions and prove some basic lemmas which are necessary to understand the structure of ATT.

Definition 1 If a sequent is derivable by the inference rules above, then it is called a *legal sequent*. If $\Gamma \vdash A \in B$ is a legal sequent, then Γ is called *legal context*, $A \in B$ is *legal judgment*, and A and B are *(legal) terms*. To emphasize the context Γ, we will often say "a preterm A is a term *under the context* Γ." □

In the following, we list basic lemmas. They all can be easily proved by standard methods. So we will omit the proofs.

Lemma 1 *(*Church-Rosser property of preterms*) If two preterms are β-convertible, then they can be reduced to a common preterm.*

Lemma 2 *If $x_1 \in A_1, \ldots, x_n \in A_n \vdash J$ is a legal sequent, then (i) if $i \neq j$, then $x_i \neq x_j$, (ii) the free variables of the term A_i are a subset of $\{x_1, \ldots, x_{i-1}\}$, and (iii) the free variables of the judgment J are a subset of $\{x_1, \ldots, x_n\}$.*

Lemma 3 *Assume that $x_1 \in A_1, \ldots, x_n \in A_n \vdash B \in C$ is a legal sequent. Then B is not the sort Kind. Furthermore, for each i, A_i is not the sort Kind and one of the followings holds:*

1. *A_i is the sort Type and $x_1 \in A_1, \ldots, x_{i-1} \in A_{i-1} \vdash A_i \in Kind$ is derivable.*
2. *$x_1 \in A_1, \ldots, x_{i-1} \in A_{i-1} \vdash A_i \in Type$ is derivable.*

Lemma 4 (substitution) *If $\Gamma_1, x \in A, \Gamma_2 \vdash J$ and $\Gamma_1 \vdash C \in A$ are legal sequents, then so is $\Gamma_1, \Gamma_2[x := C] \vdash J[x := C]$.*

Lemma 5 (generalized weakening) *When a context Γ is obtained from another context Γ' by dropping some judgments in Γ', then we say that Γ' contains Γ. If $\Gamma \vdash J$ is a legal sequent and Γ' is a legal context which contains Γ, then $\Gamma' \vdash J$ is a legal sequent.*

The first part of the following lemma is a special case of lemma 4 above and the second part is easily proved by induction on derivations. The second part was called "context replacement" by Luo [Luo90].

Lemma 6 (replacement) *(i) If $\Gamma \vdash A \in B$ and $\Gamma, x \in B \vdash x \in B'$ are derivable, then $\Gamma \vdash A \in B'$ is derivable. (ii) If $\Gamma, x \in A, \Gamma' \vdash J$ and $\Gamma, x \in A' \vdash x \in A$ are derivable, then $\Gamma, x \in A', \Gamma' \vdash J$ is derivable.*

2.3 Classification of terms

Every term of ATT (except the sort *Kind*) is classified into one of object, type and kind. This classification can be done by purely syntactical observation. Firstly, we define object, type and kind by their shapes. Then we prove that this classification is sound, e.g., if a term A is a type, then $A \in Type$ is derivable.

Assume a preterm A is legal under context Γ. By lemma 2, all free variables of A are declared in Γ. Let x be a free variable of A. If it is declared by $x \in Type$, then we call it a *type variable* of Γ. Otherwise, we call it an *object variable* of Γ. So all free variables of *any* legal term under Γ are classified into type variables and object variables. We extend this classification to legal terms.

Definition 2 (classification under a variable classification) Assume that $FV(A)$ is a union of two disjoint sets Obj and Typ. The variables of Obj are called object variables, and the variables of Typ are called type variables. Then objects, types and kinds under the classification of free variables $\langle Obj, Typ\rangle$ are defined as follows:

objects
1. An object variable is an object.
2. If A and B are objects, then so is (AB).
3. If A is an object under the classification $\langle Obj \cup \{x\}, Typ\rangle$, then $\lambda x.A$ is an object under the classification $\langle Obj, Typ\rangle$.

types
1. A type variable is a type.
2. If A and B are types, then so are $A \to B$, $A \vee B$ and $A \wedge B$.
3. If A is an object and B is a type, then $\{A\}_B$ is a type.
4. If A is a type and B is a type under the classification $\langle Obj \cup \{x\}, Typ\rangle$, then $\bigwedge x \in A.B$ and $\bigvee x \in A.B$ are types under the classification $\langle Obj, Typ\rangle$.
5. If A is a type under the classification $\langle Obj, Typ \cup \{x\}\rangle$, then $\bigwedge x \in Type.A$ and $\bigvee x \in Type.A$ are types under the classification $\langle Obj, Typ\rangle$.

kinds The constant $Type$ is a kind. □

Now we classify legal terms under a context using the classification above.

Definition 3 (classification under a context) Assume a preterm A is legal under context Γ. Let Obj be the set of object variables of Γ, and let Typ be the set of type variables of Γ. If A is an object under the classification $\langle Obj, Typ\rangle$, then A is an *object under* Γ. *Types* and *kinds under* Γ are defined in the same way. □

Lemma 7
1. *If $x_1 \in A_1, \ldots, x_n \in A_n$ is a legal context, then either A_i is a type under the context $x_1 \in A_1, \ldots, x_{i-1} \in A_{i-1}$ or A_i is a kind, i.e., $A_i \equiv Type$.*
2. *If $\Gamma \vdash A \in B$ is a legal sequent, then exactly one of the following holds:*
   (a) *A is an object and B is a type under Γ.*
   (b) *A is a type and B is a kind, i.e. $B \equiv Type$.*
   (c) *A is a kind, i.e. $A \equiv Type$, and B is the sort Kind.*

This lemma can be proved induction on derivations and shows that classification corresponds to our intutive notions of terms, types and kinds. Note that the uniqueness of the classification is trivial, as the sets of objects, types, kinds, and the sort $Kind$ are syntactically disjoint.

The sort $Type$ is an element of $Kind$ by the axiom. It is coherent with the classification above. We shall check that a type as defined above is an element of the sort $Type$, as $Type$ is intended to be the collection of types. Namely, we shall prove the following lemma.

Lemma 8 *If $\Gamma \vdash A \in B$ is derivable and B is a type, then $\Gamma \vdash B \in Type$ is derivable.*

This lemma is proved by induction on derivations using the following type inversion lemma, which is proved by induction on derivations using lemmas 1, 6 and 7.

Lemma 9 *1. Let op be one of $\to$, $\vee$, $\wedge$. If $\Gamma \vdash (A\ \mathsf{op}\ B) \in Type$ is derivable, then so are $\Gamma \vdash A \in Type$ and $\Gamma \vdash B \in Type$.*

2. Let OP be one of $\bigvee$, $\bigwedge$. If $\Gamma \vdash (\mathsf{OP}\, x \in A.B) \in Type$ is derivable, then so are $\Gamma \vdash A \in Type$ and $\Gamma, x \in A \vdash B \in Type$.

3. If $\Gamma \vdash \{A\}_B \in Type$ is derivable, then so are $\Gamma \vdash B \in Type$ and $\Gamma \vdash A \in B$.

Note that if a term is proved to be an element of *Type*, then it is one of the forms of the lemma above or a type variable. Hence, by lemma 9, any element of *Type* is formed by successive applications of the formation rules.

2.4 Semantics of ATT

A semantics of ATT can be given on any reasonable concrete model of partial untyped lambda calculus. Unfortunately, we cannot find any good reference for this kind of models of partial untyped lambda calculus. (A good reference for the typed case is [Mog88].) One possible approach is to start with models of ordinary (total) untyped lambda calculus. But this construction will invalidate an important proposition that if a term is typable then it denotes a meaningful value. So we will give a semantics using a particular domain theoretic model of the partial untyped lambda calculus by Plotkin [Plo85]. Plotkin reconstructed D_∞ in the category of cpo's and continuous partial functions. In his sense, a cpo may or may not have bottom element. His D_∞ is a cpo with two total continuous functions i and j

$$\mathsf{D}_\infty \underset{j}{\overset{i}{\rightleftarrows}} [\mathsf{D}_\infty \rightharpoonup \mathsf{D}_\infty] \qquad \text{satisfying } i \circ j = id$$

Note that $[\mathsf{D}_\infty \rightharpoonup \mathsf{D}_\infty]$ is the set of partial continuous functions from D_∞ to D_∞. We can interpret lambda terms as in the usual D_∞. Note that $[\![(AB)]\!]_\rho$ can be undefined, as $i([\![A]\!]_\rho)$ is a partial function. So an interpretation of a term may be undefined. Using Plotkin's semantics, a semantics of ATT is given as follows.

Definition 4 (interpretation) Let TYPE be the set of subsets of D_∞ and KIND be the singleton $\{\mathsf{TYPE}\}$. Let $\langle Obj, Typ\rangle$ be a partition of the variables into object variables and type variables. A function ρ from the set of variables to $\mathsf{D}_\infty \cup \mathsf{TYPE} \cup \mathsf{KIND}$ is called an *assignment*, if $\rho(x)$ is in D_∞ for an object variable x and $\rho(x)$ is in TYPE for a type variable x. For simplicity, we assume that the only constants of ATT are *Type* and *Kind*. Let M be an object, a type, or a kind under $\langle Obj, Typ\rangle$. We define interpretation $[\![M]\!]_\rho$ of M.

objects An object is a lambda term. The interpretation of objects is the same as the usual interpretation of lambda terms.

types
1. If x is a type variable, then set $[\![x]\!]_\rho = \rho(x)$.
2. $[\![A \to B]\!]_\rho = \{f \in \mathsf{D}_\infty | \forall x \in [\![A]\!]_\rho.((i(f))(x) \in [\![B]\!]_\rho)\}$.
3. $[\![A \vee B]\!]_\rho = [\![A]\!]_\rho \cup [\![B]\!]_\rho$.
4. $[\![A \wedge B]\!]_\rho = [\![A]\!]_\rho \cap [\![B]\!]_\rho$.
5. $[\![\{A\}_B]\!]_\rho = \{x \in \mathsf{D}_\infty | x = [\![A]\!]_\rho \text{ and } [\![A]\!]_\rho \in [\![B]\!]_\rho\}$.
6. $[\![\bigwedge x \in A.B]\!]_\rho = \bigcap_{v \in [\![A]\!]_\rho}([\![B]\!]_{\rho[x := v]})$.
7. $[\![\bigvee x \in A.B]\!]_\rho = \bigcup_{v \in [\![A]\!]_\rho}([\![B]\!]_{\rho[x := v]})$.

sorts $[\![\mathit{Type}]\!]_\rho = \mathsf{TYPE}$ and $[\![\mathit{Kind}]\!]_\rho = \mathsf{KIND}$. □

The soundness theorem is proved by induction on derivations.

Theorem 1 (soundness) *Assume that $\Gamma \vdash A \in B$ is derivable. Let ρ be an environment with respect to the partition of variables induced by the context Γ. Then $[\![A]\!]_\rho$ and $[\![B]\!]_\rho$ are defined and $[\![A]\!]_\rho$ is an element of $[\![B]\!]_\rho$. Hence the interpretation of any legal term is defined. Furthermore, this theorem holds for the system with the strong rule of singleton-elimination.*

Although we gave a semantics in the category of sets, it would be possible to give a semantics in other categories with some conditions. Rosolini [Ros86] and others have introduced notions of partial categories which is intended for categories of partial maps. Following D. Scott's semantics of lambda calculus in cartesian closed categories, we would be able to give a semantics of partial lambda calculus in partial cartesian closed categories. Then the partial CCC would be embeddable into a topos. In the topos, we would be able to give semantics of ATT, as we can define singleton, union and intersection by the constructive logic of the topos. This would be straightforward but is yet to be worked out.

Pierce [Pie90] has given a PER-semantics for a type system closely related to ATT. Pierce did not consider singleton type, but it would be interpreted by

$$x\ [\![\{A\}_B]\!]_\rho\ y = x\ [\![B]\!]_\rho\ y \text{ and } x\ [\![B]\!]_\rho\ [\![A]\!]_\rho.$$

This semantics works for ATT as well. Then the rule of extensionality of functions can be added to the system as in Martin-Löf's type theory of [Mar82]. But Pierce has pointed out an anomaly of this semantics: As the union of PER's is not in general a PER, we have to collapse it by taking the transitive closure to interpret union types. This causes some expected equalities between types to fail. See [Pie90] for details.

2.5 Eliminating the non-computational part

In this section, we will show that the non-computational part of the proof is eliminable by a simple translation. This translation translates derivations in ATT into another type theory, ibMPS, which is a variant of the type theory of MacQueen et al. [MPS86]. As a corollary of the soundness of the translation, we obtain a strong normalization theorem for ATT.

The system ibMPS (inhabitable basic MacQueen, Plotkin and Sethi system) is roughly defined as follows:

$$\begin{array}{rcl} \text{ibMPS} & = & \text{ATT} - \text{singleton types} \\ & & \quad - \text{first order big union and intersection types} \\ & & \quad - \text{conversion rule} \end{array}$$

The first order big union and intersection types mean types of the form $\bigwedge x \in A.B$ or $\bigvee x \in A.B$, where A is a type.

As we dropped the singleton types, types of ibMPS can be defined independently from the definition of legal terms as in the usual second order type theories. It is more natural to reformulate ibMPS in this way, but we do not do so here to avoid repetitions and to save space. We only list changes of ibMPS from ATT. Note that some rules for intersections and unions are changed. These changes come from the other changes.

Definition 5 (ibMPS) The system ibMPS is defined by changing ATT as follows:

1. Drop $\{T\}_T$ from the syntax of the preterms, and drop all the rules for singleton types.

2. Change all the rules for $\bigwedge$ and $\bigvee$, and $\vee$-elimination rule as follows:

$$\frac{\Gamma, x \in \mathit{Type} \vdash B \in \mathit{Type}}{\Gamma \vdash (\bigwedge x \in \mathit{Type}.B) \in \mathit{Type}} \qquad \frac{\Gamma, x \in \mathit{Type} \vdash B \in \mathit{Type}}{\Gamma \vdash (\bigvee x \in \mathit{Type}.B) \in \mathit{Type}}$$

$$\frac{\Gamma, x \in \mathit{Type} \vdash B \in \mathit{Type} \quad \Gamma, x \in \mathit{Type} \vdash C \in B}{\Gamma \vdash C \in (\bigwedge x \in \mathit{Type}.B)}$$

$$\frac{\Gamma \vdash C \in (\bigwedge x \in \mathit{Type}.B) \quad \Gamma \vdash D \in \mathit{Type}}{\Gamma \vdash C \in B[x := D]}$$

$$\frac{\Gamma, x \in \mathit{Type} \vdash B \in \mathit{Type} \quad \Gamma \vdash D \in \mathit{Type} \quad \Gamma \vdash C \in B[x := D]}{\Gamma \vdash C \in (\bigvee x \in \mathit{Type}.B)}$$

$$\frac{\Gamma \vdash C \in (\bigvee x \in \mathit{Type}.B) \quad \Gamma, x \in \mathit{Type}, y \in B \vdash D \in E}{\Gamma \vdash D[y := C] \in E}$$

$$\frac{\Gamma \vdash C \in A \vee B \quad \Gamma, x \in A \vdash D \in E \quad \Gamma, x \in B \vdash D \in E}{\Gamma \vdash D[x := C] \in E}$$

In the last two rules, E is not Type or Kind.

3. Drop the conversion rule. □

It is easy to prove the following lemma by induction of derivations.

Lemma 10 (inhabitation rule) *The following is an admissible rule of* ibMPS.

$$\frac{\Gamma, x \in A, \Gamma' \vdash B \in C}{\Gamma, \Gamma' \vdash B \in C},$$

where x is not a free variable of B, and A is not Type. (Note that the "first order" variable x never appear in C. So we do not need variable condition on C.)

The inhabitation rules maintains that all types are inhabited (cf. [Mit90]). Note that ATT is regarded as having an empty type; otherwise singleton-elimination is not sound. Existence of an empty type is a characteristic of type systems based on the Curry-Howard isomorphism. On the other hand, in many systems purely for types in programming languages, types can be assumed to be inhabited. In this sense, we are translating a type theory for the Curry-Howard isomorphism (verification system) into a type theory for a programming language (computation system).

Definition 6 (translation from ATT to ibMPS) We define a translation $M \mapsto M^*$ from preterms of ATT to preterms of ibMPS as follows:

1. $c^* = c$.
2. $x^* = x$.
3. $(\lambda x.A)^* = (\lambda x.A^*)$.
4. $(AB)^* = (A^* B^*)$.
5. $(A \ \mathsf{op} \ B)^* = (A^* \ \mathsf{op} \ B^*)$, where op is one of $\to$, $\wedge$, $\vee$.
6. $(\mathsf{OP}\, x \in \mathit{Type}.B)^* = \mathsf{OP}\, x \in \mathit{Type}.B^*$, where OP is one of $\bigwedge$, $\bigvee$.
7. $(\mathsf{OP}\, x \in A.B)^* = B^*$, where A is not *Type* and OP is one of $\bigwedge$, $\bigvee$.
8. $(\{A\}_B)^* = B^*$. □

The translation is naturally expanded to translations of judgments, contexts and sequents. Then the translation preserves derivability.

Theorem 2 *If $\Gamma \vdash J$ is derivable in* ATT, *then so is $\Gamma^* \vdash J^*$ in* ibMPS.

Proof. By induction of derivations. Let us check only the cases of $\bigwedge$-introduction and singleton-elimination, as the other cases are proved similarly.

Let us consider the case of $\bigwedge$-introduction. Assume that the last rule of the derivation is as follows:

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{Type} \quad \Gamma, x \in A \vdash C \in B}{\Gamma \vdash C \in (\bigwedge x \in A.B)}$$

If A is not *Type*, then the translation of the conclusion is $\Gamma^* \vdash C^* \in B^*$. By the induction hypothesis, the translation of the last premise, $\Gamma^*, x \in A^* \vdash C^* \in B^*$, is derivable. By the inhabitation rule, $\Gamma^* \vdash C^* \in B^*$ is derivable. If A is *Type*, then the translation of the rule is the $\bigwedge$-introduction rule of ibMPS.

Now consider the case of singleton-elimination. Assume that the last rule of the derivation is as follows:

$$\frac{\Gamma \vdash P_1 \in \{A\}_C \wedge \{B\}_C \quad \Gamma \vdash P_2 \in D[x := A] \quad \Gamma, x \in C \vdash D \in \mathit{Type}}{\Gamma \vdash P_2 \in D[x := B]}$$

Since $\{A\}_C$ is a legal term, C is a type and so x is an object variable. Since the translation eliminates all subterms which are objects, we see $(D[x := A])^* \equiv (D[x := B])^* \equiv D^*$. Hence the translation of the conclusion of the rule is the same as the translation of the second premise and so it is derivable from the induction hypothesis. □

Note that the strong rule of first singleton-elimination is unsound w.r.t. the translation. Let Γ be the context $X \in \mathit{Type}, Y \in \mathit{Type}, x \in X, y \in Y$. Then $\Gamma, w \in \{x\}_{(X \vee Y)} \wedge \{y\}_{(X \vee Y)} \vdash x \in X$ is provable in ATT. The strong rule infers the sequent $\Gamma, w \in \{x\}_{(X \vee Y)} \wedge \{y\}_{(X \vee Y)} \vdash y \in X$. Assume the translation works for the strong rule. Then $\Gamma, w \in (X \vee Y) \wedge (X \vee Y) \vdash y \in X$. is provable in ibMPS. By substituting x for w, we get $\Gamma \vdash y \in X$. But, this is not provable in ibMPS.

The terms of ibMPS are strongly normalizable. A proof for a related system can be found in [Lei86] and it is applicable to ibMPS. So we do not give a proof here.

Theorem 3 *Every term of* ibMPS *is strongly normalizable.*

We can prove strong normalization property of ATT from this theorem.

Corollary 1 *Every term of* ATT *is strongly normalizable.*

Proof. Let M be a term of ATT. Then M^* is strongly normalizing by the previous theorem and theorem 2. If M is an object, then M^* is identical to M. So M is strongly normalizing. If M is a type, then its subterms, which are objects, are legal terms of ATT by lemma 9. Since reduction can happen only in such subterms, M is strongly normalizing. □

The following is also obvious:

Corollary 2 ATT *is a conservative extension of* ibMPS. *Namely, any provable sequent of* ibMPS *is provable in* ATT, *and any provable sequent of* ATT *having only preterms of* ibMPS *is provable in* ibMPS.

2.6 Some definable types and judgments

In this section, we will define some derived notions including dependent products and sums at the level of types.

(1) **Dependent product.** ATT does not have a dependent product Π unlike Martin-Löf type theories. Such a dependent product of types is definable by means of intersection types and function spaces. Assume that A and B are types. Then we define

$$\Pi x \in A.B \stackrel{\text{def}}{=} \bigwedge x \in A.(\{x\}_A \to B)$$

The formation and elimination rules for Π are easily derivable.

We will derive the introduction rule. Assume that the following three sequents are derived:

$$(1)\quad \Gamma \vdash A \in \mathit{Type}, \quad (2)\quad \Gamma, x \in A \vdash B \in \mathit{Type}, \quad (3)\quad \Gamma, x \in A \vdash C \in B.$$

Let y and z be fresh variables. By renaming the variable x of (3) to z and applying generalized weakening, we can derive

$$\Gamma, x \in A, y \in \{x\}_A, z \in A \vdash C[x := z] \in B[x := z].$$

Since $y \in A$ is derivable under $\Gamma, x \in A, y \in \{x\}_A$, we can derive

$$\Gamma, x \in A, y \in \{x\}_A \vdash C[x := y] \in B[x := y]$$

by substitution. On the other hand, we can derive $y \in \{y\}_A \wedge \{x\}_A$ under the same context. Then we can derive the Π-introduction rule roughly as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma, x \in A, y \in \{x\}_A \vdash y \in \{y\}_A \wedge \{x\}_A \quad \Gamma, x \in A, y \in \{x\}_A \vdash C[x := y] \in B[x := y]}{\Gamma, x \in A, y \in \{x\}_A \vdash C[x := y] \in B[x := y][y := x]}}{\Gamma, x \in A \vdash \lambda y.(C[x := y]) \in (\{x\}_A \to B)}}{\Gamma \vdash \lambda x.C \in \bigwedge x \in A.(\{x\}_A \to B)}$$

In the second inference, we omitted the first two premises of $\to$-introduction, which are easily derived. Note that we identify the two α-convertible terms $\lambda y.C[x := y]$ and $\lambda x.C$ in the last step of the inference.

(2) **Cartesian product.** Using $\bigwedge$ at the level of types, we can define the cartesian product of two types by the standard method for polymorphic lambda calculi.

$$A \times B \stackrel{\text{def}}{=} \bigwedge X \in \mathit{Type}.((A \to B \to X) \to X)$$

Note that the arrows $\to$ above are basic constructors of ATT. Pairing $\langle A, B \rangle$ and projections, $\mathit{fst}(A), \mathit{snd}(A)$, are defined in the standard way.

(3) **Dependent sum.** The dependent sum of types is definable by means of union types and cartesian products. Assume that A and B are types. Then we define

$$\Sigma x \in A.B \stackrel{\text{def}}{=} \bigvee x \in A.(\{x\}_A \times B)$$

Note that this dependent sum is a *strong* sum [Coq86], as the first component of a member a of $\Sigma x \in A.B$ is obtained by $fst(a)$ as well as the second component. Note that a preterm $\Sigma x \in Type.B$ can be defined as above, but cannot be a type, as such a strong sum implies a contradiction [Coq86]. To derive $(\Sigma x \in Type.B) \in Type$, $\{x\}_{Type} \in Type$ has to be derived. But, to derive it by the rule of singleton-formation, $Type \in Type$ is required but it is not derivable. Note that the standard definition of dependent sum of types is also possible in ATT:

$$\Sigma x \in A.B \stackrel{\text{def}}{=} \bigwedge y \in Type.((\Pi x \in A.(B \to y)) \to y)$$

Since ATT has a strong sum, this sum turns out to be a strong sum. (This was pointed out by Luo.)

(3) **Subset Type.** A subset type is definable as follows:

$$\{x \in A|B\} \stackrel{\text{def}}{=} \bigvee x \in A. \bigvee y \in B.\{x\}_A$$

Note that subset type is a kind of dependent sum $\Sigma x \in A.B$ which exposes information from A and hides information from B. The strong sum defined above exposes information from both A and B. The union type $\bigvee x \in A.B$ hides information from A and exposes information from B.

(4) **Propositional equality.** Martin-Löf type has a basic type $Eq(A, M, N)$ called propositional equality. In higher order type theories like CoC, Leibnitz equality can be defined as a type (proposition). But it cannot be in second order theories. Having singleton types, an equality type can be defined in ATT as follows:

$$Eq(A, M, N) \stackrel{\text{def}}{=} \{M\}_A \wedge \{N\}_A$$

It is inhabited iff M and N are the same object of the type A. This equality is an equivalence relation, i.e., the following rules hold:

$$\frac{M \in A \quad A \in Type}{M \in Eq(A, M, M)} \quad \frac{C \in Eq(A, N, M)}{C \in Eq(A, M, N)} \quad \frac{C_1 \in Eq(A, M, N) \quad C_2 \in Eq(A, N, P)}{C_2 \in Eq(A, M, P)}$$

The rule of singleton-elimination serves as the equality rule. Note that our propositional equality is closer to Leibnitz equality than to Martin-Löf's, as it is a defined notion and is "not" an extensional equality. (Exactly speaking, our equality is neutral with respect to extensionality. We may add extensionality rules consistently, although extensionality is problematic as noted in 2.4.)

(5) **Equality judgment.** Martin-Löf type theory has an equality judgment $M = N \in A$. A similar judgment can be defined by

$$M = N \in A \stackrel{\text{def}}{=} M \in \{N\}_A$$

Note that the following rules hold:

$$\frac{C \in Eq(A, M, N)}{M = N \in A} \qquad \frac{M = N \in A}{M \in Eq(A, M, N)} \qquad \frac{M = N \in A}{N \in Eq(A, M, N)}$$

The following proposition ensures that our definition of equality is reasonable.

Proposition 1 (equality reflection) *Assume that $M \in A$ and $N \in A$ are derivable under the empty context. Then $M = N \in A$ is derivable under the empty context iff M and N are β-convertible.*

Proof. The "if" direction is proved by applying the conversion rule to $M \in \{M\}_A$. Let us prove the "only if" direction. Let T be the set of closed terms of untyped lambda calculus. β-convertibility is an equivalence relation on it. Let D be the set of β-conversion equivalence classes of T. Then D is a model of (total) untyped lambda calculus known as the closed term model. Interpreting *Type* as the powerset of D, an interpretation of ATT is obtained in the same way as the semantics in section 2.4. Note that the interpretation is identical, i.e., $[\![M]\!]_\rho$ is the term obtained from M by the substitution induced by the assignment ρ. Note that $[\![\{A\}_B]\!]_\rho$ is the singleton of $[\![A]\!]_\rho$. If $M = N \in A$, i.e., $M \in \{N\}_A$, is derivable under the empty context, then the equivalence classes of M and N must be identical by soundness and the interpretation of singleton. Hence M and N are β-convertible. □

Equality reflection was first proved by Luo for Leibnitz equality in his system ECC [Luo90] by a syntactical analysis of normalized terms. (Martin-Löf had considered the same problem for his inconsistent type theory [Mar71].) It seems difficult to prove it for ATT by syntactical analysis. The possibility and importance of equality reflection for ATT was pointed out by Luo.

Many other important notions can be defined in ATT, e.g., subtype judgment and a variant of the conditional formulas of PX. But we do not include them here. They will appear elsewhere together with some interpretations of logic into ATT and ATTT (variants of the Curry-Howard isomorphism).

2.7 Application to program extraction

In this section, we sketch how program extraction is achieved in ATT. By the types and judgments defined in the previous section, we can write specifications of programs as types of ATT just as in Nuprl and CoC. (In practice, these are not enough. We have to add some additional types such as recursive types and recursion operators for them. But this is another matter.) A typical specification with a subset type is $\Pi\, x \in A.\{y \in B | C\}$ (see [NPS90]). By this type, we mean the set of functions f from $A \to B$ such that $C[y := f(x)]$ is inhabited. Unfolding the definition of Π and the subset type, we have the following type:

$$\bigwedge x \in A.(\{x\}_A \to \bigvee y \in B.\bigvee z \in C.\{y\}_B).$$

The translation of this type into ibMPS is $A \to B$. Hence a function f of this type has the type $A \to B$ in ibMPS and so also in ATT. Furthermore, the condition on f is fulfilled.

Subset types are used to hide unnecessary information in specifications. If one does not need to have the information from C of $\prod x \in A.\Sigma y \in B.C$, then one replaces the Σ with a subset type. But this hides all of the information from C. In ATT, finer hiding of information is possible. Suppose that we need the information y_2 but not y_1 in $\prod x \in A.\Sigma y_1 \in B_1.\Sigma y_2 \in B_2.C$. If the Σ for y_1 is changed to a subset type, then it hides y_2 as well. When the Σ is replaced by $\bigvee$, then the specification become what we expect. Note that we cannot always exchange y_1 and y_2, as y_1 may appear in B_2. But, we do *not* know any natural and concrete example for this. This is one of the most important open problems in our approach.

3 ATTT: ATT Typed more

ATT is a type system based on untyped lambda calculus. Although untyped lambda calculus serves well as a semantics of type systems, there are some more natural semantics, e.g., coherence semantics. In this section, we introduce another type theory called ATTT based on Girard-Reynolds' $\lambda 2$. (The formal system of ATTT in this paper is different from the system in [Hay91], as the present system has a stronger singleton-elimination.) As the coherence space gives a semantics of $\lambda 2$, it also gives a semantics of ATTT.

The basic idea of ATTT is to keep types just as in $\lambda 2$ and introduce the notion of refinement (or refinement type) of types for specifications of programs. A refinement of a type T is a subset of T. In some semantics, e.g., in coherence semantics, a type is always inhabited. A refinement is an arbitrary subset of a type, so it can be empty.

The refinement approach of ATTT is closely related to the notion of a modified realizability interpretation. Modified realizability determines a set of realizers and their functionality. Let's consider the modified realizability of the formula $\forall x \in A.\exists y \in B.(f(x) =_C g(y))$. A term a is a realizer of the formula iff a is a term of type $A \to B$ and $\forall x \in A.(f(x) = g(a(x)))$ holds. In ATTT, the formula is interpreted by the following *refinement* of the type $A \to B$:

$$\bigwedge x \in A.(\{x\}_A \to \bigvee y \in B.\bigvee p \in Eq(C, f(x), g(y)).\{y\}_B).$$

The type $A \to B$ gives the functionality of the realizers of the formula and the refinement above gives the set of realizers of the formula. This will become clearer when we give a translation of ATTT into $\lambda 2$ below.

The idea of refinement types was first introduced by Freeman and Pfenning [FP91] in the context of ML. Our notion of refinement was influenced by their work, although the aim and presentation are different. Freeman and Pfenning write $A : B :: C$ iff A is

an element of the refinement type B and B is a refinement of C. We will not introduce new judgment $A : B :: C$ but will introduce a new kind $\mathit{refine}(A)$ for each type A. This denotes the set of the refinements of A. Hence, the equivalent of $A : B :: C$ is the pair of judgments $A \in B$ and $B \in \mathit{refine}(C)$.

3.1 Formulation of ATTT

The preterms of ATTT are defined by the following grammar:

$$\begin{array}{rcl} T & ::= & x \mid c \mid \lambda x \in T.T \mid TT \mid \mathit{refine}(T) \mid T \to T \mid T \wedge T \mid T \vee T \mid \\ & & \Pi\, x \in T.T \mid \bigwedge x \in T.T \mid \bigvee x \in T.T \mid \{T\}_T. \end{array}$$

Note that *refine* is not a constant but is a syntactical operator. *Sorts* of ATTT are *Kind*, *Type*, *RKind* and preterms of the form of $\mathit{refine}(A)$. In the following, s stands for a sort. Among the sorts, *RKind* and $\mathit{refine}(A)$ are called *refinement sorts* and $\mathit{refine}(A)$ is called a *refinement kind*. We will give inference rules of ATTT below:

The axiom, start rule, weakening rule and conversion rule are the same as in ATT, so we do not reproduce them here. The rule of subject reduction is an admissible rule for ATTT (see 3.5). Besides these general rules, ATTT has two rules for refinement.

refine-formation

$$\frac{\Gamma \vdash M \in \mathit{Type}}{\Gamma \vdash \mathit{refine}(M) \in \mathit{RKind}}$$

refine-subtyping

$$\frac{\Gamma \vdash C \in A \quad \Gamma \vdash A \in \mathit{refine}(B)}{\Gamma \vdash C \in B}$$

Rules for dependent product

Π-formation

$$\frac{\Gamma, x \in \mathit{Type} \vdash A \in \mathit{Type}}{\Gamma \vdash (\Pi\, x \in \mathit{Type}.A) \in \mathit{Type}} \qquad \frac{\Gamma, x \in \mathit{Type} \vdash R \in \mathit{refine}(A)}{\Gamma \vdash (\Pi\, x \in \mathit{Type}.R) \in \mathit{refine}(\Pi\, x \in \mathit{Type}.A)}$$

Π-introduction

$$\frac{\Gamma, x \in \mathit{Type} \vdash A \in \mathit{Type} \quad \Gamma, x \in \mathit{Type} \vdash B \in A}{\Gamma \vdash (\lambda x \in \mathit{Type}.B) \in (\Pi\, x \in \mathit{Type}.A)}$$

$$\frac{\Gamma, x \in \mathit{Type} \vdash R \in \mathit{refine}(A) \quad \Gamma, x \in \mathit{Type} \vdash B \in R}{\Gamma \vdash (\lambda x \in \mathit{Type}.B) \in (\Pi\, x \in \mathit{Type}.R)}$$

Π-elimination

$$\frac{\Gamma \vdash F \in (\Pi\, x \in \mathit{Type}.A) \quad \Gamma \vdash B \in \mathit{Type}}{\Gamma \vdash (FB) \in A[x := B]}$$

Rules for function space

$\rightarrow$-formation

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type}}{\Gamma \vdash A \rightarrow B \in \mathit{Type}} \qquad \frac{\Gamma \vdash C \in \mathit{refine}(A) \quad \Gamma \vdash D \in \mathit{refine}(B)}{\Gamma \vdash C \rightarrow D \in \mathit{refine}(A \rightarrow B)}$$

$\rightarrow$-introduction

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type} \quad \Gamma, x \in A \vdash C \in B}{\Gamma \vdash (\lambda x \in A.C) \in A \rightarrow B}$$

$$\frac{\Gamma \vdash C \in \mathit{refine}(A) \quad \Gamma \vdash D \in \mathit{refine}(B) \quad \Gamma, x \in C \vdash E \in D}{\Gamma \vdash (\lambda x \in A.E) \in C \rightarrow D}$$

$\rightarrow$-elimination

$$\frac{\Gamma \vdash F \in A \rightarrow B \quad \Gamma \vdash C \in A}{\Gamma \vdash (FC) \in B}$$

Rules for union

$\vee$-formation

$$\frac{\Gamma \vdash A \in \mathit{refine}(C) \quad \Gamma \vdash B \in \mathit{refine}(C)}{\Gamma \vdash A \vee B \in \mathit{refine}(C)}$$

$\vee$-introduction

$$\frac{\Gamma \vdash A \in \mathit{refine}(C) \quad \Gamma \vdash B \in \mathit{refine}(C) \quad \Gamma \vdash D \in A}{\Gamma \vdash D \in A \vee B}$$

$$\frac{\Gamma \vdash A \in \mathit{refine}(C) \quad \Gamma \vdash B \in \mathit{refine}(C) \quad \Gamma \vdash D \in B}{\Gamma \vdash D \in A \vee B}$$

$\vee$-elimination

$$\frac{\Gamma \vdash C \in A \vee B \quad \Gamma, x \in A \vdash D \in E \quad \Gamma, x \in B \vdash D \in E}{\Gamma \vdash D[x := C] \in E[x := C]}$$

where E is not a sort.

$\bigvee$-formation

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{refine}(C)}{\Gamma \vdash (\bigvee x \in A.B) \in \mathit{refine}(C)}$$

where s is not Kind.

$\bigvee$-introduction

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{refine}(C) \quad \Gamma \vdash D \in A \quad \Gamma \vdash E \in B[x := D]}{\Gamma \vdash E \in (\bigvee x \in A.B)}$$

where s is not Kind.

$\bigvee$-elimination

$$\frac{\Gamma \vdash C \in (\bigvee x \in A.B) \quad \Gamma, x \in A, y \in B \vdash D \in E}{\Gamma \vdash D[y := C] \in E[y := C]}$$

where x is not in $FV(D) \cup FV(E)$ and E is not a sort.

Rules for intersection

$\wedge$-formation

$$\frac{\Gamma \vdash A \in \mathit{refine}(C) \quad \Gamma \vdash B \in \mathit{refine}(C)}{\Gamma \vdash A \wedge B \in \mathit{refine}(C)}$$

$\wedge$-elimination

$$\frac{\Gamma \vdash C \in A \wedge B}{\Gamma \vdash C \in A} \qquad \frac{\Gamma \vdash C \in A \wedge B}{\Gamma \vdash C \in B}$$

$\wedge$-introduction

$$\frac{\Gamma \vdash A \in \mathit{refine}(C) \quad \Gamma \vdash B \in \mathit{refine}(C) \quad \Gamma \vdash D \in A \quad \Gamma \vdash D \in B}{\Gamma \vdash D \in A \wedge B}$$

$\bigwedge$-formation

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{refine}(C)}{\Gamma \vdash (\bigwedge x \in A.B) \in \mathit{refine}(C)}$$

$\bigwedge$-elimination

$$\frac{\Gamma \vdash C \in (\bigwedge x \in A.B) \quad \Gamma \vdash D \in A}{\Gamma \vdash C \in B[x := D]}$$

where s is not Kind.

$\bigwedge$-introduction

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in \mathit{refine}(C) \quad \Gamma, x \in A \vdash D \in B}{\Gamma \vdash D \in (\bigwedge x \in A.B)}$$

where x is not in $FV(D)$ and s is not Kind.

Rules for singleton

singleton-formation

$$\frac{\Gamma \vdash B \in \mathit{Type} \quad \Gamma \vdash A \in B}{\Gamma \vdash \{A\}_B \in \mathit{refine}(B)}$$

singleton-introduction

$$\frac{\Gamma \vdash B \in \mathit{Type} \quad \Gamma \vdash A \in B}{\Gamma \vdash A \in \{A\}_B}$$

singleton-elimination

$$\frac{\Gamma \vdash P_1 \in \{A\}_C \wedge \{B\}_C \quad \Gamma \vdash P_2[x := A] \in D[x := A] \quad \Gamma, x \in C \vdash D \in s}{\Gamma \vdash P_2[x := B] \in D[x := B]}$$

where s is a refinement kind. (Note that P_2 is a preterm.)

The second elimination rule of ATT is not present in ATTT, because it is derivable in ATTT. (If $\{A\}_B$ is a legal term, then $\{A\}_B \in \mathit{refine}(B)$ is derivable. Then *refine*-subtyping implies $C \in B$ from $C \in \{A\}_B$.) Note that the remained elimination rule is a strong singleton-elimination rule. As opposite to the case of ATT, this is sound for the translation from ATTT to $\lambda 2$ introduced later. The rule is stated only for refinements. The same rule for types is an admissible rule (corollary 6.)

Legal sequents and terms, etc. are defined as in ATT. Basic lemmas, e.g., the Church-Rosser property, substitution etc., hold for ATTT as well. Lemma 3 must be changed as follows:

Lemma 11 *Assume that* $x_1 \in A_1, \ldots, x_n \in A_n \vdash B \in C$ *is a legal sequent. Then* B *is neither Kind nor RKind. Furthermore, for each* i, A_i *is neither Kind nor RKind, and one of the followings holds:*

1. A_i *is the kind Type and* $x_1 \in A_1, \ldots, x_{i-1} \in A_{i-1} \vdash A_i \in Kind$ *is derivable.*
2. A_i *has the form refine*(D) *and* $x_1 \in A_1, \ldots, x_{i-1} \in A_{i-1} \vdash A_i \in RKind$ *is derivable.*
3. $x_1 \in A_1, \ldots, x_{i-1} \in A_{i-1} \vdash A_i \in Type$ *is derivable.*
4. $x_1 \in A_1, \ldots, x_{i-1} \in A_{i-1} \vdash A_i \in refine(D)$ *is derivable for some* D.

3.2 Classification of terms of ATTT

We classify the terms of ATTT as in the classification in ATT.

Definition 7 (classification under variable classification) Assume that $FV(A)$ is a union of three disjoint sets *Obj*, *Typ* and *Ref*. The variables of *Obj* are called object variables, the variables of *Ref* are called refinement variables, and the variables of *Typ* are called type variables. Then objects, types, refinements, kinds and refinement kinds (r-kinds for short) under the classification of free variables $\langle Obj, Typ, Ref\rangle$ are defined as follows:

objects
1. An object variable is an object.
2. If A and B are objects, then so is (AB).
3. If A is an object under the classification $\langle Obj \cup \{x\}, Typ, Ref\rangle$ and B is a type, then $\lambda x \in B.A$ is an object under the classification $\langle Obj, Typ, Ref\rangle$.

types
1. A type variable is a type.
2. If A and B are types, then so is $A \to B$.
3. If A is a type and B is a type under the classification $\langle Obj, Typ \cup \{x\}, Ref\rangle$, then $\Pi\, x \in Type.A$ is a type under the classification $\langle Obj, Typ, Ref\rangle$.

refinements
1. A refinement variable is a refinement.
2. If A is an object and B is a type, then $\{A\}_B$ is a refinement.
3. If A and B are refinements, then so are $A \to B$, $A \vee B$ and $A \wedge B$.
4. If A is a refinement under the classification $\langle Obj, Typ \cup \{x\}, Ref\rangle$, then $\Pi\, x \in Type.A$ is a refinement under the classification $\langle Obj, Typ, Ref\rangle$.

5. If A is a type or a refinement, and B is a refinement under the classification $\langle Obj \cup \{x\}, Typ, Ref\rangle$, then $\bigwedge x \in A.B$ and $\bigvee x \in A.B$ are refinements under the classification $\langle Obj, Typ, Ref\rangle$.
6. If A is an r-kind and B is a refinement under the classification $\langle Obj, Typ, Ref \cup \{x\}\rangle$, then $\bigwedge x \in A.B$ and $\bigvee x \in A.B$ are refinements under the classification $\langle Obj, Typ, Ref\rangle$.

kind The constant *Type* is a kind.

r-kind If A is a type, then $refine(A)$ is an r-kind. □

As in the case of ATT, any legal term of ATTT is uniquely classified into one of the classifications above, or one of *Kind* and *RKind*. ATTT-versions of the basic lemmas for ATT also hold. It should be clear how to formulate and prove them and so we will not include them here.

3.3 Translation of ATTT into $\lambda 2$

Now we give a translation of ATTT into a GTS-formulation of $\lambda 2$. The GTS-formulation of $\lambda 2$ is given by dropping $refine(T)$, $T \vee T$, $T \wedge T$, $\bigvee x \in T.T$, $\bigwedge x \in T.T$ and $\{T\}_T$ from the grammar of preterms. The conversion rule, rule of subject reduction and all of the rules for these are also dropped, e.g., the second inference rule of Π-formationis dropped. (Note that the rule of subject reduction is an admissible rule of $\lambda 2$.) The system was called $\lambda 2$ in [Bar90].

Definition 8 (translation of preterms of ATTT into $\lambda 2$) We define a *nondeterministic* translation from ATTT to $\lambda 2$. That is, we define a binary relation $M \rightsquigarrow M'$ between preterms M of ATTT and preterms M' of $\lambda 2$, as follows:

1. $RKind \rightsquigarrow Kind$ and $c \rightsquigarrow c$ for the other constant c.
2. $refine(A) \rightsquigarrow Type$.
3. $x \rightsquigarrow x$.
4. If $A \rightsquigarrow A'$ and $B \rightsquigarrow B'$, then $(\lambda x \in A.B) \rightsquigarrow (\lambda x \in A'.B')$.
5. If $A \rightsquigarrow A'$ and $B \rightsquigarrow B'$, then $(AB) \rightsquigarrow (A'B')$.
6. If $A \rightsquigarrow A'$ and $B \rightsquigarrow B'$, then $(\Pi\, x \in A.B) \rightsquigarrow (\Pi\, x \in A'.B')$ and $(A \to B) \rightsquigarrow (A' \to B')$
7. If $A \rightsquigarrow A'$ and $B \rightsquigarrow B'$, then $A\ \mathsf{op}\ B \rightsquigarrow A'$ and $A\ \mathsf{op}\ B \rightsquigarrow B'$, where op is one of $\wedge$, $\vee$.

8. If $A \rightsquigarrow A'$ and $B \rightsquigarrow B'$, then $(\mathsf{OP}\, x \in \mathit{refine}(A).B) \rightsquigarrow B'[x := A']$, where OP is one of $\bigwedge, \bigvee$.

9. If $B \rightsquigarrow B'$ and A is not an r-kind, then $(\mathsf{OP}\, x \in A.B) \rightsquigarrow B'$, where OP is one of $\bigwedge, \bigvee$.

10. If $B \rightsquigarrow B'$, then $\{A\}_B \rightsquigarrow B'$.

If $M \rightsquigarrow M'$ holds, then we call M' a *translation of* M. □

The translation of judgments is naturally defined. If $A \rightsquigarrow A'$ and $B \rightsquigarrow B'$, then $A \in B \rightsquigarrow A' \in B'$. The translation of contexts and sequents is not straightforward, however.

Definition 9 (translation of contexts and sequents) Let Γ be $x_1 \in A_1, \ldots, x_n \in A_n$ and $A_i \rightsquigarrow A_i'$ for $i = 1, \ldots, n$. Set

$$\Gamma = \Gamma_0, x_{p_1} \in \mathit{refine}(T_{p_1}), \Gamma_1, \ldots, x_{p_m} \in \mathit{refine}(T_{p_m}), \Gamma_m$$

such that any judgment in Γ_j $(j = 1, \ldots, m)$ is not of the form $x \in \mathit{refine}(T)$. That is, $x_{p_1} \in A_{p_1}, \ldots, x_{p_m} \in A_{p_m}$ is the enumeration of all of the judgments in Γ whose A_i is an r-kind and $\mathit{refine}(T_{p_j}) = A_{p_j}$.

Let Γ_j' be the context obtained from Γ_j by replacing each judgment $x_i \in A_i$ by $x_i \in A_i'$. For $j = 1, \ldots, m$, let σ_j be the substitution $[x_{p_1} := T_{p_1}, \cdots, x_{p_j} := T_{p_j}]$. (Note that x_{p_j} does not occur freely in $T_{p_{j+k}}$, as $T_{p_{j+k}}$ is a type.) In particular, we will denote σ_m by $[\Gamma]$. Set

$$\Gamma' = \Gamma_0', \Gamma_1'\sigma_1, \ldots, \Gamma_m'\sigma_m.$$

Then we call Γ' a *translation of the context* Γ and write $\Gamma \rightsquigarrow \Gamma'$.

Next, we define the translation of a sequent. Let S be $\Gamma \vdash B \in C$. Assume that $\Gamma \rightsquigarrow \Gamma'$, $B \rightsquigarrow B'$ and $C \rightsquigarrow C'$. Set

$$S' = \Gamma' \vdash B'[\Gamma] \in C'[\Gamma].$$

Then we call S' a *translation of the sequent* S and write $S \rightsquigarrow S'$. □

For example, $x \in \mathit{Type}, z \in x \vdash x \in \mathit{Type}$ is a translation of the legal sequent

$$x \in \mathit{Type}, y \in \mathit{refine}(x), z \in y \vdash (y \wedge \{z\}_x) \in \mathit{refine}(x).$$

The soundness of the translation is stated as follows:

Theorem 4 *If S is a legal sequent of* ATTT *and $S \rightsquigarrow S'$, then S' is a legal sequent of $\lambda 2$.*

Note that if M is an object or a type, then it is a preterm of $\lambda 2$, and any translation of M is identical to M. By this observation, the following is obvious:

Corollary 3 ATTT *is a conservative extension of* $\lambda 2$ *in the same sense of corollary 2. Furthermore, if* $\Gamma \vdash J$ *is provable in* ATTT *and* $\Gamma \vdash J \rightsquigarrow \Gamma' \vdash J'$, *then* $\Gamma \vdash J'$ *is also provable in* ATTT.

The second part of the theorem above is a consequence of generalized weakening and the fact that any judgment K of Γ' is derivable under the context Γ in ATTT. This fact is easily proved by the induction on the length of Γ.

An inversion principle similar to the type inversion principle for ATT (lemma 9) holds for ATTT. Hence, we can prove the strong normalization property as in the case of ATT.

Corollary 4 *Every term of* ATTT *is strongly normalizable.*

Note that strong normalization property of legal objects and types is obvious by the conservativity result.

Theorem 4 is a corollary of the following proposition.

Proposition 2 *1. If* S *is a legal sequent of* ATTT *and* $S \rightsquigarrow S'$, *then* S' *is a legal sequent of* $\lambda 2$.

2. If S *is a legal sequent of* ATTT *and* $S \rightsquigarrow S'$ *and* $S \rightsquigarrow S''$, *then* S' *and* S'' *are identical.*

3. If $\Gamma \vdash A \in \mathit{refine}(B)$ *is a legal sequent of* ATTT *and*

$$(\Gamma \vdash A \in \mathit{refine}(B)) \rightsquigarrow (\Gamma \vdash A' \in C),$$

then A' *is* B *and* C *is* Type.

This propostition can be proved by simultaneous induction on derivations. The proof is straightforward, but lengthy. So we do not include it. Note that the following lemma is used to show the soundness of the singleton-elimination rule. This lemma can be proved by induction on the complextity of P.

Lemma 12 *The following is an admissible rule of* $\lambda 2$:

$$\frac{\Gamma \vdash A \in C \quad \Gamma \vdash B \in C \quad \Gamma \vdash P[x := A] \in D \quad \Gamma \vdash D \in \mathit{Type}}{\Gamma \vdash P[x := B] \in D}$$

As a corollary of proposition 2, we can prove the following:

Corollary 5 *If* $\Gamma \vdash A \in \mathit{refine}(B)$ *and* $\Gamma \vdash A \in \mathit{refine}(B')$ *are legal sequents of* ATTT*, then* B *and* B' *are identical.*

We will call the unique type B of $A \in \mathit{refine}(B)$ *the type of* A written $\mathit{type}(A)$. Note that $\mathit{type}(A)$ depends on the context under which A is legal. When we emphasize this fact, we will write $\mathit{type}_\Gamma(A)$.

In the rule of singleton-elimination, D is restricted to refinements. The same rule for types is an admissible rule.

Corollary 6 *The rule of singleton-elimination of* ATTT *is an admissible rule, when the sort* s *is replaced by the kind* Type*.*

Proof. Assume the three premises of the rule are proveble in ATTT and s is Type. Note that $D[x := A]$ is identical to D, as D is a type and so x is not free in D. By the translation, the following sequents are provable in $\lambda 2$:

$$\Gamma' \vdash A \in C, \quad \Gamma' \vdash B \in C, \quad \Gamma' \vdash P_2[x := A] \in D, \quad \Gamma' \vdash D \in \mathit{Type}.$$

By lemma 12, $\Gamma' \vdash P_2[x := B] \in D$ is derivable in $\lambda 2$. By generalized weakening and corollary 3, the conclusion of the rule is derivable in ATTT. □

3.4 A decidability result

We conjecture that typechecking in ATT will be undecidable. On the other hand, ATTT has a nice decidability result.

Definition 10 (family) A *family* is a sort or a type. If N is a family and $\Gamma \vdash M \in N$ is provable, then N is called a family of M under Γ. It is easy to see that a family N of M under Γ uniquely exists for any legal term M under Γ. We call this family N *the family of* M *under* Γ.

Theorem 5 *Let* Γ *denote a context and* M *denote a preterm except Kind and RKind. There are algorithms* V_C *and* V_F *satisfying the followings:*

1. $V_C(\Gamma)$ *always returns* $\top$ *or* $\bot$*.*
2. $V_C(\Gamma)$ *is* $\top$ *iff* Γ *is a legal context.*
3. $V_F(\Gamma, M)$ *always returns* $\bot$ *or a family.*
4. *if* $V_F(\Gamma, M)$ *is* $\bot$*, then* Γ *is illegal or* M *is illegal under* Γ*.*
5. *if* $V_F(\Gamma, M)$ *is not* $\bot$*, then* $V_F(\Gamma, M)$ *is the family of* M *under* Γ*.*

Corollary 7 *Provability of* $\Gamma \vdash M \in N$ *is decidable, unless* N *is a refinement.*

Objects of ATTT are regarded as *programs* and refinements are regarded as *specifications* or *formulas*. For a refinement N, $\Gamma \vdash M \in N$ means that the program M meets the specification N. This notion cannot be decidable. On the other hand, types of ATTT are regarded as types in the sense of programming languages and refinements of ATTT are regarded as specifications (or formulas). These two notions and membership of types are decidable as expected.

Contrary to corollary 7, provability of a sequent is always decidable in calculi like CoC. This is because objects of such calculi contain logical information as well as computational contents (program). Namely, they are partly *proofs*. It's no wonder that proof is a decidable notion. As both of logical information and computational contents are coded by means of typed lambda-calculus, CoC cannot distinguish program-parts and proof-parts and so proof-parts must be executed in execution time. An approach to avoid this defect is keeping proof-parts in a separated term. This approach was taken in [BM91].

It is easy to define V_C by V_F, since $\Gamma, x \in M$ is legal iff $V_F(\Gamma, M)$ is not $\bot$. We present the algorithms V_F below. Their properties can be proved by means of a lexicographical induction on complexity of M and the length of Γ. The proof is left to the readers.

$V_F(\Gamma, M)$ is
  if $V_C(\Gamma)$ is $\bot$ then return $\bot$
  else if M is a variable then
      if $M \in N$ appears in Γ then
        if N is a refinement then return $type_\Gamma(A)$
        else return N
      else return $\bot$
  else if M is *Type* then return *Kind*
  else if M is $refine(N)$ then
      if $V_F(\Gamma, N)$ is *Type* then return *RKind* else return $\bot$
  else if M is $\Pi x \in M_1.M_2$ then
      if M_1 is *Type* then
        if $V_F(\Gamma, x \in M_1, M_2)$ is *Type* then return *Type*
        else if $V_F(\Gamma, x \in M_1, M_2)$ is $refine(N)$ then return $refine(\Pi x \in Type.N)$
        else return $\bot$
      else return $\bot$
  else if M is $M_1 \to M_2$ then
      if $V_F(\Gamma, M_1)$ and $V_F(\Gamma, M_2)$ is *Type* then return *Type*
      else if $V_F(\Gamma, M_1)$ is $refine(N_1)$ and $V_F(\Gamma, M_2)$ is $refine(N_2)$ then
          return $refine(N_1 \to N_2)$
      else return $\bot$
  else if M is M_1 op M_2, where op is $\wedge$ or $\vee$ then
      if $V_F(\Gamma, M_1)$ and $V_F(\Gamma, M_2)$ is $refine(N)$ then

return $refine(N)$
else return $\bot$
else if M is $\mathsf{OP}\ x \in M_1.M_2$ where OP is $\bigwedge$ or $\bigvee$ then
if $V_F(\Gamma, M_1)$ is a sort except $Kind$ and $V_F(\Gamma, x \in M_1, M_2)$ is $refine(N)$ then
return $refine(N)$
else return $\bot$
else if M is $\{M_1\}_{M_2}$ then
if $V_F(\Gamma, M_2)$ is $Type$ and $V_F(\Gamma, M_1)$ is M_2 then return M_2
else return $\bot$
else if M is $(M_1 M_2)$ then
if $V_F(\Gamma, M_1)$ is $\Pi x \in V_F(\Gamma, M_2).N$ then return $N[x := M_2]$
else if $V_F(\Gamma, M_1)$ is $V_F(\Gamma, M_2) \to N$ then return N
else return $\bot$
else if M is $\lambda x \in M_1.M_2$ then
if $V_F(\Gamma, M_1)$ is $Type$ and $V_F(\Gamma, x \in M_1, M_2)$ is a type then
return $M_1 \to V_F(\Gamma, x \in M_1, M_2)$
else if M_1 is $Type$ then
if $V_F(\Gamma, x \in Type, M_2)$ is a type then
return $\Pi x \in Type.V_F(\Gamma, x \in Type, M_2)$
else return $\bot$
else return $\bot$

3.5 RL-rule and LR-rule

An implication of singleton-elimination rule is the following RL-rule (Right-to-Left-rule).

$$\frac{\Gamma \vdash P \in \{M\}_C \wedge D}{\Gamma \vdash M \in D}$$

Note that D is a refinement. Since we can move M from the right hand side of $\in$ to the left hand side. We call the opposite rule LR-rule.

$$\frac{\Gamma \vdash M \in D}{\Gamma \vdash M \in \{M\}_C \wedge D}$$

It's a trivial implication of the introduction rules for singleton and intersection.

In set theory, a set D has a nonempty intersection with a singleton $\{M\}$, iff $M \in D$ holds. The RL-rule and LR-rule are a counterpart of this fact.

Proposition 3 *RL-rule and LR-rule are derivable in ATTT.*

Proof. Let's prove RL-rule. Assume $\Gamma \vdash P \in \{M\}_C \wedge D$ is provable. Let x be a fresh variable and s be the family of D (under Γ). Then $\Gamma \vdash P \in \{M\}_C$, $\Gamma \vdash P \in \{P\}_C$, $\Gamma \vdash P \in D$, $\Gamma, x \in C \vdash D \in s$ are derivable. Regarding P as $x[x := P]$, the conclusion of the RL-rule is derived by the singleton-elimination form these. $\square$

Note that the strong rule is essentially used in the proof above. It is easy to see the (weak) rule of singleton-elimination and LR-rule imply the strong rule of singleton-elimination. The proof of this fact is left for readers.

By means of LR-rule and RL-rule, we may regard refinement types as predicates (or sets). Let D be a refinement type of the kind $refine(A)$ and M be an object of the type A. Then, D may be regarded as a predicate on the type A and the refinement $D \wedge \{M\}$ may be regarded as the application $D(M)$ of the predicate D to the object M. We regard $D \wedge \{M\}$ a true formula iff it has *the* element M. By LR-rule and RL-rule, $D \wedge M$ is true iff $M \in D$ holds. In this sense, $D \wedge -$ is a transformation from refinements to predicates.

An implication of RL-rule and LR-rule is subject reduction.

Corollary 8 *Subject reduction holds for* ATTT.

Proof. Assume $\Gamma \vdash M \in N$ is provable and M reduces to M'. We wish to prove $\Gamma \vdash M' \in N$. If N is a term of $\lambda 2$, then we can prove it by subject reduction for $\lambda 2$ and corollary 3. If N is a refinement, then we apply LR-rule and conversion rule to it and get $\Gamma \vdash M \in N \wedge \{M'\}$. By RL-rule, we get the sequent to be proved. If N is a refinement kind, then we can prove it by easy induction on the derivation. □

Another implication of RL-rule and LR-rule is distributive laws for union and intersection discussed in 2.1.3. Let us prove the following:

$$\Gamma_0, x \in (a \vee b) \wedge c \vdash x \in (a \wedge c) \vee (b \wedge c),$$

where Γ_0 is a context which maintains a, b, c are refinements of a type. Let Γ_1 be $\Gamma_0, x \in (a \vee b) \wedge c)$. Then the following are derivable:

$$(1) \quad \Gamma_1 \vdash x \in (a \wedge \{x\}) \vee (b \wedge \{x\}) \qquad (2) \quad \Gamma_1 \vdash x \in c$$

Let Γ_2 be the context $\Gamma_1, y \in a \wedge \{x\}$. Using RL-rule, we can derive $\Gamma_2 \vdash x \in a$. From this and (2), we can derive $\Gamma_2 \vdash x \in (a \wedge c) \vee (b \wedge c)$. Similarly, $\Gamma_1, y \in b \wedge \{x\} \vdash x \in (a \wedge c) \vee (b \wedge c)$ is derivable. By applying $\vee$-elimination to (1) and these two, we can derive the judgment to be proved. The other law presented in 2.1.3 is proved similarly.

Note that a special case of the distributive law considered above is $x \in (a \vee b) \wedge \{x\} = (a \wedge \{x\}) \vee (b \wedge \{x\})$. Regarding refinements as predicates, this turns out $(a \vee b)(x) = a(x) \vee b(x)$. This is the distributivity of application and union.

3.6 Semantics of ATTT

The semantics of ATTT is fairly simple. We start with the so called BMM-semantics of $\lambda 2$ (see [BMM90] for BMM-semantics). The BMM-semantics is a "concrete" semantics, that is, each type of $\lambda 2$ is interpreted as a set. Let $[\![A]\!]_\rho$ be the interpretation of type A under assignment ρ. Then we interpret refinement kinds by

$$[\![refine(A)]\!]_\rho = Pow([\![A]\!]_\rho).$$

Pow(X) is the powerset of X. Hence, refinements of a type are subsets of the interpretation of the type. Singleton, intersection and union refinements are interpreted in the usual sense of set theory. Then it is straightforward to check the soundness. A more intrinsic semantics could be given in appropriate categories, such as the topos of Pitts [Pit87]. Furthermore, we conjecture that there is a PL-category-like second-order categorical framework for ATTT.

As remarked in section 2.4, the PER-semantics for ATT has some anomalies. But this is not so in ATTT, because we can state the extensionality rule only for types and not for refinements.

Extensionality

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma \vdash B \in \mathit{Type} \qquad \Gamma, f \in (A \to B), g \in (A \to B), x = y \in A \vdash f(x) = g(y) \in B}{\Gamma, f \in (A \to B), g \in (A \to B) \vdash f = g \in (A \to B)}$$

Note that we can define the equality judgment only for types and not for refinements, since $a = b \in A$ is $a \in \{b\}_A$ and so A must be a type. Hence, two functions from refinement $R_1 \to R_2$ must be compared in the type $\mathit{type}(R_1) \to \mathit{type}(R_2)$.

3.7 An example of a specification

Here, we present an example of a specification in ATTT. We specify the quotient and remainder algorithm for natural numbers. Let Nat be a type of natural numbers, e.g., the polymorphic natural numbers $\Pi x \in \mathit{Type}.(x \to x) \to (x \to x)$. A specification would be

$$\forall a \in \mathit{Nat}.\forall b \in \mathit{Nat}^+.\exists q \in \mathit{Nat}.\exists r \in \mathit{Nat}.(a = b * q + r \wedge r < b).$$

In the above, Nat^+ is a refinement of positive natural numbers. It may be defined by $\bigvee n \in \mathit{Nat}.\{\mathit{suc}(n)\}_{\mathit{Nat}}$. Let $R(a, b, q, r)$ be a refinement which is inhabited iff the formula holds. As we don't wish to have a proof for $R(a, b, q, r)$ in the outputs of the quotient and remainder functions to be specified, we regard $\exists r \in \mathit{Nat}$ as a subset type operation. Hence $\exists r \in \mathit{Nat}.(a = b * q + r \wedge r < b)$ should be replaced by

$$\bigvee r \in \mathit{Nat}. \bigvee \mathit{prf} \in R(a, b, q, r).\{r\}_{\mathit{Nat}}.$$

Hence the specification turns to

$$\forall a \in \mathit{Nat}.\forall b \in \mathit{Nat}^+.\exists q \in \mathit{Nat}. \bigvee r \in \mathit{Nat}. \bigvee \mathit{prf} \in R(a, b, q, r).\{r\}_{\mathit{Nat}}.$$

The universal and existential quantifiers are defined by the same way as in 2.6. The type of this specification (refinement) is $\mathit{Nat} \to \mathit{Nat} \to (\mathit{Nat} \times \mathit{Nat})$, where "$\times$" is the polymorphic cartesian product. Note that the type of the specification is irrelevant to the type of $R(a, b, q, r)$.

Another formalization of the specification without the auxiliary refinement $R(a,b,q,r)$ is

$$\begin{array}{l}\forall a \in Nat.\forall b \in Nat^{+}.\exists q \in Nat.\bigvee r \in Nat.\\ \quad \bigvee p1 \in \{a\}_{Nat} \wedge \{b * q + r\}_{Nat}.\\ \qquad \bigvee p2 \in (\bigvee x \in Nat^{+}.\{r + x\}_{Nat} \wedge \{b\}_{Nat}).\{r\}_{Nat}.\end{array}$$

The type of this specifications is again $Nat \to Nat \to (Nat \times Nat)$.

Note that the induction principle for Nat must be added to ATTT to prove these specifications as was done in systems like PX and Coq. (See 2.4.1 and 4.3 of [HN88]and 1.1.8 of [D^{+}91].)

4 Related works

Many authors have studied union and intersection types from disparate motivations, e.g., [FP91, MPS86, BDC91, Pie90, Pie91b, Rey88]. A more complete list of the literature on the subject and a fine survey of them can be found in [Pie91b]. But we here list only related work devoted to the same aim as ours: program extraction.

No literature has considered the suite of types, Singleton, Union and Intersection Types, SUIT in short, for program extraction. An exception is the work by Parigot and Krivine. They considered first order and second order intersections for a similar purpose to program extraction [Par88, Par89, KP90]. Parigot uses a type closely related to singleton types in [Par89]. But they do not consider union types, because they use functions explicitly rather than existential quantifiers.

Although none of types of SUIT is used explicitly, Paulin's modified realizability interpretation for CoC [PM89] and Burstall's deliverable [BM91] seem closely related to our work. Actually, these two were our constant sources of ideas. As we pointed out, ATTT is an axiomatization of modified realizability. The translation from ATTT to $\lambda 2$ seems to be a counterpart of Paulin's extraction operator $\mathcal{E}$ in [PM89]. There would be a formal relationship between Paulin's realization method and a higher order version of ATTT.

Burstall's deliverable is closely related to modified realizability and so to Paulin's work. His motivation was to separate proofs and programs in program development via CoC or ECC. This is close to our motivation of separating logic and computation. In ATTT, refinements serve as the device of logic and types serve as the device of computation. Unlike the deliverable approach, the logical part, i.e., inference on refinement, is not recorded in the terms developed. In deliverables, programs and proofs are separated terms of CoC. Roughly, a deliverable is a pair of a program and its correctness proof. In our case, the program part of a deliverable is the term A of the derived judgment $A \in B$ and the proof is represented as the derivation of the judgment. The exact relationships to Paulin and Burstall's work remain to be studied.

5 Future work

Full decidability of typechecking with SUIT is hopeless. Typechecking in the deliverable approach is reduced to typechecking of CoC and so is automatic. This is because of the presence of a term recording a history of proofs separately from programs. We feel that this approach separates proofs and programs more than we expect. There seems no apparent association of subparts of the proof-term and program-term of a deliverable. It would be difficult to see that a subterm of a program-term was verified by a part of the proof-term, and in what way. We are working to add annotations recording history of derivations to our calculi. Such annotations should record all the logical information of inference rules, but they should not be respected by computaion. When annotations are literally erased from the program text, then we get the pure program and the results of the pure program and the original program with annotations are the same. That is, these annotations should work like the documentation part in Knuth's WEB language. With these annotations, typechecking in ATT and ATTT must become decidable.

The application of SUIT to program extraction was briefly mentioned. We have already had some applications of SUIT for the aim, which will appear elsewhere. But SUIT in practice of program extraction has not been well studied. In particular, we have not considered SUIT in backward reasoning.

We have designed higher order versions of ATT and ATTT. Their metatheories are yet to be studied. We believe that there is a general method for adding SUIT to a type theory at least in the way of ATTT without interfering with any features of the original type theory. Such a method might give a general way to add logic to type theory.

The right semantics of ATT and ATTT is yet to investigate as noted in the paper. Furthermore, semantics of SUIT fully consistent with extensionality is yet to study. We conjecture that there is no such semantics.

Acknowledgments

I thank Zhaohui Luo, Christine Paulin, Benjamin Pierce and Yukihide Takayama for fruitful discussions on the subject and helpful comments on the paper. I am very grateful for thorough comments and criticisms on the paper by Jan Smith and Tierry Coquand. Their comments and criticisms were extremely helpful to polish the paper.

References

[Bar90] H. Barendregt. Introduction to generalized type systems. Technical Report no.90-8, University of Nijmegen, May 1990.

[BDC91] F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types. In T.Ito and A.R.Meyer, editors, *Theoretical Aspect of Computer Software (Lecture Notes in Comuter Science No. 526)*, pages 651–674, Berlin, 1991. Springer-Verlag.

[BM91] R. Burstall and J. McKinna. Deliverable: an approach to program development in the calculus of constructions. An earlier version is in the preliminary proceeding of the Logical Frameworks BRA meeting at Sophia Antipolis 1990, April 1991.

[BMM90] K.B. Bruce, A.R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 85:76–134, 1990.

[CH88] Th. Coquand and G. Huet. Calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Coq86] Th. Coquand. An analysis of Girard's paradox. In *Proceedings of First Annual Symposium on Logic in Computer Science*, 1986.

[D+91] C. Dowek et al. The coq proof assistant user's guide, version 5.6. Technical Report No. 134, INRIA, December 1991.

[FP91] T. Freeman and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Ontario*. ACM Press, June 1991.

[Hay90] S. Hayashi. Constructive mathematics and computer-aided reasoning systems. In P.P. Petkov, editor, *MATHEMATICAL LOGIC*, pages 43–52. Plenum Press, New York, 1990.

[Hay91] S. Hayashi. Singleton, union and intersection types for program extraction. In T.Ito and A.R.Meyer, editors, *Theoretical Aspect of Computer Software (Lecture Notes in Comuter Science No. 526)*, pages 701–730, Berlin, 1991. Springer-Verlag.

[HN88] S. Hayashi and H. Nakano. *PX: A Computational Logic*. The MIT Press, Cambridge, Mass., 1988.

[KP90] J.-L. Krivine and M. Parigot. Programmimg with proofs. *Journal of Information Processing and Cybernetics EIK*, 26:149–167, 1990.

[Lei86] D. Leivant. Typing and convergence in the lambda calculus. *Theoretical Computer Science*, 44:51–68, 1986.

[Luo90] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, July 1990.

[Mar71] P. Martin-Löf. A theory of types. Technical Report 71-3, University of Stockholm, 1971.

[Mar82] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology, and Philosophy of Science VI*, pages 153–179. North-Holland, 1982.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. North-Holland, 1990.

[Mog88] E. Moggi. *The Partial Lambda-Calculus*. PhD thesis, Department of Computer Science, University of Edinburgh, 1988.

[MPS86] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's type theory, an introduction*. Clarendon Press, Oxford, 1990.

[Par88] M. Parigot. Programming with proofs: a second order type theory. In *ESOP '88 (Lecture Notes in Computer Science No. 300)*. Springer-Verlag, 1988.

[Par89] M. Parigot. Recursive programming with proofs. In *Proceedings of Symposium Mathematiques et Informatique, Luminy, France*, October 1989. To appear in Theoretical Computer Science.

[Pie90] B.C. Pierce. Preliminary investigation of a calculus with intersection and union types. Unpublished manuscript, June 1990.

[Pie91a] B.C. Pierce. personal communication. 1991.

[Pie91b] B.C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, School of Computer Science, Carnegie Mellon University, Feburary 1991.

[Pit87] A.M. Pitts. Polymorphism is set theoretic constructively. In D.H. Pitt et al., editors, *Category theory and computer science (Lecture Notes in Comuter Science No. 283)*, pages 12–39, Berlin, 1987. Springer-Verlag.

[Plo85] G. Plotkin. Lectures given at ASL Stanford meeting. Unpublished lecture notes, July 1985.

[PM89] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of construction. In *Proceedings of 16th Annual ACM Symposium on Principles of Programming Languages*, 1989.

[Rey88] J.C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, School of Computer Science, Carnegie Mellon University, June 1988.

[Ros86] G. Rosolini. *Continuity and effectiveness in topoi*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1986.

[Tur89] D. Turner. A new formulation of constructive type theory. In P. Dybjer et al., editors, *Proceedings of the Workshop on Programming Logic*, pages 258–294, 1989. Report 54, Programming Methodology Group, Department of Computer Science, Chalmers University of Technology and University of Göteborg.