Logic of refinement types

Susumu Hayashi*
Department of Applied Mathematics and Informatics
Ryukoku University
Ohtsu, Shiga, JAPAN

Abstract

Refinement types are subsets of ordinary types, which are intended to be specifications of programs. Ordinary types correspond to constructive propositions by Curry-Howard isomorphism. Refinement types correspond to "classical" propositions by a semantics resembling interpretations of logics in categorical/algebraic logic. In this paper, we will study the logic of refinement types in the type system ATTT which was introduced in [9] as a framework for an "optimized" Curry-Howard isomorphism.

1 Introduction

ATTT is a conservative extension of Girard-Reynolds' second order polymorphic lambda calculus $\lambda 2$ (or F in more popular name). The aim of ATTT is to serve yet another type theoretic basis for program extraction or program development. In [9], the theory of pure ATTT was presented and its basic metatheory was developed. In this paper, we will study logics in ATTT. The main purpose of this paper is to show how logic is represented by refinements of ATTT. We will show that the refinements of each type is an "internally" complete Heyting algebra (cHa). Using the cHa structure, second order intuitionistic logic is naturally interpreted by refinements via the technique of categorical logic [15]. The second order formulas provable in the intuitionistic second order logic are provably true in ATTT under this interpretation. Conversely, the first order formulas provably true in ATTT under this interpretation are provable in the intuitionistic first order logic. We conjecture that this holds for the second order case as well and will give a sufficient condition.

The logic of refinements mentioned above is "non-informative" in the sense of Coq system [3] or "rank zero" in the sense of PX system [8]. Thus, it is a substitute of logics of Coq's propositions and PX's rank zero formulas. This means that the interpretation of formulas does not keep any computational information contrary to the ordinary BHK-interpretation (Brouwer-Heyting-Kolmogorov-interpretation) of constructive logic. Besides this non-informative interpretation, some BHK-style interpretations of logic can be coded in ATTT. They are essentially realizability interpretations. We will show that both of recursive realizability and modified realizability are represented naturally in ATTT. Then, we will show that informative induction principle is derivable by non-informative induction principle in ATTT.

We also introduce a notion of refinement classifier resembling subobject classifier of topos, and, by means of this notion, we will compare our approach to program/proof development in ATTT with related works [2], [3], [16], [20].

2 Refinement types

In this section, we will briefly review the notion of refinement types and develop cHa-structures of the refinements. The notion of refinement type was first introduced in [5]. We independently introduced a

*hayashi@rins.st.ryukoku.ac.jp

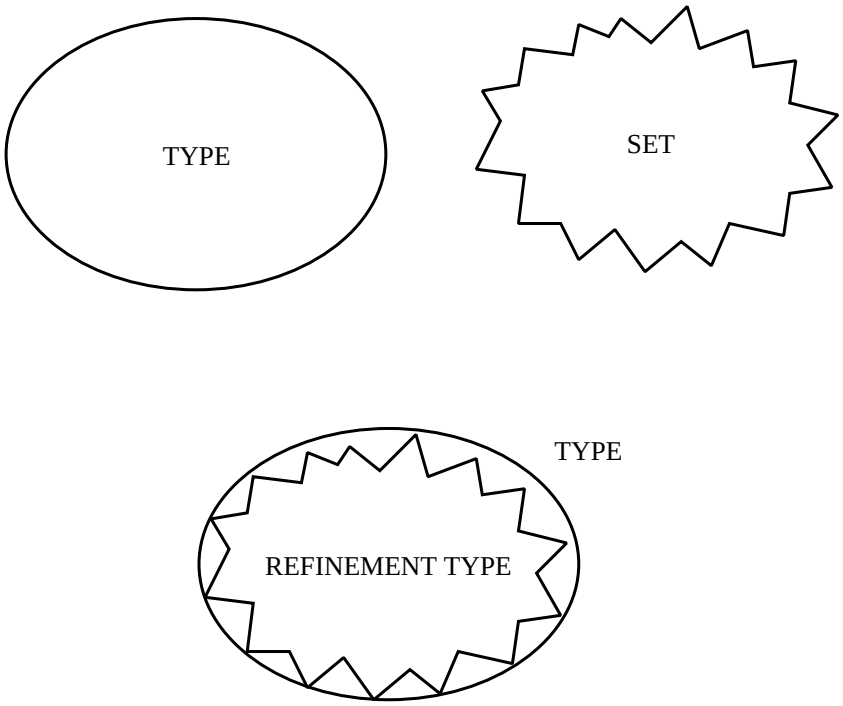

Figure 1: Type, Set and Refinement Type

very similar notion for a rather different aim [9].[1]

2.1 Types versus refinement types

A type is a collection of data which have the same uniform structure. A set is a collection of data which have the same property. Property which defines a set can be so complicated that we cannot effectively decide if a data belongs to the set. On the other hand, types of programming languages are supposed to be simple. For examples, integers, elements of free algebras, records and functions are all types. But, the primes are not a type. It is a *set*. We summarize this intuition by the following "doctrine": the border of types are smooth and the border of sets are rugged.[2] "The function f returns the next prime number to its real argument" is a specification of a function f. The boundary of the collection of data satisfying this specification is rugged, as it uses the primes in it.

Intuitively, specifications must be identified with sets. Thus, we think that the specifications should be distinguished from data types, even in the type theories for program extraction based on Curry-Howard isomorphism. This consideration led us to introduce a new kind of "types" called *refinement types*. Refinement types have rugged boundaries and are intended to serve as specifications. This refinement types refine the boundaries of types, i.e., they distinguish elements more refinedly than types. More precisely, a refinement types of type A is intended to be a *subset* of the ordinary type A. Note that a refinement type, or refinement in short, is not a subtype of A, since it may have a rugged boundary.

ATTT is a type system with refinement types designed as "ATTT = $\lambda 2$ + refinement types." In a sense , ATTT is "$\lambda 2$ + logic." In set theory, sets are defined by the aid of logical formulas. We do not introduce formulas but directly define set (refinements) by means of the rules for refinements in the framework of type theory. In categorical/algebraic logic, logics are coded through sets (subobject),

[1] As our framework looked very similar to Freeman and Pfenning, we call our notion by the same name. The exact relationship is still unknown, but it seems fair to say that these are essentially the same notion.

[2] I learnt this doctrine from Rod Burstall. According him, it's due to JÄRobinson.

e.g. the statement "$F(x)$ implies $G(x) \vee H(x)$" can be interpreted as set-theoretic inclusion $\{x|F(x)\} \subset \{x|G(x)\} \cup \{x|H(x)\}$. Thus we may think that the set part (refinement part) of ATTT is a kind of logic *without* formulas. This resembles the philosophy of Martin-Löf's type theory (at least his philosophy in 80's). The difference is that Martin-Löf used Curry-Howard isomorphism to represent logic but we use the approach via categorical/algebraic logic. Note that we have *not* abandoned Martin-Löf's philosophy. The categorical/algebraic logic approach is used only for non-informative part. For informative part, we follow Martin-Löf's philosophy. In a sense, we use Curry-Howard isomorphism, when we sit in outside Martin-Löf's subset type, and use categorical/algebraic logic approach, when we sit in inside Martin-Löf's subset type.

More formally, ATTT is described as follows. The types and terms of ATTT are exactly the ones of $\lambda 2$. If A is a type of $\lambda 2$ (and so of ATTT), we introduce the refinement kind of A, which we will write $\mathit{refine}(A)$. A refinement kind is a sort of "kind" in the sense of GTS. The elements of the refinement kind $\mathit{refine}(A)$ is intended to be the refinements of the type A. Each refinement kind $\mathit{refine}(A)$ is closed under the formation rules of singleton $\{a\}_A$ ($a \in A$), and unions $R_1 \vee R_2$, $\bigvee x \in A.R$ and intersections $R_1 \wedge R_2$, $\bigwedge x \in A.R$. Furthermore, if R_1 is in $\mathit{refine}(A)$ and R_2 is in $\mathit{refine}(B)$, then $R_1 \to R_2$ is in $\mathit{refine}(A \to B)$. This refinement type of functions represents the constructive implication. Natural introduction and elimination rules for these refinements are included. A subtle point is the introduction rule for the refinement $R_1 \to R_2$. If $e \in R_2$ for $x \in R_1$, then the term $\lambda x \in A.e$ is introduced in $R_1 \to R_2$, and not $\lambda x \in R_1.e$.[3] The description of ATTT above would be enough to understand the development below. Thus we will not explain the details. See [9] for more details.

2.2 Refinement application

For the discussions below, we will introduce a notion *refinement application*. A future of ATTT is that the realizability relation "a realizes F" is not only a judgment but also a refinement type. The relation is expressed as

$$F \wedge \{a\}.$$

This refinement is realized by a, if and only if, a belongs to F. If we consider a refinement of A as a predicate on A as we will do later, $F \wedge \{a\}$ is considered the application of F to a. So we call this refinement application and write $F\{a\}$.

Note that Martin-Löf's propositional equality $I(F, a, a)$ is a proposition which expresses realizability relation. The refinement application resemble this. The difference is $F\{a\}$ is realized by a, and $I(F, a, a)$ is realized by a fixed constant.

2.3 The algebra of refinements

The refinement kind $\mathit{refine}(A)$ is intended to be the subsets of the type A. So it has a lattice theoretic structure. $\bigwedge$ and $\bigvee$ are "internal" infimum and supremum, respectively. Actually, they are an "internally" complete Heyting algebra. Namely, they are distributed "internally" complete lattice. To show it, we define an pre-order relation on $\mathit{refine}(A)$. The relation itself is a refinements of $\mathit{refine}(A \to A)$.

Definition 1 (subset refinement) Let R_1 and R_2 be refinements of a type A. The *subset refinement* $R_1 \leq R_2$ is defined as follows:

$$R_1 \leq R_2 \stackrel{\text{def}}{=} \bigwedge x \in R_1.(R_1\{x\} \to R_2\{x\}).$$

The subset refinement is a refinement of type $A \to A$. Intuitively, the subset refinement says that R_1 is a subset of R_2. For this refinement the following proposition holds.

Proposition 1 *Let $A \in \mathit{Type}$, $R_1 \in \mathit{refine}(A)$ and $R_2 \in \mathit{refine}(A)$ be derivable under a context Γ. Then, the following are equivalent:*

[3] This rule is justified by the observation that terms of ATTT are exactly those of $\lambda 2$ and the former lambda term is a term of $\lambda 2$ but the latter is not. This rule also reflects modified realizabilities. Recently, Pfenning introduced a type theory with intersection types which has a function-introduction rule similar to ours for a different good reason [19].

1. $\Gamma, x \in R_1 \vdash x \in R_2$ *is derivable.*

2. $\Gamma \vdash Id_A \in R_1 \leq R_2$ *is derivable.*

3. $\Gamma \vdash e \in R_1 \leq R_2$ *is derivable for some term* e*.*

Id_A is the identical function $\lambda x \in A.x$. Note that the second condition shows that the identity function is the standard witness of the refinement. In this sense, the subset refinement is self-realizing or non-informative.

It is easy to see the subset refinement defines an pre-order relation on $refine(A)$. Note that this fact can be stated *internally*. Namely, the followings are derivable in an appropriate context:

1. $\bigwedge R \in refine(A).R \leq R$,

2. $\bigwedge R_1 \in refine(A).\bigwedge R_2 \in refine(A).\ \bigwedge R_3 \in refine(A).R_1 \leq R_2 \supset R_2 \leq R_3 \supset R_1 \leq R_3$.

Note that "$\supset$" is the ordinary implication defined by the function space. Since the subset refinement is self-realizing, the implication may be replaced by non-informative implication explained later.

Since the subset relation is a pre-order, we have to consider the "equivalence relation" induced from the pre-order. This can be also internalized. Let's call such an equivalence relation *extensional equality refinement*, $R_1 \geq\!\leq R_2$ in notation. It's defined by

$$R_1 \geq\!\leq R_2 \stackrel{\text{def}}{=} (R_1 \leq R_2) \wedge (R_2 \leq R_1).$$

2.4 The refinements are internally complete Heyting algebra

The finite intersection and union are infimum and supremum, respectively. These facts can be stated internally as well as externally. As we have big intersections and unions, the refinements of a type are *internally* complete. For example, we can derive the followings for the big intersection:

$$\ldots, i \in I \vdash Id_A \in (\bigwedge i \in I.R) \leq R$$

$$\frac{\ldots, X \in refine(A), i \in I \vdash e \in X \leq R}{\ldots, X \in refine(A) \vdash X \leq \bigwedge x \in A.R}(i \text{ is not free in } e)$$

Note that we have to be careful with what kind of indexes $i \in I$ are allowed. It depends on what kind of formation rules for the intersection and union are allowed. But, once a formation rule is introduced, there is no difficulty to prove the properties above.

A complete Heyting algebra is a complete lattice with the following distribution law:

$$a \wedge \bigvee i \in I.b_i = \bigvee i \in I.(a \wedge b_i).$$

The distribution law is provable in ATTT.

Lemma 1 (the distribution lemma) *Assume i is not free in R. In the contexts where R and S are refinements of a type A, the following refinement is provably inhabited.*

$$R \wedge \bigvee i \in I.S \geq\!\leq \bigvee i \in I.(R \wedge S).$$

Note that I must be an index set (in the context) for which the formation of the big intersection allowed.

This lemma is not quite trivial. The $\geq$-direction is trivial, but the $\leq$-direction is not. In some type theories with intersection and union types, the distribution law is included as an axiom, as the $\leq$-direction is not derivable from the other axioms and rules. In ATTT, the $\leq$-direction is easily derivable by the following lemma:

Lemma 2 *The following are derived rules of ATTT:*

1.

$$\frac{\Gamma \vdash e_0 \in \bigvee x \in A.B}{\Gamma \vdash e_0 \in \bigvee x \in A.(B \wedge \{e_0\})}$$

2.

$$\frac{\Gamma \vdash e_0 \in \bigvee x \in A.B \qquad \Gamma, x \in A, y \in B \wedge \{e_0\} \vdash e_1 \in C}{\Gamma \vdash e_1[y := e_0] \in C[y := e_0]},$$

where x is not in $FV(e_1) \cup FV(C)$.

The second one is a consequence of the first one.

2.5 implication refinement

In cHa, implication $a \supset b$ is defined as the greatest element c such that $a \wedge c \leq b$ holds. By the same vain, we define the *implication refinement*.

Definition 2 (implication refinement) Let R_1 and R_2 be refinements of a type A. Then we define the *implication refinement* $R_1 \mathbin{\check{\supset}} R_2$ by

$$\bigvee R \in \mathit{refine}(A).\bigvee w \in R_1 \wedge R \leq R_2.R.$$

Note that the implication refinement is of type A and the subset refinement is of type $A \to A$. For the implication refinement, the following two are equivalent:

$$\Gamma, a \in R_1 \vdash a \in R_2,$$

$$\Gamma, a \in A \vdash a \in R_1 \mathbin{\check{\supset}} R_2.$$

This implies the equivalence of the followings:

$$\Gamma, a \in R_1\{a\} \vdash a \in R_2\{a\},$$

$$\Gamma, a \in A \vdash a \in (R_1 \mathbin{\check{\supset}} R_2)\{a\}.$$

By the commutativity, we can prove that $(R_1 \mathbin{\check{\supset}} R_2)\{a\}$ and $R_1\{a\} \mathbin{\check{\supset}} R_2\{a\}$ are extensionally equal refinements. If we abuse the notation, we have

$$\Gamma, R_1\{a\} \vdash R_2\{a\},$$

$$\Gamma, a \in A \vdash R_1\{a\} \mathbin{\check{\supset}} R_2\{a\}.$$

This shows that the implication refinement satisfies the standard rules for implication.

2.6 Absurdity refinement and truth refinement

The absurdity refinement of a type A, $\bot_A$, is the least refinement and the negation refinement is defined from it and the implication.

$$\bot_A \stackrel{\text{def}}{=} \bigwedge R \in \mathit{refine}(A).R,$$

$$\check{\neg} R \stackrel{\text{def}}{=} R \mathbin{\check{\supset}} \bot_A.$$

Obviously, the absurdity refinement implies any refinement of the same type. The truth refinement $\top_A$, which is the greatest refinement is also definable by

$$\top_A \stackrel{\text{def}}{=} \bigvee R \in \mathit{refine}(A).R.$$

Normally, Heyting algebra is assumed to have at least two elements. But, we do not assume it in this paper. As a type may be empty in a semantics for $\lambda 2$, we cannot prove the truth refinement is not the same as the absurdity refinement.

Note that $\bot$ is not unique in ATTT. We have the absurdity $\bot$ for each refinement kind. Later, we will show the refinement of the unit type *unit* classifies the refinements. Thus, $\bot_{unit}$ is the most general absurdity in a sense. If $x \in A$ holds, then $R\{x\}$ for any refinement R of A under the assumption $\bot_{unit}$.

Proposition 2 *The following is derivable in ATTT:*

$$A \in Type, x \in A, R \in refine(A), w \in \bot_{unit} \vdash A \leq R.$$

This means that any refinement is true, if we assume $\bot_{unit}$.

3 Interpreting logic by refinements

In this section, we give an interpretation of second order logic by refinements. A refinement kind of a type A is the power set of A. Thus, refinement kinds resemble the algebras of subobjects in a category. We give an interpretation of second order predicate logic in ATTT using the technique of categorical logic by Makkai and Reyes [15]. But, our interpretation is not completely the same as theirs. Makkai and Reyes considered first order logic. We will consider second order logic. They used external intersections and unions, but we will use internal ones. Furthermore, products are assumed in [15]. We define product by the polymorphic product (see [9]). The polymorphic products are not real products but semi-products in the sense of [7] and [10], since $\lambda 2$-part of ATTT does not have η-conversion.[4]

3.1 The Makkai-Reyes-style interpretation

In spite of the differences mentioned above, Makkai-Reyes technique works well in our case by small modifications. Let's illustrate the interpretation briefly. Let D be a fixed type. This is intended to be the domain of first order objects (ground type). Let F be second order formulas whose *first order* free variables are among $x_1, \ldots, x_n$. Their interpretations are refinements of the type D^n (n-times product of D). F may have second order free variables. We assign refinements with appropriate types to the variables and the interpretation of F is defined depending on this assignment.

3.1.1 Interpretation of predicates

Let's see how to interpret predicates by refinements. Since refinements are sets, they represent unary predicates. Thus, we use the polymorphic products and paring to interpret n-ary predicates. For example, a binary predicate $P(x, y)$ is interpreted as a refinement R of the polymorphic product $D \times D$. More precisely, we assign a refinement R of the type $D \times D$ to P, and interpret the formula $P(x, y)$ by

$$\{\langle x, y\rangle \in D \times D | R\{\langle x, y\rangle\}\}.$$

The pair $\langle x, y\rangle$ is the polymorphic Church pairing and the subset notation is an abbreviation of

$$\bigvee x \in D. \bigvee y \in D. \bigvee w \in R\{\langle x, y\rangle\}. \{\langle x, y\rangle\}_{D \times D}.$$

See [9] for details.

3.1.2 Interpretation of logical operators

Conjunction, disjunction, implication and negation of formulas are interpreted as intersection, union, implication and negation of refinements. Note that implication is interpreted by implication refinement defined above. It is straightforward, but there is a problem.

[4] Note that we can define a refinement $\bigvee a \in A. \bigvee b \in B. \{\langle a, b\rangle\}$ of a polymorphic product $A \times B$, which is the "real" product. But, this trick would be incorrect, as a product of types should be a type.

If F and G have different free variables, this interpretation does not work, since their interpretations are of refinements of different types. Then, we embed refinements into a bigger refinements. Assume we are to form the union of the interpretations of F and G, say R_F and R_G. If F has a free variable x_1 as $F(x_1)$ and G has another free variable x_2 as $G(x_1, x_2)$. Then, $R_F \in \mathit{refine}(D)$ and $R_G \in \mathit{refine}(D \times D)$. Thus, we must embed the interpretation of $F(x_1)$ into $D \times D$ as $\{\langle x_1, x_2\rangle \in D \times D | R_F\{x_1\}\}$.

Next we interpret the universal and existential quantifiers. This is the point at which our interpretation diverges from Makkai-Reyes interpretation. Makkai and Reyes interpret first order quantifiers by images and co-images of subobject morphisms. We interpret them by the big intersection and union over the type D. Let $F(x_1, \ldots, x_n)$ be a formula and let R_F be its interpretation, which is a refinement of D^n. The interpretation of the formula $\forall x_i.F(x_1, \ldots, x_n)$ is the refinement

$$\bigwedge x_i \in D.\{\langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\rangle \in D^{n-1} | R\{\langle x_1, \ldots, x_n\rangle\}\}.$$

First order existential quantifier is interpreted in the similar way by union.

Since we have unions and intersections over refinement kinds, second order quantifiers are interpreted in the same way. For example, let P be a binary predicate variable. Then $\mathit{forallP}.F$ is interpreted by $\bigwedge P \in \mathit{refine}(D^2).R_F$. Note that P is a refinement of the polymorphic product D^2, which may have more elements than the real pairs. Thus, P may contain elements which are not pairs. But, this is not a problem, since P always appears in the form $P\{\langle s, t\rangle\}$ in F. This completes the definition of the interpretation of logic by refinements.

3.2 Provably true formulas in ATTT

A refinement R of a type A is said to be true, if and only if $Id_A \in \top_A \leq R$ holds. [5] If this judgment is provable in ATTT, we will say the refinement R is provably true in ATTT. If the interpretation of a formula F is provably true in ATTT, then we say the formula is provably true in ATTT.

3.2.1 Intuitionistic second order logic are provably true

Then, the following is easily proved:

Proposition 3 *The intuitionistic second order predicate logic (ISOL) is sound w.r.t. the interpretation. Namely, if a closed formula F of ISOL is provable, then it is provably true in ATTT under the context $D \in \mathit{Type}, x \in D$, where the type variable D is used as the domain of first order objects.*

The intuitionistic predicate logic assumes that the domain is non-empty. Thus, the assumption $x \in D$ is necessary. If we use a free logic instead, this is not necessary.

3.2.2 What are provably true formulas?

It is natural to ask what are provably true formulas in ATTT. We conjecture that provably true formulas in ATTT are exactly the formulas provable in ISOL. This means that our interpretation is adequate to ISOL and amounts to the following:

Conjecture 1 *The converse of proposition 3 holds. Namely, if a closed formula F of ISOL is provably true in ATTT under the context mentioned in proposition 3, then F is provable in ISOL.*

Somehow related result is in [14]. Although, we have not been able to prove the conjecture yet, it is easy to prove the conjecture restricted to first order formulas. Namely, the following proposition holds:

Proposition 4 *If a closed* first order *formula ϕ is provably true in ATTT under the context mentioned in the proposition above, then ϕ is provable in the intuitionistic* first order *predicate logic (IFOL).*

[5] More exactly speaking, we have to respect in which context the refinement is true.

The idea of our proof is very simple. But, full details are rather clumsy. Thus, we illustrate only the idea, below.

Our proof consists three steps. First, we show that any provably true first order formula ϕ is valid in the sense of Tarski semantics by means of a term model of $\lambda 2$. Secondly, we formalize the proof of this fact in the intuitionistic second order arithmetic (HAS) augmented with an auxiliary sort γ. The auxiliary sort is intended as the generic domain, i.e., we do not pose any axiom or rules for this except the ones of ISOL. Let us call this system HAS(γ). Then, we can show that ϕ^γ, that is ϕ made the sort to be γ, is provable in HAS(γ). Lastly, we show that if ϕ^γ is provable in HAS(γ) then ϕ is provable in IFOL. Thus, ϕ is provable in IFOL.

The first step is easy. If the judgment "the interpretation is true" is provable in ATTT under the context, we can interpret it in the set-theoretical semantics introduced in [9]. Let *Const* be a arbitrary set of new constants. We augment $\lambda 2$ by *Const* as a new type. Namely, we introduce a new type constant D and regard the constants from *Const* as new constants of D. Let denote the augmented system by $\lambda 2(D)$. Let $\mathbf{M}$ be the *closed* term model of $\lambda 2(D)$.

In [9], we took BMM-model as the basis of semantics of ATTT. But, $\mathbf{M}$ is not a BMM-model but a $\lambda 2$-algebra in the sense of [11]. This difference is not so important, since any concrete $\lambda 2$-algebra also gives a semantics of ATTT in the sense of [9]. The reason why we considered only BMM-model in [9] was only for simplicity.

The model $\mathbf{M}$ consists of the collection of the set of closed normal terms of $\lambda 2(D)$. We denote the set of closed normal terms of type A by $\mathbf{M}(A)$. By interpreting $\mathit{refine}(A)$ as the power set of the closed normal terms of type A, it becomes a model of ATTT (see [9]). Assume that a first order formula ϕ is provably true in ATTT, interpreting D is the domain of the first order objects. Since the normal closed terms of D are only the new constants, $\mathbf{M}(D)$ is identical to *Const*. Let $x_1, \ldots, x_n$ be the free variables of ϕ. Then, it is easy to check that R_ϕ (the interpretation of ϕ) coincides with the set of the list of constants $\langle c_1, \ldots, c_n \rangle$ where $\phi[c_1/x_1, \ldots, c_n/x_n]$ is valid in the sense of Tarski semantics whose domain is *Const*. Since *Const* is arbitrary, this means that ϕ is valid in any frame (a tautology) in the sense of Tarski semantics. Thus, ϕ is provable in the classical first order logic by the completeness theorem. Note that the argument above is all constructive except the completeness theorem.

The second step involves messy formalization.[6] Let us clarify the system HAS(γ) in which we do the formalization. HAS(γ) is a second order theory with two sorts, ν and γ. The sort ν is intended to be the sort of natural numbers and the sort γ is intended to be arbitrary (generic). The constants of ν is only 0, and γ does not have any constants. The only function symbol is s (successor) for ν. The arity of a predicate variable is a list $(s_1, \ldots, s_n)$ of sorts (n is possibly zero). Thus we can talk about functions from ν to γ, etc. The equality for each sort is defined by Leibniz equality. (This is only for simplicity. We may have equality symbols, instead.) The logic is the two-sorted intuitionistic second order logic, and the axioms are the Peano axiom for the sort of natural numbers ν.

Now, we formalize the argument of the first step in HAS(γ). Note that only "pointwise" formalization is possible, since we need normalization theorem of $\lambda 2$. Namely, we formalize the proof for each ϕ. First, we augment ATTT with D just as $\lambda 2(D)$. Let us denote the augmented system by ATTT(D). Assume that a first order formula ϕ is provably true in ATTT. Then it is also provably true in ATTT(D), regarding the new type D as the domain of the first order objects. Then there is a fragment of $\lambda 2(D)$ for which the strong normalization property of the terms is provable in HAS(γ) and all terms in the proof of the truth of ϕ fall in the fragment (see, e.g., [21]). Thus, we have a closed term model of the fragment of ATTT(D) formalized in HAS(γ). Since the set of constants *Const* is arbitrary, we may regard it as the sort γ. Recall that the Makkai-Reyes-style interpretation coincided with the Tarski semantics. Thus, we can conclude that ϕ^γ is provable in HAS(γ).

The third step consists of the following lemma:

Lemma 3 *A first order formula ϕ is provable in intuitionistic first order logic, if ϕ^γ is provable in HAS(γ).*

[6] It is possible to avoid the formalization by means of Kripke semantics. Then, we have to introduce a notion of Kripke semantics for $\lambda 2$ as did by Mitchell and Moggi and extend any first order Kripke frame to a Kripke frame of $\lambda 2$ by "free construction." This is not easier than the formalization.

This is an easy consequence of an infinitary normalization theorem of intuitionistic second order arithmetic, which is proved by Tait-Girard computability predicate technique, e.g., [6]. A proof of ϕ^γ of HAS(γ) can be normalized using ω-rule. Since ϕ^γ is a first order formula only with the sort γ, subformula property holds, i.e., the sort appearing in the normal proof is only γ. This means that the proof is of intuitionistic first order logic, regarding the sort γ as the ground sort. This completes the proof.

The only third step fails for the second order case. Since second order logic does not have subformula property, our proof fails to prove the lemma above for second order case. But, we conjecture that the lemma will hold for second order formulas as well.

Conjecture 2 *A formula ϕ of second order predicate logic is provable in intuitionistic second order predicate logic, iff ϕ^γ is provable in HAS(γ).*

Note that this lemma has an intrinsic meaning: if a second order formula is constructively valid under the assumption of the existence of the natural numbers, then it is provable in second order intuitionistic logic. Since this seems very plausible, the conjecture must be true.

Remark 1 The reduction to the original conjecture to the conjecture on HAS(γ) seem unnatural, as the original conjecture does not involve natural numbers. But, if the conjecture on HAS(γ) is once proved, then it will be straightforward to change it to a proof of the original conjecture via Kripke semantics of $\lambda 2$. In such a proof, arithmetic part and even formalization part will vanish. Actually, it is possible to give a sufficient condition by means of Kripke frames of $\lambda 2$. But, the sufficient condition by means of HAS(γ) is easier to understand and has an intrinsic meaning as noted above.

3.3 The excluded middle

The law of excluded middle for refinements

$$\Pi A \in \mathit{Type}. \bigwedge R \in \mathit{refine}(A).(\top_A \leq R \vee \tilde{\neg} R)$$

or

$$\Pi A \in \mathit{Type}. \bigwedge R \in \mathit{refine}(A).(\neg\neg R \leq R).$$

is not derivable in ATTT by the the adequacy result above. Thus, the Heyting algebra of a refinement kind is not boolean. But, they are compatible with boolean laws. Even if we add the law of excluded middle above as an axiom, the system is still conservative over $\lambda 2$. In [9], we defined a translation from ATTT to $\lambda 2$ which is identical on the $\lambda 2$ fragment of ATTT. We may add the excluded middle above as an axiom, whose realizer is the identity function. Then the two forms of the classical axioms above are both translated to $\mathit{Id}_A \in A \to A$. This resembles the logic of PX system, which is compatible with classical logic as stressed in [8].

3.4 The refinement classifier and comparisons to the other approaches

The Makkai-Reyes-style interpretation given above look rather different from the standard way of interpreting logic in higher order type systems as Coq system. Thus, it seems difficult to compare our approach to the standard ones. But, we can encode logic similarly to the standard way by the notion of refinement classifier. Upon the idea, we can compare our approach to the others.

3.4.1 The refinement classifier

Refinements resembles subobjects in toposes. A topos has a subobject classifiers by which subobjects are classified. Namely, a subobject of an object is represented by a map from the object to the subobject classifier. Intuitively, a subobject classifier is the power set of a singleton set. It can be considered as the set of truth values. Introducing a singleton type, say *unit*, consisting the only element $\star$, the refinement kind $\mathit{refine}(\mathit{unit})$ is a refinement classifier. Note that this is the kind of non-informative propositions, and so corresponds to *Prop* in the sense of Coq system rather than the *Prop* in the sense of the original Calculus of Constructions.

We will write the refinement kind $refine(unit)$ by Ω and call it the *refinement classifier*. Assume that $R \in refine(A)$. Then we define a term of the type $A \rightarrow \Omega$. Then the following correspondence between $A \rightarrow \Omega$ and $refine(A)$ is obtained:

$$\begin{aligned} R \in refine(A) &\longmapsto \lambda x \in A. \bigvee w \in R\{x\}.\{\star\}, \\ \phi \in A \rightarrow \Omega &\longmapsto \bigvee x \in A. \bigvee w \in \phi(x)\{\star\}.\{x\}. \end{aligned}$$

But, $A \rightarrow \Omega$ is illegal in ATTT in [9]. Thus, we have to extend ATTT to include such a type by allowing the following formation rule and corresponding introduction rule.

$$\frac{\Gamma \vdash A \in Type \qquad \Gamma \vdash B \in RKind}{\Gamma \vdash A \rightarrow B \in RKind}$$

This extension is not essential. In such a extension, any term of $RKind$ has the form $A_1 \rightarrow \ldots A_n \rightarrow refine(B)$, where $A_1, \ldots, A_n$ are types. Then, we can regard it as $refine(A_1 \times \ldots \times A_n \times B)$, by regarding $\lambda x_1 \in A_1. \cdots \lambda x_n \in A_n.R$ as $\{\langle x_1, \ldots x_n, y\rangle \in A_1 \times \cdots A_n \times B | R\{y\}\}$. Note that this is a standard technique to encode functions from first order objects to predicates in the second order logic. Thus, the extension still keep the second order character.

3.4.2 Comparisons to the other approach

Upon the notion of refinement classifier, we compare our approach to the others. In impredicative higher order type theories as Calculus of Constructions, *Type* (or often denoted as *Prop*) is often used as a substitute for the collection of the truth values. It is also regarded as the collection of the data types. This confusion leads to inefficiency of the extracted code and makes it difficult to understand encoded logic. Thus, Coq system separate these two notions by introducing two kinds *Prop* and *Set*. The kinds *Prop* and *Set* belong to the sorts (in the sense of GTS) *Type* and *Type_Set*, respectively. Note the messy confusion of symbols. *Prop* of Coq corresponds to our Ω and *Set* of Coq corresponds to our *Type*, the kind of the types. *Type* of ATTT belongs to the sort *Kind* and refinement kinds $refine(A)$ belong to the sort *RKind*. Thus, *Type* of Coq is our *Kind* and *Type_Set* is our *RKind*.

The superficial divergences of our approach from Coq's are

1. . ATTT is a second order system, but Coq is a higher order system.
2. . The types of Coq can depend on data, as it is based on Calculus of Construction. The types of ATTT cannot depend on data, as it is based on $\lambda 2$.

There is still another twist. Coq has an extraction algorithm, by which the data-dependency of types (and other logical information) is stripped out so that the extracted codes are of Girard's F_ω. At the specification level, Coq has data-dependent types, but, at the object code level, it does not. The types of ATTT are the ones of $\lambda 2$ at the both levels. In ATTT, we directly talk about object level codes and their types, i.e. the terms and types of $\lambda 2$. On the other hand, the users of Coq talk about them indirectly through the extraction procedure. This is a fate of the systems based on realizability as Coq and PX. This might not be a bad doom, as the indirectness can make users think their programs abstractly and good extraction procedure can sometime produce better codes than the users expect. This contrast between direct and indirect approaches may be compared with the contrast between assembler languages and compiler languages.

Another approach closely related to Coq is "deliverable" by Burstall and McKinna ([2], [16]). They take a direct approach. They do not consider any extraction algorithm, but directly talk about codes. To represent data types and talk about them and codes, they use Luo's Extended Calculus of Constructions ECC ([12], [13]). ECC has *Prop* and a predicative cumulative hierarchy of kinds $Type(0)$, $Type(1), \ldots$. They use *Prop* as the truth values and $Type(i)$ as data types. Thus, *Prop* corresponds to our Ω. But, there is a difference. *Prop* is an element and also a subset of $Type(0)$. This implies that the collection of truth values and each truth value is a data type. It would be possible to regard the collection of truth values as a data type by regarding them as formulas. But, a proposition (truth value) is a refinement of

a unit type from our point of view. Thus, it has a rugged border, and so it is not a data type. When it comes to data-dependency of types, deliverable approach allows it. The predicative hierarchy allows such types.

The approach most closed to ours is Erik Poll's programming logic $\lambda\omega_L$ [20]. As programming language, he uses Girard's F_ω, $\lambda\omega$ in his notation. As logic, he uses the other copy of $\lambda\omega$. His $*_s$ is *Type* and $*_p$ is Ω of ATTT. These two copies of the same type theory resembles two copies of Calculus of Constructions in the Coq system. Then he introduce function types from a data type A to $*_p$ to represent predicates. This corresponds to the refinement kind $\mathit{refine}(A)$ or $A \to \Omega$. The data-dependency of types is not allowed as he uses F_ω. In a sense, $\lambda\omega_L$ is a higher order version of ATTT without refinements. The difference is that Poll uses predicates, on the other hand, we use sets (refinement types). Since propositions and data types, and so programs and proofs, are more clearly separated in $\lambda\omega_L$ than ECC, $\lambda\omega_L$ might be a better framework for the deliverable approach.

As illustrated above, the four approaches are closely related in rather intricate ways. They do not seem very different from a theoretical point of view. The difference in practice of proof/program developments should be investigated.

4 Informative interpretations of logic

The logic of refinements given above is a non-informative interpretation. In this section, we examine two informative interpretations in ATTT.

4.1 Two informative interpretations of implication

Type theories based on "proofs as program" notion can be regarded as axiomatizations of realizability notions. For example, Martin-Löf's type theory can be regarded as an axiomatization of extensional recursive realizability as exploited in [1].

In PX system, realizers of non-informative formulas are always the empty list which are passed by extracted codes. In Coq system, realizers of non-informative formulas are the null sequence which literally disappear in extracted codes. These two treatments of non-informativeness reflect the different realizability interpretations used by those systems. PX uses recursive realizability and Coq uses modified realizability.

The difference of these realizabilities appears in the interpretation of implication. Let P be a non-informative formula and let A be an informative formula. In recursive realizability,

$$r \text{ realizes } P \supset A \stackrel{\text{def}}{=} r \downarrow \wedge P \supset r(\mathit{nil}) \text{ realizes } A.$$

In modified realizability,

$$r \text{ realizes } P \supset A \stackrel{\text{def}}{=} r \downarrow \wedge P \supset r \text{ realizes } A.$$

(Normally, only terminating terms are used in modified realizability. Then, the condition $r \downarrow$ may be omitted.) In ATTT, user can choose both interpretations locally. Let R_1 and R_2 be refinements. The standard intuitionistic implication is defined by $R_1 \to R_2$. This interpretation takes R_1 informative.

The following interpretations take R_1 non-informative:

1. $\bigwedge w \in R_1.R_2$,
2. $\{x \in \mathit{unit} | R_1\} \to R_2$,

w is a fresh variable which does not appear in R_2. The first one represents the implication of the modified realizability, and the second one represents the implication of the recursive realizability. In this sense, ATTT embodies both realizabilities in a single framework.

4.2 Induction principles

Induction principle is a very important component of a system of constructive programming, since it is the device to derive recursive programs. Below, we will examine two kinds of induction principles, one is non-informative and the other is informative.

The non-informative one is purely logical. The informative one is not only logical but also representing recursion scheme as well. For example, informative mathematical induction represents primitive recursion and its correctness. We will show that informative induction principle can be derived from non-informative induction principle in ATTT. This is a future which distinguishes ATTT from the other type theories.

4.3 Monotone inductive definition of refinements

First, we will examine non-informative induction principles. They are induction principles over refinements. The monotone inductive definition of refinements are easily defined as in the second order logic. Let R be a refinement variable of type A and let $\Psi(R)$ be a refinement of A. Then, $\mu R.\Psi(R)$ is

$$\bigwedge S \in \mathit{refine}(A).\bigwedge w \in (\Psi(S) \leq S).S.$$

If $\Psi(S) \leq S$ holds, then $\mu R.\Psi(R) \leq S$ obviously holds. If Φ is monotone, i.e., $R_1 \leq R_2 \supset \Phi(R_1) \leq \Phi(R_2)$, holds, then $\Phi(\mu R.\Psi(R)) \geq\leq \mu R.\Psi(R)$ holds. In this sense, $\mu R.\Psi(R)$ is the least fixed point of a monotone operator $\Phi(R)$.

4.4 The type of natural numbers

Recursive data types as integers, list, etc. are definable in $\lambda 2$. For example, natural numbers are definable as Church integers. But, we do not use this as data types. We should introduce such types as primitive type as in [3]. For simplicity, we consider only Nat, here.

To extend ATTT by Nat, we introduce a type constant Nat and two constants $0 \in \mathit{Nat}$ and $S \in \mathit{Nat} \to \mathit{Nat}$. Then, we add the primitive recursor rec

$$\mathit{rec} \in \Pi X \in \mathit{Type}.X \to (\mathit{Nat} \to X \to X) \to \mathit{Nat} \to X$$

together with the standard reductions.

To say Nat is built by 0 and S, we introduce the following axiom:

$$\mathit{Nat} \ \leq \mu R.(\{0\}_{\mathit{Nat}} \ \vee S''R),$$

where $S''R$ is the image of R by S, which is defined by

$$f''R \stackrel{\text{def}}{=} \bigvee x \in R.\{f(x)\}.$$

Since Nat is not a refinement, "$\mathit{Nat} \ \leq \mu R\ldots$" is not the order on refinements. This is defined by $\bigwedge x \in \mathit{Nat}.\{x\}_{\mathit{Nat}} \to (\mu R\ldots.)\{x\}$.

As the axiom is self-realizing by the identity function, the axiom may be put in the context with a variable, which is a virtual witness of the axiom.

Note that we did not introduce the informative mathematical induction. The last axiom is a kind of mathematical induction, but it is non-informative mathematical induction (NIMI). The informative mathematical induction (IMI) should be

$$\Pi A \in \mathit{Type}.\bigwedge P \in \mathit{Nat} \to \Omega.P(0) \to \Pi n \in \mathit{Nat}.(P(n) \to P(Sn)) \to \Pi n \in \mathit{Nat}.P(n).$$

In ATTT, IMI is provable from the axioms above.

Theorem 1 *The following is derivable in ATTT:*

$$\lambda A \in \mathit{Type}.\lambda a \in \mathit{Nat}.\lambda f \in \mathit{Nat} \to A \to A.\lambda n \in \mathit{Nat}.\mathit{rec}Aafn \in \mathit{IMI}.$$

Note that we have to put the abstractions, since ATTT does not have η-conversion. Note that this theorem asserts that the λ-term above is the realizer of IMI. Let $e(n)$ be the term $recAafn$. It is enough to prove $e(n) \in P(n)$ under the assumptions. To prove this, it is sufficient to prove the refinement $P(n)\{e(n)\}$. We can prove this by the non-informative mathematical induction NIMI. The proof is essentially the same as the soundness proof of mathematical induction for realizabilities.

This example shows that informative induction follows from non-informative induction in ATTT. This ability of ATTT distinguishes it from the other type theories and illustrates its set theoretical nature.

There is one thing missing in the axioms above. It is Peano's fourth axiom "$0 = 1$ implies $\bot$." There are several formulations of the axiom, since we have several ways to define implication and absurdity. One of the strongest formulation would be

$$\Pi A \in \mathit{Type}. \bigwedge w \in 0 = 1. \bot_A.$$

The type of this refinement is $\Pi A \in \mathit{Type}.A$. This is rather problematic as pointed out in [18]. If the refinement is realized by f, then $f(A)$ is a polymorphic element of A. In PX system, this problem did not arise, as it is a monotype system whose domain is inhabited. But, we are now in a type theoretic framework and so cannot use such a solution. In Coq system [18], "abort"-command is introduced which belongs polymorphically to any type A. This solution seems to lead us to a type theory with non-terminating programs. Coq system offers a solution, but it is not very clear to me.

This approach resembles the resent works on controls in Curry-Howard isomorphisms by Griffin, Murthy, Nakano. Especially, Nakano's calculus accommodate catch/throw mechanism in a natural constructive framework [17]. As "abort" is a throw to the top-level, there might be a natural way to interpret this problem by Nakano's calculus.

Anyway, the problem does not seem to have been settled satisfactory. Thus, we do not use it but consider a less ambitious formulation

$$\Pi A \in \mathit{Type}. \bigwedge w \in 0 = 1. A \to \bot_A,$$

which is equivalent to

$$\bigwedge w \in 0 = 1. \bot_{unit}.$$

In Coq system, essentially the same statement is proved by a proof by cases, which is similar to Martin-Löf's proof of the 4th axiom in his type theory. If we introduce a principle to define refinement-valued functions by cases, we can do the same thing in ATTT. Assume that we can define the following program from Nat to Ω.

$$\lambda x \in Nat.(\mathit{if}\ x = 0\ \mathit{then}\ \top_{unit}\ \mathit{else}\ \bot_{unit}).$$

By assuming $0 = 1$, we can convert $\top_{unit}$ to $\bot_{unit}$. This proves the axiom above by means of conversion rule or an appropriate equality rule. Note that the program above have the type $Nat \to \mathit{refine}(\mathit{unit})$. If we extend the type of the recursor rec so that it permits refinement kinds as well as the type A, then we can do such a definition by cases. This does not seem bad. But we do not do so here. We simply add the axiom to the system.

Note that type A is assumed to be non-empty, when we deduce $\bot_A$:

$$\Pi A \in \mathit{Type}. \bigwedge w \in 0 = 1. A \to \bot_A.$$

This means that "have an element of A and return it as the default value if something is wrong." The non-emptiness assumption would not cause a big problem in practical cases. Such an A would be the type of a specification. A specification of a program of a type whose elements are not known would be meaningless. Thus we may suppose the type is inhabited.

Acknowledgments

I am very grateful to Rod Burstall and Henk Barendregt for inviting me to the BRA workshop "Types for Proofs and Programs" and for opportunity to present the paper. I thank Christine Paulin, Erik Poll,

Kees Hemerik, Benjamin Werner, Mariko Yasugi for discussions on the subjects of the paper and/or Coq system which helped me to clarify the nature of ATTT.

References

[1] M.J. Beeson, *Foundations of Constructive Mathematics*, Springer-Verlag, 1985.

[2] R.M. Burstall and J. H. McKinna, *Deliverables: an approach to program in Constructions*, Technical Report ECS-LFCS-91-133, Department of Computer Science, The University of Edinburgh, 1991.

[3] Dowek, C. et al., *The Coq Proof Assistant User's Guide*, Version 5.6, Technical Report No. 134, INRIA, December, 1991.

[4] Th. Coquand and G. Huet, Calculus of Constructions, *Information and Computation*, vol. 76, pp. 95-120, 1988.

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

[6] S. Hayashi, On derived rules of intuitionistic second order arithmetic, *Commentariorum Mathematicorum Universitatis Sancti Pauli*, vol. XXVI, pp. 77-103, 1977.

[7] S. Hayashi, Adjunction of semifunctors: categorical structures in non-extensional lambda calculus, *Theoretical Computer Sciences,*, vol. 41, pp.95-104, 1986.

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

[9] S. Hayashi, *Singleton, Union and Intersection Types for Program Extraction*, Lecture Notes in Computer Science No. 526, pp. 701-730, T.Ito and A.R.Meyer, eds., Springer-Verlag, 1991, an extended version to appear in Information and Computation.

[10] R. Hoofman, The theory of semi-functors, *Mathematical Structures in Computer Science*, vol. 3, pp.93-128, 1993.

[11] B. Jacobs, Semantics of the second order lambda calculus, *Mathematical Structures in Computer Science*, vol. 1, pp. 327-360, 1991.

[12] Z. Luo. *ECC, an Extended Calculus of Constructions*, in Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science, Asilomar, California, 1989.

[13] Z. Luo. *An Extended Calculus of Constructions*, Ph. D. thesis, Department of Computer Science, The University of Edinburgh, 1990.

[14] Z. Luo. *A problem of adequacy: conservativity of Calculus of Constructions over higher order logic*, Technical Report ECS-LFCS-90-121, Department of Computer Science, The University of Edinburgh, 1990.

[15] M. Makkai and G. Reyes. *First Order Categorical Logic*, Lecture Note in Mathematics No. 611, 1977, Springer-Verlag.

[16] J. H. McKinna. *Deliverables: A Categorical Approach to Program Development in Type Theory*, Ph. D. thesis, Department of Computer Science, The University of Edinburgh, 1992.

[17] , H. Nakano. *A Constructive Formalization of the Catch and Throw Mechanism*, in Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, 1992.

[18] C. Paulin-Mohring and B. Werner, Synthesis of ML programs in the System Coq, Journal of Symbolic Computation, 1993.

[19] F. Pfenning, Intersection Types for a Logical Framework, in this volume, 1992.

[20] Erik Poll, A programming logic for F_ω, Computing Science Notes, 92/25, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1992.

[21] A.S. Troelstra, *Metamathematical investigations of intuitionistic arithmetic and analysis*, Lecture Notes in Mathematics, vol. 344, Springer-Verlag, 1973.