Refining Refinement Types

Ewen Denney*

LFCS, Department of Computer Science, University of Edinburgh
Kings Buildings, Edinburgh EH9 3JZ, Scotland
Fax : +44 131 667 7209
Email : `ewd@dcs.ed.ac.uk`

September 12, 1997

Abstract

We develop a theory of program specification using the notion of *refinement type*. This provides a notion of structured specification, useful for verification and program development. We axiomatise the satisfaction of specifications by programs as a generalised typing relation and give rules for refining specifications. A per semantics based on Henkin models is given, for which the system is proven to be sound and complete.

Keywords : specification, refinement, verification, type theory, Henkin models

1 Introduction

We address the question of what is a suitable notion of specification for a programming language, where the properties of interest can be expressed using some given program logic. In particular, we restrict our attention to those which can be studied using typed lambda calculi, that is, typed functional programming languages.

A number of possibilities can be considered. One is to say that a specification is a type in some expressive type theory. This is the approach taken by [Luo91] for example. The problem is that this only works for an intuitionistic logic. Classical logics are more common for specification, and cannot easily be encoded in type theories. Also, programming languages generally have a simple type system of their own, and this must somehow be related to the specification type theory.

Another possibility is to say that a specification is just a proposition of the program logic with a distinguished free variable. This is the approach traditionally taken by the program refinement community. [Mor94] describes a refinement calculus based on the use of first-order predicate logic.

However, this approach has a number of shortcomings, which we illustrate with an example below. The main point is that for compositional verification and program development is it better to put more structure on specifications.

In this paper, we suggest a third possibility, a combination of the program logic with the type theory of the programming language which we call *refinement types*. The notion of refinement type has been studied extensively in program analysis (under different names) and there are many different systems, depending on the area of interest. The general idea is to have two levels — an underlying level of program types, and a more expressive level of program properties, which are then treated like types. For us, this more expressive level will be the specifications. Hence we can exploit type-theoretic structure in our specifications, but do not need to do any encoding.

*Supported by a postgraduate research studentship from the Engineering and Physical Sciences Research Council

We describe a verification calculus based on the simply-typed lambda calculus with products ($\lambda^{\times\rightarrow}$) and some ground types such as `nat` and `bool`. The satisfaction of specifications by programs is axiomatised as a generalised typing relation. We do this by viewing specifications as refinements of an underlying type, expressed using the program logic. We use classical predicate logic as program logic here. We axiomatise an ordering on the refinement types, to be viewed as an increase in information, or refinement of specifications.

If P is a proposition in the logic containing free variable x with refinement type ϕ, where ϕ is a refinement of τ, then the subset type $(x : \phi)P$ is a refinement of type τ. It is convenient to combine this with dependent functions and products at the level of refinement types, even though the underlying types are not dependent.

Refinement types are not the same as subtypes though. For example, `nat` might be a subtype of `real` say, but not a refinement type. Though we are careful to distinguish refinement types from subtypes, equality is 'stratified' at different refinement types as in subtyping systems.

Contexts consist of both variable assumptions $x : \phi$, and propositions P. These are combined so as to make explicit the mutual dependencies in well-formedness. The dependency arises because we allow refinement types in terms, which in turn can appear in propositions.

We give a simple set-theoretic interpretation of the calculus. The main result of the paper is soundness and completeness with respect to this class of models.

In Section 2 we consider a simple example of specifying and verifying a program in order to motivate the features of our calculus. We then give the syntax and rules of the calculus in Section 3. In Section 4 we return to the `div2` example. Section 5 gives the semantics and proofs of soundness and completeness. Finally, we make some conclusions in Section 6.

Related Work

A number of authors have advocated program analysis using annotated type systems. An example in the 'non-standard type system as program logic' paradigm is [NN88], a system for binding time analysis (and optimisation). Jensen [Jen91] performs strictness analysis using intersection types and primitive types to indicate termination. Burn [Bur92] considers a more general framework, with intersection and union types. Each of these systems axiomatises property deductions using refinements $\phi \sqsubseteq \phi'$.

Pfenning, who introduced the term "refinement type", used them to express properties of mini-ML programs [FP91]. He also extended LF [Pfe93] with (possibly intensional) properties such as "in normal form" (a property of derivations). He does not allow refinement types in abstractions though. Pfenning adds primitive properties and intersection, whereas we add primitive predicate symbols.

There have been various approaches by type theorists to combining logic and types. Feferman's system of variable types [Fef85] extends $\lambda^{\times\rightarrow}$ with subset types, though equality does not depend on the type. Refinement of properties can be defined in the logic, and is not explicitly axiomatised. Talcott used a similar form of refinement types (though the underlying theory is untyped), based on Feferman's work, in order to express local information for use in transformations to introduce continuations. Since she lacks a typed equality though, local assumptions can not be discharged and conclusions take the form "$t = t'$ in the context $C[]$".

Aspinall [Asp95] presents a dependent type theory, $\lambda_{\leq\{\}}$, which is formally similar in that it has subtyping on dependent functions and products. His system is based on subtyping rather than refinement types however. Dependency in $\lambda_{\leq\{\}}$ comes from singleton types, which are a special case of subset types.

Hayashi's ATTT [Hay94b] is a rich type theory conservatively extending the polymorphic lambda calculus with singleton, union and intersection types. It is based on the refinement type philosophy, maintaining a distinction between types and specified subsets in order to eliminate non-computational information during program extraction. Refinement types are not allowed on abstractions. Dependent function and product types can be defined from the nondependent constructors [Hay94a], as well as subset types constructed using full second order intuitionistic logic.

As pointed out by Hayashi, schemas in the Z specification language can be seen as refinement types. They comprise two parts — a typing declaration, and a logical predicate given as a collection of axioms.

The deliverables approach [BM92, McK92] is to consider a program paired with its proof of correctness. We are similarly motivated in wanting to structure specifications using program types, but differ in taking proof existence as more important than the proof itself — terms do not need a witness to satisfy a refinement type. Our calculus could be regarded as an internal language for deliverables.

The work of Luo [Luo91] presents an encoding of specifications and 'specification morphisms' (corresponding to our terms) in an expressive type theory. Our work provides a more direct analysis of the concept of specification. The existential form of Martin-Löf's type theory with subset types in [NPS90] is similar.

The program refinement community has traditionally used unstructured specifications of the $(x : \tau)P$ form. For example, Morgan [Mor94] describes a refinement calculus based on the use of propositions of first order predicate logic.

2 Example

Let us consider specifying division by 2 on the naturals and verifying that a program satisfies the specification. We will take the simply-typed lambda calculus and classical first-order predicate logic as simple programming and specification languages respectively. We will use an applied $\lambda^{\times\rightarrow}$ theory extended with a constant for iteration over the naturals, where $\mathtt{natiter}\ z\ f\ n$ computes the n-th iterate $f^n(z)$. As a first approximation to specifications we use propositions with a distinguished free variable, which we write as $(x : \tau)P$ where τ is the type of the variable x in proposition P.

A program `div2` which implements division on the naturals is

$$\mathtt{div2} = \lambda n : \mathtt{nat}.\pi_1(\mathtt{div2}'\ n) : \mathtt{nat} \rightarrow \mathtt{nat}$$

where this uses the auxiliary function

$$\mathtt{div2}' = \mathtt{natiter}\ \langle 0, 0\rangle\ (\lambda p : \mathtt{nat} \times \mathtt{nat}.\langle \pi_2 p, \pi_1 p + 1\rangle)$$

Now this can be specified as

$$\mathtt{div2\_spec} = (f : \mathtt{nat} \rightarrow \mathtt{nat})\ \forall n : \mathtt{nat}.n = 2 * fn \vee n = 2 * fn + 1$$

We want to axiomatise a satisfaction relation `sat` between programs (closed terms) and specifications, so that we can prove

$$\mathtt{div2\ \ sat\ \ div2\_spec}$$

One simple way of doing this is to say that t `sat` $(x : \tau)P$ is just taken to be a notation for a typing and a proposition, with derived rule

$$\frac{t : \tau \quad P[t/x]}{t\ \ \mathtt{sat}\ \ (x : \tau)P} \tag{1}$$

so this example reduces to proving

$$\forall n : \mathtt{nat}.n = 2 * \mathtt{div2}(n) \vee n = 2 * \mathtt{div2}(n) + 1$$

Now, our specification language is rather cumbersome as it stands, so let us introduce dependent sums and products as abbreviations

$$\Sigma_{x:(x:\sigma)P}(y : \tau)Q \quad \text{for} \quad (z : \sigma \times \tau)P[\pi_1 z/x] \wedge Q[\pi_1 z/x, \pi_2 z/y]$$

$$\Pi_{x:(x:\sigma)P}(y : \tau)Q \quad \text{for} \quad (f : \sigma \rightarrow \tau)\forall x : \tau.P \supset Q[fx/y]$$

If we allow ourself the further abbreviation of viewing types as trivial specifications, so that for example, `nat` can stand for $(x : \mathtt{nat})\mathtt{true}$, then we can write our specification more compactly as

$$\mathtt{div2\_spec} = \Pi_{n:\mathtt{nat}}(m : \mathtt{nat})n = 2 * m \vee n = 2 * m + 1$$

Now, using our abbreviations, the following rule is admissible from our definition of sat

$$\frac{n : \texttt{nat} \vdash \pi_1(\texttt{div2}'\ n)\ \ \texttt{sat}\ \ (m : \texttt{nat})n = 2m \vee n = 2m+1}{\lambda n : \texttt{nat}.\pi_1(\texttt{div2}'\ n)\ \ \texttt{sat}\ \ \Pi_{n:\texttt{nat}}(m : \texttt{nat})n = 2m \vee n = 2m+1}$$

where we understand $n : \texttt{nat} \vdash t\ \ \texttt{sat}\ \ \phi$ to mean for all closed $t' : \texttt{nat}$, $t[t'/n]\ \ \texttt{sat}\ \ \phi[t'/n]$. In general then, we want to consider satisfaction in an arbitrary context. Note the similarity to a typing rule. In fact, not only are Σ and Π useful structuring devices for specifications, they are also useful for proofs, as specifications of programs often tend to be most naturally expressed and proved in a 'shape' similar to the program.

For example, the program div2 is an abstraction and the specification div2_spec is of the form $\Pi_{x:\phi}\psi$. The rule directly reflects a natural proof that div2 satisfies div2_spec. Similarly, the auxiliary function $\texttt{div2}'$ has specification

$$\texttt{div2}'\texttt{\_spec} = \Pi_{n:\texttt{nat}}\Sigma_{(m:\texttt{nat})\mathrm{n=2m\vee n=2m+1}}(m' : \texttt{nat})m + m' = n$$

The proof of this, in turn, involves showing that a pair satisfies a product specification, and an abstraction satisfies a functional specification (as above). We also have to use induction to show that an iteration satisfies some specification parameterised on the naturals. We consider this example more fully in Section 4.

We do not throw away the original rule (1) however, since not all specifications can be given in a structured form.

A significant benefit in writing specification in this more structured form is conceptual — it is preferable to structure specifications for then the task of comprehension need not be duplicated unnecessarily for specification and program. Also, separate checks of well-formedness (*i.e.* type-checking here) and correctness, will involve some duplication of effort, so it is better to combine types and correctness properties. Though we will not consider it here, in order to be the basis of a useful program development methodology, it helps for our specifications and proofs to reflect the structure of the programs.

There is one final aspect of specifications which we must consider — equality. The kind of specifications with which we are concerned here are those which specify the input-output characteristics of programs. We are only interested in programs up to *extensional equality*. The alternative, in a type-theoretic setting, is to use an intensional equality and distinguish programs on the basis of syntactic form. This would be unnatural here however, so we view specifications as coming equipped with a *partial equivalence relation* (per), which acts as an equality. A per is a symmetric and transitive relation on the set of terms at the type, or equivalently, an equivalence relation on a subset of terms at the type.

For example, the specification $\Pi_{l:\texttt{nonemptylist}}(n : \texttt{nat})\texttt{Min}(n,l)$ where the proposition $\texttt{Min}(n,l)$ says that n is the minimum element in list l, is a refinement type over type $\texttt{list} \to \texttt{nat}$. We want to regard functions $f, f' : \texttt{list} \to \texttt{nat}$ as equal solutions of this specification if they give the same results for *nonempty* lists. Any program satisfying this specification must be defined on the empty list, but we are not interested in the value it takes there.

Moreover, it is a useful abbreviation in specifications themselves to write $=_{\Pi_{x:(x:\sigma)P}(y:\tau)Q}$ where

$$t =_{\Pi_{x:(x:\sigma)P}(y:\tau)Q} t' \text{ when } \forall x, x' : \sigma.(x =_\sigma x') \wedge P[x] \wedge P[x'] \supset (tx =_\tau t'x') \wedge Q[tx] \wedge Q[t'x']$$

Now, we would attain some conceptual simplicity if specifications were to subsume types, satisfaction to subsume typing, and equality at a specification to subsume the usual equality at a type (which is often left implicit). For example, we use $(n : \texttt{nat})\texttt{true}$ in place of nat, and $\Sigma_{n:(n:\texttt{nat})\texttt{true}}(b : \texttt{bool})\texttt{true}$ for $\texttt{nat} \times \texttt{bool}$. However, $\Pi_{x:(x:\sigma)P}(y : \tau)Q$ is no longer an abbreviation since it represents a different equality now.

We believe it is misleading to regard specifications as types though, and refer to the specifications of this idealised specification language as *refinement types*. Equality is given by a per over the underlying type.

A specification therefore, is a refinement type, and consists of

- a type τ
- a per ϕ over τ

We take a program in this calculus to be

- an equivalence class of a per ϕ

The alternative would be to take a program as an element of the domain of a per, but this would be unnatural because we would then be distinguishing programs beyond extensional equality.

We use a notation for the equivalence classes of pers, by allowing refinement types on the variables in abstractions. For example, $\lambda n : \mathtt{even}.n$ is a class in the per $\mathtt{even} \to \mathtt{nat}$ but not $\mathtt{nat} \to \mathtt{nat}$, and $\lambda n : \mathtt{nat}.n$ is a class in both $\mathtt{even} \to \mathtt{nat}$ and $\mathtt{nat} \to \mathtt{nat}$. The equality $t =_\phi t'$ says that t and t' are the same equivalence class of per ϕ.

For refinement types ϕ and ϕ' over the same underlying type, we want to consider refinements $\phi \sqsubseteq \phi'$, to be though of semantically as per inclusion (*i.e.* $x =_{\phi'} y \supset x =_\phi y$). We use the square $\sqsubseteq$ symbol to indicate an information ordering — the refinement of specifications.

3 The Calculus

We now give the syntax of the system and describe its judgements. Some syntactic results are then given and we give an operational intuition for the language.

3.1 Syntax

The idea is that we construct a theory of refinement types on top of an underlying $\lambda^{\times\to}$ theory and a first-order logic theory.

Definition 3.1 *A* signature *consists of a finite collection of ground types γ, and n-ary constants k, each of which is assigned some sort $\tau_1, \ldots, \tau_n \to \tau$; a collection of n-ary predicates F, which are also assigned a sequence of types $\tau_1, \ldots, \tau_n$, indicated by $F : \mathtt{pred}\ \tau_1, \ldots, \tau_n$; and axioms, that is, well-formed propositions and refinement typings in context, $\Gamma \vdash P$, $\Gamma \vdash t : \phi$.*

Although we do not have arbitrary refinement types as primitive in a signature, we get much the same expressiveness using primitive predicate symbols. For example, with the primitive type $\mathtt{nat}$, we could have predicate $\mathtt{even} : \mathtt{pred}\ \mathtt{nat}$, and overload $\mathtt{even}$ for the refinement type $(n : \mathtt{nat})\mathtt{even}(n)$; constants $* : \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat}$ and $\mathtt{inverse} : \mathtt{nat} \to \mathtt{nat}$, with axiom $\mathtt{inverse} : \Pi_{n:(n:\mathtt{nat})\mathtt{n}\neq 0}(m : \mathtt{nat})n * m = 1$.

The raw terms of the language are given by

Refinement types	$\phi ::= \mathbf{1} \mid \gamma \mid \Sigma_{x:\phi}\psi \mid \Pi_{x:\phi}\psi \mid (x : \phi)P$
Terms	$t ::= x \mid k(t_1, \ldots, t_n) \mid * \mid \langle t, t' \rangle \mid \lambda x : \phi.t \mid \pi_i(t) \mid tt'$
Propositions	$P ::= \mathtt{false} \mid P \supset P' \mid \forall x : \phi.P \mid F(t_1, \ldots, t_n) \mid t =_\phi t' \mid \phi \sqsubseteq \phi'$
Contexts	$\Gamma ::= \langle\rangle \mid \Gamma, x : \phi \mid \Gamma, P$

We do not distinguish types and refinement types as syntactic categories. Refinement types should be viewed as being refinements of underlying types, so for example, if ϕ is over σ, and ψ is over τ, then $\Sigma_{x:\phi}\psi$ is over $\sigma \times \tau$. Formally however, types are just refinement types with no logical information, that is, without any $(x : \phi)P$. We use σ and τ as metavariables for types.

There is a unique (up to equality) term $*$ of unit type $\mathbf{1}$. The meaning of the other refinement types in terms of satisfaction is that $\langle t, t' \rangle$ satisfies $\Sigma_{x:\phi}\psi$ when t satisfies ϕ and t' satisfies $\psi[t/x]$; term

t satisfies $\Pi_{x:\phi}\psi$ when for every t' satisfying ϕ, tt' is well-formed and satisfies $\psi[t'/x]$; and t satisfies $(x:\phi)P$ when it satisfies ϕ and the proposition $P[t/x]$ holds.

Terms have their usual meaning in the lambda calculus, except that an abstraction $\lambda x:\phi.t$ should be thought of as a simple specification of terms $\lambda x:\sigma.t'$ such that for all t'' which satisfy ϕ, $t'[t''/x]$ satisfies $t[t''/x]$. We will refer to terms in the underlying $\lambda^{\times\rightarrow}$ as *total* terms. For example, $\lambda n:\texttt{even}.n$ specifies total terms of type $\texttt{nat} \rightarrow \texttt{nat}$ which are the identity on even arguments. The application $(\lambda x:\phi.t)t''$ is only well-formed for arguments t'' which satisfy ϕ so behaviour outwith ϕ is not constrained. Note that this means that although $\texttt{even} \rightarrow \texttt{even}$ is a refinement of the type $\texttt{nat} \rightarrow \texttt{nat}$, the term $\lambda n:\texttt{even}.n$ does not itself have type $\texttt{nat} \rightarrow \texttt{nat}$. Intuitively, we can say that a term t has refinement type ϕ if its behaviour 'at ϕ' is uniquely determined, that is, any two total terms which satisfy t are themselves equal at ϕ.

The propositions are typed first order predicate logic of equalities and refinements. We use classical typed first-order logic as an example of a simple expressive logic.

We sometimes abbreviate the assumption $x:(x:\phi)P$ as $x:\phi\,|\,P$.

Remark 3.1 Our choice of first-order classical logic is only significant insofar as it is an example of what we might call an *extensional* logic. We make essential use of the fact that for all terms t, t' and propositions P, if t is extensionally equal to t' (which, in general, may coincide with the equality of the logic, if it has one, but otherwise can be defined as a logical relation), then $P[t/x]$ holds if and only if $P[t'/x]$ does. In other words, we require Leibniz and observational equality to coincide.

Intuitionistic logic would be fine too, but these choices can be contrasted with, say, the use of an intensional logic such as the modal μ-calculus, where terms are viewed as transition systems through their reduction sequences.

3.2 Judgements

The judgements of the calculus are

$$\Gamma \vdash t : \phi$$

$$\Gamma \vdash P$$

where the propositions include

$$\Gamma \vdash t =_\phi t'$$

$$\Gamma \vdash \phi \sqsubseteq \phi'$$

All judgements are made in a context Γ of variable assumptions $x:\phi$ and propositions P. There are also mutually dependent well-formedness judgements

$$\vdash \Gamma \;\; \texttt{wf}$$

$$\Gamma \vdash \phi : \texttt{Ref}\;\;\tau$$

$$\Gamma \vdash P \;\; \texttt{wf}$$

We say that a term t is well-formed in context Γ when there exists a refinement type ϕ such that $\Gamma \vdash t:\phi$. Note that ϕ need not be unique, though the underlying type is unique. Sometimes we write $\Gamma \vdash t, t' : \phi$ for $\Gamma \vdash t : \phi$ and $\Gamma \vdash t : \phi'$.

The well-formedness judgement for refinement types, $\Gamma \vdash \phi : \texttt{Ref}\;\;\tau$, says that refinement type ϕ in context Γ is over the type τ. We abbreviate $\phi : \texttt{Ref}\;\;\tau$ as $\phi\;\;\texttt{wf}$ when the type τ is not significant.

We use g as a metavariable for environments, that is, tuples of terms which satisfy the refinement types and propositions in the context. We will use the abbreviations $g:\Gamma$, $g =_\Gamma g'$ and $t[g/\Gamma]$ to indicate simultaneous satisfaction, equality and substitution respectively.

We understand $\Gamma \vdash t : \phi$ to mean for all well-typed substitutions of the variables in Γ with g and g', if $g =_\Gamma g'$, then $t[g/\Gamma] =_{\phi[g/\Gamma]} t[g'/\Gamma]$. Although the use of g rather than g' in the equality seems to introduce an asymmetry, it can be shown in fact that $\phi[g/\Gamma] = \phi[g'/\Gamma]$.

3.3 Rules of the Calculus

The rules of the calculus are listed in the appendix, in figures 1-8. Well-formedness and proof judgements are mutually recursive. Figures 1 and 2 give the well-formedness rules for contexts and refinement types respectively. The well-formedness rules for propositions in figure 3 require that for an equality $t =_\phi t'$ to be well-formed, the terms t and t' have the property ϕ. This means for example, that $n =_{\mathtt{even}} n'$ is not well-formed in the context $n : \mathtt{nat}, n' : \mathtt{nat}$. Refinements are only well-formed when the refinement types are over the same type.

In figure 4 we give the refinement typing rules, which also serve as well-formedness rules for the terms. These generalise the usual typing rules of $\lambda^{\times\rightarrow}$ in the obvious way, with the addition of an introduction rule for refinement types. The natural elimination rule, $t : (x : \phi)P \Rightarrow t : \phi$, can be derived. We also need the rule (EQ) to infer a property from an equality. Finally, there is a weakening rule (WEAKEN), like the subsumption rule of subtyping systems.

Note that it is possible for the body of an abstraction to not be well-formed, just using the refinement type of the lambda, but the whole term to be well-formed at a particular refinement type. For example, we have $\lambda n : \mathtt{nat}.(\lambda m : \mathtt{even}.m)n : \mathtt{even} \rightarrow \mathtt{nat}$, although the body is not well-formed in the context $n : \mathtt{nat}$.

Figures 5 and 6 give equality rules for terms. The well-formedness hypothesis in the congruence rule (CONG-EQ) is important since with stratified equalities, we require equality at the appropriate property.

In combination with (EQ), the rule (ABS-ξ-EQ) lets us prove (as in [Asp95]), for example, that $\lambda n : \mathtt{nat}.n : \mathtt{even} \rightarrow \mathtt{even}$ even though $\mathtt{nat} \rightarrow \mathtt{nat}$ and $\mathtt{even} \rightarrow \mathtt{even}$ are incomparable (which is an obstacle for some subtyping systems).

Example 3.1

$$\dfrac{\dfrac{\dfrac{n : \mathtt{nat}, \mathtt{even}(n) \vdash n : \mathtt{nat}}{\mathtt{nat} \sqsubseteq (n : \mathtt{nat})\mathtt{even}(n)}\ \text{REFTYPE-R-REF} \qquad \dfrac{\dfrac{n : \mathtt{even} \vdash n : \mathtt{even}}{n : \mathtt{even} \vdash n =_{\mathtt{even}} n}\ \text{REFL-EQ}}{}}{\lambda n : \mathtt{even}.n =_{\mathtt{even}\rightarrow\mathtt{even}} \lambda n : \mathtt{nat}.n}\ \text{ABS-}\xi\text{-EQ}}{\lambda n : \mathtt{nat}.n : \mathtt{even} \rightarrow \mathtt{even}}\ \text{EQ-PROP}$$

Using the symmetry rule (SYMM-EQ) we can deduce the symmetric forms of (APP-ξ-EQ) and (PROJ2-ξ-EQ), *e.g.* if $t =_{\Sigma_{x:\phi}\psi} t'$ then $\pi_2(t) =_{\psi[\pi_1(t')]} \pi_2(t')$.

We seem to need the non-standard hypotheses for the η-rules. The usual hypothesis for (λ-β-EQ) would be $\Gamma \vdash t : \Pi_{x:\phi}\psi$. However, the following example makes essential use of it.

Example 3.2

$$\dfrac{\dfrac{\dfrac{f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx), n : \mathtt{nat} \vdash fn : \mathtt{nat} \qquad f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx), n : \mathtt{nat} \vdash \mathtt{even}(fn)}{f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx), n : \mathtt{nat} \vdash fn : \mathtt{even}}}{f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx) \vdash f =_{\mathtt{nat}\rightarrow\mathtt{even}} \lambda n : \mathtt{nat}.fn}}{f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx) \vdash f : \mathtt{nat} \rightarrow \mathtt{even}}$$

The η-rules enable us to combine logical and typing assumptions.

Similarly, we use (π-η-EQ) in order to prove, for example, that

$$z : \sigma \times \tau, P[\pi_1 z], Q[\pi_2 z] \vdash z : (x : \sigma)P \times (y : \tau)Q$$

Figure 7 lists the refinement rules. There are the obvious 'structural' refinement rules for units and products, with a contravariant rule for functions, and rules for refinements involving subset types. We regard refinement as a proposition because we use logical reasoning to prove refinements. The rules for the other propositions are given in figure 8. This is a natural deduction presentation of typed classical predicate logic of equalities and refinements. Assumptions in context can be used via (HYP) or (REFTYPE-ELIM). The (CONG) rule actually follows from (CONG-EQ).

In order that when $\Gamma \vdash P$ is derivable, we have well-formedness of P in Γ, the rules (`false`-ELIM) and (HYP) have explicit well-formedness hypotheses. This prevents us proving non well-formed equalities from `false`.

3.4 Metatheory

We prove a few syntactic results about the calculus. Some of the following results will be needed in Section 5 for proving completeness with respect to the semantics. Other standard metatheoretic results (not listed here) can be deduced from completeness.

We have the expected downward closure of equality with respect to refinement, as follows.

Proposition 3.1 *The following rule is admissible.*

$$\frac{\Gamma \vdash t =_{\phi'} t \quad \Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash t =_{\phi} t'}$$

Proposition 3.2 *If closed t has type σ, then there exists a total term t' such that $\vdash t =_\sigma t'$.*

Proof: Terms can only fail to be total by containing an abstraction with a refinement type. Either the abstraction is applied or it is not. If it is, reduce it (equality at the type). If not, then $\lambda x : \phi.t : \sigma \to \tau$, and we must have $\phi \sqsubseteq \sigma$, so $\lambda x : \phi.t =_{\sigma \to \tau} \lambda x : \sigma.t$. ∎

Well-formedness is a combination of typing and proving properties. This is illustrated in the following proposition.

Proposition 3.3 *Given raw term t in well-formed context Γ, the following are equivalent :*

1. $\texttt{Typify}(t)$ *is well-typed in typing context* $\texttt{Typify}(\Gamma)$*, where* $\texttt{Typify}$ *replaces* $\phi : \texttt{Ref}\ \tau$ *with* τ.
2. *There exists a ϕ such that* $\Gamma, \texttt{false} \vdash t : \phi$.
3. *There exists a type τ, such that for all* $\Gamma \vdash \phi : \texttt{Ref}\ \tau$*, we can prove* $\Gamma, \texttt{false} \vdash t : \phi$.

We say that t is well-structured.

We will see in Section 5 that $[\![\Gamma \vdash t]\!]$ is only defined for well-structured terms.

Lemma 3.1 *It is an easy proof to show that for well-formed properties $\Gamma \vdash \phi$, we have reflexivity of refinement $\Gamma \vdash \phi \sqsubseteq \phi$. Some other derived refinements (assuming well-formedness where appropriate) are*

1. $\dfrac{\Gamma, x : \phi \vdash P' \supset P \quad \Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash (x : \phi)P \sqsubseteq (x : \phi')P'}$
2. $\Gamma, P \vdash (x : \phi)P = \phi$
3. $\Gamma \vdash (x : \phi)\texttt{true} = \phi$
4. $\Gamma \vdash \Pi_{x:(x:\phi)P}(y : \psi)Q \sqsubseteq (f : \Pi_{x:\phi}\psi)\forall x : \phi.P \supset Q[fx/y]$
5. $\dfrac{\Gamma \vdash \phi\ \texttt{wf} \quad \Gamma, x : \phi \vdash \psi\ \texttt{wf} \quad \Gamma, x : \phi, y : \psi \vdash Q\ \texttt{wf}}{\Gamma \vdash \Pi_{x:\phi}(y : \psi)Q = (f : \Pi_{x:\phi}\psi)\forall x : \phi.Q[fx/y]}$
6. $\Gamma \vdash \Sigma_{x:(x:\phi)P}(y : \psi)Q = (z : \Sigma_{x:\phi}\psi)P[\pi_1 z/x] \wedge Q[\pi_1 z/x, \pi_2 z/y]$
7. $\Gamma \vdash (x : (y : \phi)P[y])Q[x] = (z : \phi)P[z] \wedge Q[z]$

Proposition 3.4 *If* $\Gamma \vdash \phi, \phi' : \mathtt{Ref}\ \tau$*, then* $\Gamma \vdash \phi \sqsubseteq (x : \phi')\mathtt{false}$.

Remark 3.2 Refinement is a definitional extension of refinement typing in the sense that $\Gamma, x : \phi' \vdash x : \phi$ whenever $\Gamma \vdash \phi \sqsubseteq \phi'$.

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \phi' \vdash x : \phi}{\Gamma, x : \phi', \mathtt{true} \vdash x : \phi}}{\Gamma \vdash \phi \sqsubseteq (x : \phi')\mathtt{true}}\ \text{REFTYPE-R-REF} \qquad \dfrac{\Gamma \vdash \phi' \sqsubseteq \phi' \quad \Gamma, x : \phi' \vdash \mathtt{true}}{\Gamma \vdash (x : \phi')\mathtt{true} \sqsubseteq \phi'}\ \text{REFTYPE-L-REF}}{\Gamma \vdash \phi \sqsubseteq \phi'}$$

However, this does not mean that the refinement rules are superfluous, and that proofs of refinement always reduce to typing. In practice, when working backwards from $\phi \sqsubseteq \phi'$, we only make use of the refinement type rules, via REFTYPE-R-REF, when ϕ' is of the form $(x : \psi)Q$.

The rules ABS-ξ-EQ and WEAKEN have refinements as hypotheses and they are needed to prove $x : (x : \phi)P \vdash x : \phi$. Therefore, as the rules stand, we cannot treat $\phi \sqsubseteq \phi'$ as syntactic sugar for $\Gamma, x : \phi' \vdash x : \phi$.

3.5 Operational intuition

The operational intuition behind equality and satisfaction of refinement types can be expressed in terms of a notion of (closed) total term. Recall that programs are to be thought of as equivalence classes of pers. We expect

$$t : \phi \text{ when } \forall u, u' \in \mathtt{Total}(t)\ .\ u \sim_\phi u'$$

$$t =_\phi t' \text{ when } \forall u \in \mathtt{Total}(t).\forall u' \in \mathtt{Total}(t')\ .\ u \sim_\phi u'$$

where we pick some definition of $\mathtt{Total}(t)$ and a notion of equivalence on total terms $\sim_\phi$.

Definition 3.2 *We define a logical relation* $\sim_\phi$ *between total terms, a function* $\mathtt{Total}$ *which takes a closed term to its set of total realizers, and relations* $\simeq_\phi$ *between arbitrary terms. For example,* $\mathtt{Total}(\langle \lambda n : \mathsf{even}.n, * \rangle)$ *is the set of pairs* $\langle f, * \rangle$*, such that total* f *is an identity on evens.*

-
  - $u_1 \sim_{\mathbf{1}} u_2$ *when* $u_1 =_{\mathbf{1}} u_2$
  - $u_1 \sim_\gamma u_2$ *when* $u_1 =_\gamma u_2$
  - $u_1 \sim_{\Sigma_{x:\phi}\psi} u_2$ *when* $\pi_1(u_1) \sim_\phi \pi_1(u_2)$ *and* $\pi_2(u_1) \sim_{\psi[\pi_1(u_1)]} \pi_2(u_2)$
  - $u_1 \sim_{\Pi_{x:\phi}\psi} u_2$ *when for all* $u_1' \sim_\phi u_2'$, $u_1 u_1' \sim_{\psi[u_1']} u_2 u_2'$
  - $u_1 \sim_{(x:\phi)P} u_2$ *when* $u_1 \sim_\phi u_2$ *and* $\tilde{P}[u_1]$, $\tilde{P}[u_2]$ *where* $\tilde{P}$ *is* P *with each* $=_\phi$ *replaced by* $\simeq_\phi$.
-
  - $\mathtt{Total}(k) = \{k\}$
  - $\mathtt{Total}(*) = \{*\}$
  - $\mathtt{Total}(\langle t, t' \rangle) = \{\langle u, u' \rangle \mid u \in \mathtt{Total}(t), u' \in \mathtt{Total}(t')\}$
  - $\mathtt{Total}(\lambda x : \phi.t) = \{f : \sigma \to \tau \mid \forall u \sim_\phi u.fu \in \mathtt{Total}(t[u/x])\}$
  - $\mathtt{Total}(\pi_i(t)) = \{\pi_i(u) \mid u \in \mathtt{Total}(t)\}$
  - $\mathtt{Total}(tt') = \{uu' \mid u \in \mathtt{Total}(t), u' \in \mathtt{Total}(t')\}$
- $t \simeq_\phi t'$ *when for all* $u \in \mathtt{Total}(t)$, $u' \in \mathtt{Total}(t')$*, we have* $u \sim_\phi u'$

Note that $\simeq_\phi$ is not the same as $=_\phi$. For example,

$$\lambda f : \mathtt{nat} \to \mathtt{nat}.3 \simeq_{(\mathtt{even} \to \mathtt{nat}) \to \mathtt{nat}} \lambda f : \mathtt{even} \to \mathtt{nat}.3$$

This gives the basis of an operational semantics, although we do not pursue this here. We use it to characterise the term model in Section 5.

Proposition 3.5 $\simeq$ *is a congruence, that is, for* $x : \phi \vdash t'' : \psi$*, if* $t \simeq_\phi t'$ *then* $t''[t/x] \simeq_{\psi[t/x]} t''[t'/x]$.

4 Division by 2 revisited

As an illustration of how refinement types can provide a useful proof technique, we give the division by 2 example from the introduction again.

First, we give the inference rules for primitive and well-founded recursion over naturals, which combine the typing rules for recursion constants with the induction rules, into a single refinement typing. The sort is

$$\mathtt{natrec} : \tau, (\mathtt{nat} \to \tau \to \tau), \mathtt{nat} \to \tau$$

and the axiom schemas are (for each ϕ)

$$\frac{\Gamma \vdash n : \mathtt{nat} \quad \Gamma \vdash t : \phi[0] \quad \Gamma \vdash t' : \Pi_{n:\mathtt{nat}} \phi[n] \to \phi[\mathtt{succ}\; n]}{\Gamma \vdash \mathtt{natrec}\; t\; t'\; n : \phi[n]}$$

$$\frac{\Gamma \vdash t : \phi[0] \quad \Gamma \vdash t' : \Pi_{n:\mathtt{nat}} \phi[n] \to \phi[\mathtt{succ}\; n]}{\Gamma \vdash \mathtt{natrec}\; t\; t'\; 0 =_{\phi[0]} t}$$

$$\frac{\Gamma \vdash t : \phi[0] \quad \Gamma \vdash t' : \Pi_{n:\mathtt{nat}} \phi[n] \to \phi[\mathtt{succ}\; n]}{\Gamma \vdash \mathtt{natrec}\; t\; t'\; \mathtt{succ}\; n = t'\; n\; (\mathtt{natrec}\; t\; t'\; n)}$$

We define iteration from the more general recursion, as

$$\mathtt{natiter}\; t\; t'\; n = \mathtt{natrec}\; t\; (\lambda x : \mathtt{nat}.t')\; n$$

where $x \notin t'$.

The program is

$$\mathtt{div2} = \lambda n : \mathtt{nat}.\pi_1(\mathtt{div2}'\; n)$$

$$\mathtt{div2}' = \mathtt{natiter}\; \langle 0, 0\rangle\; (\lambda p : \mathtt{nat} \times \mathtt{nat}.\langle \pi_2 p, \pi_1 p + 1\rangle)$$

We prove that it satisfies the specification

$$\mathtt{div2} : \Pi_{n:\mathtt{nat}}(m : \mathtt{nat}) n = 2m \vee n = 2m + 1$$

$$\mathtt{div2}' : \Pi_{n:\mathtt{nat}} \Sigma_{(m:\mathtt{nat})\mathrm{n=2m \vee n=2m+1}}(m' : \mathtt{nat}) m + m' = n$$

In fact, there is little of interest in the main part of the proof. Since refinement types explicitly indicate the structure of the specification, this enables much of the proof to be carried out in a syntax-directed fashion. This would be useful for automation.

Write $\phi[n]$ as an abbreviation for $\Sigma_{m:(m:\mathtt{nat})\mathrm{n=2m \vee n=2m+1}}(m' : \mathtt{nat}) m + m' = n$.

$$\dfrac{\langle 0,0\rangle : \phi[0] \qquad \dfrac{\dfrac{\textit{see below}}{n : \mathtt{nat}, p : \phi[n] \vdash \langle \pi_2 p, \pi_1 p + 1\rangle : \phi[n+1]} \quad n : \mathtt{nat} \vdash \mathtt{nat} \times \mathtt{nat} \sqsubseteq \phi[n]}{n : \mathtt{nat} \vdash \lambda p : \mathtt{nat} \times \mathtt{nat}.\langle \pi_2 p, \pi_1 p + 1\rangle : \phi[n] \to \phi[n+1]}}{\dfrac{n : \mathtt{nat} \vdash \mathtt{natiter}\; \langle 0,0\rangle\; (\lambda p : \mathtt{nat} \times \mathtt{nat}.\langle \pi_2 p, \pi_1 p + 1\rangle)\; n \;:\; \Sigma_{(m:\mathtt{nat})\mathrm{n=2m \vee n=2m+1}}(m' : \mathtt{nat}) m + m' = n}{\lambda n : \mathtt{nat}.\mathtt{natiter}\; \langle 0,0\rangle\; (\lambda p : \mathtt{nat} \times \mathtt{nat}.\langle \pi_2 p, \pi_1 p + 1\rangle)\; n \;:\; \Pi_{n:\mathtt{nat}} \Sigma_{(m:\mathtt{nat})\mathrm{n=2m \vee n=2m+1}}(m' : \mathtt{nat}) m + m' = n}}$$

The proof continues with

$$\dfrac{\dfrac{n : \mathtt{nat}, n = 2\pi_1 p \vee n = 2\pi_1 p + 1 \vdash n + 1 = 2\pi_2 p \vee n + 1 = 2\pi_2 p + 1}{n : \mathtt{nat}, p : \phi[n] \vdash n + 1 = 2\pi_2 p \vee n + 1 = 2\pi_2 p + 1}}{n : \mathtt{nat}, p : \phi[n] \vdash \pi_2 p : (m : \mathtt{nat}) n + 1 = 2m \vee n + 1 = 2m + 1}$$

The remainder of the proof is arithmetic reasoning. In practice, we would use a theorem prover here.

$$\frac{n : \mathtt{nat}, p : \phi[n] \vdash \pi_1 p + \pi_2 p = n}{n : \mathtt{nat}, p : \phi[n] \vdash \pi_2 p + \pi_1 p + 1 = n + 1}$$

5 Models

We give interpretations of the calculus in Henkin models with per structure. As is usual in concrete models of applied lambda calculi, we must consider general Henkin models in order to get completeness. The per structure is to account for stratified equalities at different refinement types.

A Henkin model (with products) is an applicative structure

$$\langle \{A^\sigma\}, \{\mathtt{Proj}_1^{\sigma,\tau}\}, \{\mathtt{Proj}_2^{\sigma,\tau}\}, \{\mathtt{App}^{\sigma,\tau}\}, \mathtt{Const}\rangle$$

which is also extensional and satisfies the environment model condition. See [Mit96] for details. In addition, we require a function `Pred` which interprets predicates

$$\mathtt{Pred}(F) \subseteq A^{\tau_1\times\cdots\times\tau_n}, \text{ for } F : \mathtt{pred}\ \tau_1,\ldots,\tau_n$$

It is extensionality which allows us to interpret abstractions, pairs and the unit uniquely, up to equality in the appropriate per, and the environment model condition which gives us enough elements. To each type σ we ascribe the set A^σ, and use `Const` to map constants into the appropriate sets. For each constant $k : \gamma$, we have $\mathtt{Const}(k) \in A^\gamma$. The constant $k : \tau_1,\ldots,\tau_n \to \tau$ is mapped into the set $A^{\tau_1\times\cdots\times\tau_n\to\tau}$. Note that we do not have $A^{\sigma\times\tau} = A^\sigma \times A^\tau$ in general, but have an isomorphism mediated by projection functions, $\mathtt{Proj}_1^{\sigma,\tau} : A^{\sigma\times\tau} \to A^\sigma$, and $\mathtt{Proj}_2^{\sigma,\tau} : A^{\sigma\times\tau} \to A^\tau$. Similarly, $A^{\sigma\to\tau}$ is not $A^\sigma \to A^\tau$ in general, but may be embedded in it.

The idea is that refinement types over type σ are interpreted as pers over A^σ. Primitive types are interpreted as the identity per over their set. It is easy to see that, in fact, all types are interpreted as identities.

A Henkin model $\mathcal{A}$ models an applied theory when all constants and predicates are given an interpretation, and each axiom is true in the model.

However, terms, refinement types and propositions are all interpreted in context, so we must first define environments g for context Γ, written $g \vDash \Gamma$, where g is a tuple of elements in the domains of the pers for the refinement types in Γ. We define this recursively with the interpretation of refinement types and propositions :

$$\langle\rangle \vDash \langle\rangle$$

$$\langle g, a\rangle \vDash \Gamma, x : \phi \text{ when } g \vDash \Gamma \text{ and } a \in [\![\Gamma \vdash \phi]\!](g)$$

$$g \vDash \Gamma, P \text{ when } g \vDash \Gamma \text{ and } g \in [\![\Gamma \vdash P]\!]$$

For $g, g' \vDash \Gamma$ we define $g =_\Gamma g'$ in the obvious way.

The interpretations are given in the appendix.

To avoid questions of coherence, we interpret raw terms[1], and so raw propositions and refinement types too. We define the interpretation of the refinement type in context $\Gamma \vdash \phi$, for each environment $g \vDash \Gamma$, in figure 9. There is an apparent asymmetry in the definition of the product per for $\Sigma_{x:\phi}\psi$, but this is not so, for if ϕ is a well-formed refinement type in context Γ, then if $g[\![\Gamma]\!]g'$ we have $[\![\Gamma \vdash \phi]\!]g = [\![\Gamma \vdash \phi]\!]g'$.

The raw term in context $\Gamma \vdash t$ is interpreted in environment $g \vDash \Gamma$ as a subset (its 'total realizers') of A^σ. We interpret a raw proposition in context $\Gamma \vdash P$ as the set of environments $g \vDash \Gamma$ in which P holds.

We may now say what the semantic analogues of the judgements are. For per R, write $a \in R$ to mean $a\ R\ a$. We often write the relation as $=_R$. Define $\Gamma \vDash t : \phi$ when for all models $\mathcal{A}$, for all $g =_{[\![\Gamma]\!]} g'$, for all $a \in [\![\Gamma \vdash t]\!](g)$ and $a' \in [\![\Gamma \vdash t']\!](g')$, we have $a =_{[\![\Gamma\vdash\phi]\!](g)} a'$. In other words, the interpretation is unique up to the equality of the per. We say that $\Gamma \vDash P$ when $[\![\Gamma \vdash P]\!] = \{g \mid g \vDash \Gamma\}$. In particular, the refinement $\Gamma \vDash \phi \sqsubseteq \phi'$ is valid when for all $g \vDash \Gamma$, there is an inclusion of pers $[\![\Gamma \vdash \phi']\!](g) \subseteq [\![\Gamma \vdash \phi]\!](g)$.

[1] In fact, only well-structured terms are given an interpretation. For example, $(\lambda n : \mathtt{even}.n)*$ does not have a well-defined interpretation, but $(\lambda n : \mathtt{even}.n)3$ does, even though it is not syntactically well-formed.

We define $\Gamma \vDash \phi$ `wf` to mean for all $g =_{[\![\Gamma]\!]} g'$, $[\![\Gamma \vdash \phi]\!](g) = [\![\Gamma \vdash \phi]\!](g')$. Also, $\Gamma \vDash P$ `wf` is convenient notation for, for all $g =_{[\![\Gamma]\!]} g'$, $g \in [\![\Gamma \vdash P]\!] \iff g' \in [\![\Gamma \vdash P]\!]$.

First we give a substitution lemma. Since total terms are interpreted as singleton sets, we abuse notation and write $[\![u]\!]$ for both the set and its unique inhabitant.

Lemma 5.1 *(Substitution Lemma) For total terms u, and appropriate well-formed contexts, g, g',*

1. $[\![\Gamma, x : \phi, \Gamma' \vdash \psi]\!]\langle g, [\![u]\!], g'\rangle = [\![\Gamma, \Gamma'[u] \vdash \psi[u]]\!]\langle g, g'\rangle$
2. $\langle g, [\![u]\!], g'\rangle \in [\![\Gamma, x : \phi, \Gamma' \vdash P]\!] \iff \langle g, g'\rangle[\![\Gamma, \Gamma'[u] \vdash \psi[u]]\!]\langle g, g'\rangle$
3. $[\![\Gamma, x : \phi, \Gamma' \vdash t]\!]\langle g, [\![u]\!], g'\rangle = [\![\Gamma, \Gamma'[u] \vdash t[u]]\!]\langle g, g'\rangle$

Proof: Induct simultaneously over open terms, refinement types and propositions. ∎

Lemma 5.2 *Suppose $[\![t]\!] =_{[\![\phi]\!]} [\![t']\!]$. Note that we do not require $t, t' : \phi$. Then,*

1. *If $x : \phi \vDash \psi$ `wf` then $[\![\psi[t]]\!] = [\![\psi[t']]\!]$*
2. *If $x : \phi \vDash P$ `wf` then $\vDash P[t]$ iff $\vDash P[t']$*
3. *If $x : \phi \vDash t'' : \psi$ then $[\![t''[t]]\!] =_{[\![\psi[t]]\!]} [\![t''[t']]\!]$*

Proof: Simultaneous induction over terms, refinement types and propositions. ∎

Lemma 5.3 *For $\vDash t : \phi$, and $x : \phi \vDash P$ `wf`, if $[\![t]\!] \subseteq [\![x : \phi \vdash P]\!]$ then $\vDash P[t]$.*

Proof: Let $u \in \texttt{Total}(t)$, so $u \simeq_\phi t$. Then singleton $[\![u]\!] \subseteq [\![t]\!]$. By soundness of $t : \phi$, we have $[\![t]\!] =_{[\![\phi]\!]} [\![u]\!]$, so since $[\![u]\!] \subseteq [\![x : \phi \vdash P]\!]$, by Lemma 5.1, $\vDash P[u]$. Since $[\![t]\!] =_{[\![\phi]\!]} [\![u]\!]$, Lemma 5.2 gives $\vDash P[t]$. ∎

We have

Theorem 5.1 *(Soundness) If $\Gamma \vdash t : \phi$ then $\Gamma \vDash t : \phi$, if $\Gamma \vdash \phi$ `wf` then $\Gamma \vDash \phi$ `wf`, if $\Gamma \vdash P$ `wf` then $\Gamma \vDash P$ `wf`, and if $\Gamma \vdash P$ then $\Gamma \vDash P$.*

Proof: Simultaneous induction over all derivations.

The soundness of λ-β-EQ uses Lemma 5.3. Assume that $\Gamma, x : \phi \vDash t : \psi$, $\Gamma \vDash t' : \phi$, and let $g =_{[\![\Gamma]\!]} g'$, $a \in [\![\Gamma \vdash (\lambda x : \phi.t)t']\!](g)$, $a' \in [\![\Gamma \vdash t[t'/x]]\!](g')$. Then $a =_{[\![\Gamma \vdash \psi[t'/x]]\!](g)} a'$. The soundness of $\forall$-ELIM is similar. ∎

We want to show that if a particular judgement holds in all the models (of the relevant signature) then it is provable. Although we do not have minimal models (due to having propositional assumptions), we can still use a term model construction, using a slight generalisation of the standard 'consistency implies satisfiability' argument.

First we generalise consistency and satisfiability from sets of closed propositions to arbitrary contexts. Say that context Γ is

- *consistent*, when $\Gamma \nvdash$ `false`
- *satisfiable*, when there exists a model $\mathcal{A}$ and environment g in $\mathcal{A}$ such that $g \vDash_{\mathcal{A}} \Gamma$

In the case that Γ is a context of closed propositions, these reduce to the usual definitions. Now we want to show that $\Gamma \vDash P \Rightarrow \Gamma \vdash P$, so suppose $\Gamma \nvdash P$. Then $\Gamma, \neg P$ is consistent and so, by assumption, is satisfiable. Hence $\Gamma \nvDash P$. We must stipulate that the proposition is well-formed because it is possible that $t =_\phi t$ is true in the model, but $t : \phi$ is not provable.

We must be careful with the statement of completeness. This is because it is possible for non well-formed terms to have a unique interpretation, and so, semantically, have a refinement type. For example, $(\lambda n : \texttt{even}.*)3$ is interpreted as the unique inhabitant of unit type, but cannot be typed in the system. Also, $\lambda f : \texttt{nat} \to \texttt{nat}.3$ has the refinement type $(\texttt{even} \to \texttt{nat}) \to \texttt{nat}$ in the model, but not in the calculus.

What we can show however, is that if a term t has refinement type ϕ in the model, then there exists a term t' which does satisfy refinement type ϕ, such that $t \simeq_\phi t'$, and so $\models t =_\phi t'$. We write this as $t \sim \phi$.

Theorem 5.2 *(Completeness) For* $\Gamma \vdash P$ $\texttt{wf}$*, if* $\Gamma \models P$ *then* $\Gamma \vdash P$*. For* $\Gamma \vdash \phi$ $\texttt{wf}$*, if* $\Gamma \models t : \phi$ *then* $\Gamma \vdash t \sim \phi$*.*

Proof: Let Γ be a consistent context. We sketch the construction of a model $\mathcal{A}$ and environment $g \models_{\mathcal{A}} \Gamma$ below. We will just consider unary predicate symbols.

1. Let $A = \{P \mid \Gamma \vdash P \ \texttt{wf}\}$. Construct a maximal consistent Henkin theory Δ such that $\{P \mid \Gamma \vdash P\} \subseteq \Delta \subseteq A$.

   A *Henkin*[2] *theory* T, is a collection of sentences closed under derivation, such that if the proposition $\exists x : \phi.P$ is in T, then $P[t/x]$ is in T for some $t : \phi$.

   The construction of the Henkin theory involves adding constants $c_{\phi,P} : \tau$, and axioms $c_{\phi,P} : \phi$, $P[c_{\phi,P}]$, for each refinement type $\Gamma \vdash \phi : \texttt{Ref}\ \tau$, and propositions $\Gamma \vdash P$ $\texttt{wf}$, for each $\exists x : \phi.P$ in T.

   The existence of maximal consistent Henkin extensions follows using a standard argument from Zorn's Lemma. See [vD94] for details. Note that for all $\Gamma \vdash P$ $\texttt{wf}$ exactly one of P, $\neg P$ is in Δ.

2. Let Γ_∞ be a set of infinitely many declarations $x : \phi$ for each closed inhabited ϕ, *i.e.* for which there exists t such that $\vdash t : \phi$. This is so that we can construct a model from open terms.

3. Define a family of logical equivalence relations, $R^\tau_\Delta(\_,\_)$ for each type τ, on open terms of $\lambda^{\times\to}$with type τ, $\{t \mid \Gamma_\infty \vdash t : \tau\}$, as $R^\tau_\Delta(t,t') \iff \Delta_\infty \vdash t =_\tau t'$.

4. Define A^τ as the set of equivalence classes with respect to R of open terms in τ, and construct a Henkin model $\mathcal{A}$, by interpreting constants syntactically. Interpret predicates $F : \texttt{pred}\ \tau$ as $\{[t] \mid F(t) \in \Delta\}$.

5. Hence $\mathcal{A}$ models Δ, so it models Γ. The set Δ gives rise to an environment for Γ in $\mathcal{A}$. Now for every $x : \phi \in \Gamma$ there is a proposition $x =_{\phi'} t \in \Delta$, where t is closed, and ϕ' is a closed instantiation of ϕ. So Δ is effectively a set of closed propositions.

   We use $\hat{g}$ below for the tuple of elements t and g for the total elements u.

   For $x : \phi \in \Gamma$, define the corresponding element in the environment as $g_x = [u]^\tau$, where $x =_{\phi'} t \in \Delta$ and u is any total realizer of t (which does exist).

6. We abbreviate $[\![\_]\!](g)$ with $[\![\_]\!]$ and $g \in [\![\Gamma \vdash P]\!]$ by $\mathcal{A} \models P$.

   For $\Gamma \vdash P$ $\texttt{wf}$, prove that $\mathcal{A} \models P \iff \Delta_\infty \vdash P \iff \Delta \vdash P$. The second equivalence is because for each $x : \phi \in \Gamma_\infty$, $x \notin P$ and ϕ is inhabited.

In order to prove the first equivalence in step 5, we need a number of lemmas. First we characterise the interpretation of terms and refinement types in the term model, $\mathcal{A}$. In the following, the semantic interpretation $[\![\_]\!]$ is to be understood as being in $\mathcal{A}$. As pointed out in Section 3.5, the pers do not correspond exactly to equality.

Lemma 5.4

$$[u] \ \ [\![\Gamma \vdash \phi]\!](g)\ [u'] \iff \Delta_\infty \vdash u \sim_{\phi[g/\Gamma]} u'$$

[2]Nothing to do with Henkin models.

Lemma 5.5 *For closed raw terms t,*

$$[\![t]\!] = \mathtt{Total}(t), \text{ that is, } \{[u] \mid \Delta_\infty \vdash u : \sigma,\ u \in \mathtt{Total}(t)\}$$

Hence $t =_\phi t'$ in the term model is $t \simeq_\phi t'$.

Proof of Theorem 5.2 cont.:

We now prove that for all $\Gamma \vdash J$ $\mathtt{wf}$, $\mathcal{A} \vDash J \iff \Delta_\infty \vdash \tilde{J}$ by simultaneous induction over all judgements. By $\tilde{J}$ we mean replacing $t : \phi$ with $t \sim \phi$.

By well-formedness of $t : \phi$ we mean well-formedness of ϕ. Recall that this is all with respect to an environment g for context Γ.

The inductive ordering is

$$P, P' < P \supset P'$$

$$\phi, P[t] < \forall x : \phi.P$$

$$\phi < t =_\phi t'$$

$$\phi, \phi' < \phi \sqsubseteq \phi'$$

$$\phi, \psi[t] < \Sigma_{x:\phi}\psi$$

$$\phi, \psi[t] < \Pi_{x:\phi}\psi$$

$$\phi, \psi[t] < (x : \phi)P$$

where the ϕ case means the truth or otherwise of $t : \phi$ for each t, *i.e.* for each t, $\vDash t : \phi \Rightarrow \vdash t \sim \phi$.

- $\mathcal{A} \vDash P \wedge Q \iff \mathcal{A} \vDash P$ and $\mathcal{A} \vDash Q \iff \Delta_\infty \vdash P$ and $\Delta_\infty \vdash Q \iff \Delta_\infty \vdash P \wedge Q$
- $\mathcal{A} \nvDash \mathtt{false}$, and $\Delta_\infty \nvdash \mathtt{false}$ since consistent
- $\mathcal{A} \vDash \forall x : \phi P$ when for all $a \in [\![\phi]\!]$, $a \in [\![x : \phi \vdash P]\!]$. Now if $\Gamma \vdash t : \phi$, by soundness we have $\langle g, [\![t]\!]\rangle \subseteq [\![\Gamma, x : \phi \vdash P]\!]$, and so by Lemma 5.3, $\mathcal{A} \vDash P[t]$, and by the inductive hypothesis, $\Delta_\infty \vdash P[t]$. In other words, for all $\Gamma \vdash t : \phi$ we have $\Delta_\infty \vdash P[t]$. Thus $\Delta_\infty \vdash \forall x : \phi.P[x]$, for if not, we would have $\Delta_\infty \vdash \exists x : \phi.\neg P$, and since Δ_∞ is a Henkin theory, $\Delta_\infty \vdash \neg P[t]$ for some $\Gamma \vdash t : \phi$, which contradicts the above.

  Conversely, suppose $\Delta_\infty \vdash \forall x : \phi.P$. Now let $[u] \in [\![\phi]\!]$. We cannot use completeness to infer that $u : \phi$ but we do get $\Delta_\infty \vdash u \sim \phi$, that is, there exists $t : \phi$ such that $u \simeq_\phi t$, and so $[\![u]\!] =_{[\![\phi]\!]} [\![t]\!]$.

  Now by ($\forall$-ELIM), $\Delta_\infty \vdash P[t]$ so by the inductive hypothesis, $\mathcal{A} \vDash P[t]$, and by Lemma 5.2, $\mathcal{A} \vDash P[u]$. Hence $\mathcal{A} \vDash \forall x : \phi.P$.
- $\mathcal{A} \vDash F(t)$ when $[\![t]\!] \subseteq \mathtt{Pred}(F)$. Now predicates only apply to typable terms, so $t : \sigma$ say. Although t might not be total, for all $u \in \mathtt{Total}(t)$, $u =_\sigma t$. By definition of $u \in \mathtt{Pred}(F)$ in the term model, $\Delta_\infty \vdash F(u)$, so by CONG, $\Delta_\infty \vdash F(t)$. Conversely, if $\Delta_\infty \vdash F(t)$, then $\Delta_\infty \vdash F(u)$ for all $u \in \mathtt{Total}(t)$, so $[\![t]\!] \subseteq \mathtt{Pred}(F)$.
- $\mathcal{A} \vDash t =_\phi t'$ is $t \simeq_\phi t'$, in other words, for all $u \in \mathtt{Total}(t)$ and $u' \in \mathtt{Total}(t')$, we have $u \sim_\phi u'$.

  We must show that $t \simeq_\phi t' \iff \Delta_\infty \vdash t =_\phi t'$. Recall that we have assumed $t, t' : \phi$ for well-formedness of the proposition $t =_\phi t'$.

  - For all $\Gamma \vdash t, t' : \mathbf{1}$ we have $t \simeq_{\mathbf{1}} t'$ and $\Gamma \vdash t =_{\mathbf{1}} t'$.

- Suppose $\Delta_\infty \vdash t =_\gamma t'$. Then since $\Delta_\infty \vdash g \simeq_\Gamma \Gamma$ (with the obvious meaning), and since $\simeq$ is a congruence (Proposition 3.5), we have $\Delta_\infty \vdash t[g/\Gamma] \simeq_\gamma t'[g/\Gamma]$, so $\mathcal{A} \vDash t[g/\Gamma] =_\gamma t[g/\gamma]$. Since $[\![t]\!] = [\![t'[g/\Gamma]]\!]$, and $[\![t']\!] = [\![t'[g/\Gamma]]\!]$, we get $\mathcal{A} \vDash t =_\gamma t'$.

  Conversely, suppose $t \simeq_\gamma t'$. Now Δ must contain the proposition $x = \hat{g}_x$ for each x in Γ. Then, since closed $t[\hat{g}/\Gamma] : \gamma$, we can deduce by Proposition 3.2 that there exists some total t'' such that $u \in \texttt{Total}(t[\hat{g}/\Gamma]) \Rightarrow u =_\gamma t''$. Hence $\Delta_\infty \vdash t[\hat{g}/\Gamma] =_\gamma t''$ and similarly, $\Delta_\infty \vdash t'[\hat{g}/\Gamma] =_\gamma t''$. We get the same t'' since $t[\hat{g}/\Gamma] \simeq_\gamma t'[\hat{g}/\Gamma]$ (using Proposition 3.5). Then since $\Delta_\infty \vdash t =_\gamma t[\hat{g}/\Gamma]$ and $\Delta_\infty \vdash t' =_\gamma t'[\hat{g}/\Gamma]$, we get $\Delta_\infty \vdash t =_\gamma t'$.

- Suppose $t \simeq_{\Sigma_{x:\phi}\psi} t'$. We prove that $\pi_2(t) \simeq_{\psi[\pi_1(t)]} \pi_2(t')$.

  Suppose $u_2 \in \pi_2(t)$, $u_2' \in \pi_2(t')$. Then there exist u_1, u_1' such that $\langle u_1, u_2\rangle \sim_{\Sigma_{x:\phi}\psi} \langle u_1', u_2'\rangle$, so $u_2 \sim_{\psi[u_1]} u_2'$. Since $u_1 \in \texttt{Total}(\pi_1(t))$, by soundness of $\pi_1(t) : \phi$, $u_1 \simeq_\phi \pi_1(t)$, so by Lemma 5.2, $u_2 \sim_{\psi[\pi_1(t)]} u_2'$. Hence, $\pi_2(t) \simeq_{\psi[\pi_1(t)]} \pi_2(t')$. Similarly, we can show that $\pi_1(t) \simeq_\phi \pi_1(t')$ By induction, this is equivalent to $\pi_1(t) =_\phi \pi_1(t')$ and $\pi_2(t) =_{\psi[\pi_1(t)]} \pi_2(t')$, and so $\Delta_\infty \vdash t =_{\Sigma_{x:\phi}\psi} t'$.

- First note that $t_1 =_{\Pi_{x:\phi}\psi} t_1' \iff$ for all $t_2, t_2' : \phi$, $t_2 =_\phi t_2' \supset t_1 t_2 =_{\psi[t_1]} t_1' t_2'$.

  Now suppose $t_1 \simeq_{\Pi_{x:\phi}\psi} t_1'$, that is, for all $u_1 \in \texttt{Total}(t_1)$, $u_1' \in \texttt{Total}(t_1')$, $u_1 \sim_{\Pi_{x:\phi}\psi} u_1'$. Then if $t_2 =_\phi t_2'$, for $u_2 \in \texttt{Total}(t_2)$, $u_2' \in \texttt{Total}(t_2')$, we have $u_2 \sim_\phi u_2'$, so $\texttt{App}(u_1, u_2) \sim_{\psi[u_1]} \texttt{App}(u_1', u_2')$, so by Lemma 5.2, $\texttt{App}(u_1, u_2) \sim_{\psi[t_1]} \texttt{App}(u_1', u_2')$. By the inductive hypothesis, $t_1 t_2 =_{\psi[t_1]} t_1' t_2'$. Thus, for all $t_2 =_\phi t_2'$, $t_1 t_2 =_{\psi[t_1]} t_1' t_2'$, and so $t_1 =_{\Pi_{x:\phi}\psi} t_1'$.

- Suppose $t \simeq_{(x:\phi)P} t'$. So for all $u \in \texttt{Total}(t)$, $u' \in \texttt{Total}(t')$, $u \sim_{(x:\phi)P} u'$. Therefore $u \sim_\phi u'$, and by the inductive hypothesis, $t =_\phi t'$. Now in assuming well-formedness, we have $t, t' : (x : \phi)P$ and so by CONG-EQ, $t =_{(x:\phi)P} t'$.

• $\mathcal{A} \vDash \phi \sqsubseteq \phi'$

  If ϕ' is inhabited then $x : \phi' \in \Gamma_\infty$ so $\Delta_\infty \vdash x : \phi'$ and we can deduce (see Remark 3.2) that $\Delta_\infty \vdash \phi \sqsubseteq \phi'$.

  If ϕ' is not inhabited we can prove $\Delta_\infty \vdash \phi \sqsubseteq \phi'$ directly. Now to say that ϕ' is uninhabited is to say that there does not exist a closed term $t : \phi'$. Since Δ is a maximally consistent Henkin extension, we must have $\neg\exists x : \phi'.\texttt{true} \in \Delta$. Hence $\Delta_\infty, x : \phi' \vdash \texttt{false}$. Hence by (REFTYPE-L-REF), $\Delta_\infty \vdash (x : \phi')\texttt{false} \sqsubseteq \phi'$. Now Proposition 3.4 gives us $\Delta_\infty \vdash \phi \sqsubseteq (x : \phi')\texttt{false}$ and so using (TRANS-REF) $\Delta_\infty \vdash \phi \sqsubseteq \phi'$.

  Conversely, if $\Delta_\infty \vdash \phi \sqsubseteq \phi$, then using Proposition 3.1 and the inductive hypothesis on ϕ and ϕ', we get $\mathcal{A} \vDash \phi \sqsubseteq \phi'$.

• $\mathcal{A} \vDash t : \phi$, *i.e.* $t \simeq_\phi t'$. We show $\Delta_\infty \vdash t \sim \phi$, that is, there exists $\Delta_\infty \vdash t' : \phi$ such that $t \simeq_\phi t'$.

  - $\mathcal{A} \vDash t : \mathbf{1}$ when for all $u, u' \in \texttt{Total}(t)$, $\Delta_\infty \vdash u \sim_{\mathbf{1}} u'$, so $u =_{\mathbf{1}} u'$. Hence for any $u \in \texttt{Total}(t)$, we have $t \simeq_{\mathbf{1}} u : \mathbf{1}$, and so $\Delta_\infty \vdash t \sim \mathbf{1}$.
  - The $\mathcal{A} \vDash t : \gamma$ case is similar to that of $\mathbf{1}$.
  - If $t \simeq_{\Sigma_{x:\phi}\psi} t$, then using reasoning similar to the $t =_{\Sigma_{x:\phi}\psi} t'$ case above, $\pi_1(t) \simeq_\phi \pi_1(t)$ and $\pi_2(t) \simeq_{\psi[\pi_1(t)]} \pi_2(t)$. By the inductive hypothesis, we get some t_1' such that $\pi_1(t) \simeq_\phi t_1' : \phi$, and t_2' such that $\pi_2(t) \simeq_{\psi[\pi_1(t)]} t_2'$. Using Lemma 5.2, we deduce that $\pi_2(t) \simeq_{\psi[t_1']} t_2'$. Hence, $t \simeq_{\Sigma_{x:\phi}\psi} \langle t_1', t_2'\rangle$, and so $\langle t_1', t_2'\rangle \sim \Sigma_{x:\phi}\psi$.
  - $\mathcal{A} \vDash t : \Pi_{x:\phi}\psi$. Suppose ϕ is inhabited. Then $x : \phi \in \Gamma_\infty$ so $\mathcal{A} \vDash tx : \psi$, and by induction, there exists a $t' : \psi$ such that $tx \simeq_\psi t'$. Hence $\Delta_\infty \vdash t \simeq_{\Pi_{x:\phi}\psi} \lambda x : \phi.t'$ and $\Delta_\infty \vdash t : \Pi_{x:\phi}\psi$.

    If ϕ is not inhabited, then we can prove $\Delta_\infty \vdash t : \Pi_{x:\phi}\psi$ directly (which implies $t \sim \Pi_{x:\phi}\psi$). Since t must be well-structured for its interpretation to be well-defined, we can use Proposition 3.3.

– $\mathcal{A} \models t : (x : \phi)P \iff \mathcal{A} \models t : \phi$ and $\mathcal{A} \models P[t]$. The $\Rightarrow$ follows from Lemma 5.3. For $\Leftarrow$, use soundness of $x : \phi \vdash P$ `wf` and $t : \phi$.
Then, since $\phi < (x : \phi)P$, there exists $t' : \phi$ such that $t \simeq_\phi t'$. By Lemma 5.2, $\mathcal{A} \models P[t']$. Now we assumed $\Gamma \vdash (x : \phi)P$ `wf` and $\Gamma \vdash t : \phi$, so $\Gamma \vdash P[t']$ `wf`, and since $P[t'] < (x : \phi)P$, we get $\Delta_\infty \vdash P[t']$. Hence $\Delta_\infty \vdash t' : (x : \phi)P$, and $\Delta_\infty \vdash t \sim (x : \phi)P$.

∎

Corollary 5.1 *The calculus is a conservative extension of $\lambda^{\times\rightarrow}$: If $\Gamma \vdash t =_\tau t'$ is a well-formed equation in $\lambda^{\times\rightarrow}$, then it is provable in $\lambda^{\times\rightarrow}$, if and only if it is provable in the calculus of refinement types.*

Proof: Both calculi are complete for the class of Henkin Models. ∎

Similarly,

Corollary 5.2 *The calculus is a conservative extension of first-order logic : If $\Gamma \vdash P$ `wf` does not contain any refinement types, then it is provable in first-order logic, if and only if it is provable in the calculus of refinement types.*

Proof: Both systems are complete for the class of Henkin Models. ∎

The significance of these corollaries is that we are free to use the specification language for proving program equivalences, and for reasoning about programs using the program logic, in the knowledge that it faithfully reflects the equality in the underlying programming language, and proofs in the program logic.

6 Conclusions

We have described the refinement type methodology of specification. This is a way of combining the type system of a programming language with a program logic to give a specification language. This is an alternative to approaches which rely on encoding a logic into an expressive type theory, and those which simply use a program logic.

The two-level nature of the calculus suggests the construction of a modular tool in which checking program correctness is a combination of type checking and theorem proving. The modularity would come from constructing a 'specification checker' from an existing theorem prover and a type checker, for the program logic and programming language respectively. Indeed, this is similar to what is done in the interactive proof development systems, Nuprl and PVS.

The calculus could provide a foundation for other specification based formalisms. In the proof that `div2` satisfied its specification we used the proof for `div2'`. There is an implicit element of refinement here. We envisage separate extensions to transformation and refinement calculi, and an embedding in a calculus of full program annotations.

Although we give a refinement relation $\phi \sqsubseteq \phi'$ on specifications, this does not constitute a full *refinement calculus* (such as in [Mor94]). The idea there is to internalise specifications into programs and consider a refinement relation on mixtures of specification and program. This is carried out in the author's forthcoming PhD thesis. The structure imposed on specifications would help to partially automate verification and derivation.

By extending the type system of $\lambda^{\times\rightarrow}$ to refinement types, we gain a simple notion of program annotation, where variables on abstractions are labelled with logical information. Expressing local contextual information is useful, as we often need to be able to make use of facts which are true at some point in a program. For example, if it is known that variable n must be within certain bounds, then a programmer (or compiler) may be able to perform some partial evaluation or optimisation.

The ability to express equality at a refinement type is useful in program transformation. For example, we might want to transform a function of type `nat` $\rightarrow$ `nat`, with the prescription "maintain value on evens, and improve on odds (in some way)". We can express (part of) this by saying that the terms are equal at the refinement type `even` $\rightarrow$ `nat`.

This account of specifications which brings equality to the fore should be especially useful in data refinement, where it is natural to consider different equalities at the abstract and concrete types.

We believe that the principles outlined here are general enough to be applied to structures other than those traditionally studied — data flow diagrams for example. Since the logic is arbitrary (up to a point) we are not constrained by the type theory. It would be an interesting line of research to see how type-theoretic and semantic ideas could help there.

Acknowledgements

This work has benefited from discussions with my supervisors — Gordon Plotkin, John Power and Marcelo Fiore. Conversations with Alex Bunkenburg, Healfdene Goguen and James McKinna have also been useful.

References

[Asp95] David Aspinall. Subtyping with singleton types. In *Proceedings of Computer Science Logic '94*, volume 933 of *LNCS*, 1995.

[BM92] R. Burstall and J. McKinna. Deliverables : a categorical approach to program construction in type theory. LFCS Technical Report ECS-LFCS-92-242, Department of Computer Science, University of Edinburgh, 1992.

[Bur92] G.L. Burn. A logical framework for program analysis. In J. Launchbury and P. Sansom, editors, *Proceedings of the 1992 Glasgow Functional Programming Workshop*, pages 30–42. Springer-Verlag Workshops in Computer Science series, 6–8 July 1992.

[Fef85] Solomon Feferman. A theory of variable types. *Revista Colombiana de Matématicas*, 19, 1985.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIG-PLAN'91 Symposium on Language Design and Implementation*, pages 268–277. ACM Press, 1991.

[Hay94a] Susumu Hayashi. Logic of refinement types. In *Types for Proofs and programs*, volume 806 of *Lecture Notes in computer Science*. Springer Verlag, 1994.

[Hay94b] Susumu Hayashi. Singleton, union, and intersection types for program extraction. *Information and Computation*, 109:174–210, 1994.

[Jen91] Thomas Jensen. Strictness analysis in logical form. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, volume 523 of *LNCS*, pages 352–366, 1991.

[Luo91] Z. Luo. Program specification and data refinement in type theory. LFCS Technical Report ECS-LFCS-90-131, Department of Computer Science, University of Edinburgh, 1991.

[McK92] James McKinna. *Deliverables : A Categorical Approach to Program Development in Type Theory.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[Mit96] J. Mitchell. *Foundations for Programming Languages.* Foundations of Computing Series. MIT Press, 1996.

[Mor94] C. Morgan. *Programming from Specifications.* Prentice Hall, 1994.

[NN88] Hanne Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *Monographs on Computer Science*. Oxford University Press, 1990.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, 1993.

[vD94] Dirk van Dalen. *Logic and Structure*. Springer-Verlag, 1994.

A Rules of the calculus

$$\vdash \langle\rangle \ \mathtt{wf} \qquad \text{(EMPTY-WF)}$$

$$\frac{\Gamma \vdash \phi \ \mathtt{wf}}{\vdash \Gamma, x : \phi \ \mathtt{wf}} \ x \notin \Gamma \qquad \text{(VARCON-WF)}$$

$$\frac{\Gamma \vdash P \ \mathtt{wf}}{\vdash \Gamma, P \ \mathtt{wf}} \qquad \text{(PROPCON-WF)}$$

Figure 1: Well-formedness of contexts

$$\frac{\vdash \Gamma \ \mathtt{wf}}{\Gamma \vdash \mathbf{1} : \mathtt{Ref} \ \mathbf{1}} \qquad (\mathbf{1}\text{-WF})$$

$$\frac{\Gamma \vdash \phi : \mathtt{Ref} \ \sigma \quad \Gamma, x : \phi \vdash \psi : \mathtt{Ref} \ \tau}{\Gamma \vdash \Sigma_{x:\phi}\psi : \mathtt{Ref} \ \sigma \times \tau} \qquad (\Sigma\text{-WF})$$

$$\frac{\Gamma \vdash \phi : \mathtt{Ref} \ \sigma \quad \Gamma, x : \phi \vdash \psi : \mathtt{Ref} \ \tau}{\Gamma \vdash \Pi_{x:\phi}\psi : \mathtt{Ref} \ \sigma \to \tau} \qquad (\Pi\text{-WF})$$

$$\frac{\Gamma \vdash \phi : \mathtt{Ref} \ \tau \quad \Gamma, x : \phi \vdash P \ \mathtt{wf}}{\Gamma \vdash (x : \phi)P : \mathtt{Ref} \ \tau} \qquad \text{(REFTYPE-WF)}$$

$$\frac{\vdash \Gamma \ \mathtt{wf}}{\Gamma \vdash \gamma : \mathtt{Ref} \ \gamma} \ (\gamma \text{ primitive}) \qquad \text{(PRIMTYPE-WF)}$$

Figure 2: Well-formedness of refinement types

$$\frac{\vdash \Gamma \ \mathtt{wf}}{\Gamma \vdash \mathtt{false}\ \mathtt{wf}} \qquad (\mathtt{false}\text{-WF})$$

$$\frac{\Gamma \vdash P \ \mathtt{wf} \quad \Gamma \vdash P' \ \mathtt{wf}}{\Gamma \vdash P \supset P' \ \mathtt{wf}} \qquad (\supset\text{-WF})$$

$$\frac{\Gamma, x : \phi \vdash P \ \mathtt{wf}}{\Gamma \vdash \forall x : \phi . P \ \mathtt{wf}} \qquad (\forall\text{-WF})$$

$$\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash F(t_1, \ldots, t_n) \ \mathtt{wf}} \qquad (F : \mathtt{pred}\ \tau_1, \ldots, \tau_n) \qquad (\text{PRED-WF})$$

$$\frac{\Gamma \vdash t : \phi \quad \Gamma \vdash t' : \phi}{\Gamma \vdash t =_\phi t' \ \mathtt{wf}} \qquad (=\text{-WF})$$

$$\frac{\Gamma \vdash \phi : \mathtt{Ref}\ \tau \quad \Gamma \vdash \phi' : \mathtt{Ref}\ \tau}{\Gamma \vdash \phi \sqsubseteq \phi' \ \mathtt{wf}} \qquad (\sqsubseteq\text{-WF})$$

Figure 3: Well-formedness of propositions

$$\frac{\vdash \Gamma, x:\phi, \Gamma' \ \mathtt{wf}}{\Gamma, x:\phi, \Gamma' \vdash x:\phi} \qquad \text{(VAR)}$$

$$\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash k(t_1, \ldots, t_n) : \tau} \ (k : \tau_1, \ldots, \tau_n \to \tau \text{ primitive}) \qquad \text{(CONST)}$$

$$\frac{\vdash \Gamma \ \mathtt{wf}}{\Gamma \vdash * : \mathbf{1}} \qquad \text{(UNIT)}$$

$$\frac{\Gamma \vdash t : \phi \quad \Gamma \vdash t' : \psi[t/x] \quad \Gamma, x:\phi \vdash \psi \ \mathtt{wf}}{\Gamma \vdash \langle t, t' \rangle : \Sigma_{x:\phi}\psi} \qquad \text{(PAIR)}$$

$$\frac{\Gamma, x:\phi \vdash t : \psi}{\Gamma \vdash \lambda x:\phi.t : \Pi_{x:\phi}\psi} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash t : \Sigma_{x:\phi}\psi}{\Gamma \vdash \pi_1(t) : \phi} \qquad \text{(PROJ1)}$$

$$\frac{\Gamma \vdash t : \Sigma_{x:\phi}\psi}{\Gamma \vdash \pi_2(t) : \psi[\pi_1(t)/x]} \qquad \text{(PROJ2)}$$

$$\frac{\Gamma \vdash t : \Pi_{x:\phi}\psi \quad \Gamma \vdash t' : \phi}{\Gamma \vdash tt' : \psi[t'/x]} \qquad \text{(APP)}$$

$$\frac{\Gamma \vdash t : \phi \quad \Gamma \vdash P[t/x]}{\Gamma \vdash t : (x:\phi)P} \qquad \text{(REFTYPE-INTRO)}$$

$$\frac{\Gamma \vdash t =_\phi t'}{\Gamma \vdash t : \phi} \qquad \text{(EQ)}$$

$$\frac{\Gamma \vdash t : \phi' \quad \Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash t : \phi} \qquad \text{(WEAKEN)}$$

Figure 4: Refinement typings

$$\frac{\Gamma, x:\phi \vdash t:\psi \quad \Gamma \vdash t':\phi}{\Gamma \vdash (\lambda x:\phi.t)t' =_{\psi[t'/x]} t[t'/x]} \qquad (\lambda\text{-}\beta\text{-EQ})$$

$$\frac{\Gamma, x:\phi \vdash tx:\psi}{\Gamma \vdash \lambda x:\phi.tx =_{\Pi_{x:\phi}\psi} t}\ x \notin t \qquad (\lambda\text{-}\eta\text{-EQ})$$

$$\frac{\Gamma \vdash t_1:\phi_1 \quad \Gamma \vdash t_2:\phi_2}{\Gamma \vdash \pi_i\langle t_1, t_2\rangle =_{\phi_i} t_i} \qquad (\pi\text{-}\beta\text{-EQ})$$

$$\frac{\Gamma \vdash \pi_1(t):\phi \quad \Gamma \vdash \pi_2(t):\psi[\pi_1(t)/x]}{\Gamma \vdash \langle\pi_1(t), \pi_2(t)\rangle =_{\Sigma_{x:\phi}\psi} t} \qquad (\pi\text{-}\eta\text{-EQ})$$

$$\frac{\Gamma \vdash t:\mathbf{1}}{\Gamma \vdash t =_{\mathbf{1}} *} \qquad (*\text{-}\eta\text{-EQ})$$

Figure 5: $\beta\eta$ equality rules

$$\Gamma \vdash t =_\phi t' \qquad (Ax \rhd \Gamma \vdash t =_\phi t') \qquad (\text{AXIOM-EQ})$$

$$\frac{\Gamma \vdash \phi' \sqsubseteq \phi \quad \Gamma, x:\phi \vdash t =_\psi t'}{\Gamma \vdash \lambda x:\phi.t =_{\Pi_{x:\phi}\psi} \lambda x:\phi'.t'} \qquad (\text{ABS-}\xi\text{-EQ})$$

$$\frac{\Gamma \vdash t_1 =_{\Pi_{x:\phi}\psi} t_1' \quad \Gamma \vdash t_2 =_\phi t_2'}{\Gamma \vdash t_1 t_2 =_{\psi[t_2/x]} t_1' t_2'} \qquad (\text{APP-}\xi\text{-EQ})$$

$$\frac{\Gamma \vdash t_1 =_\phi t_1' \quad \Gamma \vdash t_2 =_{\psi[t_1/x]} t_2'}{\Gamma \vdash \langle t_1, t_2\rangle =_{\Sigma_{x:\phi}\psi} \langle t_1', t_2'\rangle} \qquad (\text{PAIR-}\xi\text{-EQ})$$

$$\frac{\Gamma \vdash t =_{\Sigma_{x:\phi}\psi} t'}{\Gamma \vdash \pi_1(t) =_\phi \pi_1(t')} \qquad (\text{PROJ1-}\xi\text{-EQ})$$

$$\frac{\Gamma \vdash t =_{\Sigma_{x:\phi}\psi} t'}{\Gamma \vdash \pi_2(t) =_{\psi[\pi_1(t)/x]} \pi_2(t')} \qquad (\text{PROJ2-}\xi\text{-EQ})$$

$$\frac{\Gamma \vdash t:\phi}{\Gamma \vdash t =_\phi t} \qquad (\text{REFL-EQ})$$

$$\frac{\Gamma \vdash t =_\phi t' \quad \Gamma \vdash P[t/x] \quad \Gamma, x:\phi \vdash P\ \texttt{wf}}{\Gamma \vdash t =_{(x:\phi)P} t'} \qquad (\text{CONG-EQ})$$

$$\frac{\Gamma \vdash t =_\phi t'}{\Gamma \vdash t' =_\phi t} \qquad (\text{SYMM-EQ})$$

$$\frac{\Gamma \vdash t =_\phi t' \quad \Gamma \vdash t' =_\phi t''}{\Gamma \vdash t =_\phi t''} \qquad (\text{TRANS-EQ})$$

Figure 6: Other equalities

$$\frac{\vdash \Gamma \ \texttt{wf}}{\Gamma \vdash \mathbf{1} \sqsubseteq \mathbf{1}} \quad (\mathbf{1}\text{-REF})$$

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma, x : \phi \vdash \psi \sqsubseteq \psi'}{\Gamma \vdash \Sigma_{x:\phi}\psi \sqsubseteq \Sigma_{x:\phi'}\psi'} \quad (\Sigma\text{-REF})$$

$$\frac{\Gamma \vdash \phi' \sqsubseteq \phi \quad \Gamma, x : \phi \vdash \psi \sqsubseteq \psi'}{\Gamma \vdash \Pi_{x:\phi}\psi \sqsubseteq \Pi_{x:\phi'}\psi'} \quad (\Pi\text{-REF})$$

$$\frac{\Gamma \vdash \phi \sqsubseteq \psi \quad \Gamma, x : \psi \vdash P}{\Gamma \vdash (x : \phi)P \sqsubseteq \psi} \quad (\text{REFTYPE-L-REF})$$

$$\frac{\Gamma, x : \psi, Q \vdash x : \phi}{\Gamma \vdash \phi \sqsubseteq (x : \psi)Q} \quad (\text{REFTYPE-R-REF})$$

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma \vdash \phi' \sqsubseteq \phi''}{\Gamma \vdash \phi \sqsubseteq \phi''} \quad (\text{TRANS-REF})$$

Figure 7: Refinement rules

$$\frac{\Gamma \vdash P \ \texttt{wf}}{\Gamma \vdash P} \quad (Ax \triangleright \Gamma \vdash P) \qquad \text{(AXIOM)}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \qquad (\wedge\text{-INTRO})$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad (\wedge\text{-ELIM1})$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \qquad (\wedge\text{-ELIM2})$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \qquad (\vee\text{-INTRO1})$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \qquad (\vee\text{-INTRO2})$$

$$\frac{\Gamma \vdash P \vee Q \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R} \qquad (\vee\text{-ELIM})$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q} \qquad (\supset\text{-INTRO})$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash P \supset Q}{\Gamma \vdash Q} \qquad (\supset\text{-ELIM})$$

$$\frac{\Gamma \vdash \forall x : \phi . P \quad \Gamma \vdash t : \phi}{\Gamma \vdash P[t/x]} \qquad (\forall\text{-ELIM})$$

$$\frac{\Gamma, x : \phi \vdash P}{\Gamma \vdash \forall x : \phi . P} \qquad (\forall\text{-INTRO})$$

$$\frac{\Gamma \vdash P \ \texttt{wf}}{\Gamma, \texttt{false} \vdash P} \qquad (\texttt{false}\text{-ELIM})$$

$$\frac{\Gamma, \neg P \vdash \texttt{false}}{\Gamma \vdash P} \qquad (\neg\text{-ELIM})$$

$$\frac{\Gamma \vdash P \ \texttt{wf}}{\Gamma, P \vdash P} \qquad \text{(HYP)}$$

$$\frac{\Gamma \vdash t =_{\phi} t' \quad \Gamma, x : \phi \vdash P \ \texttt{wf} \quad \Gamma \vdash P[t/x]}{\Gamma \vdash P[t'/x]} \qquad \text{(CONG)}$$

$$\frac{\Gamma \vdash t : (x : \phi) P}{\Gamma \vdash P[t/x]} \qquad \text{(REFTYPE-ELIM)}$$

Figure 8: Natural Deduction rules for classical first order logic

B Interpretation in Henkin models

$$a \,[\![\Gamma \vdash \mathbf{1} : \texttt{Ref}\ \mathbf{1}]\!](g)\ a' \iff a, a' \in A^{\mathbf{1}}$$

$$\frac{[\![\Gamma \vdash \phi : \texttt{Ref}\ \sigma]\!] = R \quad [\![\Gamma, x : \phi \vdash \psi : \texttt{Ref}\ \tau]\!] = S}{a\,[\![\Gamma \vdash \Sigma_{x:\phi}\psi : \texttt{Ref}\ \sigma \times \tau]\!](g)\ a' \iff \begin{array}{l}\texttt{Proj}_1^{\sigma,\tau}(a)\ \ Rg\ \ \texttt{Proj}_1^{\sigma,\tau}(a') \text{ and} \\ \texttt{Proj}_2^{\sigma,\tau}(a)\ \ S\langle g, \texttt{Proj}_1^{\sigma,\tau}(a)\rangle\ \ \texttt{Proj}_2^{\sigma,\tau}(a')\end{array}}$$

$$\frac{[\![\Gamma \vdash \phi]\!] = R \quad [\![\Gamma, x : \phi \vdash \psi]\!] = S}{f\ [\![\Gamma \vdash \Pi_{x:\phi}\psi]\!](g)\ f' \iff \text{for all } a\ Rg\ a', \texttt{App}(f,a)\ S\langle g, a\rangle\ \texttt{App}(f',a')}$$

$$\frac{[\![\Gamma \vdash \phi : \texttt{Ref}\ \sigma]\!] = R \quad [\![\Gamma, x : \phi \vdash P\ \ \texttt{wf}]\!] = A}{a\ [\![\Gamma \vdash (x:\phi)P : \texttt{Ref}\ \sigma]\!](g)\ a' \iff a\ Rg\ a', \langle g, a\rangle \in A, \langle g, a'\rangle \in A}$$

$$a\ [\![\Gamma \vdash \gamma : \texttt{Ref}\ \gamma]\!](g)\ a' \iff a, a' \in A^{\gamma} \text{ and } a = a'$$

Figure 9: Interpretation of refinement types

$$[\![\Gamma, x : \phi, \Gamma' \vdash x]\!]\langle g, a, g'\rangle = \{a\}$$

$$[\![\Gamma \vdash k]\!](g) = \{\texttt{Const}(k)\}$$

$$[\![\Gamma \vdash *]\!](g) = A^{\mathbf{1}}$$

$$\frac{[\![\Gamma \vdash t]\!] = m \quad [\![\Gamma \vdash t']\!] = m'}{[\![\Gamma \vdash \langle t, t'\rangle]\!](g) = \{a \in A^{\sigma\times\tau} \mid \texttt{Proj}_1^{\sigma,\tau}(a) \in mg, \texttt{Proj}_2^{\sigma,\tau}(a) \in m'g\}}$$

$$\frac{[\![\Gamma, x : \phi \vdash t]\!] = m}{[\![\Gamma \vdash \lambda x : \phi.t]\!](g) = \{f \in A^{\sigma\to\tau} \mid \forall a \in [\![\Gamma \vdash \phi]\!](g).\texttt{App}(f,a) \in m\langle g, a\rangle\}}$$

$$\frac{[\![\Gamma \vdash t]\!] = m}{[\![\Gamma \vdash \pi_1(t)]\!](g) = \{\texttt{Proj}_1^{\sigma,\tau}(a) \mid a \in mg\}}$$

$$\frac{[\![\Gamma \vdash t]\!] = m}{[\![\Gamma \vdash \pi_2(t)]\!](g) = \{\texttt{Proj}_2^{\sigma,\tau}(a) \mid a \in mg\}}$$

$$\frac{[\![\Gamma \vdash t]\!] = m \quad [\![\Gamma \vdash t']\!] = m'}{[\![\Gamma \vdash tt']\!](g) = \{\texttt{App}(f,a) \mid f \in mg, a \in m'g\}}$$

Figure 10: Interpretation of raw terms

$$[\![\Gamma \vdash \texttt{false}]\!] = \emptyset$$

$$[\![\Gamma \vdash P \supset P']\!] = \{g \vDash \Gamma \mid g \notin [\![\Gamma \vdash P]\!] \text{ or } g \in [\![\Gamma \vdash P']\!]\}$$

$$[\![\Gamma \vdash \forall x : \phi . P]\!] = \{g \vDash \Gamma \mid \forall a \in [\![\Gamma \vdash \phi]\!](g) . \langle g, a \rangle \in [\![\Gamma, x : \phi \vdash P]\!]\}$$

$$\frac{[\![\Gamma \vdash t]\!] = m}{[\![\Gamma \vdash F(t)]\!] = \{g \vDash \Gamma \mid mg \subseteq \texttt{Pred}(F)\}}$$

$$\frac{[\![\Gamma \vdash t]\!] = m \quad [\![\Gamma \vdash t']\!] = m' \quad [\![\Gamma \vdash \phi]\!] = R}{[\![\Gamma \vdash t =_{\phi} t']\!] = \{g \vDash \Gamma \mid \forall a \in mg . \forall a' \in m'g \;.\; a \; Rg \; a'\}}$$

$$\frac{[\![\Gamma \vdash \phi]\!] = R \quad [\![\Gamma \vdash \phi']\!] = R'}{[\![\Gamma \vdash \phi \sqsubseteq \phi']\!] = \{g \vDash \Gamma \mid Rg \supseteq R'g\}}$$

Figure 11: Interpretation of propositions