Refinement types for specification

E. Denney
LFCS, Department of Computer Science, University of Edinburgh
Kings Buildings, Edinburgh EH9 3JZ, Scotland
Tel. : +44 131 650 5151
Fax : +44 131 667 7209
Email : `ewd@dcs.ed.ac.uk`

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 languages 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 Luo (1991) for example. An integer square root function might be specified as the existential type $\exists f : \texttt{nat} \to \texttt{nat} \ . \ (f\ n)^2 = n \ \vee \ (f\ n)^2 + 1 = n$, where the logic is encoded in the type theory. 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. Our square root example would be the proposition $(f\ n)^2 = n \ \vee \ (f\ n)^2 + 1 = n$, where f is a free variable of type $\texttt{nat} \to \texttt{nat}$. This is the approach traditionally taken by the program refinement community, and Morgan (1994) 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 it is 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 known as *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.

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 typed classical predicate logic as program logic here, and axiomatise an ordering on the refinement types, to be viewed as an increase in information, or refinement of specifications. 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 example. Section 5 gives the semantics and proofs of soundness and completeness. Finally, we make some conclusions in Section 6.

Related Work

There have been a number of papers in the 'non-standard type system as program logic' paradigm. Nielson & Nielson (1988) and (Burn 1992) axiomatise consequence relations on properties as a form of refinement. Pfenning, who introduced the term "refinement type", used them to express properties of mini-ML programs (Freeman & Pfenning 1991).

There have been various approaches by type theorists to combining logic and types. Feferman's (1985) system of variable types extends $\lambda^{\times\rightarrow}$ with subset types, though equality does not depend on the type, as it does here. Refinement can be defined in the logic, and is not explicitly axiomatised. Other type-theoretic approaches include (Aspinall 1995, Aspinall & Compagnoni 1996), which differ from the present work in being concerned with subtyping type families. Dependency there is at the level of types themselves, whereas we only allow dependent structure at the refinement type level.

The deliverables approach (Burstall & McKinna 1992, McKinna 1992) 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 an explicit witness to satisfy a refinement type. Our calculus could be regarded as an internal language for deliverables.

The work of Luo (1991) 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 (Nordström, Petersson & Smith 1990) is similar. The work of Hayashi (1994) is also related.

The program refinement community has traditionally used unstructured specifications. For example, Morgan (1994) 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

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

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 $t\ \mathtt{sat}\ (x : \tau)P$ is just taken to be a notation for a typing and a proposition, with the rule that $t\ \mathtt{sat}\ (x : \tau)P$ when $t : \tau$ and $P[t/x]$. This example reduces then 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 products and functions as abbreviations

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

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

The dependent function $\Pi_{x:(x:\sigma)P}(y : \tau)Q$ specifies some function which for all $x : \sigma$ such that P, returns a $y : \tau$ such that Q. This has combined the two quantifications in $(f : \sigma \to \tau)\forall x : \sigma \,.\, P \supset Q[fx/y]$, which we read as some $f : \sigma \to \tau$ such that for all $x : \sigma$, if P then $Q[fx/y]$. 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`

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

where we understand the sequent $n : \mathtt{nat} \vdash t\ \ \mathtt{sat}\ \ \phi$ to mean for all closed $t' : \mathtt{nat}$, $t[t'/n]\ \ \mathtt{sat}\ \ \phi[t'/n]$. 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 $\mathtt{div2}'$ has specification

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

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.

A significant benefit in writing specification in this more structured form is conceptual – it is preferable to structure specifications in such a way that it aids understanding of both specification and program. 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. We do not throw away the original rule that t satisfies $(x : \tau)P$ when $t : \tau$ and $P[t/x]$, however, since not all specifications can be given in a structured form.

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.

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 $\texttt{nat}$, and $\Sigma_{n:(n:\texttt{nat})\texttt{true}}(b : \texttt{bool})\texttt{true}$ for $\texttt{nat} \times \texttt{bool}$. At this point, we must cease to regard $\Pi_{x:(x:\sigma)P}(y : \tau)Q$ as an abbreviation, since we want it to have a different equality from $(f : \sigma \to \tau)\forall x : \sigma\ .\ P \supset Q[fx/y]$. 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, together with 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, $\lambda n : \texttt{even}.n$ is a class in the per $\texttt{even} \to \texttt{nat}$ but not $\texttt{nat} \to \texttt{nat}$, and $\lambda n : \texttt{nat}.n$ is a class in both $\texttt{even} \to \texttt{nat}$ and $\texttt{nat} \to \texttt{nat}$. 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.* equality at ϕ' implies equality at ϕ). We use the square $\sqsubseteq$ symbol to indicate an information ordering – the refinement of specifications. Note that this convention for refinement is the opposite direction to the usual subtyping relation.

3 THE CALCULUS

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. This is generated from a signature of types, constants and predicate symbols (in the underlying theory) and axioms (in the full theory). The well-formedness conditions on axioms will be explained in Section 3.3 below.

Definition 1 *A* signature *consists of a finite collection of ground types γ, n-ary constants k, each of which is assigned some sort $\tau_1, \ldots, \tau_n \to \tau$, and n-ary predicates F, which are assigned a sequence of types $\tau_1, \ldots, \tau_n$, indicated by $F : \mathtt{pred}\ \tau_1, \ldots, \tau_n$. There are two forms of axiom. We write propositional axioms as $Ax \triangleright \Gamma \vdash P$ and require that $\Gamma \vdash P\ \mathtt{wf}$. Axioms on the constants are given in the form $Ax \triangleright \Gamma \vdash k : \phi_1, \ldots, \phi_n \to \psi$, for constant k with sort $\tau_1, \ldots, \tau_n \to \tau$, and such that $\Gamma \vdash \phi_i : \mathtt{Ref}\ \tau_i$, $\Gamma \vdash \psi : \mathtt{Ref}\ \tau$.*

Note that the sorting $k : \tau \to \tau'$ and axiom $k : \phi \to \psi$ do not say that unary constant k is a well-formed term without the necessary number of arguments. For $t : \tau$, we have $k(t) : \tau'$, and if $t : \phi$ then $k(t) : \psi$.

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 ground type `nat`, we could have predicate `even : pred nat`, and then overload `even` for the refinement type $(n : \mathtt{nat})\mathtt{even}(n)$.

For ground type `bool`, we can have constants $\mathtt{true} : \mathtt{bool}, \mathtt{false} : \mathtt{bool}$ and $\mathtt{cond}_\tau : \mathtt{bool}, \tau, \tau \to \tau$, for each type τ. The axiom schema for conditionals can be given as $\mathtt{cond}_\tau : P + P', (x : \phi)P \supset Q[x], (y : \phi)P' \supset Q[y] \to (z : \phi)Q[z]$, where $P + P'$ abbreviates $(b : \mathtt{bool})b = \mathtt{true} \supset P\ \wedge\ b = \mathtt{false} \supset P'$.

The raw expressions of the language are given by a mutual recursion over refinement types, terms, propositions and contexts :

$$\phi ::= \mathbf{1}\ \mid\ \gamma\ \mid\ \Sigma_{x:\phi}\phi'\ \mid\ \Pi_{x:\phi}\phi'\ \mid\ (x : \phi)P$$

$$t ::= x\ \mid\ k(t_1, \ldots, t_n)\ \mid\ *\ \mid\ \langle t, t' \rangle\ \mid\ \lambda x : \phi.t\ \mid\ \pi_i(t)\ \mid\ tt'$$

$$P ::= \mathtt{false}\ \mid\ P \supset P'\ \mid\ \forall x : \phi.P\ \mid\ F(t_1, \ldots, t_n)\ \mid\ t =_\phi t'\ \mid\ \phi \sqsubseteq \phi'$$

$$\Gamma ::= \langle\rangle\ \mid\ \Gamma, x : \phi\ \mid\ \Gamma, P$$

Conceptually, it is simpler not to 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 a refinement of (we will just say '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, not containing any propositions. We use σ and τ as metavariables for types.

There is a unique (up to equality) term $*$ of unit type **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.

We think of terms of the calculus as being simple specifications of terms in the underlying $\lambda^{\times\to}$. We will refer to terms of $\lambda^{\times\to}$ as *total* terms. Terms of the calculus uniquely specify total terms up to equality at some refinement type. 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 $t[t''/x]$.

For example, $\lambda n : \texttt{even}.n$ specifies total terms of type $\texttt{nat} \to \texttt{nat}$ 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 $\texttt{even} \to \texttt{even}$ is a refinement of the type $\texttt{nat} \to \texttt{nat}$, the term $\lambda n : \texttt{even}.n$ does not itself have type $\texttt{nat} \to \texttt{nat}$. 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.

Remark 2 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. It does not matter whether the logic is classical or intuitionistic.

This 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

$$\Gamma \vdash t : \phi \qquad\qquad \Gamma \vdash P$$

where the atomic propositions are $\Gamma \vdash t =_\phi t'$ and $\Gamma \vdash \phi \sqsubseteq \phi'$. Equality and refinement are not separate judgement classes from the other propositions. All judgements are made in a context Γ of variable assumptions $x : \phi$ and propositions P. There are also mutually dependent well-formedness judgements

$$\vdash \Gamma\ \texttt{wf} \qquad \Gamma \vdash \phi : \texttt{Ref}\ \tau \qquad \Gamma \vdash P\ \texttt{wf}$$

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. We understand $\Gamma \vdash t : \phi$ to mean that for all the variables in the context Γ, if they satisfy the relevant refinement types, then the term t has refinement type ϕ. 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.

3.3 Rules of the Calculus

Rather than give an exhaustive listing of all the rules of the calculus, we restrict the discussion to a subset of the rules and refer the interested reader to (Denney 1997) and (Denney 1998) for more details.

One distinctive feature of the calculus is the mutual dependencies of the different syntactic categories, and hence of the different judgement classes. Refinement types can contain propositions, which can contain terms, and these in turn can contain refinement types in the abstractions.

First we describe the well-formedness rules, starting with contexts. The empty context is well-formed, and there are two rules for extending an existing context.

$$\frac{\Gamma \vdash \phi \ \mathtt{wf}}{\vdash \Gamma, x : \phi \ \mathtt{wf}}\ x \notin \Gamma \qquad\qquad \frac{\Gamma \vdash P \ \mathtt{wf}}{\vdash \Gamma, P \ \mathtt{wf}}$$

The well-formedness rules for refinement types essentially involve stripping off the logic while checking that everything fits together correctly. For example,

$$\frac{\Gamma \vdash \phi : \mathtt{Ref}\ \sigma \quad \Gamma, x : \phi \vdash \psi : \mathtt{Ref}\ \tau}{\Gamma \vdash \Pi_{x:\phi}\psi : \mathtt{Ref}\ \sigma \to \tau} \qquad\qquad \frac{\Gamma \vdash \phi : \mathtt{Ref}\ \tau \quad \Gamma, x : \phi \vdash P \ \mathtt{wf}}{\Gamma \vdash (x : \phi)P : \mathtt{Ref}\ \tau}$$

There are checks on the well-formedness of the context for the base cases so as to ensure that all provable judgements are well-formed.

$$\frac{\vdash \Gamma \ \mathtt{wf}}{\Gamma \vdash \mathbf{1} : \mathtt{Ref}\ \mathbf{1}} \qquad\qquad \frac{\vdash \Gamma \ \mathtt{wf}}{\Gamma \vdash \gamma : \mathtt{Ref}\ \gamma}\ (\gamma \text{ ground type})$$

Similar conditions are made for the base cases of the other judgement classes.

It is straightforward to formulate well-formedness rules for propositions. We select two rules for discussion.

$$\frac{\Gamma \vdash t : \phi \quad \Gamma \vdash t' : \phi}{\Gamma \vdash t =_\phi t' \ \mathtt{wf}} \qquad\qquad \frac{\Gamma \vdash \phi : \mathtt{Ref}\ \tau \quad \Gamma \vdash \phi' : \mathtt{Ref}\ \tau}{\Gamma \vdash \phi \sqsubseteq \phi' \ \mathtt{wf}}$$

For $t =_\phi t'$ to be well-formed we require that t and t' both have refinement type ϕ. The appeal to refinement typing is why well-formedness involves logical reasoning, and this propagates through the well-formedness rules for the other syntactic categories. Similarly, the refinement $\phi \sqsubseteq \phi'$ is only well-formed when ϕ and ϕ' are over the same type.

The refinement typing rules are the natural generalisations of the usual typing rules for the simply-typed lambda calculus with products. For example, the introduction rules for abstractions and pairs are

$$\frac{\Gamma, x : \phi \vdash t : \psi}{\Gamma \vdash \lambda x : \phi . t : \Pi_{x:\phi}\psi} \qquad \frac{\Gamma \vdash t : \phi \quad \Gamma \vdash t' : \psi[t/x] \quad \Gamma, x : \phi \vdash \psi \ \mathtt{wf}}{\Gamma \vdash \langle t, t' \rangle : \Sigma_{x:\phi}\psi}$$

and the corresponding elimination rules are

$$\frac{\Gamma \vdash t : \Pi_{x:\phi}\psi \quad \Gamma \vdash t' : \phi}{\Gamma \vdash tt' : \psi[t'/x]} \qquad \frac{\Gamma \vdash t : \Sigma_{x:\phi}\psi}{\Gamma \vdash \pi_1(t) : \phi} \qquad \frac{\Gamma \vdash t : \Sigma_{x:\phi}\psi}{\Gamma \vdash \pi_2(t) : \psi[\pi_1(t)/x]}$$

Where the calculus differs from $\lambda^{\times\rightarrow}$ is in the logical reasoning which pervades the rules. This is evident in the rule for forming terms with a constant, where well-formedness uses a logical axiom

$$\frac{\Gamma \vdash t_i : \phi_i}{\Gamma \vdash k(t_1, \ldots, t_n) : \psi} \quad (Ax \triangleright \Gamma \vdash k : \phi_1, \ldots, \phi_n \rightarrow \psi)$$

We have the obvious introduction rule for proving that a term inhabits a refinement type, and a weakening rule:

$$\frac{\Gamma \vdash t : \phi \quad \Gamma \vdash P[t/x]}{\Gamma \vdash t : (x : \phi)P} \qquad \frac{\Gamma \vdash t : \phi' \quad \Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash t : \phi}$$

We do not include an elimination rule (allowing us to conclude that $t : \phi$ from $t : (x : \phi)P$) as primitive, since this follows from the weakening rule and the refinement rules which we give below.

One further rule is

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

Inferring a refinement typing from an equality may seem strange, but it saves us a few rules. The reason for this is that in proving refinement typings and equalities we need to be able to combine assumptions on subterms. Since equalities are subscripted with a refinement type, the rule lets us use equality rules to prove a refinement typing. For example, the congruence rule for abstractions is

$$\frac{\Gamma \vdash \phi' \sqsubseteq \phi \quad \Gamma, x : \phi \vdash t =_\psi t'}{\Gamma \vdash \lambda x : \phi.t =_{\Pi_{x:\phi}\psi} \lambda x : \phi'.t'}$$

which lets us prove that $\lambda n : \mathtt{even}.n =_{\mathtt{even}\rightarrow\mathtt{even}} \lambda n : \mathtt{nat}.n$, and so we can infer that $\lambda n : \mathtt{nat}.n : \mathtt{even} \rightarrow \mathtt{even}$.

The η rules for abstractions and pairs have unconventional hypotheses, enabling us to combine logical and typing assumptions.

$$\frac{\Gamma, x : \phi \vdash tx : \psi}{\Gamma \vdash \lambda x : \phi.tx =_{\Pi_{x:\phi}\psi} t}\ x \notin t \qquad \frac{\Gamma \vdash \pi_1(t) : \phi \quad \Gamma \vdash \pi_2(t) : \psi[\pi_1(t)/x]}{\Gamma \vdash \langle \pi_1(t), \pi_2(t)\rangle =_{\Sigma_{x:\phi}\psi} t}$$

The usual hypothesis for the abstraction rule would be $\Gamma \vdash t : \Pi_{x:\phi}\psi$. The following example makes essential use of this rule.

$$\dfrac{\dfrac{\dfrac{\begin{array}{l} f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx), n : \mathtt{nat} \vdash fn : \mathtt{nat} \\ \quad f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx), n : \mathtt{nat} \vdash \mathtt{even}(fn) \end{array}}{f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx), n : \mathtt{nat} \vdash fn : \mathtt{even}}}{f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx) \vdash f =_{\mathtt{nat}\rightarrow\mathtt{even}} \lambda n : \mathtt{nat}.fn}\ (\lambda\text{-}\eta\text{-EQ})}{f : \mathtt{nat} \rightarrow \mathtt{nat}, \forall x : \mathtt{nat}.\mathtt{even}(fx) \vdash f : \mathtt{nat} \rightarrow \mathtt{even}}$$

Similar examples can be given making use of the η rule for pairs.

The refinement rules are of two kinds – ‘structural’ and logical. The obvious structural rules are

$$\frac{\vdash \Gamma \ \texttt{wf}}{\Gamma \vdash \mathbf{1} \sqsubseteq \mathbf{1}} \qquad \frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma, x : \phi \vdash \psi \sqsubseteq \psi'}{\Gamma \vdash \Sigma_{x:\phi}\psi \sqsubseteq \Sigma_{x:\phi'}\psi'} \qquad \frac{\Gamma \vdash \phi' \sqsubseteq \phi \quad \Gamma, x : \phi \vdash \psi \sqsubseteq \psi'}{\Gamma \vdash \Pi_{x:\phi}\psi \sqsubseteq \Pi_{x:\phi'}\psi'}$$

The interesting rules, however, are for refinement involving propositions. We must say when an arbitrary refinement type is a refinement of a type with a proposition, and when it refines to one.

$$\frac{\Gamma \vdash \phi \sqsubseteq \psi \quad \Gamma, x : \psi \vdash P}{\Gamma \vdash (x : \phi)P \sqsubseteq \psi} \qquad \frac{\Gamma, x : \psi, Q \vdash x : \phi}{\Gamma \vdash \phi \sqsubseteq (x : \psi)Q}$$

The only other refinement rule is transitivity of refinement.

Finally, there are the rules of the logic. As the idea behind the calculus is that the logic should be orthogonal to the rules of the calculus, we do not list most of the (standard) rules of our example logic, typed classical first-order logic, but just indicate where refinement types are involved. One point is that the contexts are different from the usual formulation. This is made clear by the two introduction rules

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q} \qquad \frac{\Gamma, x : \phi \vdash P}{\Gamma \vdash \forall x : \phi . P}$$

We need a refinement typing for the $\forall$-elimination rule

$$\frac{\Gamma \vdash \forall x : \phi . P \quad \Gamma \vdash t : \phi}{\Gamma \vdash P[t/x]}$$

and refinement typings are also used to infer propositions with the rules

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

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

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

and the axiom schemas are (for each ϕ)

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

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

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

The axiom can also be expressed in the standard form of Definition 1. 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

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

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

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

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

The proof continues with

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

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

5 MODELS

We give interpretations of the calculus in general models, known as *Henkin models* (Henkin 1950), with additional per structure. As is usual in concrete models of applied lambda calculi, we must consider such general models in order to get completeness. The per structure is to account for stratified equalities at different refinement types.

We define Henkin models in two stages. First, to each type σ (not just ground types) we ascribe the set A^σ, and to each constant $k : \tau_1, \ldots, \tau_n \to \tau$, an element $\mathtt{Const}(k)$ in the set $A^{\tau_1 \times \cdots \times \tau_n \to \tau}$. An applicative structure (with products) is a tuple

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

with families of projection and application maps. In addition, we require a function Pred which interprets predicates

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

Now, a Henkin Model is an applicative structure with two additional conditions, namely, that it is extensional, and that it satisfies the environment model condition. See (Mitchell 1996) for details. 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 enough elements in the model.

We do not have $A^{\sigma \times \tau} = A^\sigma \times A^\tau$ in general, but have an isomorphism mediated by the projection functions, $\mathtt{Proj}_1^{\sigma,\tau} : A^{\sigma \times \tau} \to A^\sigma$, and $\mathtt{Proj}_2^{\sigma,\tau} : A^{\sigma \times \tau} \to A^\tau$. In general, $A^{\sigma \to \tau}$ is embedded in $A^\sigma \to A^\tau$, but not equal.

A Henkin model models an applied theory when all constants and predicates are given an interpretation, and each axiom is true in the model, as defined below.

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

Now, expressions 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. For per R, we write $a \in R$ to mean $a\, R\, a$.

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

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

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

For $g, g' \vDash \Gamma$ we define equality of environments in the obvious way, as simultaneous equality of elements in the corresponding per, written $g\, [\![\Gamma]\!]\, g'$.

To avoid questions of coherence, we interpret raw terms*, and so raw propositions and refinement types too. The notation $\Gamma \vdash E$ indicates raw expression E in context Γ.

Now as mentioned above, refinement types are interpreted as pers over the associated type. The interpretation is given in Figure 1. The unit and ground types are interpreted as identities; the product and function refinement types are interpreted as the expected combination of pers, and $(x : \phi)P$ is interpreted as the restriction of ϕ to the elements for which P holds.

There is an apparent asymmetry in the definition of the product per for $\Sigma_{x:\phi}\psi$, but

*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.

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

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

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

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

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

Figure 1 Interpretation of refinement types

in fact, if ϕ is a well-formed refinement type in context Γ, then the soundness result below states that 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}. This is given in Figure 2. The idea is that terms are

$$\frac{[\![\Gamma \vdash \phi]\!] = R}{[\![\Gamma, x:\phi, \Gamma' \vdash x]\!]\langle g, a, g'\rangle = \{a' \mid a'\;Rg\;a\}}$$

$$\frac{[\![\Gamma \vdash t_i]\!] = m_i}{[\![\Gamma \vdash k(t_1,\ldots,t_n)]\!](g) = \{\mathtt{App}(\mathtt{Const}(k), \langle a_1,\ldots,a_n\rangle) \mid a_i \in m_i(g)\}}$$

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

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

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

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

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

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

Figure 2 Interpretation of terms

interpreted as morphisms of pers, that is, maps of equivalence classes, and so the interpretation of a variable is a map from an element to its equivalence class (in the relevant refinement type). It is because of the refinement type in abstractions that we interpret terms as sets rather than as single elements. For example, $\lambda n : \texttt{even}.n$ is interpreted as the set of elements in $A^{\texttt{nat}\to\texttt{nat}}$ which are the identity for even arguments. In Figure 3 we give the interpretation of propositions. We interpret a raw proposition in context $\Gamma \vdash P$ as the set of environments $g \vDash \Gamma$ in which P holds.

$$
\begin{gathered}
[\![\Gamma \vdash \texttt{false}]\!] = \emptyset \\
[\![\Gamma \vdash P \supset P']\!] = \{g \vDash \Gamma \mid g \notin [\![\Gamma \vdash P]\!] \text{ or } g \in [\![\Gamma \vdash P']\!]\} \\
[\![\Gamma \vdash \forall x : \phi.P]\!] = \{g \vDash \Gamma \mid \forall a \in [\![\Gamma \vdash \phi]\!](g) \,.\, \langle g, a\rangle \in [\![\Gamma, x : \phi \vdash P]\!]\} \\
\frac{[\![\Gamma \vdash t_i]\!] = m_i}{[\![\Gamma \vdash F(t_1, \ldots, t_n)]\!] = \{g \vDash \Gamma \mid \forall a_i \in m_i g \,.\, \langle a_1, \ldots, a_n\rangle \in \texttt{Pred}(F)\}} \\
\frac{[\![\Gamma \vdash t]\!] = m \quad [\![\Gamma \vdash t']\!] = m' \quad [\![\Gamma \vdash \phi]\!] = R}{[\![\Gamma \vdash t =_\phi t']\!] = \{g \vDash \Gamma \mid \forall a \in mg \,.\, \forall a' \in m'g \,.\, a\, Rg\, a'\}} \\
\frac{[\![\Gamma \vdash \phi]\!] = R \quad [\![\Gamma \vdash \phi']\!] = R'}{[\![\Gamma \vdash \phi \sqsubseteq \phi']\!] = \{g \vDash \Gamma \mid Rg \supseteq R'g\}}
\end{gathered}
$$

Figure 3 Interpretation of propositions

We may now say what the semantic analogues of the judgements are. 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 $[\![\Gamma \vdash \phi']\!](g) \subseteq [\![\Gamma \vdash \phi]\!](g)$. We define $\Gamma \vDash \phi\ \texttt{wf}$ to mean : for all $g\ [\![\Gamma]\!]\ g'$, $[\![\Gamma \vdash \phi]\!](g) = [\![\Gamma \vdash \phi]\!](g')$, and $\Gamma \vDash P\ \texttt{wf}$ to mean : for all $g\ [\![\Gamma]\!]\ g'$, $g \in [\![\Gamma \vdash P]\!] \iff g' \in [\![\Gamma \vdash P]\!]$.

5.1 Soundness and Completeness

Having given the meaning to the expressions of the calculus via an interpretation, we must verify that this respects the inference rules, that is, the calculus is sound with respect to the intended interpretation. A consequence of this is that since we can give nontrivial models the calculus is consistent.

Theorem 3 *(Soundness) If* $\Gamma \vdash t : \phi$ *then* $\Gamma \vDash t : \phi$*, if* $\Gamma \vdash \phi\ \texttt{wf}$ *then* $\Gamma \vDash \phi\ \texttt{wf}$*, if* $\Gamma \vdash P\ \texttt{wf}$ *then* $\Gamma \vDash P\ \texttt{wf}$*, and if* $\Gamma \vdash P$ *then* $\Gamma \vDash P$*.*

Proof: Simultaneous induction over all derivations. The soundness of β-reduction and $\forall$-elimination is by a substitution lemma. ∎

A more challenging question is whether the calculus is in any sense complete, that is, if a particular judgement holds in all the models (of the relevant signature) then it is provable. In fact, the calculus is also complete, with a couple of provisos. Firstly, due to the way in which well-formedness is combined with satisfying logical properties, we must assume that the judgement is well-formed. 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.

The second point arises with higher-order terms, and is due to the calculus requiring arguments to an abstraction to have the refinement type on the abstraction, but the model just needing equality of arguments at that refinement type to give equal results. For example, $\lambda f : \texttt{nat} \to \texttt{nat}.3$ 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 in context, $\Gamma \vdash t$, has refinement type ϕ in the model, then there exists a term t' which does have refinement type ϕ, such that $\Gamma \vDash t =_\phi t'$. We write this as $\Gamma \vdash t \sim \phi$.

Theorem 4 *(Completeness) For* $\Gamma \vdash P \;\; \texttt{wf}$*, if* $\Gamma \vDash P$ *then* $\Gamma \vdash P$*. For* $\Gamma \vdash \phi \;\; \texttt{wf}$*, if* $\Gamma \vDash t : \phi$ *then* $\Gamma \vdash t \sim \phi$*.*

Proof: Although we do not have minimal term models (due to having propositional assumptions), we can still use a term model construction to prove completeness, by 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 $\Gamma \nvdash \texttt{false}$, and *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 of consistency and satisfiability. 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 $\Gamma \nvDash P$.

Let Γ be a consistent context. We sketch the construction of a model $\mathcal{A}$ and environment $g \vDash_{\mathcal{A}} \Gamma$.

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*[*] *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.

[*] These have nothing to do with Henkin models.

The existence of maximal consistent Henkin extensions follows using a standard argument from Zorn's Lemma. See (van Dalen 1994) for details. Note that for all $\Gamma \vdash P$ `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 a 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 $\lambda^{\times\rightarrow}$ 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 $F : \mathtt{pred}\, \tau_1, \ldots, \tau_n$ as $\{[\langle t_1, \ldots, t_n \rangle] \mid F(t_1, \ldots, t_n) \in \Delta\}$.
5. The set Δ gives rise to an environment for Γ in $\mathcal{A}$. 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. Define the corresponding element in the environment as $g_x = [u]^\tau$, where u is any total realizer of t (which must exist).
6. For $\Gamma \vdash P$ `wf`, prove that $g \in [\![\Gamma \vdash P]\!] \iff \Delta_\infty \vdash P \iff \Delta \vdash P$. To prove the first equivalence we directly characterise the interpretation of terms and refinement types in the term model, $\mathcal{A}$. The second equivalence is because for each $x : \phi \in \Gamma_\infty$, $x \notin P$ and ϕ is inhabited.

Hence $\mathcal{A}$ models Δ, and so it models Γ. ∎

Now since first-order logic, the simply-typed lambda calculus, and the refinement types calculus are all complete for the class of Henkin models, we have

Corollary 5 *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.*

Corollary 6 *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.*

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 $\mathtt{div2}'$. 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 have given a refinement relation $\phi \sqsubseteq \phi'$ on specifications, this does not constitute a full *refinement calculus* (such as in (Morgan 1994)). 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. The ability to express information within a program context is useful. Program reasoning and manipulation often requires facts which are true at some local program point. 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 $\mathtt{nat} \rightarrow \mathtt{nat}$, 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 $\mathtt{even} \rightarrow \mathtt{nat}$.

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. The presentation has been improved by numerous suggestions from the anonymous referees.

The author was supported by a postgraduate research studentship from the Engineering and Physical Sciences Research Council while carrying out this research.

REFERENCES

Aspinall, D. (1995) Subtyping with singleton types, *in* 'Proceedings of Computer Science Logic '94', Vol. 933 of *LNCS*.

Aspinall, D. & Compagnoni, A. (1996) Subtyping dependent types, *in* 'Proceedings of the eleventh IEEE Symposium on Logic in Computer Science'.

Burn, G. L. (1992) A logical framework for program analysis, *in* J. Launchbury & P. Sansom, eds, 'Proceedings of the 1992 Glasgow Functional Programming Workshop', Springer-Verlag Workshops in Computer Science series, pp. 30–42.

Burstall, R. & McKinna, J. (1992) Deliverables : A categorical approach to program development in type theory, *in* 'Mathematical Foundations of Computer Science : 18th International Symposium', Vol. 711 of *Lecture Notes in Computer Science*, pp. 32–67. An earlier version appeared as LFCS Technical Report ECS-LFCS-92-242.

Denney, E. (1997) Refining Refinement Types, *in* 'Informal Proceedings of Types Workshop on Subtyping, Inheritance and Modular Development of Proofs', University of Durham.

Denney, E. (1998) A General Theory of Program Refinement. PhD thesis, Department of Computer Science, University of Edinburgh. Forthcoming.

Feferman, S. (1985) A theory of variable types, *in* 'Proceedings of the Fifth Latin American Symposium on Mathematical Logic', Vol. 19 of *Revista Colombiana de Matemáticas*.

Freeman, T. & Pfenning, F. (1991) Refinement types for ML, *in* 'Proceedings of the SIGPLAN'91 Symposium on Language Design and Implementation', ACM Press, pp. 268–277.

Hayashi, S. (1994) Logic of refinement types, *in* 'Types for Proofs and programs', Vol. 806 of *Lecture Notes in Computer Science*, Springer Verlag.

Henkin, L. (1950) Completeness in the theory of types. *Journal of Symbolic Logic* **15**(2), 81–91.

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

McKinna, J. (1992) Deliverables : A Categorical Approach to Program Development in Type Theory. PhD thesis, Department of Computer Science, University of Edinburgh.

Mitchell, J. (1996) *Foundations for Programming Languages*, Foundations of Computing Series, MIT Press.

Morgan, C. (1994) *Programming from Specifications*. Prentice Hall.

Nielson, H. & Nielson, F. (1988) Automatic binding time analysis for a typed λ-calculus, *in* 'Proceedings of the Fifteenth Annual ACM Symposium on Prin-

ciples of Programming Languages'.
Nordström, B., Petersson, K. & Smith, J. M. (1990) *Programming in Martin-Löf's Type Theory*. Vol. 7 of *Monographs on Computer Science*, Oxford University Press.
van Dalen, D. (1994) *Logic and Structure*. Springer-Verlag.

BIOGRAPHY

Ewen Denney is a final year PhD student at the Laboratory for the Foundations of Computer Science in Edinburgh University, under the supervision of Professor Gordon Plotkin and Doctor Marcelo Fiore. His research is on the theory of program refinement, with an aim to characterising the logical and semantic structures involved. He holds a Masters Degree from Imperial College and a Bachelors from the University of Glasgow.