A Theory of Program Refinement

Ewen W.K.C. Denney

Doctor of Philosophy
University of Edinburgh
1998

Do mo phàrantan

Abstract

We give a canonical program refinement calculus based on the lambda calculus and classical first-order predicate logic, and study its proof theory and semantics. The intention is to construct a metalanguage for refinement in which basic principles of program development can be studied.

The idea is that it should be possible to induce a refinement calculus in a generic manner from a programming language and a program logic. For concreteness, we adopt the simply-typed lambda calculus augmented with primitive recursion as a paradigmatic typed functional programming language, and use classical first-order logic as a simple program logic.

A key feature is the construction of the refinement calculus in a modular fashion, as the combination of two orthogonal extensions to the underlying programming language (in this case, the simply-typed lambda calculus).

The crucial observation is that a refinement calculus is given by extending a programming language to allow indeterminate expressions (or 'stubs') involving the construction 'some program x such that P'. Factoring this into 'some $x\ldots$' and '... such that P', we first study extensions to the lambda calculus providing separate analyses of what we might call 'true' stubs, and structured specifications. The questions we are concerned with in these calculi are how do stubs interact with the programming language, and what is a suitable notion of structured specification for program development.

The full refinement calculus is then constructed in a natural way as the combination of these two subcalculi. The claim that the subcalculi are orthogonal extensions to the lambda calculus is justified by a result that a refinement can actually be factored into simpler judgements in the subcalculi, that is, into logical reasoning and simple decomposition.

The semantics for the calculi are given using Henkin models with additional structure. Both simply-typed lambda calculus and first-order logic are interpreted using Henkin models themselves. The two subcalculi require some extra structure and the full refinement calculus is modelled by Henkin models with a combination of these extra requirements. There are soundness and completeness results for each calculus, and by virtue of there being certain embeddings of models we can infer that the refinement calculus is a conservative extension of both of the subcalculi which, in turn, are conservative extensions of the lambda calculus.

Acknowledgements

Thanks to Gordon Plotkin and John Power for supervising this thesis and for providing advice and encouragement. I hope their suffering over my writing has not been in vain. Marcelo Fiori also supervised the early stages. Thanks also to John for his famous chats.

I had useful conversations with Alex Bunkenburg, Joe Morris, Álvaro Moreira, Masahito Hasegawa, Thomas Kleymann, David Aspinall, and Jitka Stříbrná.

This work was supported by an EPSRC studentship and, in the final year, by a part-time research contract with Gordon Plotkin.

This thesis was examined by Don Sannella and Peter O'Hearn. Thanks to them for all their helpful comments.

Thanks to Yukki and my parents for their support. 多謝 agus mòran taing.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Ewen W.K.C. Denney*)

Table of Contents

Chapter 1

Introduction

Program refinement is a programming methodology in which a formal description of what a program should do — a specification — is gradually *refined* into an executable program satisfying that specification.

This thesis is a study of program refinement for a simple idealised programming and specification language. Although our analysis is theoretical, we give motivation from practical considerations.

The kind of issues with which we are concerned are:

- What logical machinery and semantic principles are involved in program refinement?

- Are there interesting fragments of refinement calculus which have practical uses?

- Given a program logic, what is a suitable specification language based on it for program refinement?

- Understanding the general relationship between a refinement calculus, a programming language, and a program logic.

- How might the structure of a calculus inform the architecture of tools for program development?

Although we pose these questions here in general terms, for concreteness we use typed λ-calculus as a paradigmatic functional programming language, and classical first-order logic as a program logic. It is our hope that by making clear how these choices affect our analysis, we will attain some degree of generality. The significance of these decisions is discussed below.

In order to address these questions, we carry out a modular analysis of a simple refinement calculus. We suggest that a refinement calculus can be understood as a combination of two extensions to the underlying programming language — one accounting for specifications, and the other for what we call ‘pure’ refinement.

Just as the lambda calculus can be used as a metalanguage for studying functional programming languages, so the lambda calculus based refinement calculus we develop could be used as a metalanguage for studying refinement in functional programming languages. The interest in studying metalanguages is that they provide a simple setting in which fundamental issues can be studied.

The purpose of this analysis then is not to justify or consider refinement as a viable methodology, nor to develop an industrial strength refinement calculus[1], but to fully work out the theory of refinement in one limited area. Ultimately this knowledge could help to delimit the boundary between manual and automated program development and so be used in theorem provers.

Certainly, a theory of specification and programming is a necessary precursor to having tools which support automatic or manual refinement and synthesis. We will suggest later how the theory can inform the construction of a tool.

In this introduction we give an overview of the issues involved and how our refinement calculus is built up. We start with a description of the refinement methodology of program development in Section 1.1, then in Section 1.2 describe the related notion of program verification in terms of a simple satisfaction relation between programs and specifications, and the relevance of program logics for refinement. Then we will see that there are essentially two themes in refinement — structure on specifications, and internalisation of development — and this is discussed in Section 1.3, where we give an overview of the calculi in this thesis. In Section 1.4 we compare the particular form of refinement studied here with other methodologies. In Section 1.5 we discuss the significance of the choices we have made for programming language and program logic. Then in Sections 1.6 and 1.7 we describe related work and summarise the thesis.

1.1 Refinement Methodology

We now describe how program refinements are constructed, and consider what features a calculus should have in order to formalise refinement. There are two uses for refinement. First, it is a methodology for the construction of correct programs. Second, refinement can form the basis of a framework in which programming knowledge can be presented (as collections of refinements).

[1] A contradiction in terms, some would say!

1.1.1 Stepwise Development

When programmers write programs they often use a mixture of top-down and bottom-up development. Central to the method of top-down development is the idea of a *stub*. Suppose we are writing some program, the main body of which needs to use a sorting function on lists. We prefer to get the structure of the main body correct first before writing the sorting function, but in order to get the program to compile, or even to type-check, we must at least declare the sorting function, and give it a dummy body — a stub.

$$\texttt{fun Sort l} = \texttt{nil};$$

Any use of the function elsewhere is now well-defined

$$\texttt{l}' = \texttt{Sort some\_list};$$

Conceptually, however, this seems rather inelegant. Practically, although the program will compile and will run, the programmer must always bear in mind that this is not 'true' code. When testing the behaviour of the partially written program this must be taken into account. One way of avoiding behaviour which depends on dummy bodies would be to use some kind of error or exception mechanism as a stub. This is not entirely satisfactory, however, since it is possible that the programmer might forget that various stubs are left throughout the program. It would be better to have a language construct that allows true stubs to be written. The sorting function would be written as

$$\texttt{fun Sort l} = ?_{\texttt{int list}};$$

The idea is that $?_{\texttt{int list}}$ stands for some unwritten code of type `int list`. This can then be type-checked and the system would give a warning message to the effect that certain lines of code still remain to be implemented. We would like to be able to compile and run such partially written programs. In general, this is not possible since our program might just be one big stub. However, we would like an understanding of programs with stubs which lets us do this whenever possible.

Now, this much is a true description of (part of) what programmers do in practice. However, it is little more than a convenient notation for recording the development of a program through a sequence of abstractions. The essence of refinement lies in being able to use the full expressive power of a program logic. Rather than just have a stub for 'some program' we use the logic to say 'some program such that P'. So whereas above we could do no more than just specify the type of a program using $?_\tau$, we now extend this to $?_{(x:\tau)P}$, meaning 'some program x such that P', where proposition P can contain free variable x of type τ. For example, suppose we have defined predicates `Sorted` and `Permutation` in first-order logic. We can then write

$$\texttt{fun Sort l} = ?_{(l':\texttt{int list})\,\texttt{Sorted}(l') \wedge \texttt{Permutation}(l,l')}\ ;$$

This should be thought of as a description of some program which takes a list l and returns a list l' such that l' is sorted and a permutation of l, and not as a nondeterministic function which returns all such l'. This contrast between *nondeterminism* and, what we will refer to as *underdeterminism*, is of central importance in this thesis and is discussed in more detail below.

We can think of these combinations of logical specifications with program code as a kind of 'abstract program'. Languages which combine program and specification constructs in this way are called *wide-spectrum* languages.

Now, we proceed by doing a case analysis on the input. If the input is the nil list then clearly we just return the nil list. If it is of the form $x :: xs$ then we must decide how this is to be sorted. We can decompose the sortedness of a nonempty list into the correct insertion of some element into a sorted list, since,

$$\texttt{Sorted}\,(l_1)\ \wedge\ \texttt{Sorted}\,(l_2)\ \wedge\ l_1 \leq x\ \wedge\ x \leq l_2\ \supset\ \texttt{Sorted}\,(l_1 +\!\!+ (x :: l_2))$$

Let us write $\texttt{Insertion}\,(l, x, l')$ for the proposition that the list l' is the insertion of x into list l, that is,

$$(\exists l_1 : \texttt{int list}\,.\,\exists l_2 : \texttt{int list}\,.\,l = l_1 +\!\!+ l_2 \wedge l_1 \leq x \wedge x \leq l_2 \wedge l' = l_1 +\!\!+ (x :: l_2)$$

Using the above proposition, we can prove that

$$\texttt{Sorted}\,(l)\ \wedge\ \texttt{Insertion}\,(l, x, l')\ \supset\ \texttt{Sorted}\,(l')$$

Using this idea, we replace the specification of sorting with this more constructive form.

$$\texttt{fun Sort l} =$$
$$?_{(l' : \texttt{int list})}\ l = \texttt{nil}\ \supset\ l' = \texttt{nil} \wedge$$
$$\exists x : \texttt{int}\,.\,\exists xs : \texttt{int list}\,.\,l = x :: xs\ \supset\ \exists xs' : \texttt{int list}.\texttt{Sorted}(xs') \wedge \texttt{Permutation}(xs, xs') \wedge \texttt{Insertion}\,(xs', x, l')$$

Now, this specification has a recursive form. Assuming the existence of a refinement rule which lets us replace a *recursive specification* with a *recursion over a specification*, we aim to shift the recursion from specification to code. This is more clearly explained by the example. Let us write `Insert` for the specification

$$(f : \texttt{int list} \rightarrow \texttt{int} \rightarrow \texttt{int list})\ \forall l : \texttt{int list}\,.\,\forall x : \texttt{int}\,.\,\texttt{Insertion}\,(l, x, f\ l\ x)$$

After introducing the recursion, the sorting specification is

$$\texttt{fn}\ l : \texttt{int list} \Rightarrow \texttt{listrec}\ (\texttt{nil}, ?_{\texttt{Insert}}, \texttt{l})$$

This is a specification of insertion sort. Although this may seem rather a large step, we will cover a similar example in more detail in Chapter 5.

Thus, we *decompose* the specification into the simpler specification of an insertion function, and some code to carry out the recursion over the list.

The development so far can be represented as a process of *refinement* of a piece of 'unwritten code' corresponding to the original specification, which we summarise as

$$\begin{array}{lll} ?_{\texttt{Sort}} & \sqsubseteq & \texttt{fn}\ l : \texttt{int list} \Rightarrow \\ & & \quad ?_{(l' : \texttt{int list})\ l = \texttt{nil}\ \supset\ l' = \texttt{nil}\ \wedge} \\ & & \quad\quad \exists x : \texttt{int} . \exists xs : \texttt{int list} . l = x :: xs\ \supset \\ & & \quad\quad \exists xs' : \texttt{int list}.\texttt{Sorted}(xs') \wedge \texttt{Permutation}(xs, xs') \wedge \texttt{Insertion}\ (xs', x, l') \\ & \sqsubseteq & \texttt{fn}\ l : \texttt{int list} \Rightarrow \texttt{listrec}\ (\texttt{nil}, ?_{\texttt{Insert}}, \texttt{l}) \end{array}$$

This hierarchical separation of concerns illustrates the *stepwise refinement* methodology of program development. The idea is to gradually implement the specification in stages. At each stage, we either simplify a specification, possibly by decomposing it into a combination of subspecifications, or introduce some program code. Thus refinement can be seen as consisting of two alternating phases. On the one hand, logical specifications are replaced with something equivalent or more specific, so that they are more amenable to implementation. On the other, there are decomposition rules, where a specification is split into a number of simpler specifications while introducing a program constructor.

Now, although refinement is an inherently top-down methodology, there is also a bottom-up aspect. We sometimes want to use programs which have already been written, from a library say. So-called *problem reduction* is when a program is used to directly implement a specification. For example, we might have written the insertion function before, or be able to take it from a library. We illustrate this by simply assuming a free variable

$$ins : \texttt{int list} \to \texttt{int} \to \texttt{int list}$$

for which $\forall l : \texttt{int list} . \forall x : \texttt{int} . \texttt{Insertion}\ (l, x, ins\ l\ x)$.

The final step in the refinement then is

$$\begin{array}{l} ins : \texttt{int list} \to \texttt{int} \to \texttt{int list} \\ \vdash \lambda l : \texttt{int list} . \texttt{listrec}\ (\texttt{nil}, ?_{\texttt{insert}}, \texttt{l}) \sqsubseteq \lambda \texttt{l} : \texttt{int list} . \texttt{listrec}\ (\texttt{nil}, ins, \texttt{l}) \end{array}$$

This is typical of the general form of programming problems. In practice, the question we ask is *given* a library of component programs, how can we implement a specification? Even if we start with an empty context, during decomposition of a specification we will often construct subprograms which can be used elsewhere.

We can represent this formally by a global context of assumptions of the form $x : \tau \,|\, P$, meaning that program x has type τ and that proposition P (which can contain x) holds. For example, the insertion function is assumed as:

$$ins : \texttt{int list} \to \texttt{int} \to \texttt{int list} \;|\; \forall l : \texttt{int list} \,.\, \forall x : \texttt{int} \,.\, \texttt{Insertion}\,(l, x, ins\ l\ x)$$

The final step in the refinement above follows from the following simple rule for problem reduction:

$$x : \tau \,|\, P \vdash ?_{(x:\tau)P} \sqsubseteq x$$

Two important characteristics of refinement are that it is *stepwise* and *piecewise*. By stepwise, we mean that the specification can be refined into a program in a number of small steps, while piecewise means that a large specification can be refined one piece at a time, in the knowledge that this leads to a valid refinement of the whole specification. This means that the calculus should have rules so that the refinement relation is transitive and compositional (with respect to the program constructors). Moreover, the refinement rules should be syntax-directed (as far as this is possible).

There are some other features we would like in a refinement calculus. If we can place requirements on the output, it is convenient to also make assumptions about the input. For example, the specification of a search function might assume that the input list was sorted. This could be expressed with a notation like

$$\texttt{fun Search l n where Sorted}(l) = ?_{(b:\texttt{bool})\ b=\texttt{true} \iff \texttt{In}(n,l)}\ ;$$

We are allowed to use the fact that l is sorted when implementing the body. However, $(l : \texttt{int list})\ \texttt{Sorted}\,(l)$ is not a type of the programming language We must still produce a function which works for all arguments of type `int list`. In particular, any implementation of this specification must produce a result for unsorted lists. The point is, however, that for the purpose of implementing the specification it does not matter what the result is for unsorted lists. In a sense, we can regard two implementations t and t' as being the same when they give the same results for sorted lists. We can write this as

$$t =_{((l:\texttt{int list})\ \texttt{Sorted}\,(l))\to\texttt{int}\to\texttt{bool}} t'$$

where we think of $(l : \texttt{int list})\ \texttt{Sorted}\,(l) \to \texttt{int} \to \texttt{bool}$ as the specification of functions from $(l : \texttt{int list})\ \texttt{Sorted}\,(l)$ and `int` to `bool`, and of the equality as being ‘at the specification’.

Now, the intention behind annotating stubs with logical properties is, of course, that they will eventually be replaced with code satisfying that property. The slogan of the refinement methodology is that it is *correctness preserving* — programs will be automatically correct by construction. By requiring that each refinement step preserve properties, the program eventually constructed in a chain of refinement steps from a specification will be guaranteed to satisfy the original specification. The proof of correctness has been carried out essentially as a side-effect of the program's construction. This is easier than verifying the completed program independent of its construction.

However, if we start a refinement from a mixture of specification and code, then what is a correct refinement? Indeed, what does it mean for the individual rules to be correct? This is related to the question of how the specifications, logic and refinement all relate to each other. It is often most natural to specify a program using a mixture of logic and algorithm. We will discuss specifications in more detail below.

Just as we want to be able to reason about programs, so we would like to be able to reason about abstract programs. We can formalise all of this by defining a notion of satisfaction of specifications by programs, and extending this to abstract programs. As this is all carried out in a context of assumptions, the form of the satisfaction judgement is

$$x_1 \texttt{ sat } (x : \tau_1)P_1, \ldots, x_n \texttt{ sat } (x : \tau_n)P_n \vdash r \texttt{ sat } (x : \tau)P$$

Then to say that a refinement rule, $\Gamma \vdash r \sqsubseteq r'$, is correct means: for all specifications $(x : \tau)P$, if $\Gamma \vdash r \texttt{ sat } (x : \tau)P$ then $\Gamma \vdash r' \texttt{ sat } (x : \tau)P$.

The notation for refinement is suggestive of the fact that refinement is an inequality. This raises the question of the relationship it bears to the underlying equality[2] of the programming language. In general, we will address questions of conservativity of refinement calculi over program logics and the equational theory of a programming language.

In addition we would like completeness of refinement with respect to the logic, if at all possible: if $\Gamma \vdash t \texttt{ sat } (x : \tau)P$ then we should be able to obtain the program by refinement: $\Gamma \vdash ?_{(x:\tau)P} \sqsubseteq t$.

1.1.2 Programming Knowledge

Although some researchers have portrayed refinement as a framework for eventual automated programming tools, this should not be seen as the sole selling point of

[2] That is, some notion of equivalence of programs.

the refinement methodology. There is no doubt that constructing a program by refinement involves more work than just writing it directly. The point is that the discipline this imposes offers a framework in which programming *as a whole* can be more easily carried out. We feel that this aspect of refinement has not been sufficiently emphasised in the literature.

Actual programming practice involves a sequence of decisions, in which the programmer figures out how to solve some problem. In figuring out how to implement a specification, the programmer will use a number of insights in order to (informally) justify these decisions. This sequence of decisions is usually discarded, just leaving the final program. Such an approach is reasonable if the program is only intended for consumption by computer. However, program comprehension is necessary whenever a program is intended to be processed by humans. Moreover, there are two activities for which it is essential — verification and maintenance. The difficulty in understanding programs arises from the need to rediscover the insights that the programmer used in writing the program in the first place, and so it would be best if these were retained. Hence we are led to study a paradigm of programming in which derivations are primary. Scherlis and Scott [SS83] discuss the need for a *logic of programming*, as distinct from a *logic of programs*. An early advocate of refinement and its use for program comprehension was Wirth [Wir71].

The idea of explaining a program by its refinement can, in principle, be applied to program optimisation. Although an executable program may consist of optimised 'spaghetti', there will be a level of structured code above this in the derivation. The relationship between the two is justified by some optimising transformations.

The particular knowledge of some application domain can be collected in a library of programs, each paired with its specification, `prog sat spec`. This can be viewed as a refinement, $\texttt{spec} \sqsubseteq \texttt{prog}$, but this does not give all the information that might be useful to a programmer. It is more insightful to read such relationships as a sequence of abstractions. Similarly, for teaching purposes, algorithms and general programming principles can often be best explained as a refinement through several levels of abstraction.

The knowledge of experienced programmers is essentially having solutions to generic problems and how to apply them in particular situations. Empirical studies have shown that programming knowledge can be encapsulated as collections of derivations [SE84].

To a certain extent, modern programming languages encourage programmers to indicate levels of abstraction through the use of abstract data types and structured programming methods, but it is not always possible to express all the structure in the program text. The refinement paradigm is applicable to all programming languages and development methodologies.

1.2 Program Logics and Specification

The specification of a computer program is a formal description of the essential properties it is to have. *Correctness* of a program means that it satisfies some intended specification and *verification* is the task of establishing correctness. Verifying that a program satisfies a specification is, in some sense, dual to refining specifications to programs.

Program properties can be expressed in a number of ways. We will first consider logics in which the propositions refer to programs — *program logics*. The term 'program logic' is often used specifically for the first-order dynamic logic of Harel [Har79] (see the survey in [vL90] for example), but we use it more generally, for any logic of programs (although we will only consider one ourselves). Harel [Har80] gives a survey of different techniques for proving program correctness.

However, a basic distinction can be drawn between *extensional* and *intensional* properties. Extensional (or functional) properties are those concerned with a program's input-output behaviour, that is, by viewing it as a black box. Formally, we can say that they are the properties preserved under extensional equality (which has a suitable inductive definition). Intensional properties, on the other hand, depend on the structure of the program itself. Examples of such notions include complexity, program style, and so on. Here we will be concerned with extensional properties and say more about this later in Section 1.4.

Program analysis [Nie96] is the task of taking a program and finding which, of a specific class of properties, it satisfies. These are typically computational in nature, such as strictness properties, or binding times.

A specification, on the other hand, is generally of more complex properties. For concreteness, let us use the classical first-order equational theory of simply-typed λ-calculus as a program logic. We will lay the basis of this in Chapter 2, and give an outline here. The pre-expressions are:

Types	$\tau ::= \mathbf{1} \mid \gamma \mid \tau \times \tau \mid \tau \to \tau$
Terms	$t ::= x \mid k(t_1, \ldots, t_n) \mid * \mid \langle t, t' \rangle \mid \lambda x : \tau . t \mid \pi_i(t) \mid tt'$
Propositions	$P ::= \bot \mid P \supset P' \mid \forall x : \tau . P \mid F(t_1, \ldots, t_n) \mid t =_\tau t'$
Variable contexts	$\Gamma ::= \langle\rangle \mid \Gamma, x : \tau$
Propositional contexts	$\Delta ::= \langle\rangle \mid \Delta, P$

This is with respect to a collection of ground types γ, constants k, and primitive predicates F (extensional predicates such as `Iszero` and `Even`).

There are well-formedness judgements

$$\Gamma \vdash t : \tau$$

$$\Gamma \vdash P \ \mathtt{wf}$$

and a proof judgement

$$\Gamma; \Delta \vdash P$$

meaning: for all values in the types of Γ, if each proposition in Δ holds then P holds.

Now, the next question is how to write specifications using the program logic. The simplest choice is to write $(x : \tau)P$, where x is allowed to be free in P, for the specification of the property of the program x of type τ such that P holds.

The satisfaction judgement can be formulated, then, as

$$\Gamma; \Delta \vdash t \ \mathtt{sat} \ (x : \tau)P$$

meaning: for all values in the types of Γ, if each proposition in Δ holds then t has type τ and $P[t/x]$ holds.

Now either we take this judgement as derived, and an abbreviation for

$$\Gamma \vdash t : \tau \quad \textit{and} \quad \Gamma; \Delta \vdash P[t/x]$$

or we can axiomatise it directly on the structure of P, and have this correspondence as a theorem. For example, we would have rules like

$$\frac{\Gamma; \Delta \vdash t \ \mathtt{sat} \ (x : \tau)P \quad \Gamma; \Delta \vdash t \ \mathtt{sat} \ (x : \tau)P'}{\Gamma; \Delta \vdash t \ \mathtt{sat} \ (x : \tau)P \wedge P'}$$

This is really just an alternative presentation of the logic, so we do not consider it when we give the satisfaction rules.

It is also possible to have rules on the structure of the terms, such as

$$\frac{\Gamma; \Delta \vdash t \ \texttt{sat} \ (x : \tau)P \qquad \Gamma; \Delta \vdash t' \ \texttt{sat} \ (y : \tau')P'}{\Gamma; \Delta \vdash \langle t, t' \rangle \ \texttt{sat} \ (z : \tau \times \tau')P[\pi_1(z)/x] \wedge P'[\pi_2(z)/y]}$$

This is in the spirit of the refinement types which we introduce later as our notion of specification.

In Chapter 2 we will give a set-theoretic semantics to the logic (making the extensionality clear) and prove it sound and complete.

The real difference between a specification and a proposition of a program logic, however, is that a specification possesses structure 'in the large' []. It is important to structure specifications in order to handle the large scale and complexity involved in real systems. Structure allows us to reason about specifications in a compositional way and, in particular, to carry out component-wise refinement.

We can distinguish two approaches to structuring specifications — algebraic and type-theoretic. The distinction is best illustrated by considering how a datatype would be specified. There is a signature of basic types and operations, and axioms over this signature. The axioms are given using some base logic, so specifications are constructed on top of this. This generates a theory for the datatype consisting of all theorems provable from the axioms. In algebraic specification the generated theory is regarded as the specification, and structuring of specifications takes place at the level of the theory. For example, we might take the union of two theories.

This theory-level structuring should be contrasted with structuring the signature itself. In type theory, signatures with axioms can be given as existential types, and then specifications are combined using the type-theoretic constructors. For example, we can take the product of two specifications. The axioms are given within the type theory itself.

In Section 1.3.2 below we will suggest a third approach which combines aspects of both algebraic and type-theoretic specification.

1.3 Calculi

We observed in the discussion of refinement in Section 1.1 that the basic construct in the calculus is some means of expressing "some x such that P". We can factor this into "some x" and "... such that P", and study separate extensions of $\lambda^{\times\rightarrow}$ with each construct. We believe these extensions to be of independent interest providing, respectively, analyses of ‘pure’ refinement and of structured specifications.

Coming from another direction, we would expect refinement to subsume verification, and so should be able to extend the verification calculus of Section 1.2 in some minimal way to get a refinement calculus. We can ask the question, given a satisfaction system, what are the minimum additions needed to get a refinement calculus? There are essentially two things which need to be added — a notion of structured proof, so that structure can be transferred from the proof to a program, and a means of internalising backwards search. These two additions correspond to the extended calculi which we have mentioned:

- Structured proofs — refinement types
- Internalisation of backwards search — refinement terms.

1.3.1 Refinement Terms

So we first develop a simple equational theory of refinement based on a lambda calculus with true stubs. Rather than use the ML-like notation of `fun Sort l` $= ?_{\texttt{int list}}$, we will write the lambda term $\lambda l : \texttt{int list}.?_{\texttt{int list}}$. We will call lambda terms with the possibility of such stubs, *refinement terms*, and refer to this possibility of terms only being partially determined as *underdeterminism*. We are careful to make a distinction between underdeterminism and *nondeterminism*, which we regard as a computational, as opposed to a specificational, phenomenon. For example, we do not regard abstractions with underdetermined body as being determined, which is what some authors are led to do, by viewing terms as being nondeterministic. This difference is not just one of intuition — different axioms are satisfied. Moreover, we could imagine nondeterminism and underdeterminism arising together in a concurrent setting, for example. Then it would be particularly important to maintain a distinction. We will elaborate on these differences in Chapter 3.

Without logic, the calculus may seem too simple to be interesting but it is worth studying for a number of reasons. The full refinement calculus is quite complicated and the subcalculus can act as a stage towards understanding the full system. As we will see, this is justified by virtue of the full calculus being a conservative extension.

Secondly, by not having any logical annotations in the stubs, it is possible to automatically check for well-formedness and, in fact, to evaluate terms in some cases. The calculus could serve as a basis for a simple practical program development system.

There are two forms of judgement in the calculus. Letting r range over refinement terms, we give judgements for typing $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash r : \tau$ and refinement $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash r \sqsubseteq_\tau r'$. Refinement subsumes equality, which can be defined as mutual refinement.

As for the lambda calculus, it would be possible to regard typing as a derived judgement, with $r : \tau$ meaning $r \sqsubseteq_\tau r'$. However, it is clearer to keep the judgements separate.

There are refinement rules for decomposition

$$?_{\sigma\times\tau} \sqsubseteq_{\sigma\times\tau} \langle ?_\sigma, ?_\tau \rangle$$

and problem reduction

$$\Gamma, x : \tau \vdash ?_\tau \sqsubseteq_\tau x$$

Intuitively, we think of programs either in terms of how they evaluate, or as computing some mathematical value. We will think of abstract programs in terms of how they can be implemented. An intuitive way to think of this is that a term r corresponds to the set of realizers obtained by all possible ways of filling in the stubs with program code. Refinement, then, is the subset relation. We axiomatise refinement so that it is complete with respect to this semantics.

Based on this intuition, we can see that only determined terms can be substituted for variables, as substituting directly would duplicate underdeterminism. For example, the term $(\lambda n : \mathtt{nat}.\langle n, n\rangle)?_{\mathtt{nat}}$ should not be equal to $\langle ?_{\mathtt{nat}}, ?_{\mathtt{nat}}\rangle$. However, we do expect $(\lambda n : \mathtt{nat}.n)?_{\mathtt{nat}} = ?_{\mathtt{nat}}$. For similar reasons to those of the computational lambda calculus [Mog91], this leads us to introduce a 'let' construction, albeit with a different axiomatisation than there.

A direct motivation for introducing a let-construct comes from considering the result of combining underdetermined terms. Suppose two programs are being developed one of which depends on the other. Let the partially developed programs be $r_1[x]$ and r_2 where the free variable x is intended to be replaced by the program, the current state of development of which is r_2. If we want to consider the system as a whole, so as to prove some property say, then we cannot just substitute r_2 for x in r_1. The combined system can be represented by the `let` term

$$\mathtt{let}\ x : \tau\ \mathtt{be}\ r_2\ \mathtt{in}\ r_1$$

where τ is the type of r_2.

1.3.2 Refinement Types

We give a calculus in which we formalise the specification language and program logic. This task can be phrased in general terms as addressing 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. If we are just interested in input-output relations of programs then classical first-order logic will suffice for program logic.

In Section 1.2, we used the program logic to give specifications directly, but noted in Section 1.1.1 that $(x : \sigma)P \to (y : \tau)Q$ is a useful abbreviation for combining specifications. We will write $\Pi_{x:\phi}\psi$ for the specification of those functions which for all arguments x which satisfy ϕ return a result which satisfies ψ. This formulation of specifications is useful for reasoning inductively.

The general idea is to use type constructors to combine specifications. Similarly, we write $\Sigma_{x:\phi}\psi$ for the specification of pairs such that the left component x satisfies ϕ and the right satisfies ψ.

We also saw in Section 1.1.1 that it is natural to introduce a notion of equality at a specification. This leads us to take specifications as primitive rather than types.

This combination of the program logic with the type theory of the programming language is a form of *refinement types*. The general idea of refinement types 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.

This provides an alternative to simply using a program logic, or to using a type theory irrespective of any logic, and we discuss the advantages in Chapter 4.

This simple extension — replacing types by refinement types — affords a considerable degree of conceptual simplicity. Satisfaction of specifications by programs can then be formalised as a (refinement) typing judgement. Rather than write t `sat` ϕ we regard satisfaction as generalised typing and write $t : \phi$.

The terms of the λ-calculus are extended by allowing refinement types in abstractions. This subsumes the idea of indicating assumptions on the input by annotating the types in abstractions.

Contexts consist of variable assumptions of the form $x : \phi$. This means that the (refinement) typing judgement formalises the satisfaction of specifications by programs, under the assumption that some other programs satisfy specifications.

It is also convenient, though not essential, to also allow propositions as assumptions in the context.

Finally, we need to define a notion of refinement on specifications, *i.e.* on refinement types. This is the replacement of a specification by one more logically specific (and not the replacement of stubs by code).

If we just use the program logic, then refinement of specifications is no more than logical implication. With refinement types, however, we must define a refinement judgement, $\phi \sqsubseteq_\tau \phi'$, which we give as a form of subtyping on specifications over a type τ.

1.3.3 Refinement Calculus

We combine the calculi of stubs and specifications to get a logical refinement calculus. The fundamental construct is the logical stub, $?_\phi$, where ϕ is a refinement type. Using logical stubs to combine specifications with code gives us a useful means of specifying programs.

For example, using a refinement type `Sorted_List`, we can specify a search function as

$$\lambda n : \mathtt{nat}.?_{\Pi_{l:\mathtt{Sorted\_List}}(b:\mathtt{bool})b=\mathtt{true} \iff \mathtt{In(l,n)}}$$

The rules of the refinement calculus are the natural generalisations from the subcalculi. The satisfaction of specifications by programs is generalised to abstract programs, with the idea that an abstract program has a property if every program to which it could refine has that property. This lets us ask of partially developed programs what properties they are guaranteed to have, when fully written.

The refinement rules make use of the underlying logic, and just as we introduced a notion of equality at a specification, so we have refinement at a specification. We write $r \sqsubseteq_\phi r'$ for the refinement of r to r' at refinement type ϕ, but often omit the ϕ when not significant. In fact, the idea of the ϕ not being significant can be made formal.

So in the refinement calculus there are two kinds of refinement, one inherited from each subcalculus. This does reflect programming practice. A partially implemented program, r consists of a mixture of logical specification and program code. At any stage there are two options open: write some more code, replacing a piece of specification with concrete program, or modify some specification, either by replacing it with an equivalent specification, or with a more constrained one.

The connection between logic and refinement is given by the rule

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash ?_\phi \sqsubseteq ?_{\phi'}} \tag{1.1}$$

a particular case of which is

$$\frac{\Gamma, x : \tau \vdash P' \supset P}{\Gamma \vdash ?_{(x:\tau)P} \sqsubseteq ?_{(x:\tau)P'}}$$

and the rule

$$\frac{\Gamma, x : \phi \vdash r \sqsubseteq_{\psi} r'}{\Gamma \vdash \lambda x : \phi . r \sqsubseteq_{\Pi_{x:\phi}\psi} \lambda x : \phi . r'}$$

For example, in the search function specified above, we can use the information that l is sorted, expressed as the refinement typing l : `Sorted_List`, when refining the body.

These are sufficient to derive all uses of the logic in refinement. For example, refinement steps can generate proof obligations such as

$$?_{(z:\sigma\times\tau)P[z]} \sqsubseteq \langle ?_{(x:\sigma)Q[x]}, ?_{(y:\tau)R[y]} \rangle$$

when

$$\forall x : \sigma . \forall y : \tau . Q[x] \wedge R[y] \supset P[\langle x, y \rangle]$$

but this can be factored into an instance of (1.1) and decomposition.

To summarise then, we want a calculus where we can prove propositions, P, where we can prove refinement typings, $r : \phi$, and where we have correctness preserving refinement rules to prove refinements of the form, $r \sqsubseteq_{\phi} r'$, which generalise ordinary program equality. So we would like an extension of the simply-typed lambda calculus, where the usual equivalences hold, and in addition, we can express refinements.

We construct the refinement calculus in a modular fashion, as the combination of these two calculi. We are justified in understanding the refinement calculus in this way since it is a conservative extension of both of the subcalculi.

In Figure 1.1 we summarise the important judgements of the different calculi, and indicate the connections between the calculi and their classes of models. The upwards arrows are intended to indicate conservative extensions of calculi, and 'inclusions' of models, in the sense of there being a correspondence between the meanings of judgements.

1.4 Other Methodologies

Refinement is just one of many methodologies for formal program development: other approaches include program extraction and deliverables for example. The idea in all of these is to start with a formal description of the behaviour of a

Judgements

Refinement Terms

$$\Gamma \vdash r : \tau$$
$$\Gamma \vdash r \sqsubseteq_\tau r'$$

Refinement Types

$$\Gamma \vdash t : \phi$$
$$\Gamma \vdash t =_\phi t'$$
$$\Gamma \vdash \phi \sqsubseteq_\tau \phi'$$

Refinement Calculus

$$\Gamma \vdash r : \phi$$
$$\Gamma \vdash r \sqsubseteq_\phi r'$$
$$\Gamma \vdash \phi \sqsubseteq \phi'$$

r refinement terms $\quad$ ϕ refinement types
t determined terms $\quad$ τ types

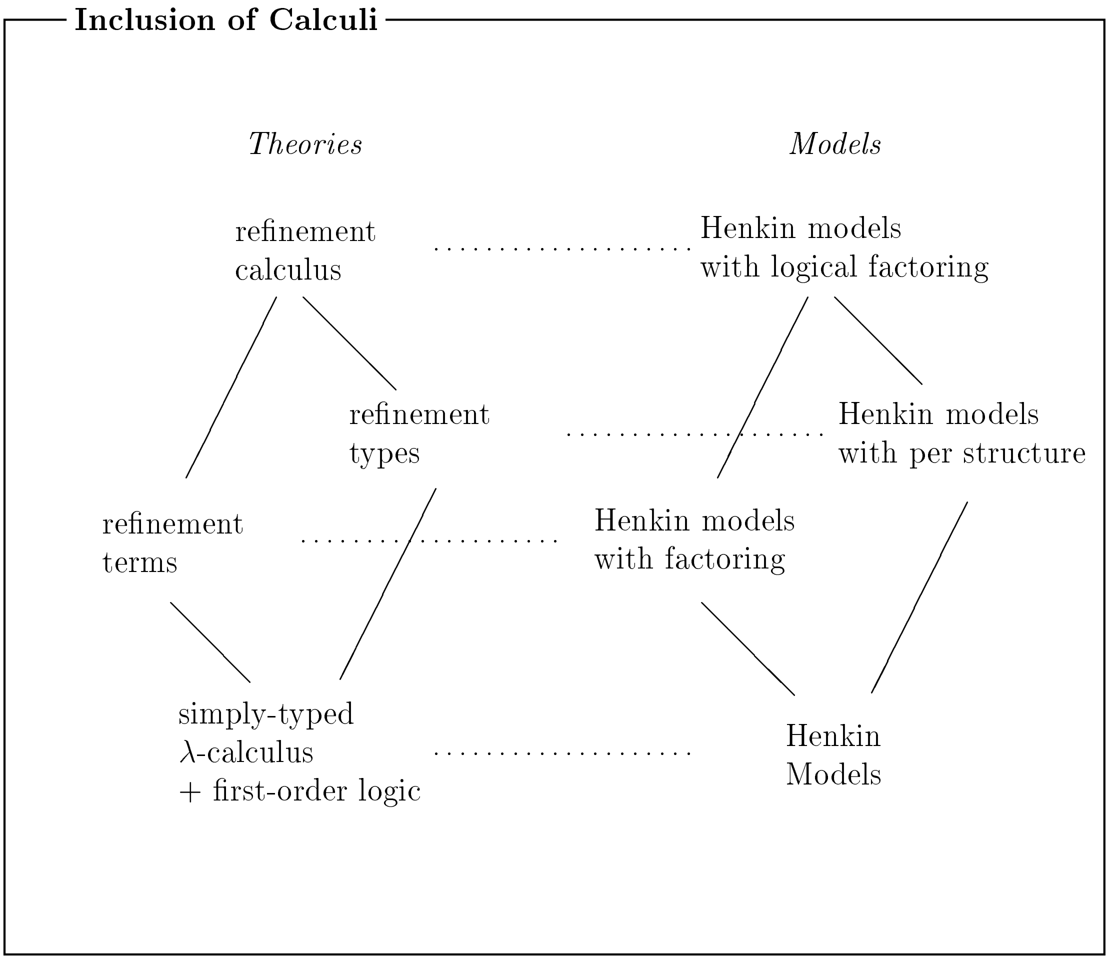

Figure 1.1: Overview of Calculi

program, usually expressed in some logical language, and construct a program which meets that description.

The idea in program refinement is to construct the program in a stepwise manner from the specification, and to have an explicit record of the stage of development in a wide-spectrum language. An advantage of using a wide-spectrum language is that it is often convenient to specify using a mixture of logic and algorithm. Refinement is a formal program development methodology since each refinement step preserves correctness, so the program is guaranteed to meet its description.

It is possible to study refinement via an encoding in a type theory (assuming that the programming language constructs can be suitably encoded — not necessarily the case in the presence of recursion). Luo [Luo91] gives an encoding of data refinement in the Extended Calculus of Constructions. An explicit calculus for refinement, however, has the advantage that it forces us to think directly about the formalism and the semantics.

Now, a contrast can be drawn between program and *data refinement*. In data refinement, a program written using an abstract data type, such as stacks, is rewritten to use a more concrete data type, lists say. The abstract data type `Stack` can be seen as a specification, and the implementation by lists, as a refinement of the program. Program refinement on the other hand, starts with a specification of a program, rather than a datatype. In this thesis we are concerned with program refinement.

Some authors see data refinement as being the central concept in program development. This is the basis of the VDM methodology [Jon90] for example. However, we believe that any calculus for data refinement would have to incorporate program refinement anyway, as the stepwise development of a datatype must include the stepwise development of its operations, that is, of programs, and so program refinement seems a natural starting point.

A similar concept to what we have called abstract programs, also used in program development, is that of program *skeletons*. These are templates of code expressing useful algorithm schemes, such as divide-and-conquer. We can regard refinement terms as a simple formalisation of skeletons, rather than taking them to be some kind of meta-entity. We cannot in general express skeletons using parameterisation, since parameters cannot range over code which can contain local variables.

To many people, "program refinement" and "program transformation" are synonyms. While the basic idea of either can be generalised to include the other, it is useful to draw a distinction between refinement of a logical specification into concrete code, and transformation of concrete program code into 'better' code. In this thesis then, our notion of refinement will not incorporate program transformation.

The other key aspect of our approach to refinement is that we work within an equational paradigm, viewing refinement as a kind of generalised (in)equality: $\mathtt{spec} \sqsubseteq \mathtt{prog}$. Although equality seems a natural thing to consider when studying lambda calculi, a notion of program equivalence is not primary for reasoning about programs in some languages. It is not so important, for example, in hardware derivation. Nevertheless, we study refinement in this higher-order manner: given two terms, prove one refines the other.

This is not the only possibility however. We could have a search-oriented system where the user starts with the specification and directly refines it to a program without explicitly indicating a refinement. This is the more likely to be useful in practice but, curiously, the equational paradigm has received far more attention from the refinement calculus community. A search-oriented system might be a more natural formalism for work on program synthesis. We leave it as conjecture, for the time being, that an equational refinement calculus is the theory generated from a search calculus.

1.5 Choices

The use of $\lambda^{\times\rightarrow}$ involves two choices: a typed language and a functional language. By providing an element of syntax-direction, types help in narrowing down the number of refinement rules which can be applied at any stage. As explained above, types can be used to structure specifications.

We choose to use a functional language, simply because the theory is better understood. The lambda calculus is a paradigmatic functional language, and comes in many 'flavours'. As part of a longer term research plan, we can tackle the problem of finding refinement calculi for complicated computational scenarios in a modular way by first finding a refinement calculus for the simply-typed lambda calculus, and then extending it in a suitable way.

We do not consider full recursion. This may seem like a significant omission, but non-termination would be a significant addition to the calculus and require various choices not central to the basic theory of refinement. Instead, we use *well-founded recursion* which is sufficiently expressive to get interesting programs. The reasoning involved in constructing fully recursive *terminating* programs is the same as in constructing programs which use well-founded recursion.

We use classical first-order logic for our program logic. The intention is that we should be able to choose an arbitrary logic, and the corresponding refinement calculus would be induced by the general rules. However, it turns out that in fact, we must choose an *extensional* logic. That is, a logic for which t and t' are extensionally equal whenever for all propositions P, $P[t/x] \iff P[t'/x]$. The refinement calculus makes essential use of replacing 'equals by (extensional) equals' which would not be possible if our specifications described intensional properties of programs. Intuitionistic logic also has this property; all that matters is that the atomic propositions are extensional. Classical first-order logic is an example of a simple and expressive logic.

As for lambda calculi, extensional logics are better understood than more intensional calculi, so it is more likely that connections can be made between this and other work.

The approach we have taken here is to start with a programming language, a specification language, an equational theory and a notion of satisfaction. It is reasonable to understand the language through its equational theory when we are only concerned with extensional properties. However, an alternative approach would be to start with an operational semantics for the programming language, and understand satisfaction operationally. For example, the equational theory considered here could be generated from a call-by-name operational semantics. The sort of question that might be addressed then would be to find the natural refinement calculus corresponding to a call-by-value semantics, say.

1.6 Related Work

We group relevant work by way of comparison with the calculi of this thesis. We describe related concepts to underdeterminism, work on structured specifications, and some calculi of program development.

1.6.1 Refinement Terms

The study of indefinite descriptions — 'some x such that P' — goes right back to the earliest work on modern logic (*e.g.* [Ros39]) but the idea there is that a description ranges over semantic values. Hilbert and his collaborators introduced the ϵ-operator (see the monograph [Lei69]), as a formalisation of indefinite descriptions, in order to provide an alternative formulation of mathematical logic. The expression $\epsilon x.P$, meaning 'some x such that P', is always defined, and denotes some unknown, but fixed, element which satisfies P, if one exists, and otherwise denotes some arbitrary, but fixed, element.

The logic of ϵ-expressions is modelled using some arbitrary, but fixed, choice function which picks out a member of each nonempty set and returns anything for the empty set. This means, then, that the value of $[\![x \vdash \epsilon y.\top]\!](n)$ is fixed, for any given interpretation of terms, and does not depend on n. Thus, the abstraction $\lambda x.\epsilon y.\top$ denotes a constant function. Thus, the ϵ-operator for indefinite descriptions is essentially a localisation of global variables. Although descriptions can appear embedded anywhere in a term, this is just like using global variables since the denotation must be a value.

We might imagine that a similar technique could be used for our purposes as the terms of our calculus can be thought of a kind of parameterised programs. The simplest choice would be to just consider terms with a free variable in the global context, representing a 'hole' to be filled in with program code. This will not work however, since variable capture prevents the variable being replaced by programs which contain local variables. Therefore we must embed underdeterminism locally in the terms with the $?_\tau$ construct. We will discuss this further in Chapter 3.

Hermida and Jacobs' study of indeterminates in the lambda calculus [HJ95] is essentially the same form of global indeterminacy and does not account for substitution allowing variable capture.

Although much work has been published on refinement calculi, there seem to be no fully axiomatised systems. Morgan [Mor94] describes the 'classical' refinement calculus, developed independently by Back, Morgan and Morris. This is an imperative language extended with specification constructs. Their system however, uses *nondeterminism* to express specification constructs. We believe this to be a mistake, as nondeterminism is a computational phenomenon distinct from our view of underdeterminism as a specificational phenomenon at a level above the programming language. Moreover, we might want to consider a combination of nondeterminism and underdeterminism, for example when developing a logic program.

Proof development systems, such as Lego [LP92], allow users to interactively construct a proof by refinement. Intermediate states in a proof development may be modelled as underdetermined terms. The idea there of allowing *existential* variables in terms is similar to our refinement terms. In a similar vein, so-called *logical* variables have been used in artificial intelligence, and are essentially the same concept.

We make a more extensive comparison with the refinement calculi of Back *et.al.*, and with Lego, in Section 5.4.

The concept of underdeterminism also arises in linguistics (with the name 'underspecification'), where semantically ambiguous statements such as "every student is $?_{\texttt{noun\_phrase}}$" are studied. Bos [Bos95] for example, considers a language with metavariables for representing such statements.

1.6.2 Refinement Types

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", gave a refinement type system for expressing properties of mini-ML programs [FP91]. In another work [Pfe93] he gave an extension to LF with (possibly intensional) properties such as "in normal form" (a property of derivations), given as refinement types. He does not allow refinement types in abstractions though. In both works, the idea is that refinement types offer greater expressivity but carefully restricted to retain desirable properties.

The paper of Coppo, Damiani and Giannini [CDG96] is quite similar to our approach, using refinement types for dead code elimination. They also give a semantics using pers.

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 refinement types) can be defined in the logic, but is not explicitly axiomatised. Talcott [Tal90] 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. However, since she lacks a typed equality, local assumptions cannot be discharged and conclusions take the form "$t = t'$ if x satisfies ϕ", rather than $\lambda x.t =_{\phi\rightarrow\psi} \lambda x.t'$, say.

Other type-theoretic approaches include [Asp95, AC96], 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. Aspinall's [Asp95] dependent type theory, $\lambda_{\leq\{\}}$, is formally similar in that it has subtyping on dependent functions and products. Dependency in $\lambda_{\leq\{\}}$ comes from singleton types, which are a special case of subset types. The purpose of Aspinall's system is not to be a specification language, however, but to give a type structure to specification building operations. His system is based on subtyping rather than refinement types. We discuss Aspinall's thesis further in 5.4.2.

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. The type theory can also internalise notions of realisability and refinement.

As pointed out by Hayashi, schemas in the Z specification language [Spi92] 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. In the conclusion to [McK92], McKinna suggests dropping the requirement for proof existence and, moreover, that implementations should be regarded as being equal up to some extensional equivalence. He proposes a definition of specification which includes an explicit definition of per. Our calculus could be regarded as an internal language for this notion.

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 by giving an explicit syntax and axiomatisation. The existential form of Martin-Löf's type theory with subset types in [NPS90] is similar, and indeed, our work on refinement types could be regarded as providing an alternative interpretation of their system.

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.

1.6.3 Refinement Calculi

Although there have been many papers on refinement calculi, no authors in this area seem to have presented explicit *proof-theoretic* axiomatisations of refinement, or given a logic for reasoning about refinement terms. Laws are usually introduced as needed. Proving properties of partially developed programs seems not to have been considered before, except in the sense of regarding abstract programs as specifications so that satisfying a property amounts to refinement.

The classical refinement calculus of Back, Morgan and Morris [Bac88, Mor94, Mor87], based on Dijkstra's Guarded Command Language [Dij76], is a calculus for deriving imperative programs from specifications expressed in terms of pre- and postconditions in first-order logic. They do not consider refinement on expressions. The Guarded Command Language is nondeterministic and, though not a refinement calculus, seems to have influenced later refinement calculi in their use of nondeterminism for specification.

Bunkenburg [Bun97] continued their approach for a functional language, retaining some imperative features using a state monad in the style of the computational lambda calculus. Norvell and Hehner [NH92] and Ward [War94] consider functional languages based on the untyped lambda calculus.

All these authors have based their calculi on nondeterminism which, we will see, has consequences for the axiomatisation of refinement.

In the presence of nontermination, this use of nondeterminism gives rise to a choice between demonic and angelic nondeterminism, a choice arising from computational considerations which we believe to be unnecessary. However, we do not consider nontermination here. Most authors have considered total correctness, which leads to the use of demonic nondeterminism. Ward also adds angelic nondeterminism, though it is doubtful whether this brings any advantages when not considering concurrency.

We mentioned the algebraic approach to specification in Section 1.2 above. This is a program specification and development methodology [vL90] centred on the use of abstract data types. An ADT is specified to be an algebra (in the sense of universal algebra) with a given signature, which satisfies a collection of axioms. Thus, a simple specification, $\langle \Sigma, E \rangle$, consists of a signature, Σ, and a collection of axioms, E, over that signature. There are various theory-level operators for combining such specifications. In the simplest approach, programs are thought of as total algebras, so there are various extensions to cope with partiality, errors and so on.

There are two styles of semantics. On the one hand, a specification can be viewed as an exact description of a program, so the semantics is defined as some specific algebra (such as the initial or terminal algebra for this specification, with respect to algebra homomorphisms). On the other hand, the specification can be taken to be a description of the required properties of the program, but leaving some possibilities open. In this 'loose' approach, the semantics of a specification is taken to be the collection of all algebras which satisfy the specification, or possibly some restriction on this (such as all reachable algebras).

The loose approach is appropriate for program development. Here, refinement is thought of as "the implementation of one specification by another", and is defined formally as: SP' is an implementation of SP if $sig(SP') = sig(SP)$ and $mod(SP') \subseteq mod(SP)$, that is, as model inclusion over the same signature. This is a very general semantic definition of refinement. There are no axioms for actually proving refinements, for example. More elaborate notions in terms of 'constructors' and abstraction have been developed by Sannella and Tarlecki [ST87].

Extended ML [San91, KST97] is a wide-spectrum language which extends (a subset of) the functional programming language ML. It is similar to our calculus in that there are essentially two specification features — a place holder, ?, and the facility to incorporate logical axioms in signatures. See Section 5.4.1 for further discussion.

Bednarczyk and Borzyszkowski [BB95] present a system of rules for finding programs which inhabit specifications. Although they have partial terms representing intermediate steps in the search for inhabitation, they do not have an explicit refinement relation. The Lego proof system [LP92] has a notion of refinement of proof state. If we regard refinement terms as being representations of such a proof state, then Lego's notion of refinement is like ours.

The relational calculus, Ruby [JS91], is essentially an untyped functional language extended with inverses. It is the use of inverses which gives the language specificational power. Nevertheless, specifications in Ruby are usually functions, and refinement amounts to equational transformation.

1.7 Summary of Thesis

In Chapter 2 we make some preliminary definitions which will be used in the rest of the thesis. We give the basic equational theory of the simply-typed lambda calculus, and describe the applied theory of booleans and naturals. We give a first-order logic theory over this, explaining how induction is formalised. The calculus and logic can be given a semantics using Henkin models, a class of non-standard set-theoretic models which we define. We prove soundness and completeness of the equational and logical theories with respect to this class of models.

In Chapter 3 we give the calculus of refinement terms. We add $?_\tau$ terms to the simply-typed lambda calculus, as a formalisation of true stubs, getting a system for the study of what we call *simply-typed underdeterminism.* The judgements of the calculus are

$$\Gamma \vdash r : \tau$$

$$\Gamma \vdash r \sqsubseteq_\tau r'$$

We show that terms can be expressed in a particular canonical form and use this to derive some results about refinement. We then show that all refinements can be given in a standard form in which replacement of stubs by code precedes all equational reasoning.

The terms can be interpreted in Henkin models, with each type σ being ascribed a set $\sigma^{\mathcal{A}}$, and terms in context $\Gamma \vdash r : \sigma$ interpreted, given the appropriate notion of environment, as subsets of $\sigma^{\mathcal{A}}$. We show that terms can be expressed in a canonical form, and use this to prove completeness of the calculus with respect to the class of models.

We study a first-order logical theory of refinement, where the atomic propositions are refinements, and give a semantics using Henkin models, for which the logic is proven sound and complete. We infer that the logical theory is conservative over the equational theory.

In fact, since the first-order logic of equality in Chapter 2 is also complete with respect to this class of models, we are able to show that every refinement is equivalent to a proposition in first-order logic involving only equality.

In Chapter 4 we give the calculus of refinement types. This involves an analysis of a suitable notion of specification for refinement, independently of considerations of underdeterminism. We argue that it is natural to use (a syntax representing) *partial equivalence relations* (pers) for specifications. Terms, then, denote equivalence classes of pers. Refinement typing subsumes typing and formalises the satisfaction of specifications by programs. There is a refinement relation on refinement types.

The judgements are

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

$$\Gamma \vdash P$$

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

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

The calculus is interpreted using a per structure over the sets in a Henkin model. We prove soundness and completeness with respect to such 'Henkin pers', which lets us conclude that the system is a conservative extension of both the simply-typed lambda calculus, and of first-order logic.

In Chapter 5 we present the combination of the two subcalculi, giving a refinement calculus for the refinement of specifications in first-order logic into lambda terms. The syntax of the full refinement calculus is

$$\begin{array}{rcl}
\phi & ::= & \mathbf{1} \mid \gamma \mid \Sigma_{x:\phi}\psi \mid \Pi_{x:\phi}\psi \mid (x:\phi)P \\
r & ::= & x \mid k(r_1,\ldots,r_n) \mid * \mid \langle r, r'\rangle \mid \lambda x:\phi.r \mid ?_\phi \mid \pi_1(r) \mid \pi_2(r) \mid \\
 & & rr' \mid \mathtt{let}\ x:\phi\ \mathtt{be}\ r\ \mathtt{in}\ r' \\
P & ::= & \bot \mid P \supset P' \mid \forall x:\phi.P \mid F(r_1,\ldots,r_n) \mid r \sqsubseteq_\phi r' \\
\Gamma & ::= & \langle\rangle \mid \Gamma, x:\phi \mid \Gamma, P
\end{array}$$

We show that the subcalculi can be embedded in the full refinement calculus by defining relations between terms which generalise the nonlogical refinement and logical equality of those systems. We show that logical refinement can be factored into these two relations. We also extend the result from Chapter 3 on standardisation of refinements to the full calculus.

Unlike the calculus of simple underdeterminism, refinement terms here can not be interpreted as sets. Both refinement terms and refinement types are interpreted as pers in Henkin Models, and we show how this generalises the semantics of the subcalculi.

We prove soundness of the calculus, but have to be careful with how completeness is formulated. For a restricted class of terms, omitting any higher-order features, we have completeness of the various judgements with respect to interpretation in the class of models.

In Chapter 6 we make some conclusions and suggestions for future work, suggesting how the simple notion of refinement described here could be extended to other situations, and how refinement itself might be incorporated into a larger theory of program development.

1.8 Notation

Following Martin-Löf [Mar96] we refer to the atomic statements of a theory as *basic judgements*. These are either judgements of well-formedness or of truth. The most general form of judgement is in *hypothetico-general* form, that is, under the assumption of hypotheses and in a context of free variables. In this thesis we will work in contexts consisting of a combination of variables and propositional assumptions. We will use Γ as a metavariable for contexts, appropriate to whichever calculus is under consideration. We write $\langle\rangle \vdash B$ for a judgement in the empty context, or just B.

The metavariable conventions used throughout this thesis are listed in Appendix A. The top-level grammar of the syntactic categories is

$$\begin{array}{lrcl}
\text{Expressions} & U & ::= & r \mid \phi \mid P \\
\text{Basic Judgements} & B & ::= & r : \phi \mid P \mid \phi : \mathtt{Ref}\,(\tau) \mid P\ \mathtt{wf} \mid \Gamma\ \mathtt{wf} \\
\text{Judgements} & J & ::= & \Gamma \vdash B
\end{array}$$

Chapter 2

Preliminaries

In this chapter we give the theoretical basis on which we will build the refinement calculus. We first describe the simply-typed lambda calculus with products and explain how to formulate applied theories. We use a theory of booleans and naturals with primitive recursion as a particular example. We define Henkin interpretations and models, and use them to give a semantics to the lambda calculus. We then describe first-order logic over the equational theory of the simply-typed lambda calculus and show how it can also be modelled using Henkin interpretations, for which we prove completeness theorems.

2.1 Simply-typed Lambda Calculus

We base the theory of refinement on an explicitly typed (Church style) formulation of the simply-typed lambda calculus [Cro93, Mit96]. Here "simply-typed lambda calculus" means with finite products and some axioms over a signature with ground types γ and constants k. The emphasis on axioms will be a feature of this thesis.

2.1.1 Syntax

The terms of an applied lambda theory are with respect to some signature of ground types and constant symbols.

Definition 2.1.1 *A* type signature *consists of a collection of symbols, which we call* ground types*. We define the* simple types *over a type signature, $\mathcal{G}$, by the grammar*

$$\tau ::= \mathbf{1} \mid \gamma \mid \tau \times \tau \mid \tau \to \tau$$

where $\gamma \in \mathcal{G}$.

For technical reasons, we will assume that all signatures in this thesis are countable, but will not bother to repeat this assumption.

In order to define the notion of constant over a type signature, we follow Mitchell [Mit96] in making a distinction between *types* and *sorts*. To each primitive constant we ascribe a *sort* — a metalevel construct which explains how to form well-formed terms using the constant.

Definition 2.1.2 *A* sort *over type signature, $\mathcal{G}$, with* arity n*, is a list of $n+1$ simple types over $\mathcal{G}$. We write the sort $[\tau_1, \ldots, \tau_n, \tau]$ using the functional notation $\tau_1, \ldots, \tau_n \to \tau$ (where $n \geq 0$).*

Definition 2.1.3 *Let $\mathcal{G}$ be a type signature. We define a* constant signature*, $\mathcal{K}$, over $\mathcal{G}$, to be a collection of symbols —* constants *— each of which is ascribed a sort over $\mathcal{G}$.*

Definition 2.1.4 *A* $\lambda^{\times\to}$-signature *Sg consists of a type signature, $\mathcal{G}$, and a constant signature, $\mathcal{K}$, over $\mathcal{G}$.*

We write $k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K}$ to indicate that constant k in signature $Sg = \langle \mathcal{G}, \mathcal{K} \rangle$ has sort $\tau_1, \ldots, \tau_n \to \tau$. This does not mean that k itself is a well-typed term, but that with n well-formed arguments of the correct types, $t_1 : \tau_1$ up to $t_n : \tau_n$, the term $k(t_1, \ldots, t_n)$ is well-formed with type τ. For example, the conditional, `if _ then _ else _`, has sort $\texttt{bool}, \tau, \tau \to \tau$, but is not itself a well-formed term with type $\texttt{bool} \times \tau \times \tau \to \tau$.

Alternatively, we could have chosen to give all constants a functional type, but the choice is not significant here. In practice, we will drop the brackets around the arguments to constants and allow ourselves to use any form of mixfix notation.

Rather than write the sort of nullary constants as $k :\to \tau$, we will just write $k : \tau$. Since we will not consider any nullary constants of functional type, there is no ambiguity with unary constants.

Definition 2.1.5 *The* preterms *over signature Sg are given by the grammar*

$$t ::= x \mid k(t_1, \ldots, t_n) \mid * \mid \langle t, t' \rangle \mid \lambda x : \tau . t \mid \pi_1(t) \mid \pi_2(t) \mid tt'$$

For each type we assume a countably infinite number of variables drawn from some set so, strictly speaking, the set of preterms is parameterised on both the signature and the variables. We adopt the convention that in writing $x : \tau$, the variable x is drawn from the set of variables of type τ.

We will use γ and k as metavariables for ground types and constant symbols respectively. Henceforth, we will assume the existence of some signature when writing k and γ.

The contexts are constructed from types and variables via the grammar:

$$\Gamma ::= \langle\rangle \mid \Gamma, x : \tau$$

We use the notation $\Gamma, x : \tau, \Gamma'$ (with the obvious meaning) to construct contexts, and assume that all the variables in a context are distinct. This assumption of well-formedness is external to the system (or 'logicistic'). In later calculi we will give explicit rules for the well-formedness of contexts. The ordering of variables is not actually important in the simply-typed lambda calculus, but is significant in the extended calculi of later chapters.

The typing judgement $\Gamma \vdash t : \tau$ means "under assumption Γ, term t has type τ." The typing rules of $\lambda^{\times\rightarrow}$ are standard [Cro93, Mit96] and we do not repeat them here. All judgements are given in a context of variable typings, $x_1 : \tau_1, \ldots, x_n : \tau_n$. We write $Sg \triangleright \Gamma \vdash t : \tau$ when the judgement $\Gamma \vdash t : \tau$ is derivable from signature Sg, but will drop the $Sg \triangleright$ when it is clear which signature is intended.

We will write $FV(t)$ for the set of free variables in preterm t, and use the notation $t[t'/x]$ to indicate the (capture avoiding) substitution of t' for each free occurrence of variable x in t. The simultaneous substitution of tuple g (or *syntactic environment*) for the variables in context Γ in t is indicated similarly by $t[g/\Gamma]$. We sometimes write $t[x]$ to distinguish all the free occurrences of variable x in t. This does not mean that x is the only free variable, nor that it actually appears free in t. When writing $t[x]$, we may use $t[t']$ to indicate the substitution $t[t'/x]$.

We adopt the usual notational conventions of the lambda calculus to avoid excessive bracketing: for example, $\lambda x : \tau.tx$ means $\lambda x : \tau.(tx)$. Round brackets ( ) will sometimes be used to increase readability. We will write $\equiv$ for syntactic equivalence (that is, up to bracketing and α-equivalence), and contrast this with $=$ for the provable equality defined below.

The obvious extensions of these conventions hold for the other syntactic categories introduced later.

Remark 2.1.6 Weakening, permutation and substitution rules are derivable for the typing judgement. This is not the case with the equality judgement, because of the presence of axioms, so we will add rules for these in the next section.

2.1.2 $\lambda^{\times\to}$-Axiom Systems

We give extralogical axioms (on top of the logical axioms of the basic theory) with respect to a particular signature. It is common in the literature to not distinguish between a collection of axioms and the theory they generate, but we will do so here.

Definition 2.1.7 *A* $\lambda^{\times\to}$-axiom system *consists of a* $\lambda^{\times\to}$*-signature,* Sg*, and a collection,* Ax*, of equations in context,* $\Gamma \vdash t =_\tau t'$, well-typed with respect to Sg*, that is,* $Sg \triangleright \Gamma \vdash t : \tau$ *and* $Sg \triangleright \Gamma \vdash t' : \tau$*.*

The theorems of $\lambda^{\times\to}$ are generated using rules of two kinds — rules for the pure theory of λ-calculus, and rules for inferring theorems from these and the axiom system.

In Figures 2.1 and 2.2 we give the rules for the pure theory of the λ-calculus, and in Figure 2.3 we give the additional rules necessary for inferring theorems from an arbitrary axiom system. It is possible to give a stronger rule of **Substitution** (Figure 2.3) which would let us derive the congruence rules, but we prefer to give them explicitly thus establishing the pattern for later calculi.

Definition 2.1.8 *Let* $\langle Sg, Ax\rangle$ *be a* $\lambda^{\times\to}$*-axiom system. We define the theorems of* $\langle Sg, Ax\rangle$ *to be the equations which can be inferred using the rules of Figures 2.1, 2.2 and 2.3. We write* $\langle Sg, Ax\rangle \triangleright \Gamma \vdash t =_\tau t'$ *to indicate that equation* $\Gamma \vdash t =_\tau t'$ *is a theorem of axiom system* $\langle Sg, Ax\rangle$*.*

In general, we do not assume that all types are inhabited (by closed terms). Recall here that the (complete) equational theory of the simply-typed lambda calculus differs depending on whether or not empty types are allowed in the semantics [MM91, MMMS87]. We will explicitly state the assumption of inhabitation when necessary, such as when giving a completeness theorem.

2.1.3 Booleans and Naturals

We will use booleans and natural numbers as a running example of an axiom system throughout the thesis. Although we do not consider more complex data types, this could be regarded as a simple case study in how datatypes are treated in refinement calculi. We add booleans and naturals as ground types:

$$\gamma ::= \texttt{bool} \mid \texttt{nat}$$

Reflexivity

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash t =_\tau t}$$

Symmetry

$$\frac{\Gamma \vdash t =_\tau t'}{\Gamma \vdash t' =_\tau t}$$

Transitivity

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

Figure 2.1: Rules for Equational Reasoning

The constants are:

$$k ::= \mathtt{0} \mid \mathtt{succ} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\ \_\ \mathtt{then}\ \_\ \mathtt{else}\ \_ \mid \mathtt{natrec}$$

In this section we take some time to explain the meaning of the various constants. We use $\bar{\mathtt{b}}$ to signify either of the boolean truth values, and $\bar{\mathtt{n}}$ for numerals. We use b and n as metavariables for expressions of type `bool` and `nat` respectively.

In fact, we have a family of recursion operators, $\mathtt{natrec}_\tau$ and so on. Rather than make this explicit or introduce polymorphism we just ignore this (unimportant) point. Similarly, there are separate conditionals for each type τ but we ignore this too.

We give the sortings and equations for booleans and naturals in Figures 2.5 and 2.4. There is a constant for *primitive recursion.*

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

Although we axiomatise constants equationally (as opposed to giving an operational semantics), the idea is that $\mathtt{natrec}\ z\ s\ n$ computes a loop in which s is applied n times to z, where s can also use the stage n. For example, a function to add up the first n naturals is $\mathtt{sum}\ n = \mathtt{natrec}\ 0\ (\lambda x : \mathtt{nat} \,.\, \lambda y : \mathtt{nat} \,.\, \mathtt{add}\ x\ y)\ n$. We will allow ourselves the abuse of language referring to loops and termination.

The primitive recursion $\mathtt{natrec}\ z\ s\ n$ will loop at most n times, so is guaranteed to 'terminate'. This corresponds to `for` loops in imperative languages.

Function Equations

$$\frac{\Gamma, x : \sigma \vdash t : \tau \quad \Gamma \vdash t' : \sigma}{\Gamma \vdash (\lambda x : \sigma . t)t' =_\tau t[t'/x]} \qquad (\beta)$$

$$\frac{\Gamma \vdash t : \sigma \to \tau}{\Gamma \vdash \lambda x : \sigma.(tx) =_{\sigma \to \tau} t} \; (x \notin FV(t)) \qquad (\eta)$$

Product Equations

$$\frac{\Gamma \vdash t_1 : \sigma_1 \quad \Gamma \vdash t_2 : \sigma_2}{\Gamma \vdash \pi_1 \langle t_1, t_2 \rangle =_{\sigma_1} t_1} \qquad \frac{\Gamma \vdash t_1 : \sigma_1 \quad \Gamma \vdash t_2 : \sigma_2}{\Gamma \vdash \pi_2 \langle t_1, t_2 \rangle =_{\sigma_2} t_2}$$

$$\frac{\Gamma \vdash t : \sigma \times \tau}{\Gamma \vdash \langle \pi_1(t), \pi_2(t) \rangle =_{\sigma \times \tau} t}$$

Unit Equation

$$\frac{\Gamma \vdash t : \mathbf{1}}{\Gamma \vdash t =_{\mathbf{1}} *}$$

Congruence Equations

$$\frac{\Gamma, x : \sigma \vdash t =_\tau t'}{\Gamma \vdash \lambda x : \sigma . t =_{\sigma \to \tau} \lambda x : \sigma . t'}$$

$$\frac{\Gamma \vdash t_1 =_{\tau_1} t_1' \;\cdots\; \Gamma \vdash t_n =_{\tau_n} t_n'}{\Gamma \vdash k(t_1, \ldots, t_n) =_\tau k(t_1', \ldots, t_n')} \; (k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K})$$

$$\frac{\Gamma \vdash t_1 =_{\sigma \to \tau} t_1' \quad \Gamma \vdash t_2 =_\sigma t_2'}{\Gamma \vdash t_1 t_2 =_\tau t_1' t_2'} \qquad \frac{\Gamma \vdash t_1 =_\sigma t_1' \quad \Gamma \vdash t_2 =_\tau t_2'}{\Gamma \vdash \langle t_1, t_2 \rangle =_{\sigma \times \tau} \langle t_1', t_2' \rangle}$$

$$\frac{\Gamma \vdash t =_{\sigma \times \tau} t'}{\Gamma \vdash \pi_1(t) =_\sigma \pi_1(t')} \qquad \frac{\Gamma \vdash t =_{\sigma \times \tau} t'}{\Gamma \vdash \pi_2(t) =_\tau \pi_2(t')}$$

Figure 2.2: Equality rules

Axioms

$$\frac{}{\Gamma \vdash t =_\tau t'} \ (\Gamma \vdash t =_\tau t' \in Ax)$$

Weakening

$$\frac{\Gamma_1, \Gamma_2 \vdash t =_\tau t'}{\Gamma_1, x : \sigma, \Gamma_2 \vdash t =_\tau t'}$$

Permutation

$$\frac{\Gamma_1, x_1 : \sigma_1, \Gamma_2, x_2 : \sigma_2, \Gamma_3 \vdash t =_\tau t'}{\Gamma_1, x_2 : \sigma_2, \Gamma_2, x_1 : \sigma_1, \Gamma_3 \vdash t =_\tau t'}$$

Substitution

$$\frac{\Gamma, x : \sigma \vdash t_1 =_\tau t_1' \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1[t_2/x] =_\tau t_1'[t_2/x]}$$

Figure 2.3: Theorems Generated from an Axiom System $\langle Sg, Ax \rangle$

Sortings

$$\texttt{true} : \texttt{bool}$$

$$\texttt{false} : \texttt{bool}$$

$$\texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_ : \texttt{bool}, \tau, \tau \to \tau$$

Axioms

$$x : \tau, y : \tau \vdash \texttt{if true then}\ x\ \texttt{else}\ y =_\tau x \qquad (\beta_{\texttt{true}})$$

$$x : \tau, y : \tau \vdash \texttt{if false then}\ x\ \texttt{else}\ y =_\tau y \qquad (\beta_{\texttt{false}})$$

$$f : \texttt{bool} \to \tau, b : \texttt{bool} \vdash f(b) =_\tau \texttt{if}\ b\ \texttt{then}\ f(\texttt{true})\ \texttt{else}\ f(\texttt{false}) \qquad (\eta)$$

Figure 2.4: Axiom System for Booleans

Sortings

$$\texttt{0} : \texttt{nat}$$

$$\texttt{succ} : \texttt{nat} \to \texttt{nat}$$

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

Axioms

$$z : \sigma, s : \texttt{nat} \to \sigma \to \sigma \vdash \texttt{natrec}\ z\ s\ \texttt{0} =_\sigma z$$

$$z : \sigma, s : \texttt{nat} \to \sigma \to \sigma, n : \texttt{nat} \vdash \texttt{natrec}\ z\ s\ (\texttt{succ}\ n) =_\sigma s\ n\ (\texttt{natrec}\ z\ s\ n)$$

Figure 2.5: Axiom System for Naturals

Iteration is a special case of (primitive) recursion, in which the stage number is not used.

$$\texttt{natiter}\ z\ s\ n = s^n(z)$$

where s^n is the n-th composite of s. Formally,

$$\texttt{natiter} = \lambda z : \tau \,.\, \lambda s : \tau \to \tau \,.\, \lambda n : \texttt{nat} \,.\, \texttt{natrec}\ z\ (\lambda a : \tau . \lambda x : \tau . s\ x)\ n$$

In the subsequent chapters we will explain how this treatment of constants is extended to richer calculi, using naturals and booleans as examples.

There are two β-equalities for booleans, and one η-equality. From the η-equality we can deduce

$$\texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t =_\sigma t$$

as well as the commuting conversion

$$f(\texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t') = \texttt{if}\ b\ \texttt{then}\ f(t)\ \texttt{else}\ f(t')$$

2.2 Models of Simply-typed Lambda Calculus

We give interpretations of the calculus in Henkin models. These are a form of ‘non-standard’ set-theoretic model for which simply-typed lambda calculi (in particular, systems containing arithmetic) are complete. As is usual in concrete models of applied lambda calculi, we must consider Henkin models in order to get completeness.

We give a Church style semantics by interpreting typing derivations, and write $\Gamma \vdash t : \tau$ as a linear shorthand for the derivation of that judgement. (In Chapters 4 and 5 we will interpret 'pre-judgements' rather than derivations.)

For a fixed $\lambda^{\times\rightarrow}$-signature, we define Henkin interpretations in two stages.

Definition 2.2.1 *Let Sg be a $\lambda^{\times\rightarrow}$-signature. A Sg-*applicative structure *(with products) is a tuple of families indexed by the types generated by Sg:*

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

To each type σ (not just ground types) we ascribe a set $\sigma^{\mathcal{A}}$, and to each constant $k : \tau_1, \ldots, \tau_n \to \tau$, a function $k^{\mathcal{A}} : \tau_1^{\mathcal{A}} \times \cdots \times \tau_n^{\mathcal{A}} \to \tau^{\mathcal{A}}$. There are projection *and* application *maps:*

$$\mathtt{Proj}_1^{\sigma,\tau} : (\sigma \times \tau)^{\mathcal{A}} \to \sigma^{\mathcal{A}}$$

$$\mathtt{Proj}_2^{\sigma,\tau} : (\sigma \times \tau)^{\mathcal{A}} \to \tau^{\mathcal{A}}$$

$$\mathtt{App}^{\sigma,\tau} : (\sigma \to \tau)^{\mathcal{A}} \times \sigma^{\mathcal{A}} \to \tau^{\mathcal{A}}$$

A Henkin interpretation is an applicative structure with two additional conditions, namely, that it is extensional, and that it satisfies the environment model condition.

Definition 2.2.2 *An applicative structure with products*

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

is extensional *when*

- $\mathbf{1}^{\mathcal{A}}$ *has exactly one element*
- *for all $f, f' \in (\sigma \to \tau)^{\mathcal{A}}$, if $\mathtt{App}^{\sigma\to\tau}(f, a) = \mathtt{App}^{\sigma\to\tau}(f', a)$ for all $a \in \sigma^{\mathcal{A}}$, then $f = f'$*
- *for all $p, p' \in (\sigma \times \tau)^{\mathcal{A}}$, if $\mathtt{Proj}_1^{\sigma,\tau}(p) = \mathtt{Proj}_1^{\sigma,\tau}(p')$ and $\mathtt{Proj}_2^{\sigma,\tau}(p) = \mathtt{Proj}_2^{\sigma,\tau}(p')$ then $p = p'$*

Meanings are given in an environment. For contexts $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$ ($n \geq 0$), a Γ-*environment* in $\mathcal{A}$ is a tuple $\langle a_1, \ldots, a_n \rangle$ such that $a_i \in \sigma_i^{\mathcal{A}}$ for $i = 1, .., n$. We will use η to range over environments, and use notation like $\langle \eta, a, \eta' \rangle$ to form the obvious environment. We write $\eta \models^{\mathcal{A}} \Gamma$ to indicate that η is a Γ-environment in interpretation $\mathcal{A}$, but will generally drop the $\mathcal{A}$ when it is obvious which interpretation is intended.

Extensionality allows us to use implicit interpretations for abstractions, pairs and the unit which are unique. However, this does not guarantee that there are actually enough elements in the structure to interpret terms at all. The second condition forces models to have enough elements, and is simply given by saying that the interpretation exists. In Figure 2.6 we define the interpretation of terms in an extensional applicative structure. There we write $[\![\Gamma \vdash t : \tau]\!]^{\mathcal{A}}(\eta)$ for the meaning of the term-in-context $\Gamma \vdash t : \tau$ with environment η in Henkin interpretation $\mathcal{A}$, and again, usually drop the $\mathcal{A}$. Strictly speaking, for an arbitrary applicative structure this only defines a partial meaning function. To give a semantics, we must assume that it is, in fact, total.

Definition 2.2.3 *An applicative structure satisfies the* environment model condition *when the interpretation in Figure 2.6 in well-defined.*

Definition 2.2.4 *We say that an applicative structure is a* $\lambda^{\times\to}$-Henkin interpretation *when it is extensional and satisfies the environment model condition.*

We do not have $(\sigma \times \tau)^{\mathcal{A}} = \sigma^{\mathcal{A}} \times \tau^{\mathcal{A}}$ in general, but we do have a bijection mediated by the projection functions, $\mathtt{Proj}_1^{\sigma,\tau} : (\sigma \times \tau)^{\mathcal{A}} \to \sigma^{\mathcal{A}}$ and $\mathtt{Proj}_2^{\sigma,\tau} : (\sigma \times \tau)^{\mathcal{A}} \to \tau^{\mathcal{A}}$, and the induced pairing map. In general, $(\sigma \to \tau)^{\mathcal{A}}$ is embedded in, but not bijective with, $\sigma^{\mathcal{A}} \to \tau^{\mathcal{A}}$. For $a_1 \in \sigma^{\mathcal{A}}$, $a_2 \in \tau^{\mathcal{A}}$, we write $\langle a_1, a_2 \rangle$ for the unique element $a \in (\sigma \times \tau)^{\mathcal{A}}$ such that $\mathtt{Proj}_1^{\sigma,\tau}(a) = a_1$ and $\mathtt{Proj}_2^{\sigma,\tau}(a) = a_2$.

We write $\Gamma \vDash^{\mathcal{A},\eta} t =_\tau t'$ when $[\![\Gamma \vdash t : \tau]\!]^{\mathcal{A}}(\eta) = [\![\Gamma \vdash t' : \tau]\!]^{\mathcal{A}}(\eta)$, and write $\Gamma \vDash^{\mathcal{A}} t =_\tau t'$ when $\Gamma \vDash^{\mathcal{A},\eta} t =_\tau t'$ for every Γ-environment, η, in $\mathcal{A}$.

Definition 2.2.5 *Let $\mathcal{A}$ be a Henkin interpretation of $\lambda^{\times\to}$-signature Sg. We say that $\mathcal{A}$ is a* Henkin model *of the axiom system $\langle Sg, Ax \rangle$ when for each axiom $\Gamma \vdash t =_\tau t'$ in Ax, $\Gamma \vDash^{\mathcal{A}} t =_\tau t'$.*

Theorem 2.2.6 *(Soundness) Let $\mathcal{A}$ be a Henkin model of axiom system $\langle Sg, Ax \rangle$. If $\langle Sg, Ax \rangle \rhd \Gamma \vdash t =_\tau t'$ then $\Gamma \vDash^{\mathcal{A}} t =_\tau t'$.*

Proof: The proof is by induction on the derivation of the judgement. ∎

As we remarked above, a complete axiomatisation of the lambda calculus depends on whether or not nonemptiness of types is assumed. If types are allowed to be empty then a form of case analysis on emptiness is require, either by having a special rule (as in [MMMS87]) or by using a logic powerful enough to derive this (as in the next section). Alternatively, a wider class of models could be used [MM91]. Here, we will just assume that all types are syntactically inhabited, that is, there are closed terms at every type. In fact, for completeness, a slightly weaker assumption would suffice, namely, that the system is closed under strengthening.

$$[\![\Gamma, x:\sigma, \Gamma' \vdash x:\sigma]\!]\langle \eta, a, \eta' \rangle = a$$

$$\frac{[\![\Gamma \vdash t_1 : \tau_1]\!](\eta) = a_1 \ \cdots \ [\![\Gamma \vdash t_n : \tau_n]\!](\eta) = a_n}{[\![\Gamma \vdash k(t_1, \ldots, t_n) : \tau]\!](\eta) = k^{\mathcal{A}}(a_1, \ldots, a_n)}$$

$$[\![\Gamma \vdash * : \mathbf{1}]\!](\eta) = \text{ the unique } a \text{ in } \mathbf{1}^{\mathcal{A}}$$

$$\frac{[\![\Gamma \vdash t : \sigma]\!](\eta) = a \quad [\![\Gamma \vdash t' : \tau]\!](\eta) = a'}{[\![\Gamma \vdash \langle t, t' \rangle : \sigma \times \tau]\!](\eta) = \begin{array}{c}\text{the unique } p \text{ in } (\sigma \times \tau)^{\mathcal{A}} \text{ such that} \\ \mathtt{Proj}_1^{\sigma,\tau}(p) = a \text{ and } \mathtt{Proj}_2^{\sigma,\tau}(p) = a'\end{array}}$$

$$\frac{[\![\Gamma, x:\sigma \vdash t:\tau]\!]\langle \eta, a \rangle = m_a \text{ for each } a \text{ in } \sigma^{\mathcal{A}}}{[\![\Gamma \vdash \lambda x:\sigma.t : \sigma \to \tau]\!](\eta) = \begin{array}{c}\text{the unique } f \text{ in } (\sigma \to \tau)^{\mathcal{A}} \text{ such that} \\ \forall a \in \sigma^{\mathcal{A}} \,.\, \mathtt{App}(f, a) = m_a\end{array}}$$

$$\frac{[\![\Gamma \vdash t : \tau \times \tau']\!](\eta) = p}{[\![\Gamma \vdash \pi_1(t) : \tau]\!](\eta) = \mathtt{Proj}_1^{\tau,\tau'}(p)}$$

$$\frac{[\![\Gamma \vdash t : \tau \times \tau']\!](\eta) = p}{[\![\Gamma \vdash \pi_2(t) : \tau']\!](\eta) = \mathtt{Proj}_2^{\tau,\tau'}(p)}$$

$$\frac{[\![\Gamma \vdash t : \sigma \to \tau]\!](\eta) = f \quad [\![\Gamma \vdash t' : \sigma]\!](\eta) = a}{[\![\Gamma \vdash tt' : \tau]\!](\eta) = \mathtt{App}(f, a)}$$

Figure 2.6: Interpretation of Terms of Simply-typed Lambda Calculus

Theorem 2.2.7 *(Completeness of equational system) Let $\langle Sg, Ax\rangle$ be a $\lambda^{\times\rightarrow}$-axiom system for which all types are syntactically inhabited. If $\Gamma \models^{\mathcal{A}} t =_\tau t'$ for all Henkin models $\mathcal{A}$ of $\langle Sg, Ax\rangle$, then $\langle Sg, Ax\rangle \triangleright \Gamma \vdash t =_\tau t'$.*

Proof : We give a sketch of the proof. The idea is to construct a minimal term model $\mathcal{T}$ for our signature of ground types, constants and equational assumptions (with no empty types).

1. Fix an infinite context Γ_∞ with an infinite number of variables at each type. We use Γ_∞ in judgements to mean some finite $\Gamma' \subseteq \Gamma_\infty$ so, for example, $\Gamma_\infty \vdash t : \tau$ means $\Gamma' \vdash t : \tau$ for some $\Gamma' \subseteq \Gamma_\infty$.

2. Define $\tau^{\mathcal{T}}$ as the set of $=_\tau$-equivalence classes of open terms of type τ. We write $[t]$ for the equivalence class of t, so $\tau^{\mathcal{T}} = \{[t] \mid \Gamma_\infty \vdash t : \tau\}$. The set $\tau^{\mathcal{T}}$ is nonempty since we have variables at each type. The projection, application and constant interpretation mappings are interpreted syntactically.

3. We use the fact that Γ_∞ contains an infinite number of variables at each type to show that $\mathcal{T}$ is extensional. If $\mathtt{App}([t], a) = \mathtt{App}([t'], a)$ for all $a \in \sigma^{\mathcal{T}}$, then since $x : \sigma \in \Gamma_\infty$, we infer that $\Gamma_\infty \vdash tx =_\tau t'x$, and so by **Congruence Equations** and **Function Equations** (η), we infer that $\Gamma_\infty \vdash t =_{\sigma\rightarrow\tau} t'$, *i.e.* $[t] = [t']$. The condition for products is straightforward.

4. Prove that $[\![\Gamma \vdash t : \tau]\!]^{\mathcal{T}}(\eta) = [t[\eta/\Gamma]]$, where $t[\eta/\Gamma]$ has the obvious meaning. Hence the environment model condition holds and $\mathcal{T}$ is a Henkin interpretation.

5. Now $\Gamma \models^{\mathcal{T}} t =_\tau t'$ means for all $\eta \models^{\mathcal{T}} \Gamma$, $[\![\Gamma \vdash t : \tau]\!]^{\mathcal{T}}(\eta) = [\![\Gamma \vdash t' : \tau]\!]^{\mathcal{T}}(\eta)$, that is $\Gamma_\infty \vdash t[\eta/\Gamma] =_\tau t'[\eta/\Gamma]$ for all η, (more correctly, $\langle Sg, Ax\rangle \triangleright \Gamma \vdash t[\eta/\Gamma] =_\tau t'[\eta/\Gamma]$). Then if $\Gamma \vdash t =_\tau t' \in Ax$ we infer $\Gamma_\infty \vdash t[\eta/\Gamma] =_\tau t'[\eta/\Gamma]$, so $\Gamma \models_{\mathcal{A},\eta} t =_\tau t'$ for all $\eta \models^{\mathcal{T}} \Gamma$ and the interpretation $\mathcal{T}$ is a Henkin model of $\langle Sg, Ax\rangle$.

6. Finally, if $\Gamma \vdash t =_\tau t'$, is an arbitrary equation which is true in the term model, then (for $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$) setting $\eta = \langle [x_1], \ldots, [x_n]\rangle$ gives $\Gamma_\infty \vdash t =_\tau t'$ which, because of inhabitation, implies $\Gamma \vdash t =_\tau t'$.

   Hence, we conclude completeness: if an equation is true in all models, it is true in the term model, and so it is provable.

∎

2.3 First-order Logic of Simply-typed Lambda Calculus

We follow the pattern of Section 2.1 and first define the notion of signature, and then axioms over a signature. We now extend signatures with primitive predicate symbols, sometimes called relation symbols.

Definition 2.3.1 *A* first-order $\lambda^{\times\to}$-signature *consists of a $\lambda^{\times\to}$-signature, and a collection of predicate symbols, F, each of which has an arity n, and a sort, given by a list of types, which we write as $F : \mathtt{Pred}\,(\tau_1, \ldots, \tau_n)$.*

Definition 2.3.2 *Let $Sg = \langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$ be a first-order $\lambda^{\times\to}$-signature. The pre-propositions over Sg are:*

$$P ::= \bot \mid F(t_1, \ldots, t_n) \mid P \supset P' \mid \forall x : \tau . P \mid t =_\tau t'$$

where $F \in \mathcal{F}$, and τ and t are types and preterms over $\langle \mathcal{G}, \mathcal{K} \rangle$ respectively.

The atomic propositions are equalities and predications. The other constructors are sufficient to define $\top$, $\wedge$, $\vee$, $\exists$ and $\neg$.

We need a well-formedness judgement

$$Sg \rhd \Gamma \vdash P \ \mathtt{wf}$$

but omit the rules here. In particular, the proposition $t =_\tau t'$ is well-formed when t and t' have type τ and, for $F : \mathtt{Pred}\,(\tau_1, \ldots, \tau_n)$, the predication $F(t_1, \ldots, t_n)$ is well-formed when $t_i : \tau_i$ for each i. For Δ a list of propositions, we write

$$Sg \rhd \Gamma \vdash \Delta \ \mathtt{wf}$$

when for each P in Δ, $Sg \rhd \Gamma \vdash P \ \mathtt{wf}$.

Definition 2.3.3 *A* first-order $\lambda^{\times\to}$-axiom system consists of *a first-order $\lambda^{\times\to}$-signature, Sg, and a collection, Ax, of closed propositions, well-formed in Sg, that is, $Sg \rhd \langle\rangle \vdash P \ \mathtt{wf}$.*

We take axioms in first-order logic to be closed propositions for simplicity's sake, but this is not important, as quantification gives the same expressiveness as allowing propositions in context. Allowing arbitrary closed propositions as axioms subsumes the equational axioms of $\lambda^{\times\to}$-axiom systems.

We will adopt the convention here, and in subsequent chapters, that when schematic rules are 'included' from one calculus to another, the rules should be understood in the latter calculus: that is, metavariables range over expressions in the latter calculus.

In Figures 2.7 and 2.8 we give the rules for classical natural deduction, modified to allow for the possibility of empty types. Only the rules for $\supset$, $\forall$ and $\bot$ are necessary, but we give the derived rules for some other connectives as well.

The form of the judgement is $\Gamma; \Delta \vdash P$, which means: for all variables in Γ, if each proposition in Δ in true, then P is true.

One property which we want the proof system to have is that all provable judgements are well-formed. It is necessary, therefore, to place well-formedness conditions on those formulae which appear in conclusions but not hypotheses. Otherwise, for example, we could infer that $P \vdash P$ for any pre-proposition P.

Definition 2.3.4 *Let $\langle Sg, Ax\rangle$ be a first-order $\lambda^{\times\rightarrow}$-axiom system. We define the theorems of $\langle Sg, Ax\rangle$ to be the propositions which can be inferred using the rules of Figures 2.1, 2.2 and 2.3, together with Figures 2.7 and 2.8.*

We write $\langle Sg, Ax\rangle \rhd \Gamma; \Delta \vdash P$ to indicate that $\Gamma; \Delta \vdash P$ is a theorem of axiom system $\langle Sg, Ax\rangle$.

Remark 2.3.5 In this thesis, we will make an important distinction between booleans and propositions. Booleans are terms of a computational datatype and can be evaluated, whereas propositions are the expressions of a logical language. It is common, however, to blur the distinction in program development frameworks, so as to reason about branches of conditionals (using booleans as propositions), and to refine a proposition into a conditional (with a proposition as the condition). However, without making the distinction, it is not clear that the refinement of propositions (as specifications) into booleans is itself a part of program development.

Figure 2.9 gives the first-order axioms for booleans and naturals. These are induction rules, expressed as schemas of closed propositions, but they can also be given as inference rules. We can derive the general rules:

$$\frac{Sg \rhd \Gamma \vdash \Delta\ \ \texttt{wf} \qquad Sg \rhd \Gamma, b : \texttt{bool} \vdash P\ \ \texttt{wf}}{\Gamma; \Delta \vdash P[\texttt{true}/b] \wedge P[\texttt{false}/b]\ \supset\ \forall b : \texttt{bool} \,.\, P}$$

$$\frac{Sg \rhd \Gamma \vdash \Delta\ \ \texttt{wf} \qquad Sg \rhd \Gamma, n : \texttt{nat} \vdash P\ \ \texttt{wf}}{\Gamma; \Delta \vdash P[0/n] \wedge (\forall n : \texttt{nat} \,.\, P \supset P[(\texttt{succ}\ n)/n])\ \supset\ \forall n : \texttt{nat} \,.\, P}$$

We retain the equational axioms in Figures 2.4 and 2.5 as part of the first-order axiom systems.

Conjunction

$$\frac{\Gamma;\Delta \vdash P \quad \Gamma;\Delta \vdash Q}{\Gamma;\Delta \vdash P \wedge Q}$$

$$\frac{\Gamma;\Delta \vdash P \wedge Q}{\Gamma;\Delta \vdash P} \qquad \frac{\Gamma;\Delta \vdash P \wedge Q}{\Gamma;\Delta \vdash Q}$$

Disjunction

$$\frac{\Gamma;\Delta \vdash P}{\Gamma;\Delta \vdash P \vee Q} \qquad \frac{\Gamma;\Delta \vdash Q}{\Gamma;\Delta \vdash P \vee Q}$$

$$\frac{\Gamma;\Delta \vdash P \vee Q \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma;\Delta \vdash R}$$

Implication

$$\frac{\Gamma;\Delta, P \vdash Q}{\Gamma;\Delta \vdash P \supset Q}$$

$$\frac{\Gamma;\Delta \vdash P \quad \Gamma;\Delta \vdash P \supset Q}{\Gamma;\Delta \vdash Q}$$

Universal Quantification

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

$$\frac{\Gamma, x:\tau;\ \Delta \vdash P}{\Gamma;\Delta \vdash \forall x:\tau.P}\ (x \notin \Delta)$$

Falsehood

$$\frac{Sg \rhd \Gamma \vdash P\ \texttt{wf} \quad Sg \rhd \Gamma \vdash \Delta\ \texttt{wf}}{\Gamma;\Delta, \bot \vdash P}$$

$$\frac{\Gamma;\Delta, \neg P \vdash \bot}{\Gamma;\Delta \vdash P}$$

Figure 2.7: Natural Deduction Rules for Classical First-order Logic

Equality

$$\frac{\Gamma; \Delta \vdash t =_\tau t' \quad \Gamma; \Delta \vdash P[t/x]}{\Gamma; \Delta \vdash P[t'/x]}$$

Assumptions

$$\frac{Sg \triangleright \Gamma \vdash P \ \texttt{wf} \quad Sg \triangleright \Gamma \vdash \Delta \ \texttt{wf}}{\Gamma; \Delta, P \vdash P}$$

Axioms

$$\frac{Sg \triangleright \Gamma \vdash \Delta \ \texttt{wf}}{\Gamma; \Delta \vdash P} \ (P \in Ax)$$

Figure 2.8: Natural Deduction Rules cont.

Remark 2.3.6 Although we have defined first-order logic over the simply-typed lambda calculus to include all the rules given in Section 2.1, some of these are derivable. The convention that axioms can be used in arbitrary contexts means that the rules of Figure 2.3 are superfluous. Moreover, the general **Equality** rule, together with **Reflexivity**, is enough to derive the other rules in Figure 2.1 and all the **Congruence Equations**.

The induction rule for booleans says that `true` and `false` are the only (closed) booleans. We can prove

$$b : \texttt{bool} \vdash b = \texttt{true} \ \vee \ b = \texttt{false}$$

In fact, this also implies the η-equality for booleans.

Using the rule of mathematical induction in Figure 2.9, we can deduce computational induction

$$\frac{P[z] \quad \forall n : \texttt{nat}.\forall x : \tau \,.\, P[x] \supset P[s\ n\ x]}{\forall n : \texttt{nat} \,.\, P[\texttt{natrec}\ z\ s\ n]} \begin{pmatrix} Sg \triangleright z : \tau, s : \texttt{nat} \to \tau \to \tau \\ Sg \triangleright x : \tau \vdash P[x] \ \texttt{wf} \end{pmatrix}$$

and well-founded induction

$$\frac{\forall n : \texttt{nat} \,.\, (\forall n' < n.P[n']) \supset P[n]}{\forall n : \texttt{nat} \,.\, P[n]} \ (Sg \triangleright n : \texttt{nat} \vdash P[n] \ \texttt{wf})$$

Booleans

$$P[\mathtt{true}/b] \wedge P[\mathtt{false}/b] \;\supset\; \forall b : \mathtt{bool}\,.\,P \quad (\text{for } Sg \triangleright b : \mathtt{bool} \vdash P \;\; \mathtt{wf})$$

Naturals

$$P[0/n] \wedge (\forall n : \mathtt{nat}\,.\,P \supset P[(\mathtt{succ}\; n)/n]) \;\supset\; \forall n : \mathtt{nat}\,.\,P$$
$$(\text{for } Sg \triangleright n : \mathtt{nat} \vdash P \;\; \mathtt{wf})$$

Figure 2.9: First-order Logic of Booleans and Naturals

2.4 Models of First-order Logic

We give Henkin models of first-order logic. First we define interpretations for a particular signature.

Definition 2.4.1 *Let $\langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$ be a first-order $\lambda^{\times\rightarrow}$-signature. A* first-order $\lambda^{\times\rightarrow}$-Henkin interpretation*, $\mathcal{A}$, of $\langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$ is a $\lambda^{\times\rightarrow}$-Henkin interpretation of $\langle \mathcal{G}, \mathcal{K} \rangle$ together with a family of subsets to interpret predicate symbols in $\mathcal{F}$*

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

Figure 2.10 gives the interpretation of propositions in a first-order $\lambda^{\times\rightarrow}$-Henkin interpretation. There we interpret well-formed propositions in context, $\Gamma \vdash P \;\mathtt{wf}$, as the set, $[\![\Gamma \vdash P]\!]^{\mathcal{A}}$ of environments, $\eta \models^{\mathcal{A}} \Gamma$, in which P holds, though we usually drop the superscript $\mathcal{A}$. We write $\Gamma \models^{\mathcal{A},\eta} P$ to mean $\eta \in [\![\Gamma \vdash P]\!]^{\mathcal{A}}$. If $\mathcal{A}$ is a Henkin interpretation, we say that $\Gamma; \Delta \models^{\mathcal{A},\eta} P$, if for all $\eta \models^{\mathcal{A}} \Gamma$, if $\Gamma \models^{\mathcal{A},\eta} A$ for each A in Δ, then $\Gamma \models^{\mathcal{A},\eta} P$. We write $\Gamma; \Delta \models^{\mathcal{A}} P$ when $\Gamma; \Delta \models^{\mathcal{A},\eta} P$ for each $\eta \models^{\mathcal{A}} \Gamma$.

Definition 2.4.2 *Let $\langle Sg, Ax \rangle$ be a first-order $\lambda^{\times\rightarrow}$-axiom system. A Henkin interpretation $\mathcal{A}$ of signature Sg is a* model *of $\langle Sg, Ax \rangle$ when for each axiom P in Ax, $\models^{\mathcal{A}} P$.*

One reason for studying the logic is that it is complete over the same class of models as the equational theory. Completeness is with respect to the class of Henkin models (of an axiom system).

In order to prove completeness for an arbitrary axiom system we will construct a term model from its theory. The main problem lies in constructing witnesses for

$$[\![\Gamma \vdash \bot]\!] = \emptyset$$
$$[\![\Gamma \vdash F(t_1,\ldots,t_n)]\!] = \{\eta \vDash \Gamma \mid \langle [\![\Gamma \vdash t_1 : \tau_1]\!](\eta),\ldots,[\![\Gamma \vdash t_n : \tau_n]\!](\eta)\rangle \in F^{\mathcal{A}}\}$$
$$[\![\Gamma \vdash P \supset P']\!] = \{\eta \vDash \Gamma \mid \eta \notin [\![\Gamma \vdash P]\!] \text{ or } \eta \in [\![\Gamma \vdash P']\!]\ \}$$
$$[\![\Gamma \vdash \forall x : \tau.P]\!] = \{\eta \vDash \Gamma \mid \text{ for all } a \text{ in } \tau^{\mathcal{A}} \,.\, \langle \eta, a\rangle \in [\![\Gamma, x:\tau \vdash P]\!]\ \}$$
$$[\![\Gamma \vdash t =_\tau t']\!] = \{\eta \vDash \Gamma \mid [\![\Gamma \vdash t : \tau]\!](\eta) = [\![\Gamma \vdash t' : \tau]\!](\eta)\}$$

Figure 2.10: Interpretation of Well-formed Propositions

existentials. We will achieve this by using the notion of *Henkin theory*[1] [vD94]. A Henkin theory, T, has the property that if the proposition $\exists x : \tau.P$ is in T, then $P[t/x]$ is in T for some $t : \tau$.

Definition 2.4.3 *A* first-order $\lambda^{\times\to}$-Henkin theory*, T, over an axiom system $\langle Sg, Ax\rangle$, in context Γ, is a collection of propositions well-formed in Γ, and closed under derivation from $\langle Sg, Ax\rangle$, such that for every proposition $\exists x : \tau\,.\,P$ in T, there is a term $\Gamma \vdash t : \tau$ such that $P[t/x]$ is in T.*

We can construct Henkin theories by adding witness variables for existentials, taking care with empty types. As pointed out after Definition 2.1.5 we assume a countably infinite set of variables at each type.

Definition 2.4.4 *Let Γ be a context and Δ a set of propositions. We define the* Henkin closure *of $\Gamma;\Delta$ by the following procedure:*

1. *First enumerate all the types. Then for each type, τ_i ($i \geq 1$), we decide if it is to be inhabited or not. For $i \geq 1$, we define the proposition $\mathtt{Inh}_i$ as*

   $$\begin{cases} \top, \text{ if } \Gamma;\Delta,(\mathtt{Inh}_1 \supset \exists x:\tau_1.\top),\ldots,(\mathtt{Inh}_{i-1} \supset \exists x:\tau_{i-1}.\top), \exists x:\tau_i.\top \nvdash \bot \\ \bot, \text{ otherwise} \end{cases}$$

2. *Then we make a list of all well-formed propositions of the form $\exists x : \sigma_n.P_n$ for inhabited σ_n, i.e. those for which $\mathtt{Inh}_n = \top$. This is countable since the signature is countable.*

3. *Then we make a list of variables $\{y_n : \sigma_n\}$ such that $y_n \notin \Gamma$ and $y_n \notin P_{n'}$ for $n' \leq n$.*

4. *We define the Henkin closure of $\Gamma;\Delta$ as $\Gamma_H;\Delta_H$, where $\Gamma_H = \Gamma \cup \{y_n : \sigma_n\}$, and $\Delta_H = \Delta \cup \{\exists x : \sigma_n\,.\,P_n \supset P_n[y_n/x]\}$.*

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

Although an infinite supply of variables in the context is not necessary to meet the definition of Henkin theory, it is used in the proof below. The point of the completeness proof is to construct the maximal such Henkin theory.

Theorem 2.4.5 *(Soundness and Completeness of logical system) Let $\langle Sg, Ax\rangle$ be a first-order $\lambda^{\times\rightarrow}$-axiom system. Then $\langle Sg, Ax\rangle \triangleright \Gamma; \Delta \vdash P$ iff $\Gamma; \Delta \models^{\mathcal{A}} P$ for every Henkin model $\mathcal{A}$ of $\langle Sg, Ax\rangle$*

Proof: Soundness is straightforward to prove. As for completeness, we do not have a minimal (Henkin) model for the logical system, but nevertheless, we can still use a term model in order to prove deductive completeness, by showing that any consistent theory is satisfiable. For axiom system $\langle Sg, Ax\rangle$, we want to show that $\langle Sg, Ax\rangle \triangleright \Gamma; \Delta \vdash P$ iff $\Gamma; \Delta \models^{\mathcal{A}} P$ in all Henkin models $\mathcal{A}$ of $\langle Sg, Ax\rangle$. (Note that we do not assume that types are nonempty.)

1. Given $\langle Sg, Ax\rangle \not\triangleright \Gamma; \Delta \vdash P$ we want to find a Henkin model $\mathcal{A}$ of $\langle Sg, Ax\rangle$, and Γ-environment, η, in $\mathcal{A}$ such that $\Gamma \models^{\mathcal{A},\eta} A$ for each A in $\Delta, \neg P$.

2. We construct a maximal consistent theory Δ_∞ and infinite context Γ_∞ such that $Ax \cup \Delta \cup \{\neg P\} \subseteq \Delta_\infty$, $\Gamma \subseteq \Gamma_\infty$, and Δ_∞ is a Henkin theory in Γ_∞.

   First let $\Gamma_H; \Delta_H$ be the Henkin closure of $\Gamma; Ax \cup \Delta \cup \{\neg P\}$. Now we consider consistent theories in Γ_H which extend Δ_H.

   We form the partial order of such theories, ordered by pairwise inclusion. The poset is nonempty since it contains the deductive closure of Δ_H. Each member is a Henkin theory. The poset is closed under taking unions, and so each chain has an upper bound. Thus, by Zorn's Lemma, the collection has a maximal element, Δ_∞.

3. We define the term interpretation on equivalence classes of open terms provably equal in $\Gamma_H; \Delta_\infty$.

   By renaming variables, we can assume without loss of generality that all terms are open in Γ_H.

   We write $[t]$ for the equivalence class of t, so that $[t] = [t']$ iff $\Gamma_H; \Delta_\infty \vdash t =_\tau t'$ (*i.e.* $t =_\tau t' \in \Delta_\infty$).

   First, we show that $\mathcal{A}$ is extensional. The interesting case is for function types. We must show that if $\Gamma_H; \Delta_\infty \vdash ta =_\tau t'a$ for every $\Gamma_H \vdash a : \sigma$, then $\Gamma_H; \Delta_\infty \vdash t =_{\sigma\rightarrow\tau} t'$. We reason on whether or not $\exists x : \sigma.\top$ is in Δ_∞, that is, σ is inhabited.

If it is then, since Γ_H contains infinitely many variables for each inhabited type, there exists a variable $x : \sigma \in \Gamma_H$ which does not appear in t or t'. Hence, $\Gamma_H; \Delta_\infty \vdash tx =_\tau t'x$, so $\Gamma_H; \Delta_\infty \vdash \lambda x : \sigma.tx =_{\sigma\to\tau} \lambda x : \sigma.t'x$ and $\Gamma_H; \Delta_\infty \vdash t =_{\sigma\to\tau} t'$.

If it is not, then by maximality, $\neg\exists x : \tau.\top \in \Delta_\infty$, and since Δ_∞ is consistent, there are no terms $\Gamma_H \vdash a : \sigma$ and so the implication trivially holds.

4. Prove that $[\![\Gamma \vdash t : \tau]\!]^{\mathcal{A}}(\eta) = [t[\eta/\Gamma]]$, and so $\mathcal{A}$ satisfies the environment model condition and is a Henkin interpretation.

5. For all $\eta' \models \Gamma'$, prove that $\Gamma' \models^{\mathcal{A},\eta'} Q$ iff $Q[\eta'/\Gamma'] \in \Delta_\infty$, by induction over Q. It is here that the proof rules for each construct are used. We use the fact that for a maximal consistent theory T and well-formed (in Γ_H) proposition Q, exactly one of Q, $\neg Q$ is in T. The crucial case is $\exists x : \tau \,.\, Q$, which goes through by virtue of Δ_∞ being a Henkin theory.

   From this it follows that $\mathcal{A}$ is a Henkin model of $\langle Sg, Ax\rangle$, since for each axiom Q, clearly $Q \in \Delta_\infty$.

6. Finally, using the extension property of $\Gamma_H; \Delta_\infty$, if $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$, then we define the Γ-environment, η, to be simply $\langle [x_1], \ldots, [x_n]\rangle$, and then $\Gamma \models^{\mathcal{A},\eta} A$, for each A in $\Delta, \neg P$.

∎

As a corollary of these completeness results, we can deduce that the first-order calculus is conservative over the equational calculus.

Corollary 2.4.6 *Let $\langle Sg, Ax\rangle$ be a $\lambda^{\times\to}$-axiom system. If all types are inhabited, then $\langle Sg, Ax\rangle \triangleright \Gamma \vdash t =_\tau t'$ iff $\langle Sg, Ax\rangle \triangleright \Gamma; \langle\rangle \vdash t =_\tau t'$.*

Proof: Both systems are complete with respect to Henkin models, and the statements have the same interpretation. ∎

Chapter 3

Refinement Terms

In this chapter we develop a theory of 'simple' refinement. We see that, in addition to aspects of refinement, the use of stubs and skeletons in top-down program development can also be studied in this language. We give a calculus, $\lambda_?^{\times\rightarrow}$ ($\lambda_?$ for short), in which we can express such constructs and a simple semantics using Henkin models for which the calculus is proven sound and complete.[1]

3.1 Introduction

We introduce an extension of the simply-typed lambda calculus ($\lambda^{\times\rightarrow}$) with constructs for expressing a notion we call *underdeterminism*. Consider the term $\lambda x:\sigma.\langle 2, x\rangle$ of type $\sigma \rightarrow \mathtt{nat} \times \sigma$. This term is *determined* in the sense that we have complete knowledge about it. Suppose now, we know that some term of the same type always returns pairs of which the left component is 2, but we know nothing about the right component. We might write this as $\lambda x:\sigma.\langle 2, ?_\sigma\rangle$, where $?_\sigma$ means 'some unknown of type σ' (possibly depending on x). We allow $?_\sigma$ to stand for any subterm which is well-formed in the local context, so these unknowns can contain the variable x. This is in contrast to the use of variables as indeterminates. If, instead of the stub $?_\sigma$, we were to use a free variable y of type σ as a parameter, writing the above term as $\lambda x:\sigma.\langle 2, y\rangle$, then we can substitute any term for y which is well-formed in the global context. The point is, though, that because of variable capture, we cannot substitute x to get $\lambda x:\sigma.\langle 2, x\rangle$. This, of course, is crucial to the use of stubs.

Now, a still less determined term would be $\lambda x:\sigma.\langle ?_{\mathtt{nat}}, ?_\sigma\rangle$. This is a term of type $\sigma \rightarrow (\mathtt{nat} \times \sigma)$ which returns ... what? Evaluation does not make sense in general for such partially constructed terms anyway, but our intuition tells us that the term $\lambda x:\sigma.\langle ?_{\mathtt{nat}}, ?_\sigma\rangle$ carries the same information as $\lambda x:\sigma.?_{\mathtt{nat}\times\sigma}$

[1] Some of the work of this chapter was presented in [Den97b].

and $?_{\sigma\to(\mathtt{nat}\times\sigma)}$. We would like to prove equivalences such as this, and in general consider a specialisation ordering at each type, such that

$$?_{\sigma\to(\mathtt{nat}\times\sigma)} \sqsubseteq_{\sigma\to(\mathtt{nat}\times\sigma)} \lambda x:\sigma.\langle 2, ?_\sigma\rangle \sqsubseteq_{\sigma\to(\mathtt{nat}\times\sigma)} \lambda x:\sigma.\langle 2, x\rangle$$

Moreover, we would like to study how underdeterminism interacts with the usual equational rules of $\lambda^{\times\to}$. Our interest in such a calculus comes from our belief that this is a fundamental aspect of program development. The refinement methodology of program development consists of writing a term meaning 'a program which satisfies specification ϕ', and transforming it step by step into an actual program satisfying the specification ϕ. We view types as rudimentary specifications, and defer study of the logic to the next chapter. We believe it is worthwhile to study underdeterminism in isolation from logic, as much of the difficulty in reasoning in refinement calculi is in understanding how underdeterminism interacts with the programming features.

In Section 3.2 we describe the language and its refinement rules. The equational theory is then studied as part of a simple logic. We give a simple denotational semantics in Section 3.4, and show that our calculus is complete for proving refinements valid in this class of models.

3.2 The Calculus

We give the syntax of the language and the classes of judgements. Then we give some syntactic results and a short example of refinement.

3.2.1 Syntax

We start by defining the notion of $\lambda_?$-signature. In fact, signatures and axioms are the same as we defined in the previous chapter for $\lambda^{\times\to}$. The significance of this is discussed later, in Remark 3.3.4.

Definition 3.2.1 *A* $\lambda_?$-signature *Sg consists of a collection, $\mathcal{G}$, of ground types (ranged over by γ) and a collection, $\mathcal{K}$, of constant symbols (ranged over by k), each of which is assigned some sort $\tau_1,\ldots,\tau_n \to \tau$.*

We extend the simply-typed lambda calculus with an underdeterminism construct, $?_\tau$, meaning 'some term with type τ'. We view the type τ as a rudimentary specification and refer to terms with such 'holes' for programs to be supplied later as *refinement terms*. We assume throughout some fixed signature of types and constants.

The types, preterms and contexts are given by:

$$\begin{array}{rcl}
\tau & ::= & \mathbf{1} \mid \gamma \mid \tau \times \tau \mid \tau \to \tau \\
r & ::= & x \mid k(r_1, \ldots, r_n) \mid * \mid \langle r, r' \rangle \mid \lambda x : \tau . r \mid ?_\tau \mid \pi_1(r) \mid \pi_2(r) \mid \\
 & & r r' \mid \mathtt{let}\ x : \tau\ \mathtt{be}\ r\ \mathtt{in}\ r' \\
\Gamma & ::= & \langle\rangle \mid \Gamma, x : \tau
\end{array}$$

We say that a term is *determined* if it contains no stubs, that is, subterms of the form $?_\tau$. Otherwise, we call a term *underdetermined*. We use the metavariable t to range over determined terms, and r over arbitrary underdetermined terms. Note that primitive constants are determined. In fact, each term of the simply-typed lambda calculus is determined. The converse does not hold, however, since we allow $\mathtt{let}\ x : \sigma\ \mathtt{be}\ t\ \mathtt{in}\ t'$ to be determined. We will see, though, that every determined term is provably equal to a term of the simply-typed lambda calculus.

Intuitively, we can think of a term r as being a kind of description or specification of determined terms, so for example, $\langle 2, ?_{\mathtt{nat}} \rangle$ is a term which specifies pairs of terms $\langle 2, t \rangle$, where t is any term of type $\mathtt{nat}$. Conversely, we say that $\langle 2, t \rangle$ satisfies $\langle 2, ?_{\mathtt{nat}} \rangle$.

Formally, r denotes a set of values. Each program, t, to which r can refine, denotes a member of this set. The abstraction $\lambda x : \tau . r$ denotes a set of functions, rather than one nondeterministic function. Intuitively, it refines to those abstractions, $\lambda x : \tau . t$, such that for each argument $t' : \tau$, the result $t[t'/x]$ is a refinement of $r[t'/x]$. It is *not* a nondeterministic program which takes an argument t' and returns a term $r[t'/x]$.

Since $\mathtt{let}\ x : \sigma\ \mathtt{be}\ t\ \mathtt{in}\ t'$ is provably equal to $t'[t/x]$, we can eliminate all determined let-subterms, and show that every determined term is provably equal to an ordinary term of the simply-typed lambda calculus.

Although terms of the language denote sets of values, we want to regard variables as ranging over single values in order to retain the familiar rules of $\lambda^{\times\to}$. Because of this we will only allow determined terms to be substituted for variables. We use an axiomatisation of $\mathtt{let}$ expressions, $\mathtt{let}\ x : \tau\ \mathtt{be}\ r\ \mathtt{in}\ r'$, as in the computational lambda calculus [Mog91], as a way of discharging an arbitrary underdetermined term r, without substituting directly for a variable x. The idea is that $\mathtt{let}\ x : \tau\ \mathtt{be}\ r\ \mathtt{in}\ r'$ defers the substitution of r for x until r has been refined into some determined t, but still lets us reason about the substitution. The expression $\mathtt{let}\ x : \tau\ \mathtt{be}\ r\ \mathtt{in}\ r'$ refines to t', then, when t' is a refinement of $r'[t/x]$ for some refinement, t, of r.

Although we do not assume that all types are inhabited (by closed terms), for various statements below we will make this restriction. This avoids the semantic complications mentioned in Chapter 2. Nevertheless, although empty types may or may not be appropriate for any particular programming language, this assumption is independent of the use of the calculus for specification. The reader might assume, though, that this assumption means that our calculus is of no interest in studying program specification, where in traditional type-theoretic approaches, specifications are viewed as possibly empty types. However, the idea in the next chapter is not to use types themselves as specifications, but that a specification (which may be unsatisfiable) is something 'over' an ordinary program type. *e.g.* the specification $(n : \texttt{nat})\ \texttt{even}(n)$, of the set of even natural numbers (in the notation of the subsequent chapters) is over `nat`.

3.2.2 Judgements

We axiomatise an equational theory with two basic judgements

Typing	$\Gamma \vdash r : \tau$
Refinement	$\Gamma \vdash r \sqsubseteq_\tau r'$

where Γ is a context of variable assumptions. As is usual with lambda calculi, (in)equations are at a type, which we sometimes drop when not significant. We write the refinement of r to r' (at τ) as $r \sqsubseteq_\tau r'$, to indicate that r' is more determined than r. Note that some authors use $\sqsupseteq$ for refinement.

We take equality to be the derived notion defined as mutual refinement. This is reasonable because, as we show by a semantic argument below, the calculus is a conservative extension of the simply-typed lambda calculus, as determined terms are mutually refinable if and only if they are provably equal in $\lambda^{\times\rightarrow}$.

Contexts are well-formed

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

if and only if they contain distinct variables. We adopt the convention that in writing a judgement we assume its context to be well-formed.

The typing judgement $\Gamma \vdash r : \tau$ is axiomatised in Figure 3.1. This just extends the rules of the simply-typed lambda calculus with typing rules for the ? and `let` constructs. We write $Sg \triangleright \Gamma \vdash r : \tau$ to indicate that the typing judgement $\Gamma \vdash r : \tau$ is derivable from signature Sg.

3.2.3 $\lambda_?$-Axiom Systems

We can define the notion of axiom system with respect to a signature as a set of well-typed equations in context between determined terms.

Variables

$$\Gamma, x : \sigma, \Gamma' \vdash x : \sigma$$

Constants

$$\frac{\Gamma \vdash r_1 : \tau_1 \quad \cdots \quad \Gamma \vdash r_n : \tau_n}{\Gamma \vdash k(r_1, \ldots, r_n) : \tau} \; (k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K})$$

Unit

$$\Gamma \vdash * : \mathbf{1}$$

Stubs

$$\Gamma \vdash ?_\sigma : \sigma$$

Product Terms

$$\frac{\Gamma \vdash r : \sigma \quad \Gamma \vdash r' : \tau}{\Gamma \vdash \langle r, r' \rangle : \sigma \times \tau}$$

$$\frac{\Gamma \vdash r : \sigma \times \tau}{\Gamma \vdash \pi_1(r) : \sigma} \qquad \frac{\Gamma \vdash r : \sigma \times \tau}{\Gamma \vdash \pi_2(r) : \tau}$$

Function Terms

$$\frac{\Gamma, x : \sigma \vdash r : \tau}{\Gamma \vdash \lambda x : \sigma . r : \sigma \to \tau}$$

$$\frac{\Gamma \vdash r : \sigma \to \tau \quad \Gamma \vdash r' : \sigma}{\Gamma \vdash r r' : \tau}$$

Let Terms

$$\frac{\Gamma \vdash r : \tau \quad \Gamma, x : \tau \vdash r' : \tau'}{\Gamma \vdash \texttt{let}\ x : \tau\ \texttt{be}\ r\ \texttt{in}\ r' : \tau'}$$

Figure 3.1: Typing Rules

Definition 3.2.2 *A $\lambda_?$-*axiom system*, $\langle Sg, Ax\rangle$, consists of a $\lambda_?$-signature, Sg, and a collection, Ax, of equations in context, $\Gamma \vdash t =_\tau t'$, well-typed with respect to the signature, that is, $Sg \rhd \Gamma \vdash t : \tau$ and $Sg \rhd \Gamma \vdash t' : \tau$.*

We will discuss below (Remark 3.2.4) why we do not allow axioms to be arbitrary refinements. We assume some fixed axiom system $\langle Sg, Ax\rangle$ throughout.

Definition 3.2.3 *Let $\langle Sg, Ax\rangle$ be a $\lambda_?$-axiom system. We define the theorems of $\langle Sg, Ax\rangle$ to be the refinements which can be inferred using the rules of Figures 3.2, 3.3, 3.4, 3.5 and 3.6. We write $\langle Sg, Ax\rangle \rhd \Gamma \vdash r \sqsubseteq_\tau r'$ to indicate that refinement $\Gamma \vdash r \sqsubseteq_\tau r'$ is a theorem of axiom system $\langle Sg, Ax\rangle$.*

We write the equality, $r =_\tau r'$, to mean the mutual refinement, $r \sqsubseteq_\tau r'$ and $r' \sqsubseteq_\tau r$, and extend the rule convention mentioned in the previous chapter, so that when we include rules from $\lambda^{\times\rightarrow}$ in $\lambda_?$, equality rules are to be taken as mutual refinements.

Figure 3.2 gives the rules for inferring theorems from a $\lambda_?$-axiom system. These are the natural extension of the rules in Chapter 2, with the condition that substitution is restricted to determined terms.

The equality rules of Figures 3.3 and 3.4 are on top of those of the simply-typed lambda calculus in Figure 2.2, Chapter 2, which should now be read as mutual refinements.

The rules are given for determined terms. Although we show below various generalisations of these to arbitrary underdetermined terms, we prefer to give the axioms of the calculus in this minimal form as it more clearly shows that refinement is an axiomatisation on top of the underlying equational theory.

Figures 3.3 and 3.4 axiomatise how underdeterminism combines with program constructs via the `let` expressions. Most of these rules are taken from the computational lambda calculus. The exception is the rule for **Abstractions**, which makes explicit the 'hidden dependency' of specifications on variables in the context. A specification, $?_\tau$, under an abstraction, $\lambda x : \sigma$, can be refined to terms which contain the x. We remarked on p. 24 that this is an important difference between $?$ and ϵ. Now, this is equivalent to specifying some term $?_{\sigma\to\tau}$ outside the λ, which is then applied to x under the λ. This rule is the only addition to Moggi's computational lambda calculus, and has significant consequences (see Lemma 3.2.10). Logically, we can think of the rule as a form of skolemisation, where abstractions correspond to universal quantifications and `let`'s to existential quantifications.

Axioms

$$\frac{}{\Gamma \vdash t =_\tau t'} \; (\Gamma \vdash t =_\tau t' \in Ax)$$

Weakening

$$\frac{\Gamma_1, \Gamma_2 \vdash r \sqsubseteq_\tau r'}{\Gamma_1, x : \sigma, \Gamma_2 \vdash r \sqsubseteq_\tau r'}$$

Permutation

$$\frac{\Gamma_1, x_1 : \sigma_1, \Gamma_2, x_2 : \sigma_2, \Gamma_3 \vdash r \sqsubseteq_\tau r'}{\Gamma_1, x_2 : \sigma_2, \Gamma_2, x_1 : \sigma_1, \Gamma_3 \vdash r \sqsubseteq_\tau r'}$$

Substitution

$$\frac{\Gamma, x : \sigma \vdash r \sqsubseteq_\tau r' \quad \Gamma \vdash t : \sigma}{\Gamma \vdash r[t/x] \sqsubseteq_\tau r'[t/x]}$$

Figure 3.2: Theorems Generated from a $\lambda_?$-Axiom System $\langle Sg, Ax \rangle$

In view of **Let Associativity**, we use $\texttt{let}\ x_1 : \sigma_1, x_2 : \sigma_2\ \texttt{be}\ r_1, r_2\ \texttt{in}\ r$ as an abbreviation of the nested let-term $\texttt{let}\ x_1 : \sigma_1\ \texttt{be}\ r_1\ \texttt{in}\ (\texttt{let}\ x_2 : \sigma_2\ \texttt{be}\ r_2\ \texttt{in}\ r)$. Note that because of the assumption of well-formedness of contexts, we can omit side-conditions on the occurrence of variables. For example, in **Let Associativity**, since $\Gamma, y : \tau$ is well-formed, y is not in Γ and so y is not in r''.

Figures 3.5 and 3.6 axiomatise the refinement relation. The intuition behind the refinement relation is that it should correspond to an increase in information, and a decrease in the possible programs to which a term can refine. There are top-down rules for decomposing a specification by refinement into a combination of simpler ones. We also have a weakening rule **Let Weakening** which may be thought of as claiming an auxiliary lemma, and a congruence rule for let-expressions, which lets us derive the corresponding rules for pairs, applications and projections.

A number of similar refinement rules for the destructors, and sequent style bottom-up rules for making use of the context are derived below (Propositions 3.2.19 and 3.2.20, respectively).

The equality rules in Figures 2.1 and 2.2, Chapter 2 together with:

Let Beta

$$\frac{\Gamma \vdash t : \tau \quad \Gamma, x : \tau \vdash r : \tau'}{\Gamma \vdash \texttt{let}\ x : \tau\ \texttt{be}\ t\ \texttt{in}\ r[x] =_{\tau'} r[t]}$$

Let Eta

$$\frac{\Gamma \vdash r : \tau}{\Gamma \vdash \texttt{let}\ x : \tau\ \texttt{be}\ r\ \texttt{in}\ x =_{\tau} r}$$

Let Associativity

$$\frac{\Gamma \vdash r : \tau \quad \Gamma, y : \tau \vdash r' : \tau' \quad \Gamma, x : \tau' \vdash r'' : \tau''}{\begin{array}{l}\Gamma \vdash \texttt{let}\ x : \tau'\ \texttt{be}\ (\texttt{let}\ y : \tau\ \texttt{be}\ r\ \texttt{in}\ r'[y])\ \texttt{in}\ r''[x] \\ \qquad =_{\tau''} \texttt{let}\ y : \tau\ \texttt{be}\ r\ \texttt{in}\ (\texttt{let}\ x : \tau'\ \texttt{be}\ r'[y]\ \texttt{in}\ r''[x])\end{array}}$$

Figure 3.3: Equality Rules

Example 3.2.4 Since the definitions of signatures and axiom systems are the same for $\lambda_?$ and $\lambda^{\times\rightarrow}$, we just use the axiomatisation of booleans and naturals in Chapter 2, given in Figures 2.4 and 2.5. It is an important point that axioms are given as equations between determined terms, and not as refinements.

The practical implication of this is that if we enrich the language with some new operations, then no extra work is required to give refinement rules, except for equalities (which are a trivial form of refinement). Since all valid refinements are derivable from the rules of the calculus it is unnecessary to have to come up with new rules. The theoretical justification for this will follow from the completeness theorems. For any axiom system, the rules we give are sufficient to prove all true refinements.

We indicate now how this is done in the case of the booleans. We need only use the general rule for combining let-terms and constants. For constant $k : \tau_1, \ldots, \tau_n \rightarrow \tau$, we have

$$\texttt{let}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \texttt{be}\ r_1, \ldots, r_n\ \texttt{in}\ k(x_1, \ldots, x_n) = k(r_1, \ldots, r_n)$$

For booleans this gives for example

$$\begin{array}{l}\texttt{let}\ x : \texttt{bool}, y : \sigma, z : \sigma\ \texttt{be}\ r_1, r_2, r_3\ \texttt{in}\ (\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z) \\ \hfill =_{\sigma} \texttt{if}\ r_1\ \texttt{then}\ r_2\ \texttt{else}\ r_3\end{array}$$

Constants

$$\frac{\Gamma \vdash r_1 : \tau_1 \quad \cdots \quad \Gamma \vdash r_n : \tau_n}{\begin{array}{l}\Gamma \vdash \texttt{let}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \texttt{be}\ r_1, \ldots, r_n \\ \qquad\quad \texttt{in}\ k(x_1, \ldots, x_n) =_\tau k(r_1, \ldots, r_n)\end{array}}\ (k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K})$$

Applications

$$\frac{\Gamma \vdash r : \sigma \to \tau \quad \Gamma \vdash r' : \sigma}{\Gamma \vdash \texttt{let}\ x : \sigma \to \tau, x' : \sigma\ \texttt{be}\ r, r'\ \texttt{in}\ xx' =_\tau rr'}$$

Pairs

$$\frac{\Gamma \vdash r : \tau \quad \Gamma \vdash r' : \tau'}{\Gamma \vdash \texttt{let}\ x : \tau, x' : \tau'\ \texttt{be}\ r, r'\ \texttt{in}\ \langle x, x' \rangle =_{\tau \times \tau'} \langle r, r' \rangle}$$

Projections

$$\frac{\Gamma \vdash r : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{let}\ x : \tau_1 \times \tau_2\ \texttt{be}\ r\ \texttt{in}\ \pi_i(x) =_{\tau_i} \pi_i(r)}\ (i = 1, 2)$$

Abstractions

$$\frac{\Gamma, x : \sigma, y : \tau \vdash r[x, y] : \tau'}{\begin{array}{l}\Gamma \vdash \texttt{let}\ z : \sigma \to \tau\ \texttt{be}\ ?_{\sigma \to \tau}\ \texttt{in}\ \lambda x : \sigma . r[x, zx] \\ \qquad\qquad =_{\sigma \to \tau'} \lambda x : \sigma . (\texttt{let}\ y : \tau\ \texttt{be}\ ?_\tau\ \texttt{in}\ r[x, y])\end{array}}$$

Figure 3.4: Equality rules cont.

Variables

$$\Gamma, x : \sigma, \Gamma' \vdash ?_\sigma \sqsubseteq_\sigma x$$

Constants

$$\Gamma \vdash ?_\tau \sqsubseteq_\tau k(?_{\tau_1}, \ldots, ?_{\tau_n}) \qquad (k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K})$$

Unit

$$\Gamma \vdash ?_{\mathbf{1}} \sqsubseteq_{\mathbf{1}} *$$

Pairs

$$\Gamma \vdash ?_{\sigma \times \tau} \sqsubseteq_{\sigma \times \tau} \langle ?_\sigma, ?_\tau \rangle$$

Abstractions

$$\Gamma \vdash ?_{\sigma \to \tau} \sqsubseteq_{\sigma \to \tau} \lambda x : \sigma . ?_\tau$$

Figure 3.5: Refinement Rules

Congruence

$$\frac{\Gamma \vdash r_1 \sqsubseteq_{\tau_1} r_1' \quad \Gamma, x : \tau_1 \vdash r_2 \sqsubseteq_{\tau_2} r_2'}{\Gamma \vdash \texttt{let } x : \tau_1 \texttt{ be } r_1 \texttt{ in } r_2 \sqsubseteq_{\tau_2} \texttt{let } x : \tau_1 \texttt{ be } r_1' \texttt{ in } r_2'} \quad \text{(let)}$$

$$\frac{\Gamma, x : \sigma \vdash r \sqsubseteq_\tau r'}{\Gamma \vdash \lambda x : \sigma.r \sqsubseteq_{\sigma \to \tau} \lambda x : \sigma.r'} \quad \text{(abs)}$$

Reflexivity

$$\frac{\Gamma \vdash r : \tau}{\Gamma \vdash r \sqsubseteq_\tau r}$$

Transitivity

$$\frac{\Gamma \vdash r \sqsubseteq_\tau r' \quad \Gamma \vdash r' \sqsubseteq_\tau r''}{\Gamma \vdash r \sqsubseteq_\tau r''}$$

Let Weakening

$$\frac{\Gamma \vdash r' : \tau \quad \Gamma \vdash r : \sigma}{\Gamma \vdash r' \sqsubseteq_\tau \texttt{let } x : \sigma \texttt{ be } r \texttt{ in } r'} \ (x \notin FV(r'))$$

Figure 3.6: Refinement Rules cont.

There is one refinement rule for each constant. If $Sg \rhd k : \tau_1, \ldots, \tau_n \to \tau$ then

$$\Gamma \vdash ?_\tau \sqsubseteq_\tau k(?_{\tau_1}, \ldots, ?_{\tau_n})$$

In combination with the congruence rule for let-terms, we have, in particular then:

$$?_{\texttt{bool}} \sqsubseteq \bar{\texttt{b}}$$

$$?_{\texttt{nat}} \sqsubseteq \bar{\texttt{n}}$$

$$?_\tau \sqsubseteq \texttt{if } ?_{\texttt{bool}} \texttt{ then } ?_\tau \texttt{ else } ?_\tau$$

Now, we say that a term, r is *satisfiable* if there exists a determined t, such that r refines to t. Otherwise r is *unsatisfiable*. A consequence of the way we have axiomatised constants and, in particular, the conditional, is that if $\texttt{empty}_{\texttt{nat}}$ is an unsatisfiable term (see Remark 3.2.18) of type `nat`, then the conditional $\texttt{if true then } 3 \texttt{ else empty}_{\texttt{nat}}$ is unsatisfiable. We can not apply the equation $\beta_{\texttt{true}}$ of Figure 2.4 since $\texttt{empty}_{\texttt{nat}}$ is not determined.

This is in contrast to refinement calculi based on nondeterminism (*e.g.*[Bun97, Mor94]) so it is worth considering why we should expect this term to be unsatisfiable.

Although the calculus is (in)equational, the idea is that terms represent stages in the search for a program. We would only expect such a term to have arisen during refinement if the intention is to refine into a conditional, and so both branches must be refined to program code. Since this is not possible, the whole term is unsatisfiable. The fact that the satisfiability of a term depends on the satisfiability of all its subterms means that we can reason about specifications compositionally. In order to implement a specification, we need just implement its components. The alternative would be if we had to do some implementation, combine the resulting specifications somehow, then do some more implementation and so on. Thus we adhere to the 'principle of modular decomposition' advocated in [SST92]. The same principle applies when refining into the application of two terms (see Remark 5.2.7).

Remark 3.2.5 Let us consider why a 'naive' approach using free variables is not sufficient. Suppose we represent a stage of refinement as a term $t[x_1, \ldots, x_n]$ with free variables $x_1, \ldots, x_n$ such that $Q_1[x_1] \wedge \cdots \wedge Q_n[x_n] \supset P[t[x_1, \ldots, x_n]]$. The free variables stand for unwritten programs and to refine we replace a free variable with a term, possibly introducing more free variables and constraints. We could refine x_1 to a term $t_1[y_1, \ldots, y_m]$, say, by introducing new free variables, $y_1, \ldots, y_m$, with constraints such that

$$R_1[y_1] \wedge \cdots \wedge R_m[y_m] \supset Q_1[t_1[y_1, \ldots, y_m]]$$

However, this does not address the possibility of refining under an abstraction. If $t_1[y]$ is of the form $\lambda x : \sigma . t_2[y]$, then the constraint could be given as $\forall x : \sigma . \exists y : \tau \,.\, R[x, y] \supset Q_1[\lambda x : \sigma . t_2[y]]$, so now y is not a (global) free variable. If, instead, we represent variables under abstractions using functional variables, and write $\exists f : \sigma \to \tau . \forall x : \sigma \,.\, R[x, fx] \supset Q_1[\lambda x : \sigma . t_2[fx]]$, then this just avoids the issue: the unwritten program has the same representation as a variable of type $\sigma \to \tau$ and we make no progress!

Logically, this leads to arbitrarily nested quantifiers. In fact, the logic of refinement is a formalism for just that. This justifies the need for a 'theory of refinement' which can handle such reasoning more naturally.

Remark 3.2.6 Although a naive use of global variables is unable to account for variable capture, we could some form of variable labelled with the local context.

However, we would then need to make a distinction between variables representing something taken as given, and those representing something which remains to be implemented. We will make some suggestions for such a system, based on logical variables, in Section 6.6.

However, refinement calculi have traditionally been formulated in terms of some kind of specification construct, variations on the stubs we use here. The equivalence with, and axiomatisation in terms of logical variables is a subject for future work.

Remark 3.2.7 The ? is not the same as a nonterminating or undefined term, $\perp$. If it was, then in a call-by-value operational semantics, we would have $(\lambda x : \sigma.2)?_\sigma = ?_{\mathtt{nat}}$, which is not true in $\lambda_?$ if σ is inhabited; in a call-by-name semantics we would have $(\lambda x : \sigma.\langle x, x\rangle)?_\sigma = \langle ?_\sigma, ?_\sigma\rangle$ and this is not true in $\lambda_?$ if σ has more than one inhabitant.

More significantly, we show below in Remark 3.4.8 that interpreting ? as $\perp$ in a cpo does not even provide a sound model of $\lambda_?$.

Remark 3.2.8 In Chapter 1, we noted that Hilbert's ϵ-operator differs from ?. In particular, because models are given using a global choice function, the abstraction $\lambda x.\epsilon y.\top$ will always be interpreted as a constant function.

The equational theory of $\lambda_?$ developed in this chapter is based on the idea that we cannot substitute arbitrary underdetermined terms for variables, so we use let-terms. Since ϵ-expressions denote individuals, they can be substituted like other terms. For example, $(\lambda x.\langle x, x\rangle)\epsilon x.\top = \langle \epsilon x.\top, \epsilon x.\top\rangle$ is sound. In fact, the ϵ-operator can be axiomatised by adding $P[t/x] \supset P[\epsilon x.P/x]$ for each t and P, which is not the case for underdetermined terms.

The logic of the HOL proof assistant contains a (polymorphic) ϵ-operator, $\epsilon : (\alpha \to \mathtt{bool}) \to \alpha$. It is modelled using a choice function, and there is no explicit refinement.

The following lemma provides some insight into underdeterminism. Although stubs can be embedded anywhere in a term, we can give a canonical form with all the underdeterminism moved 'to the outside'. For example, $\lambda x : \sigma.\langle 2, ?_{\mathtt{nat}}\rangle$ is equal to $\mathtt{let}\ f : \sigma \to \mathtt{nat}\ \mathtt{be}\ ?_{\sigma\to\mathtt{nat}}\ \mathtt{in}\ \lambda x : \sigma.\langle 2, fx\rangle$. Thus, each term can be viewed as a simple combination of program and specification.

Lemma 3.2.9 *For all terms in context $\Gamma \vdash r : \tau$, there exists a determined term in context $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : \tau$ such that each x_i appears exactly once in t, and $\Gamma \vdash (\mathtt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \mathtt{be}\ ?_{\sigma_1}, \ldots, ?_{\sigma_n}\ \mathtt{in}\ t) =_\tau r$.*

Proof: Use `let` rules to move the underdeterminism outwards, the important case being the abstractions. ∎

Note the linearity – each stub counts exactly once, so $\langle ?_\sigma, ?_\sigma \rangle$ has canonical form $\texttt{let}\ x : \sigma, y : \sigma\ \texttt{be}\ ?_\sigma, ?_\sigma\ \texttt{in}\ \langle x, y \rangle$ and not $\texttt{let}\ z : \sigma\ \texttt{be}\ ?_\sigma\ \texttt{in}\ \langle z, z \rangle$ (to which it refines though). We will not, however, use the linearity in appeals to the lemma below. Note also that such canonical forms need not be unique. We sometimes use the useful abbreviation $\texttt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \texttt{in}\ t$ for the canonical form. So, for example, $\lambda x : \sigma . \langle 2, ?_{\texttt{nat}} \rangle$ is equal to $\texttt{let}\ f : \sigma \to \texttt{nat}\ \texttt{in}\ \lambda x : \sigma . \langle 2, fx \rangle$. In fact, by repeated pairing, we can always express terms in the simpler form $\texttt{let}\ x : \sigma\ \texttt{in}\ t$.

It is often convenient to write $\texttt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \texttt{in}\ t$ as $\texttt{let}\ \Gamma\ \texttt{in}\ t$, where Γ stands for the local context $x_1 : \sigma_1, \ldots, x_n : \sigma_n$. We also write $t[\Gamma]$ as informal notation for $t[x_1, \ldots, x_n]$.

By expressing terms in canonical form, we can prove a few results about `let` expressions. None of these results hold in the computational lambda calculus.

Lemma 3.2.10 *The following rules are admissible:*

1. *(*`let`*-commutativity)*

$$\frac{\Gamma \vdash r : \tau \qquad \Gamma \vdash r' : \tau' \qquad \Gamma, x : \tau, y : \tau' \vdash r'' : \tau''}{\Gamma \vdash \texttt{let}\ x : \tau, y : \tau'\ \texttt{be}\ r, r'\ \texttt{in}\ r'' =_{\tau''} \texttt{let}\ y : \tau', x : \tau\ \texttt{be}\ r', r\ \texttt{in}\ r''}$$

2. *(*`let`*-contraction)*

$$\frac{\Gamma \vdash r : \sigma \qquad \Gamma, x : \sigma, y : \sigma \vdash r' : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma, y : \sigma\ \texttt{be}\ r, r\ \texttt{in}\ r' \sqsubseteq_\tau \texttt{let}\ z : \sigma\ \texttt{be}\ r\ \texttt{in}\ r'[z/x, z/y]}$$

3. *(Strengthen local context)*
   For $x \notin FV(r')$,

$$\frac{\Gamma \vdash r' : \tau' \qquad \Gamma, x : \tau, y : \tau' \vdash r : \tau''}{\Gamma \vdash \lambda x : \tau . (\texttt{let}\ y : \tau'\ \texttt{be}\ r'\ \texttt{in}\ r) \sqsubseteq_{\tau \to \tau''} \texttt{let}\ y : \tau'\ \texttt{be}\ r'\ \texttt{in}\ \lambda x : \tau . r}$$

Proof: We prove (3).
First we show that

$$\lambda x : \sigma . \texttt{let}\ y : \tau\ \texttt{in}\ r[y] \sqsubseteq \texttt{let}\ y : \tau\ \texttt{in}\ \lambda x : \sigma . r[y] \qquad (3.4)$$

This is:

$$
\begin{array}{lll}
\lambda x:\tau.\mathtt{let}\ y:\tau'\ \mathtt{in}\ r[y] & & \\
\quad = & \mathtt{let}\ z:\tau\to\tau'\ \mathtt{in}\ \lambda x:\tau.r[zx] & (\textbf{Abstractions}) \\
\quad \sqsubseteq & \mathtt{let}\ y:\tau'\ \mathtt{in}\ \mathtt{let}\ z:\tau\to\tau'\ \mathtt{in}\ \lambda x:\tau.r[zx] & (\textbf{Let Weakening}) \\
\quad \sqsubseteq & \mathtt{let}\ y:\tau'\ \mathtt{in} & (y:\tau'\vdash ?_{\tau\to\tau'}\sqsubseteq\lambda x:\tau.y, \\
 & \qquad \mathtt{let}\ z:\tau\to\tau'\ \mathtt{be}\ \lambda x:\tau.y\ \mathtt{in}\ \lambda x:\tau.r[zx] & \text{and } \textbf{Congruence}) \\
\quad = & \mathtt{let}\ y:\tau'\ \mathtt{in}\ \lambda x:\tau.r[y] & (\textbf{Congruence})
\end{array}
$$

Then:

$$
\begin{array}{lll}
\lambda x:\tau.\mathtt{let}\ y:\tau'\ \mathtt{be}\ r'\ \mathtt{in}\ r[x,y] & & \\
\quad = & \lambda x:\tau.\mathtt{let}\ y:\tau'\ \mathtt{be}\ (\mathtt{let}\ z:\sigma\ \mathtt{in}\ t)\ \mathtt{in}\ r[x,y] & (\text{Lemma 3.2.9; can. form}) \\
\quad = & \lambda x:\tau.\mathtt{let}\ z:\sigma\ \mathtt{in}\ (\mathtt{let}\ y:\tau'\ \mathtt{be}\ t\ \mathtt{in}\ r[x,y]) & (\textbf{Let Associativity}) \\
\quad \sqsubseteq & \mathtt{let}\ z:\sigma\ \mathtt{in}\ \lambda x:\tau.(\mathtt{let}\ y:\tau'\ \mathtt{be}\ t\ \mathtt{in}\ r[x,y]) & (3.4) \\
\quad = & \mathtt{let}\ z:\sigma\ \mathtt{in}\ \mathtt{let}\ y:\tau'\ \mathtt{be}\ t\ \mathtt{in}\ \lambda x:\tau.r[x,y] & (\textbf{Let Beta}) \\
\quad = & \mathtt{let}\ y:\tau'\ \mathtt{be}\ (\mathtt{let}\ z:\sigma\ \mathtt{in}\ t)\ \mathtt{in}\ \lambda x:\tau.r[x,y] & (\textbf{Let Associativity}) \\
\quad = & \mathtt{let}\ y:\tau'\ \mathtt{be}\ r'\ \mathtt{in}\ \lambda x:\tau.r[x,y] & (\text{can. form})
\end{array}
$$

The proofs of (1) and (2) are carried out similarly, by expressing terms in canonical form. ∎

Commutativity of `let`'s corresponds to the idea that it does not matter what order we solve subproblems in (so long as they do not depend on each other), and contraction of `let`'s says that we can solve two identical problems by just solving the problem once and using the solution twice. The third rule illustrates the dependence of underdeterminism on the context. There are more determined terms which satisfy the term on the left, since x can be used in refining r', but this is not possible when r' is outside the bound variable on the right.

If all types are inhabited then, in fact, all terms are satisfiable. Because of this, the rule **Let Weakening** can be strengthened to an equality, *i.e.* for $x \notin FV(r')$, we have $r' = \mathtt{let}\ x:\sigma\ \mathtt{be}\ r\ \mathtt{in}\ r'$. We use this fact to derive strengthened forms of the β-equality for products and η-equality for units (Chapter 2) for arbitrary underdetermined terms (of appropriate type).

Proposition 3.2.11 *If all types are inhabited:*

1. $\dfrac{\Gamma\vdash r_1:\tau_1 \quad \Gamma\vdash r_2:\tau_2}{\Gamma\vdash \pi_i\langle r_1,r_2\rangle =_{\tau_i} r_i}\ (i=1,2)$

2. $\dfrac{\Gamma\vdash r:\mathbf{1}}{\Gamma\vdash r =_{\mathbf{1}} *}$

Proof:

1. The first is derived as:

$$
\begin{array}{lll}
\pi_1\langle r_1,r_2\rangle & = & \pi_1\langle \mathtt{let}\ x:\sigma\ \mathtt{in}\ t_1, \mathtt{let}\ y:\tau\ \mathtt{in}\ t_2\rangle \qquad (\text{for some } t_1,t_2) \\
 & = & \mathtt{let}\ x:\sigma, y:\tau\ \mathtt{in}\ \pi_1\langle t_1,t_2\rangle \\
 & = & \mathtt{let}\ x:\sigma, y:\tau\ \mathtt{in}\ t_1 \\
 & = & \mathtt{let}\ x:\sigma\ \mathtt{in}\ t_1 \\
 & = & r_1
\end{array}
$$

The $\sqsubseteq$ of the second last equality is by refining with some determined term of type τ, and the $\sqsupseteq$ is an instance of **Let Weakening**.

2. Suppose r has type $\mathbf{1}$. Now r has canonical form $\texttt{let}\ x : \sigma\ \texttt{in}\ t$, say, where t has type $\mathbf{1}$. By **Unit Equation** we can prove $x : \sigma \vdash t =_{\mathbf{1}} *$ and so $\texttt{let}\ x : \sigma\ \texttt{in}\ t =_{\mathbf{1}} \texttt{let}\ x : \sigma\ \texttt{in}\ *$, which since σ is inhabited, equals $*$. ∎

We can strengthen the β-equality for abstractions, without the assumption of nonemptiness.

Proposition 3.2.12 *The following is admissible:*

$$\frac{\Gamma, x : \sigma \vdash r : \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash (\lambda x : \sigma . r)t =_{\tau} r[t/x]}$$

Proof: We use the auxiliary result that for any t of type σ

$$\texttt{let}\ f : \sigma \to \tau\ \texttt{in}\ r[ft/x] = \texttt{let}\ x : \tau\ \texttt{in}\ r$$

Then,

$$\begin{array}{rcl}
(\lambda x : \sigma . r)t & = & (\lambda x : \sigma . \texttt{let}\ z : \tau'\ \texttt{in}\ t')t \\
& = & \texttt{let}\ f : \sigma \to \tau'\ \texttt{in}\ (\lambda x : \sigma . t'[fx/z])t \\
& = & \texttt{let}\ f : \sigma \to \tau'\ \texttt{in}\ t'[ft/z, t/x] \\
& = & \texttt{let}\ z : \tau'\ \texttt{in}\ t'[t/x] \\
& = & r[t/x]
\end{array}$$

∎

Remark 3.2.13 We can use Proposition 3.2.12 to derive Landin's Equation:

$$(\lambda x : \sigma . r)r' = \texttt{let}\ x : \sigma\ \texttt{be}\ r'\ \texttt{in}\ r$$

In principle, therefore, it would be possible to define `let` in terms of abstraction and application. However, this would lead to unnatural looking equivalents for the let axioms. A more significant reason for making `let` primitive is that this equation fails when we incorporate logic in the $\lambda_{\sqsubseteq}$ calculus in Chapter 5.

Because of Proposition 3.2.12, we can β-reduce applications with arbitrary function bodies. This is significant as it means that underdetermined terms can be executed, up to a point, as ordinary programs. In general, this is not possible as evaluation can not proceed when a stub is encountered. For example, $\pi_1(?_{\sigma \times \tau})$ cannot be reduced. This observation could form the basis for a single-step operational semantics, and we discuss this in Chapter 6.

At this point, we pause to review our motivation for studying this calculus. Program refinement is a stepwise decomposition of logical specifications and their gradual replacement with code. The calculus which we are studying here formalises refinement for a limited form of specification, with no logic, and is a fragment of a larger calculus studied in Chapter 5.

The statement corresponding to Proposition 3.2.12 does not hold in the full system (see Remark 5.2.7) and the auxiliary result fails too. That is, in the presence of logic, we cannot β-reduce with arbitrary function bodies. This is not a problem, as such, since the specification language is not intended to be evaluated. Rather, it is a bonus that β-reduction does make sense here. We believe that this is motivation for studying this subcalculus of a full logical refinement calculus. In an implementation of a program development system, we would like to be able to evaluate partially developed programs such as is formalised in the dynamic semantics of Extended ML [KST97]. If the system is based on the logical refinement calculus of Chapter 5, then we cannot directly evaluate terms. We can, however, use the fragment based on the $\lambda_?$-calculus.

Now, Propositions 3.2.11 and 3.2.12 show that we can deduce general forms of β-equality, although the axioms for pairs and abstractions are given for determined terms. We do not, however, have η-equalities for arbitrary pairs and abstractions.

To see why this should be so, recall, first, that we think intuitively of $\lambda_?$-terms as describing a set of values. Equality of terms corresponds to equality of the sets of values. Now, it is possible for two different sets of functions to return the same set of results for each argument. Thus the equation $\lambda x : \sigma . rx =_{\sigma \to \tau} r$ cannot be valid. Likewise, different sets of pairs can have the same set of first (or second) projections.

To illustrate this, we give two terms, r_1 and r_2, which have the same set of results for each argument, but such that r_1 does not equal r_2. Let $r_1 \equiv \texttt{let}\ b : \texttt{bool in}\ \lambda x : \sigma . b$ and $r_2 \equiv \lambda x : \sigma . ?_{\texttt{bool}}$. These terms are different, since one can refine to any function in $\sigma \to \texttt{bool}$, and the other to any such constant function. We can prove that they have the same set of results, for each argument $x : \sigma$, however. In fact, we have $\lambda x : \sigma . r_1 x =_{\sigma \to \tau} r_2$.

$$
\begin{array}{ll}
 & \lambda x : \sigma . (\texttt{let}\ b : \texttt{bool be}\ ?_{\texttt{bool}}\ \texttt{in}\ \lambda x : \sigma . b) x \\
= & \lambda x : \sigma . \texttt{let}\ b : \texttt{bool be}\ ?_{\texttt{bool}}\ \texttt{in}\ (\lambda x : \sigma . b) x \\
= & \lambda x : \sigma . ?_{\texttt{bool}}
\end{array}
$$

Remark 3.2.14 As in the computational lambda-calculus, application distributes over `let`'s. Using the rules of **Let Associativity** (twice) and **Applica-**

tions we get:

$$\begin{array}{rcl} (\texttt{let}\ x\ \texttt{be}\ r\ \texttt{in}\ r')t & = & \texttt{let}\ x_1\ \texttt{be}\ (\texttt{let}\ x\ \texttt{be}\ r\ \texttt{in}\ r')\ \texttt{in}\ x_1 t \\ & = & \texttt{let}\ x\ \texttt{be}\ r\ \texttt{in}\ (\texttt{let}\ x_1\ \texttt{be}\ r'\ \texttt{in}\ x_1 t) \\ & = & \texttt{let}\ x\ \texttt{be}\ r\ \texttt{in}\ r't \end{array}$$

Although the η-equalities do not hold for arbitrary pairs and abstractions, inequalities are admissible.

Proposition 3.2.15 *The following are admissible:*

$$\frac{\Gamma, x : \sigma \vdash r : \tau}{\Gamma \vdash \lambda x : \sigma.rx \sqsubseteq_{\sigma\to\tau} r} \qquad \frac{\Gamma \vdash r : \sigma \times \tau}{\Gamma \vdash \langle \pi_1 r, \pi_2 r\rangle \sqsubseteq_{\sigma\times\tau} r}$$

Proof: We sketch the proof of the first statement. By Lemma 3.2.9, r has canonical form $\texttt{let}\ y : \sigma'\ \texttt{in}\ t$. Then, $\lambda x : \sigma.(\texttt{let}\ y : \sigma'\ \texttt{in}\ t)x = \lambda x : \sigma.\texttt{let}\ y : \sigma'\ \texttt{in}\ tx$, by the distribution of application over `let`'s. By Lemma 3.2.10(3), this refines to $\texttt{let}\ y : \sigma'\ \texttt{in}\ \lambda x : \sigma.tx$, which by **Function Equations (η)** and **Congruence**, equals $\texttt{let}\ y : \sigma'\ \texttt{in}\ t$, that is, r. ∎

Proposition 3.2.16 *The axiom* **Abstraction** *follows from the simpler*

$$\frac{\Gamma, x : \sigma, y : \tau \vdash t[x, y] : \tau'}{\Gamma \vdash \texttt{let}\ z : \sigma \to \tau\ \texttt{in}\ \lambda x : \sigma.t[x, zx] =_{\sigma\to\tau'} \lambda x : \sigma.(\texttt{let}\ y : \tau\ \texttt{in}\ t[x, y])} \tag{3.5}$$

where the term t is determined.

Proof: The first step is to show, by induction over terms, that (3.5) is sufficient to prove that all terms have a canonical form. For the abstraction case, we have $\lambda x : \sigma.r = \lambda x : \sigma.\texttt{let}\ y : \tau\ \texttt{in}\ t[x, y]$, by induction, and using (3.5) we get $\texttt{let}\ f : \sigma \to \tau\ \texttt{in}\ \lambda x : \sigma.t[x, fx]$. Now we prove the full axiom.
Assume $\lambda x : \sigma.\texttt{let}\ y : \tau\ \texttt{in}\ r\ :\ \sigma \to \tau'$.

$$\begin{array}{rcl} \lambda x : \sigma.\texttt{let}\ y : \tau\ \texttt{in}\ r[x, y] & = & \lambda x : \sigma.\texttt{let}\ y : \tau, z : \tau'\ \texttt{in}\ t[y, z] \\ & = & \lambda x : \sigma.\texttt{let}\ p : \tau \times \tau'\ \texttt{in}\ t[\pi_1 p, \pi_2 p] \\ & = & \texttt{let}\ f'' : \sigma \to \tau \times \tau'\ \texttt{in}\ \lambda x : \sigma.t[\pi_1(f''x), \pi_2(f''x)] \\ & = & \texttt{let}\ f : \sigma \to \tau, f' : \sigma \to \tau'\ \texttt{in}\ \lambda x : \sigma.t[fx, f'x] \\ & = & \texttt{let}\ f : \sigma \to \tau\ \texttt{in}\ \lambda x : \sigma.\texttt{let}\ z : \tau'\ \texttt{in}\ t[fx, z] \\ & = & \texttt{let}\ f : \sigma \to \tau\ \texttt{in}\ \lambda x : \sigma.r[x, fx] \end{array}$$

∎

The next lemma says that our refinement rules are complete, in the sense that they allow us to construct by refinement any program which satisfies a specification (recalling that, for now, we view types as rudimentary specifications).

Lemma 3.2.17 *If $\Gamma \vdash t : \sigma$ then $\Gamma \vdash ?_\sigma \sqsubseteq_\sigma t$.*

Proof:
By Proposition 3.2.12, $?_\sigma = (\lambda x : \sigma.?_\sigma)t$. Then $(\lambda x : \sigma.?_\sigma)t \sqsubseteq (\lambda x : \sigma.x)t = t$. ∎

The fact that this can be proven trivially motivates restricted calculi better suited to proof search, where construction must be on the structure of t. We will discuss this in Chapter 6.

Remark 3.2.18 Using booleans, we can define a form of binary choice on terms. For $r, r' : \sigma$, define

$$r \| r' \triangleq \texttt{let}\ b : \texttt{bool}\ \texttt{in}\ (\texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ r')$$

We can prove that $\|$ is commutative, associative, and idempotent, and so is a reasonable notion of choice. This definition is useful because it helps to illustrate the differences between underdeterminism and *nondeterminism* (*e.g.*[Dij76]). If the reader is unfamiliar with nondeterminism, then this remark can be safely ignored.

We compare our axiomatisation of underdeterminism with a notion of external nondeterminism, that is, where the nondeterminism arises from the environment making the choice. In particular, we compare a nondeterministic choice operator, $\texttt{+}$, with $\|$. For example, we would intuitively expect to have $\lambda x : \texttt{nat}.2 \texttt{ + } 3 = \lambda x : \texttt{nat}.2 \texttt{ + } \lambda x : \texttt{nat}.3$ (using some constants for naturals). This contrasts with the properties of $\|$ under a binding, since $2\|3$ can refine to anything well-formed in the local context. For example, $2\|3 \sqsubseteq \texttt{if}\ x > 3\ \texttt{then}\ 2\ \texttt{else}\ 3$, so $\lambda x : \texttt{nat}.2\|3 \sqsubseteq \lambda x : \texttt{nat}.\texttt{if}\ x > 3\ \texttt{then}\ 2\ \texttt{else}\ 3$. Then we have $\lambda x : \texttt{nat}.2\|3 \sqsubseteq \lambda x : \texttt{nat}.2\|\lambda x : \texttt{nat}.3$, but not the reverse.

Another difference is between nondeterministic failure terms and the analogous idea for underdeterminism — unsatisfiable terms. Suppose the type τ is uninhabited. Then there are unsatisfiable terms at every type. Any term of the form $\texttt{let}\ x : \tau\ \texttt{in}\ r$, where r is any term of type σ, will be unsatisfiable. Now, for nondeterministic failure, we expect $r \texttt{ + } 0 = r$. However, if the term $\texttt{empty}_\sigma$ is unsatisfiable, then $r\|\texttt{empty}_\sigma$ is also unsatisfiable. Informally, '$r\|\texttt{empty}_\sigma = \texttt{empty}_\sigma$'.

We now show that 'underdeterminism commutes with determinism', in the sense that underdeterminism at a particular type can be expressed at a lower type using the relevant term constructor. This offers some conceptual justification for regarding underdeterminism as being a feature at a level above a programming language. There is no interaction with computation.

Proposition 3.2.19 *The following are derivable:*

1. $?_{\sigma\times\tau} = \langle ?_\sigma, ?_\tau \rangle$

2. $\langle \pi_1(?_{\sigma\times\tau}), \pi_2(?_{\sigma\times\tau}) \rangle = ?_{\sigma\times\tau}$

3. *If τ is inhabited, then $\pi_1(?_{\sigma\times\tau}) = ?_\sigma$, and if σ is inhabited, then $\pi_2(?_{\sigma\times\tau}) = ?_\tau$.*

4. $?_{\sigma\to\tau} = \lambda x : \sigma.?_\tau$ *and* $?_{\sigma\to\tau} r =_\tau ?_\tau$ *for all satisfiable* $r : \sigma$.

Proof: We prove (1), that $?_{\sigma\times\tau} =_{\sigma\times\tau} \langle ?_\sigma, ?_\tau\rangle$. Clearly, the refinement rule for **Pairs** lets us refine from left to right. We prove the other direction.

$$\begin{array}{rcl}
\langle ?_\sigma, ?_\tau\rangle & = & \texttt{let}\ x:\sigma, y:\tau\ \texttt{be}\ ?_\sigma, ?_\tau\ \texttt{in}\ \langle x,y\rangle \\
& \sqsubseteq & \texttt{let}\ z:\sigma\times\tau\ \texttt{be}\ ?_{\sigma\times\tau}\ \texttt{in} \\
& & \qquad \texttt{let}\ x:\sigma, y:\tau\ \texttt{be}\ ?_\sigma, ?_\tau\ \texttt{in}\ \langle x,y\rangle \\
& \sqsubseteq & \texttt{let}\ z:\sigma\times\tau\ \texttt{be}\ ?_{\sigma\times\tau}\ \texttt{in} \\
& & \qquad \texttt{let}\ x:\sigma, y:\tau\ \texttt{be}\ \pi_1 z, \pi_2 z\ \texttt{in}\ \langle x,y\rangle \\
& = & \texttt{let}\ z:\sigma\times\tau\ \texttt{be}\ ?_{\sigma\times\tau}\ \texttt{in}\ \langle \pi_1 z, \pi_2 z\rangle \\
& = & ?_{\sigma\times\tau}
\end{array}$$

The proofs of the other statements are carried out in a similar way, by expressing the terms in canonical form, and using the `let`-axioms and Lemma 3.2.10 to manipulate the terms. ∎

We can derive some useful refinement rules. As mentioned above, it is possible to derive *bottom-up* style refinement rules, complementing the top-down rules given as primitive, as well as various congruence rules. Bottom-up rules put together existing programs, whereas top-down rules decompose specifications. We formulate the bottom-up rules in terms of manipulating variables in the context.

Proposition 3.2.20 *The following are derivable:*

Bottom-up Refinement

$$\frac{\Gamma \vdash t : \sigma \quad \Gamma, x:\tau \vdash r \sqsubseteq_\nu r'[x]}{\Gamma, y : \sigma\to\tau \vdash r \sqsubseteq_\nu r'[yt]}\ x \notin FV(r)$$

$$\frac{\Gamma, x:\tau \vdash r \sqsubseteq_\sigma r'[x]}{\Gamma, z:\tau\times\tau' \vdash r \sqsubseteq_\sigma r'[\pi_1 z]}\ x \notin FV(r) \qquad \frac{\Gamma, y:\tau' \vdash r \sqsubseteq_\sigma r'[y]}{\Gamma, z:\tau\times\tau' \vdash r \sqsubseteq_\sigma r'[\pi_2 z]}\ y \notin FV(r)$$

Congruence

$$\frac{\Gamma \vdash r_1 \sqsubseteq_\sigma r_1' \quad \Gamma \vdash r_2 \sqsubseteq_\tau r_2'}{\Gamma \vdash \langle r_1, r_2\rangle \sqsubseteq_{\sigma\times\tau} \langle r_1', r_2'\rangle} \qquad \frac{\Gamma \vdash r_1 \sqsubseteq_{\tau_1} r_1' \ \cdots\ \Gamma \vdash r_n \sqsubseteq_{\tau_n} r_n'}{\Gamma \vdash k(r_1,\ldots,r_n) \sqsubseteq_\tau k(r_1',\ldots,r_n')}$$

$$\frac{\Gamma \vdash r \sqsubseteq_{\tau_1\times\tau_2} r'}{\Gamma \vdash \pi_i(r) \sqsubseteq_{\tau_i} \pi_i(r')}\ (i=1,2) \qquad \frac{\Gamma \vdash r_1 \sqsubseteq_{\sigma\to\tau} r_1' \quad \Gamma \vdash r_2 \sqsubseteq_\sigma r_2'}{\Gamma \vdash r_1 r_2 \sqsubseteq_\tau r_1' r_2'}$$

Proof: The bottom-up refinement rules are obtained by substitution. The first projection rule, for example, is derived as:

$$\cfrac{\cfrac{\cfrac{\Gamma, x : \tau \vdash r \sqsubseteq_\sigma r'[x]}{\Gamma \vdash \lambda x : \tau . r \sqsubseteq_{\tau \to \sigma} \lambda x : \tau . r'[x]}}{\Gamma, z : \tau \times \tau' \vdash (\lambda x : \tau . r)\pi_1 z \sqsubseteq_\sigma (\lambda x : \tau . r'[x])\pi_1 z}}{\Gamma, z : \tau \times \tau' \vdash r \sqsubseteq_\sigma r'[\pi_1 z]}$$

Alternatively, this can be derived directly using **Substitution**. The congruence rules all follow from the rule for let-terms. For example, the rule for pairs follows since $\langle r_1, r_2 \rangle$ equals $\texttt{let}\ x_1 : \tau_1, x_2 : \tau_2\ \texttt{be}\ r_1, r_2\ \texttt{in}\ \langle x_1, x_2 \rangle$. Then using **Congruence (let)**, this refines to $\texttt{let}\ x_1 : \tau_1, x_2 : \tau_2\ \texttt{be}\ r_1', r_2'\ \texttt{in}\ \langle x_1, x_2 \rangle$, which equals $\langle r_1', r_2' \rangle$. ∎

We make a similar point here to that made after Lemma 3.2.17. In a search-directed refinement calculus (as discussed in Chapter 1), where the rules are given for direct refinement, we would expect these rules would be primitive. Our intention here, though, is to give a system complete for proving arbitrary refinements of the form $r \sqsubseteq_\tau r'$ and not goal-directed refinements $r \sqsubseteq_\tau t$.

Another point is that although we can infer forms of the bottom-up rules in which the substitution takes place in both terms (for example, if $y : \tau' \vdash r[y] \sqsubseteq_\sigma r'[y]$ then $z : \tau \times \tau' \vdash r[\pi_2 z] \sqsubseteq_\sigma r'[\pi_2 z]$) the forms we gave are more suitable for directed refinement.

Example 3.2.21 We can use **Let Weakening** (or the derived rule of contraction in Lemma 3.2.10) to combine two equivalent stubs into one:

$$\langle ?_\sigma, ?_\sigma \rangle \ \sqsubseteq\ \texttt{let}\ z : \sigma\ \texttt{be}\ ?_\sigma\ \texttt{in}\ \langle ?_\sigma, ?_\sigma \rangle \ \sqsubseteq\ \texttt{let}\ z : \sigma\ \texttt{be}\ ?_\sigma\ \texttt{in}\ \langle z, z \rangle$$

The first step uses **Let Weakening**, and the second uses the refinement rules **Variables**, **Reflexivity**, **Pairs** and **Congruence (let)**.

Example 3.2.22 We give a short example, using refinement to derive a swap function. Transitivity of refinement means that we can often present much of a refinement derivation as a form of equational reasoning. Here there are two main steps.

$$?_{\sigma \times \tau \to \tau \times \sigma} \ \sqsubseteq\ \lambda z : \sigma \times \tau . ?_{\tau \times \sigma} \ \sqsubseteq\ \lambda z : \sigma \times \tau . \langle \pi_2 z, \pi_1 z \rangle$$

Formally, this is

$$\cfrac{\cfrac{}{\vdash ?_{\sigma \times \tau \to \tau \times \sigma} \ \sqsubseteq\ \lambda z : \sigma \times \tau . ?_{\tau \times \sigma}} \quad \cfrac{}{\vdash \lambda z : \sigma \times \tau . ?_{\tau \times \sigma} \ \sqsubseteq\ \lambda z : \sigma \times \tau . \langle \pi_2 z, \pi_1 z \rangle}}{\vdash ?_{\sigma \times \tau \to \tau \times \sigma} \ \sqsubseteq\ \lambda z : \sigma \times \tau . \langle \pi_2 z, \pi_1 z \rangle}\ \textbf{Trans.}$$

The first hypothesis follows from **Congruence (abs)**, and the second is derived as

$$\dfrac{\dfrac{\dfrac{}{z : \sigma \times \tau \vdash ?_{\tau\times\sigma} \sqsubseteq \langle ?_\tau, ?_\sigma \rangle}\,\textbf{Pairs} \qquad \dfrac{\textit{see below}}{z : \sigma \times \tau \vdash \langle ?_\tau, ?_\sigma \rangle \sqsubseteq \langle \pi_2 z, \pi_1 z \rangle}}{z : \sigma \times \tau \vdash ?_{\tau\times\sigma} \sqsubseteq \langle \pi_2 z, \pi_1 z \rangle}\,\textbf{Trans.}}{\vdash \lambda z : \sigma \times \tau . ?_{\tau\times\sigma} \sqsubseteq \lambda z : \sigma \times \tau . \langle \pi_2 z, \pi_1 z \rangle}\,\textbf{Congruence}$$

$$\dfrac{\dfrac{\dfrac{}{y : \tau \vdash ?_\tau \sqsubseteq y}\,\textbf{Variables}}{z : \sigma \times \tau \vdash ?_\tau \sqsubseteq \pi_2 z}\,\textbf{Congruence} \qquad \dfrac{\dfrac{}{x : \sigma \vdash ?_\sigma \sqsubseteq x}\,\textbf{Variables}}{z : \sigma \times \tau \vdash ?_\sigma \sqsubseteq \pi_1 z}\,\textbf{Congruence}}{z : \sigma \times \tau \vdash \langle ?_\tau, ?_\sigma \rangle \sqsubseteq \langle \pi_2 z, \pi_1 z \rangle}\,\textbf{Congruence}$$

3.3 Metatheory

In this section we prove some results which illustrate the fine structure of the refinement relation. Intuitively, refinement is a combination of coding, where stubs are replaced with program code, and equational reasoning using the rules of the calculus. It is possible to formalise this by defining an explicit coding relation, $\rightsquigarrow$, and factorising refinement into a combination of $\rightsquigarrow$ and $=$.

Definition 3.3.1 *We define the* coding *relation on well-typed terms,* $\Gamma \vdash r \rightsquigarrow r'$*, as the reflexive, transitive, congruence closure of the following one-step relation:*

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash ?_\tau \rightsquigarrow t}$$

We can express any term, r , in the form $t[?^{\Gamma_1}_{\sigma_1}, \ldots, ?^{\Gamma_n}_{\sigma_n}]$, where $?^{\Gamma_i}_{\sigma_i}$ means that the subterm $?_{\sigma_i}$ appears in 'local context' (of `let`'s and λ's) Γ_i, and each such subterm appears exactly once. Then $\Gamma \vdash r \rightsquigarrow t'$ means that for each $i = 1 \ldots n$, there exists a determined term $\Gamma, \Gamma_i \vdash t_i : \sigma_i$, such that $\Gamma \vdash t[t_1, \ldots, t_n] =_\tau t'$.

We now show that any refinement to a program can be given as a 'standard refinement sequence' consisting of coding followed by equational reasoning.

Lemma 3.3.2 *Let* $\langle Sg, Ax \rangle$ *be a* $\lambda_?$*-axiom system and suppose that* $\langle Sg, Ax \rangle \triangleright \Gamma \vdash r \sqsubseteq_\tau t'$*. Then there exists a term* $Sg \triangleright \Gamma \vdash t : \tau$ *such that* $\Gamma \vdash r \rightsquigarrow t$ *and* $\langle Sg, Ax \rangle \triangleright \Gamma \vdash t =_\tau t'$*.*

Proof: We can prove the lemma by induction on the derivation of refinement. The inductive case involves showing that the coding can be extended back along each rule. We consider two cases. The reasoning is similar for the others.

- **Let Weakening**

  Suppose $\Gamma \vdash r \sqsubseteq \texttt{let}\ x : \sigma\ \texttt{be}\ r'\ \texttt{in}\ r \rightsquigarrow t''$, where $r \equiv t_1[?^{\Gamma_1}_{\sigma_1}, \ldots, ?^{\Gamma_n}_{\sigma_n}]$. By the inductive hypothesis, there exists a t' and $t_2[x]$ such that $\Gamma \vdash r' \sqsubseteq t'$ (the details of this refinement do not matter), and $\Gamma, x : \sigma \vdash r \rightsquigarrow t_2[x]$ with $\Gamma \vdash \texttt{let}\ x : \sigma\ \texttt{be}\ t'\ \texttt{in}\ t_2 = t''$, so $\Gamma \vdash t_2[t'] = t''$. Now the coding of r gives terms $\Gamma, x : \sigma, \Gamma_i \vdash u_i : \sigma_i$ for each $i = 1 \ldots n$, and so $\Gamma, \Gamma_i \vdash u_i[t'/x] : \sigma_i$.

  Hence, $\Gamma \vdash r \rightsquigarrow t_2[t'] = t''$, by refining $?^{\Gamma_i}_{\sigma_i}$ to $u_i[t_i/x]$.

- **Abstractions**

  Since this rule is given as an equation, we must consider the two directions of refinement separately. Suppose

  $$\Gamma \vdash \texttt{let}\ z : \sigma \to \tau\ \texttt{in}\ \lambda x : \sigma . t'[?^{\Gamma_i}_{\sigma_i}, x, zx] \sqsubseteq \lambda x : \sigma .(\texttt{let}\ y : \tau\ \texttt{in}\ t'[?^{\Gamma_i}_{\sigma_i}, x, y]) \rightsquigarrow t$$

  (using the abbreviated notation for let-terms). For clarity, we just indicate the one specification $?_{\sigma_i}$. Here Γ_i records the context in t'.

  By induction, there exist terms $\Gamma, x : \sigma, y : \tau, \Gamma_i \vdash u_i[x, y] : \sigma_i$ and $\Gamma, x : \sigma \vdash u[x] : \tau$ such that $\Gamma \vdash \lambda x : \sigma .(\texttt{let}\ y : \tau\ \texttt{be}\ u\ \texttt{in}\ t'[u_i, x, y]) = t$, that is, $\Gamma \vdash \lambda x : \sigma . t'[u_i[x, u], x, u] = t$.

  Then, $\Gamma \vdash \lambda x : \sigma . u[x] : \sigma \to \tau$ and $\Gamma, z : \sigma \to \tau, x : \sigma, \Gamma_i \vdash u_i[x, zx] : \sigma_i$ so we can refine the left term to $\Gamma \vdash \texttt{let}\ z : \sigma \to \tau\ \texttt{be}\ \lambda x : \sigma . u[x]\ \texttt{in}\ \lambda x : \sigma . t'[u_i[x, zx], x, zx]$, which equals $\lambda x : \sigma . t'[u_i[x, u[x]], x, u[x]]$, and this equals t.

  Now we consider the reverse refinement. Suppose

  $$\Gamma \vdash \lambda x : \sigma .(\texttt{let}\ y : \tau\ \texttt{in}\ t'[?^{\Gamma_i}_{\sigma_i}, x, y]) \sqsubseteq \texttt{let}\ z : \sigma \to \tau\ \texttt{in}\ \lambda x : \sigma . t'[?^{\Gamma_i}_{\sigma_i}, x, zx] \rightsquigarrow t$$

  By induction, we have terms $\Gamma \vdash u : \sigma \to \tau$ and $\Gamma, z : \sigma \to \tau, x : \sigma, \Gamma_i \vdash u_i[z, x] : \sigma_i$ such that

  $$\Gamma \vdash \texttt{let}\ z : \sigma \to \tau\ \texttt{be}\ u\ \texttt{in}\ \lambda x : \sigma . t'[u_i[z, x], x, zx] = t$$

  so $\Gamma \vdash \lambda x : \sigma . t'[u_i[u, x], x, ux] = t$.

  Hence we have terms

  $$\Gamma, x : \sigma \vdash ux : \tau$$

  $$\Gamma, x : \sigma, y : \tau, \Gamma_i \vdash u_i[u, x] : \sigma_i$$

  for which

  $$\Gamma \vdash \lambda x : \sigma .(\texttt{let}\ y : \tau\ \texttt{be}\ ux\ \texttt{in}\ t'[u_i[u, x], x, y]) = t$$

  and so we can refine the left term in the standard way.

$\blacksquare$

We can use this lemma to deduce that all auxiliary 'claims' made using **Let Weakening** can be immediately satisfied.

Lemma 3.3.3 *Let $\langle Sg, Ax\rangle$ be a $\lambda_?$-axiom system.*
If $\langle Sg, Ax\rangle \triangleright \Gamma \vdash \texttt{let}\ z : \sigma\ \texttt{be}\ r\ \texttt{in}\ r' \sqsubseteq_\tau t''$ then there exists a determined term $Sg \triangleright \Gamma \vdash t : \sigma$ such that $\langle Sg, Ax\rangle \triangleright \Gamma \vdash r \sqsubseteq_\sigma t$ and $\langle Sg, Ax\rangle \triangleright \Gamma \vdash r'[t/z] \sqsubseteq_\tau t''$.

Proof: Suppose that $r \equiv u[?^{\Gamma_i}_{\sigma_i}]$ and $r' \equiv u'[z, ?^{\Gamma_j}_{\tau_j}]$. Then by Lemma 3.3.2, there exist t_i, u_j such that $\Gamma \vdash \texttt{let}\ z : \sigma\ \texttt{be}\ t[t_i]\ \texttt{in}\ t'[z, u_j] =_\tau t''$. Hence, $\Gamma \vdash r \sqsubseteq_\sigma t[t_i]$ and $r'[t[t']] \sqsubseteq_\tau t'[t[t_i], u_j] =_\tau t''$. $\blacksquare$

Remark 3.3.4 For the above two lemmas to hold it is crucial that the axioms of a $\lambda_?$-axiom system just consist of determined equations, $t =_\tau t'$. If, for example, we had constants $k_1, k_2, k_3 : \tau$ and axiom $k_1 \| k_2 \sqsubseteq_\tau k_3$, that is, $\texttt{let}\ b : \texttt{bool}\ \texttt{in}\ (\texttt{if}\ b\ \texttt{then}\ k_1\ \texttt{else}\ k_2) \sqsubseteq_\tau k_3$, then we would not be able to find a specific $t : \texttt{bool}$ such that $\texttt{if}\ t\ \texttt{then}\ k_1\ \texttt{else}\ k_2 =_\tau k_3$.

In the introduction, we said that refinement should be thought of intuitively as a reduction in the set of programs which satisfy a specification (or refinement term, rather). We can formalise this by defining an ordering $r \lesssim^\Gamma_\tau r'$, on well-formed terms $\Gamma \vdash r : \tau$ and $\Gamma \vdash r' : \tau$, to mean: for all $\Gamma' \supseteq \Gamma$, for all determined $\Gamma' \vdash t : \tau$, if $\Gamma' \vdash r' \sqsubseteq_\tau t$ then $\Gamma' \vdash r \sqsubseteq_\tau t$.

Lemma 3.3.5 *(Refinement Mappings)*
If $\texttt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \texttt{in}\ t \lesssim^\Gamma_\tau \texttt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \texttt{in}\ t'$ then for all $i = 1 \ldots n$, there exist terms $\Gamma, y_1 : \tau_1, \ldots, y_m : \tau_m \vdash t_i : \sigma_i$, such that $\Gamma, y_1 : \tau_1, \ldots, y_m : \tau_m \vdash t[t_1/x_1, \ldots, t_n/x_n] =_\tau t'$.

Proof: First of all, note that this can be reduced by repeated pairing to the one variable case. Now if $\texttt{let}\ x : \sigma\ \texttt{in}\ t \lesssim^\Gamma_\upsilon \texttt{let}\ y : \tau\ \texttt{in}\ t'$, then since $\Gamma, y : \tau \vdash \texttt{let}\ y : \tau\ \texttt{in}\ t' \sqsubseteq t'[y]$, we have $\Gamma, y : \tau \vdash \texttt{let}\ x : \sigma\ \texttt{in}\ t \sqsubseteq t'[y]$, so by Lemma 3.3.3, there must exist a term $\Gamma, y : \tau \vdash u[y]$ such that $\Gamma, y : \tau \vdash t[u[y]] =_\upsilon t'$. $\blacksquare$

Because of the definition of $\lesssim^\Gamma_\tau$ in terms of all $\Gamma' \supseteq \Gamma$ we do not need to assume that all types are inhabited. If ϵ_1 and ϵ_2 are both empty types, then it might seem that $\texttt{let}\ x : \epsilon_1\ \texttt{in}\ * \lesssim^{\langle\rangle}_{\mathbf{1}} \texttt{let}\ x : \epsilon_2\ \texttt{in}\ *$ since neither term can refine to a determined term in the empty context, yet we cannot produce a term $t : \epsilon_2$. The point is, though, that since we can use the context, $x : \epsilon_2$, to refine `let` $x : \epsilon_2$ `in` $*$ but not `let` $x : \epsilon_1$ `in` $*$ the terms are not related by $\lesssim_{\mathbf{1}}^{\langle\rangle}$.

These metatheoretic results will be used in the completeness proof of the next section.

3.4 Models

We can interpret the calculus using a simple generalisation of Henkin models. In Chapter 2, Section 2.2, we used $\lambda^{\times\rightarrow}$-Henkin Interpretations to give models of the simply-typed lambda calculus. There, terms of type τ were interpreted as elements of a set $\tau^{\mathcal{A}}$. We will use the same apparatus, but interpret our underdetermined terms as subsets of the $\tau^{\mathcal{A}}$, rather than elements.

We make one additional assumption of our models. We require that the function sets be 'closed under factoring'.

Definition 3.4.1 *We say that a $\lambda^{\times\rightarrow}$-Henkin interpretation satisfies the* factoring condition *if, letting $f' \in (\tau \rightarrow \tau')^{\mathcal{A}}$, $f \in (\sigma \rightarrow \tau')^{\mathcal{A}}$: if for all $b \in \tau^{\mathcal{A}}$, there exists an $a \in \sigma^{\mathcal{A}}$ such that $\mathtt{App}(f', b) = \mathtt{App}(f, a)$, then there exists an element $g \in (\tau \rightarrow \sigma)^{\mathcal{A}}$ such that for all $b \in \tau^{\mathcal{A}}$, $\mathtt{App}(f', b) = \mathtt{App}(f, \mathtt{App}(g, b))$. In other words, writing $\bar{f}$ for the function associated with $f \in A^{\sigma\rightarrow\tau}$, that is, $(a \in \sigma^{\mathcal{A}} \mapsto \mathtt{App}(f, a))$: if there exists a function $h : \tau^{\mathcal{A}} \rightarrow \sigma^{\mathcal{A}}$ such that $\bar{f}' = h; \bar{f}$, then there exists an element $g \in (\tau \rightarrow \sigma)^{\mathcal{A}}$ such that $\bar{f}' = \bar{g}; \bar{f}$. Note that we do not require $\bar{g} = h$.*

The factoring condition is essentially a form of choice axiom and is necessary in order to prove soundness.

Recall that signatures are the same for $\lambda_?$ and $\lambda^{\times\rightarrow}$.

Definition 3.4.2 *Let $\langle Sg, Ax\rangle$ be a $\lambda_?$-signature. A $\lambda_?$-*Henkin interpretation *of $\langle Sg, Ax\rangle$ is a $\lambda^{\times\rightarrow}$-Henkin interpretation of $\langle Sg, Ax\rangle$ with the factoring condition.*

Note that we do not require an environment model condition directly on the interpretation of $\lambda_?$ terms, but rather on the underlying $\lambda^{\times\rightarrow}$-Henkin interpretation. The condition lifts in the sense that satisfiable terms are given nonempty interpretations. Moreover, the extensionality condition means that programs have unique interpretations (as singleton sets).

Before proceeding further, we give an example of a Henkin interpretation with the factoring condition.

Example 3.4.3 (Full set-theoretic function hierarchy) The sets of the full set-theoretic function hierarchy for a given signature are defined inductively as

$$\gamma^{\mathcal{A}} = \text{ any set}$$

$$\mathbf{1}^{\mathcal{A}} = \{*\}$$

$$(\sigma \times \tau)^{\mathcal{A}} = \sigma^{\mathcal{A}} \times \tau^{\mathcal{A}}$$

$$(\sigma \to \tau)^{\mathcal{A}} = \sigma^{\mathcal{A}} \to \tau^{\mathcal{A}}$$

The projection and application maps are the usual set-theoretic maps, and constants are interpreted as any elements in the appropriate sets.

We will see below that the open term model is another example. First though, we give an example of a Henkin interpretation which does not satisfy the factoring condition.

Example 3.4.4 The applied $\lambda^{\times\to}$ theory with primitive types $\mathtt{nat}$ and $\mathtt{bool}$, and constants $\mathtt{0} : \mathtt{nat}$, $\mathtt{succ} : \mathtt{nat} \to \mathtt{nat}$, $\mathtt{true} : \mathtt{bool}$, $\mathtt{false} : \mathtt{bool}$, $\mathtt{cond} : \mathtt{bool}, \mathtt{nat}, \mathtt{nat} \to \mathtt{nat}$ and $\mathtt{eq} : \mathtt{nat}, \mathtt{nat} \to \mathtt{bool}$ has a Henkin model in which $\mathtt{nat}^{\mathcal{A}} = \mathcal{N}$, the set of natural numbers, $\mathtt{bool}^{\mathcal{A}} = \mathcal{B}$, the set of boolean truth values, and the constants have the expected interpretations, where the function sets are subsets of the space $(\sigma \to \tau)^{\mathcal{A}} \subseteq \sigma^{\mathcal{A}} \to \tau^{\mathcal{A}}$. The elements of the model are just those generated by the environment model condition, that is, just those elements required to interpret the terms of the calculus. In particular, there is no element corresponding to the predecessor function, although the function *pred* such that $\mathit{pred}; \overline{\mathtt{succ}} = \overline{\mathtt{id}_{\mathtt{pos}}}$, where $\mathtt{id}_{\mathtt{pos}}$ is the identity on positive naturals $\lambda n : \mathtt{nat}.\mathtt{cond}(\mathtt{eq}(n, \mathtt{0}), \mathtt{1}, n)$, clearly exists.

As for $\lambda^{\times\to}$ in Chapter 2, meanings are given in an environment. For context $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$ $(n \geq 0)$, a Γ-*environment*, η, in interpretation $\mathcal{A}$, is a tuple of elements defined as for $\lambda^{\times\to}$, and *not* a tuple of subsets. We write $\eta \vDash^{\mathcal{A}} \Gamma$ when η is a Γ-environment in $\mathcal{A}$.

Now we interpret $\Gamma \vdash r : \tau$ inductively on the structure of the typing derivation. In Figure 3.7 we define the interpretation of terms in context, $\Gamma \vdash r : \tau$, at Γ-environment, η, written $[\![\Gamma \vdash r : \tau]\!]^{\mathcal{A}}(\eta)$. We write the $\lambda_?$-interpretation of $*$ as $\{*\}$ rather than (the equivalent) $\mathbf{1}^{\mathcal{A}}$ to emphasise the fact that the interpretations are subsets. Similarly, we write the interpretation of $?_\sigma$ as $\{a \mid a \in \sigma^{\mathcal{A}}\}$.

We say that the typing judgement, $\Gamma \vdash r : \tau$, is *true in interpretation* $\mathcal{A}$ *and* Γ-*environment,* η, written $\Gamma \vDash^{\mathcal{A},\eta} r : \tau$, when $[\![\Gamma \vdash r : \tau]\!]^{\mathcal{A}}(\eta) \subseteq \tau^{\mathcal{A}}$. We say that

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

$$\frac{[\![\Gamma \vdash r_1 : \tau_1]\!] = m_1 \ \cdots \ [\![\Gamma \vdash r_n : \tau_n]\!] = m_n}{[\![\Gamma \vdash k(r_1, \ldots, r_n) : \tau]\!](\eta) = \{k^{\mathcal{A}}(a_1, \ldots a_n) \mid a_i \in m_i(\eta)\}}$$

$$[\![\Gamma \vdash * : \mathbf{1}]\!](\eta) = \{*\}$$

$$\frac{[\![\Gamma \vdash r : \sigma]\!] = m \quad [\![\Gamma \vdash r' : \tau]\!] = m'}{[\![\Gamma \vdash \langle r, r' \rangle : \sigma \times \tau]\!](\eta) = \{a \in (\sigma \times \tau)^{\mathcal{A}} \mid \mathtt{Proj}_1^{\sigma,\tau}(a) \in m(\eta), \mathtt{Proj}_2^{\sigma,\tau}(a) \in m'(\eta)\}}$$

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

$$[\![\Gamma \vdash ?_\sigma : \sigma]\!](\eta) = \{a \mid a \in \sigma^{\mathcal{A}}\}$$

$$\frac{[\![\Gamma \vdash r : \tau \times \tau']\!] = m}{[\![\Gamma \vdash \pi_1(r) : \tau]\!](\eta) = \{\mathtt{Proj}_1^{\tau,\tau'}(a) \mid a \in m(\eta)\}}$$

$$\frac{[\![\Gamma \vdash r : \tau \times \tau']\!] = m}{[\![\Gamma \vdash \pi_2(r) : \tau']\!](\eta) = \{\mathtt{Proj}_2^{\tau,\tau'}(a) \mid a \in m(\eta)\}}$$

$$\frac{[\![\Gamma \vdash r : \sigma \to \tau]\!] = m \quad [\![\Gamma \vdash r' : \sigma]\!] = m'}{[\![\Gamma \vdash rr' : \tau]\!](\eta) = \{\mathtt{App}(f, a) \mid f \in m(\eta), a \in m'(\eta)\}}$$

$$\frac{[\![\Gamma \vdash r : \sigma]\!] = m \quad [\![\Gamma, x:\sigma \vdash r' : \tau]\!] = m'}{[\![\Gamma \vdash \mathtt{let}\ x:\sigma\ \mathtt{be}\ r\ \mathtt{in}\ r' : \tau]\!](\eta) = \bigcup_{a \in m(\eta)} m'(\langle \eta, a \rangle)}$$

Figure 3.7: Interpretation of Well-formed Terms

$\Gamma \vdash r : \tau$ is *true in* $\mathcal{A}$, written $\Gamma \models^{\mathcal{A}} r : \tau$, when it is true for all Γ-environments in $\mathcal{A}$. It is easily seen that we have soundness of typing, that is, if $\Gamma \vdash r : \tau$, then $\Gamma \models^{\mathcal{A}} r : \tau$.

Similarly, we say that the refinement, $\Gamma \vdash r \sqsubseteq_\tau r'$, is *true in interpretation* $\mathcal{A}$ *and environment* η, written $\Gamma \models^{\mathcal{A},\eta} r \sqsubseteq_\tau r'$, when $[\![\Gamma \vdash r : \tau]\!]^{\mathcal{A}}(\eta) \supseteq [\![\Gamma \vdash r' : \tau]\!]^{\mathcal{A}}(\eta)$, and define truth in an interpretation to be for all environments. We will usually drop explicit annotation of an interpretation, $\mathcal{A}$.

Definition 3.4.5 *Let* $\langle Sg, Ax \rangle$ *be a* $\lambda_?$*-axiom system, and let* $\mathcal{A}$ *be a* $\lambda_?$*-Henkin interpretation of* Sg*. We say that* $\mathcal{A}$ *is a* $\lambda_?$-Henkin model *of* $\langle Sg, Ax \rangle$ *when each axiom in* Ax *is true in* $\mathcal{A}$*.*

We will prove soundness of refinement below, but first, it is easy to see that for determined t, the interpretation $[\![\Gamma \vdash t : \tau]\!](\eta)$ is a singleton set. Now we prove a standard lemma.

Lemma 3.4.6 *(Substitution Lemma) For well-formed terms in context $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash r : \tau$ and $\Gamma \vdash t_i : \tau_i$ $(i = 1, \ldots, n)$, we have*

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash r : \tau]\!](\langle a_1, \ldots, a_n \rangle) = [\![\Gamma \vdash r[t_i/x_i]]\!](\eta)$$

where a_i is the unique inhabitant of $[\![\Gamma \vdash t_i : \tau_i]\!](\eta)$.

Proof: The proof is a straightforward induction over the typing judgement $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash r : \tau$. ∎

This may be compared with the analogous form for 'substituting' an underdetermined term, which follows directly from the semantics of let-terms:

$$\bigcup_{a \in m(\eta)} [\![\Gamma, x : \sigma \vdash r' : \tau]\!](\langle \eta, a \rangle) = [\![\Gamma \vdash \texttt{let}\ x : \sigma\ \texttt{be}\ r\ \texttt{in}\ r' : \tau]\!](\eta)$$

where $m = [\![\Gamma \vdash r : \sigma]\!]$.

Theorem 3.4.7 *(Soundness) Let $\mathcal{A}$ be a $\lambda_?$-Henkin model of $\lambda_?$-axiom system, $\langle Sg, Ax \rangle$. If $\langle Sg, Ax \rangle \rhd \Gamma \vdash r \sqsubseteq_\tau r'$ then $\Gamma \models^{\mathcal{A}} r \sqsubseteq_\tau r'$.*

Proof: The proof is by induction on the derivation of the judgement. Most cases are straightforward. We prove two key cases.

- **Let Beta**

  The interpretation $[\![\Gamma \vdash \texttt{let}\ x : \tau\ \texttt{be}\ t\ \texttt{in}\ r : \tau']\!](\eta)$ is defined to be $\bigcup_{a \in [\![\Gamma \vdash t : \tau]\!](\eta)} [\![\Gamma, x : \tau \vdash r]\!]\langle \eta, a \rangle$. Since t is determined, this is $[\![\Gamma, x : \tau \vdash r : \tau']\!]\langle \eta, a \rangle$ where a is the unique member of $[\![\Gamma \vdash t : \tau]\!](\eta)$, so by the substitution lemma, the interpretation is $[\![\Gamma \vdash r[t/x] : \tau']\!](\eta)$.

- **Abstractions** (Equality rules)

  It is easier to prove the simpler rule (3.5) of Proposition 3.2.16 sound (implies the soundness of the full rule).

  Let $\eta \models \Gamma$.

  We aim to prove that

  $$\begin{aligned}[\![\Gamma \vdash \texttt{let}\ z : \sigma \to \tau\ \texttt{be}\ ?_{\sigma\to\tau}\ \texttt{in}\ \lambda x : \sigma.t[x, zx] : \sigma \to \tau']\!](\eta)\\ = [\![\Gamma \vdash \lambda x : \sigma.(\texttt{let}\ y : \tau\ \texttt{be}\ ?_\tau\ \texttt{in}\ t[x, y]) : \sigma \to \tau']\!](\eta)\end{aligned}$$

  Now if $f \in [\![\Gamma \vdash \texttt{let}\ z : \sigma \to \tau\ \texttt{be}\ ?_{\sigma\to\tau}\ \texttt{in}\ \lambda x : \sigma.t[x, zx] : \sigma \to \tau']\!](\eta)$, that is,

  $\bigcup_{a \in (\sigma\to\tau)^{\mathcal{A}}} [\![\Gamma, z : \sigma \to \tau \vdash \lambda x : \sigma.t[x, zx] : \sigma \to \tau']\!]\langle \eta, a \rangle$, then

  $$\exists a \in (\sigma \to \tau)^{\mathcal{A}} \text{ such that } \forall b \in \sigma^{\mathcal{A}}, fb \in [\![\Gamma, z : \sigma \to \tau, x : \sigma \vdash t[x, zx] : \tau']\!]\langle \eta, a, b \rangle \quad (3.6)$$

And if $f \in [\![\Gamma \vdash \lambda x:\sigma.(\texttt{let}\ y:\tau\ \texttt{be}\ ?_\tau\ \texttt{in}\ t[x,y]) : \sigma \to \tau']\!](\eta)$, then

$$\forall b \in \sigma^{\mathcal{A}}, \exists a_b \in \tau^{\mathcal{A}} \text{ such that } fb \in [\![\Gamma, x:\sigma, y:\tau \vdash t[x,y] : \tau']\!]\langle \eta, b, a_b\rangle \tag{3.7}$$

We must prove (3.6) and (3.7) are equivalent. Suppose (3.6). Now let $b \in \sigma^{\mathcal{A}}$. We can define a_b as $\texttt{App}(a,b)$. We have

$$fb \in [\![\Gamma, z:\sigma \to \tau, x:\sigma \vdash t[x,zx] : \tau']\!]\langle \eta, a, b\rangle = [\![\Gamma, x:\sigma, y:\tau \vdash t[x,y] : \tau']\!]\langle \eta, b, a_b\rangle$$

by the substitution lemma, so (3.7) holds.

Now suppose (3.7). Define $h : \sigma^{\mathcal{A}} \to (\sigma \times \tau)^{\mathcal{A}}$ to be $(b \in \sigma^{\mathcal{A}} \mapsto \langle b, a_b\rangle)$, where a_b is any witness of the existential in (3.7). Note that h is a function. We now use the factoring condition to construct a corresponding element of the Henkin model.

First define $f' : (\sigma \times \tau \to \tau')^{\mathcal{A}}$ as the unique inhabitant of $[\![\Gamma \vdash \lambda p:\sigma \times \tau.t[\pi_1 p, \pi_2 p] : \sigma \times \tau \to \tau']\!](\eta)$.

Then we have $h; \bar{f}'(b) = \bar{f}'\,\langle b, a_b\rangle$. Now, this equals $[\![\Gamma, p:\sigma \times \tau \vdash t[\pi_1 p, \pi_2 p] : \tau']\!](\eta, \langle b, a_b\rangle)$ which, by the substitution lemma, is $[\![\Gamma, x:\sigma, y:\tau \vdash t[x,y]]\!](\langle \eta, b, a_b\rangle)$. Then (3.7) can be read as $\bar{f} = h; \bar{f}'$.

By the factoring condition, there exists an element $g \in (\sigma \to \sigma \times \tau)^{\mathcal{A}}$ such that $\bar{f} = \bar{g}; \bar{f}'$. Now define $a = \texttt{App}(\texttt{App}(comp, g), p_2) \in (\sigma \to \tau)^{\mathcal{A}}$, where the elements $comp = [\![\lambda j:\sigma \to \sigma \times \tau\,.\,\lambda k:\sigma \times \tau \to \tau\,.\,\lambda x:\sigma\,.\,k(jx)]\!]$ and $p_2 = [\![\lambda p:\sigma \times \tau\,.\,\pi_2(p)]\!]$ exist by the environment model condition.

We now prove (3.6). Let $b \in \sigma^{\mathcal{A}}$. Then, by assumption (3.7), $fb \in [\![\Gamma, x:\sigma, y:\tau \vdash t[x,y] : \tau']\!]\langle \eta, b, a_b\rangle$ which, by the substitution lemma, is $[\![\Gamma, z:\sigma \to \tau, x:\sigma \vdash t[x,zx] : \tau']\!]\langle \eta, a, b\rangle$, so (3.6) holds.

∎

Remark 3.4.8 A naive interpretation of $\lambda_?$ in cpo's where ? is interpreted as $\perp$ is not sound. If we use a cpo with strict functions, then $\texttt{let}\ x:\sigma\ \texttt{in}\ 2$ would be interpreted as $\perp_{\texttt{nat}}$, and the let-weakening axiom would fail. For example, $2 \sqsubseteq \texttt{let}\ n:\texttt{nat}\ \texttt{in}\ 2$ would not be sound. On the other hand, if we use non-strict functions, then this conflicts with the fact that the variables of the theory should range over values (*i.e.* not $\perp$) so that the determined equations are extensional. For example, using the eta rule for booleans, we can deduce that $\langle 1, 2 [\!] 3\rangle = \langle 1, 2\rangle [\!] \langle 1, 3\rangle$. However, $b [\!] b'$ would be interpreted as $\perp$, so $\langle 1, 2 [\!] 3\rangle$ and $\langle 1, 2\rangle [\!] \langle 1, 3\rangle$ would be interpreted as $\langle 1, \perp_{\texttt{nat}}\rangle$ and $\perp_{\texttt{nat}\times\texttt{nat}}$ respectively, and the equation would not be sound.

More significantly, however, we have completeness of the equational theory of simply-typed underdeterminism with respect to the class of Henkin models with factoring. This implies that the system is a conservative extension of the simply-typed lambda calculus.

For the same reasons as in the case of Theorem 2.2.7, we only get a completeness result if we restrict to nonempty types.

Theorem 3.4.9 *(Completeness of equational system) Let $\langle Sg, Ax\rangle$ be a $\lambda_?$-axiom system for which all types are inhabited. If $\Gamma \vDash^{\mathcal{A}} r \sqsubseteq_\tau r'$ for all $\lambda_?$-Henkin models $\mathcal{A}$ of $\langle Sg, Ax\rangle$, then $\langle Sg, Ax\rangle \triangleright \Gamma \vdash r \sqsubseteq_\tau r'$*

Proof: We give a sketch of the proof. The idea is to construct a minimal term model for our signature of ground types, constants and equational assumptions (with no empty types).

1. Define the term interpretation $\mathcal{T}$ as in Theorem 2.2.7. That is, define an infinite context Γ_∞ with an infinite number of variables at each type. Define $\tau^{\mathcal{A}}$ as the set of $=_\tau$-equivalence classes of open (with respect to Γ_∞) determined terms of type τ, that is, $\tau^{\mathcal{A}} = \{[t] \mid \Gamma_\infty \vdash t : \tau\}$. The projection, application and constant interpretation mappings are interpreted syntactically. Recall from Theorem 2.2.7 that this gives a well-defined Henkin interpretation.

   To see that the interpretation satisfies the factoring condition, suppose $h; \bar{f} = \bar{f}'$, where $f = [x : \sigma \vdash t[x] : \tau']$ and $f' = [y : \tau \vdash t'[y] : \tau']$. This says that for all terms $u' : \tau$ there exists a term $h(u') : \sigma$ such that $t[h(u')] = t'[u']$. In particular then, for the variable $y : \tau$, there exists some term $u[y] : \sigma$ such that $t[u[y]] = t'[y]$. We can define η to be $\lambda y : \tau.u[y]$, and then $\bar{g}; \bar{f} = \bar{f}'$.

2. Prove that $[\![\Gamma \vdash r : \tau]\!](\eta) = \{[t] \mid \Gamma_\infty \vdash r[\eta/\Gamma] \sqsubseteq_\tau t\}$, where $r[\eta/\Gamma]$ has the obvious meaning. The $?_\sigma$ case uses Lemma 3.2.17.

   For abstractions:

   $$[\![\Gamma \vdash \lambda x : \sigma.r : \sigma \to \tau]\!](g) = \{[u] \mid \text{ for all } \Gamma_\infty \vdash t' : \sigma\ ,\ \Gamma_\infty \vdash r[\eta/\Gamma, t'/x] \sqsubseteq_\tau ut'\}$$

   The result follows since $\Gamma_\infty \vdash \lambda x : \sigma.r \sqsubseteq_{\sigma\to\tau} \lambda x : \sigma.t$ iff for each $\Gamma_\infty \vdash t' : \tau$, $\Gamma_\infty \vdash r[t'/x] \sqsubseteq_\tau t[t'/x]$.

   For let-terms, the interpretation is

   $$\begin{array}{rcl} [\![\Gamma \vdash \mathtt{let}\ z : \sigma\ \mathtt{be}\ r\ \mathtt{in}\ r']\!](\eta) & = & \bigcup_{a\in[\![\Gamma\vdash r:\sigma]\!](\eta)} [\![\Gamma, z : \sigma \vdash r']\!]\langle \eta, a\rangle \\ & = & \bigcup_{a\in\{\ [t]\ \mid\ r[\eta/\Gamma]\sqsubseteq t\}} \{[t'] \mid r'[\eta/\Gamma, a/z] \sqsubseteq t'\} \end{array}$$

Now t' is in the set when there exists a t such that $r[\eta/\Gamma] \sqsubseteq t$ and $r'[\eta/\Gamma, t/z] \sqsubseteq t'$. Hence $(\texttt{let}\ z : \sigma\ \texttt{be}\ r\ \texttt{in}\ r')[\eta/\Gamma] \sqsubseteq t'$.

Conversely, if $\texttt{let}\ z : \sigma\ \texttt{be}\ r[\eta/\Gamma]\ \texttt{in}\ r'[\eta/\Gamma] \sqsubseteq t'$, then by Lemma 3.3.3, there exists a t such that $r[\eta/\Gamma] \sqsubseteq t$ and $r'[\eta/\Gamma, t/z] \sqsubseteq t'$.

3. Prove that $\Gamma \models^{\mathcal{T}} r \sqsubseteq_\tau r'$ iff $\langle Sg, Ax\rangle \triangleright \Gamma \vdash r \sqsubseteq_\tau r'$.

   By step 2, $\Gamma \models^{\mathcal{T}} r \sqsubseteq_\tau r'$ is equivalent to: for all $\eta \models^{\mathcal{A}} \Gamma$, $r[\eta/\Gamma] \lesssim_\tau^{\Gamma\infty} r'[\eta/\Gamma]$. We show that this, in turn, is equivalent to $\Gamma \vdash r \sqsubseteq_\tau r'$.

   Clearly $\Gamma \vdash r \sqsubseteq_\tau r' \Rightarrow r[\eta/\Gamma] \lesssim_\tau^{\Gamma\infty} r'[\eta/\Gamma]$. To get completeness, we need to prove the converse. The crucial step uses the canonical form of refinement terms. Suppose $r[\eta/\Gamma] \lesssim_\tau^{\Gamma\infty} r'[\eta/\Gamma]$ for all $\eta \models^{\mathcal{A}} \Gamma$. In particular, then, $r \lesssim_\tau^{\Gamma\infty} r'$. We can express this using canonical forms as

   $$\texttt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \texttt{in}\ t \quad \lesssim_\tau^{\Gamma\infty} \quad \texttt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \texttt{in}\ t'$$

   so assume $r \equiv \texttt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \texttt{in}\ t$ and $r' \equiv \texttt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \texttt{in}\ t$. Now using Lemma 3.3.5 we deduce the existence of terms $\Gamma_\infty, y_1 : \tau_1, \ldots, y_m : \tau_m \vdash t_i : \sigma_i$ (for $i = 1..n$) such that $\Gamma_\infty, y_1 : \tau_1, \ldots, y_m : \tau_m \vdash t[t_1/x_1, \ldots, t_n/x_n] =_\tau t'$, and so, since by **Let Weakening**, $\Gamma_\infty \vdash r \sqsubseteq_\tau \texttt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \texttt{in}\ r$, we have

   $$\begin{aligned}\Gamma_\infty \vdash r \sqsubseteq_\tau \texttt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \texttt{in}\ r &\sqsubseteq_\tau \\ \texttt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \texttt{in}\ t[t_1/x_1, \ldots, t_n/x_n]\end{aligned}$$

   Then this is equal to $\texttt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \texttt{in}\ t'$, which equals r'. Hence $\Gamma_\infty \vdash r \sqsubseteq_\tau r'$, and since types are nonempty, we can substitute closed (determined) terms for each variable of Γ_∞ that is not in Γ, getting $\Gamma \vdash r \sqsubseteq_\tau r'$.

Hence the interpretation $\mathcal{T}$ is a model of $\langle Sg, Ax\rangle$, from which we conclude completeness. ∎

The first two steps are standard in completeness proofs; the third is particular to our calculus.

Corollary 3.4.10 *If* $\Gamma \vdash r : \sigma$ *then* $\Gamma \vdash ?_\sigma \sqsubseteq_\sigma r$.

Corollary 3.4.11 *For axiom systems with inhabited types,* $\lambda_?$ *is a conservative extension of* $\lambda^{\times\rightarrow}$.

Proof: Clearly term models for $\lambda^{\times\rightarrow}$-axiom systems satisfy the factoring condition and so $\lambda^{\times\rightarrow}$ is actually complete for $\lambda_?$-Henkin models. Since both calculi are complete for $\lambda_?$-Henkin models, and the determined equation $\Gamma \vdash t =_\tau t'$ has the same interpretation, the result follows. ∎

In fact, conservativity probably holds without the restriction to nonempty types. This could be shown using a more general notion of model, such as Kripke models [MM91].

Remark 3.4.12 An alternative (and probably equivalent) approach would be to interpret the term in context, $\Gamma \vdash r : \tau$, as a set of mappings from $\Gamma^{\mathcal{A}}$ to $\tau^{\mathcal{A}}$, rather than as a single map taking an environment in $\Gamma^{\mathcal{A}}$ to a subset of $\tau^{\mathcal{A}}$. The first approach would avoid the factoring condition.

3.5 First-order Logic of Simply-typed Refinement

Just as we presented a first-order logic over the simply-typed lambda calculus in Section 2.3, now we give a first-order logic of the equational theory of simple refinement.

This combination of logic and refinement is not the same as internalising logic into the stubs themselves; that will be carried out in Chapter 5. In this section we present an 'external' logic for reasoning about refinement.

We should regard the logic as being orthogonal to refinement. Again, we use classical first-order logic over a signature of primitive predicate symbols, and constant symbols and ground types. We use first-order $\lambda^{\times\rightarrow}$-axiom systems, as in Definition 2.3.3, though now the atomic propositions are predications of the form $F(r_1, \ldots, r_n)$ and refinements $r \sqsubseteq_\tau r'$.

In addition to the rules for refinement, we assume some (extralogical) axioms. Given that we have universal quantification, we can take these, without loss of generality, to be closed propositions.

Definition 3.5.1 *A* first-order $\lambda_?$-signature *is the same as a first-order $\lambda^{\times\rightarrow}$-signature.*

Definition 3.5.2 *Let $Sg = \langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$ be a first-order $\lambda_?$-signature. The pre-propositions over Sg are:*

$$P ::= \bot \mid F(r_1, \ldots, r_n) \mid P \supset P' \mid \forall x : \tau . P \mid r \sqsubseteq_\tau r'$$

where $F \in \mathcal{F}$, and τ and r are types and $\lambda_?$-preterms over $\langle \mathcal{G}, \mathcal{K} \rangle$ respectively.

The well-formedness judgement

$$Sg \rhd \Gamma \vdash P \ \texttt{wf}$$

is the natural extension of that given in Section 2.3. For Δ a list of propositions, we write

$$Sg \rhd \Gamma \vdash \Delta \ \texttt{wf}$$

when for each P in Δ, $Sg \rhd \Gamma \vdash P \ \texttt{wf}$.

Definition 3.5.3 *A* first-order $\lambda_?$-axiom system consists of *a first-order $\lambda_?$-signature, Sg, and a collection, Ax, of closed $\lambda_?$-propositions, well-formed in Sg, that is, $Sg \rhd \langle\rangle \vdash P \ \texttt{wf}$.*

We will continue to write equations between determined terms as $t =_\tau t'$. Formally, these are propositions of the form $t \sqsubseteq_\tau t' \wedge t' \sqsubseteq_\tau t$. Given our intuition of constructing a refinement theory on top of a $\lambda^{\times\rightarrow}$-theory, it might seem natural to restrict axioms to only involve equations. However, if we allow arbitrary propositions, even without refinement, then we will see below that we can encode refinements as propositions of the form $\forall x : \sigma \,.\, \exists y : \tau . t =_{\tau'} t'$ anyway.

Definition 3.5.4 *Let $\langle Sg, Ax\rangle$ be a first-order $\lambda_?$-axiom system. We define the first-order $\lambda_?$-theorems of $\langle Sg, Ax\rangle$ to be the judgements which can be inferred using the natural deduction rules of first-order logic in Figures 2.7 and 2.8, Chapter 2 (extended to $\lambda_?$-propositions), and Figures 3.2 to 3.6, with the convention that $\Gamma \vdash r \sqsubseteq_\tau r'$ means $\Gamma; \Delta \vdash r \sqsubseteq_\tau r'$ for all well-formed contexts, in order to include the equational theory of refinement in the logic. As before, the judgements are of the form*

$$\Gamma; \Delta \vdash P$$

where Γ is a variable context, Δ is a list of propositions well-formed in Γ, and P a proposition well-formed in Γ. The meaning is: in context Γ, if each proposition in Δ is true, then P is true.

We write $\langle Sg, Ax\rangle \rhd \Gamma; \Delta \vdash P$ to indicate that proposition in context $\Gamma \vdash P$ is a theorem of axiom system $\langle Sg, Ax\rangle$.

As in Chapter 2, the logic is complete over the same class of models as the equational theory (now the first-order $\lambda_?$-Henkin models). After giving a semantics and proving completeness below, we will be able to conclude that refinement can be encoded in the logic using just equality. This does not mean that the notion of refinement is superfluous. Rather, we can consider $r \sqsubseteq_\tau r'$ to be a useful high-level notation for some Π_2-proposition.

Figure 3.8 gives the interpretation of propositions in a first-order $\lambda_?$-Henkin interpretation. There we interpret well-formed propositions in context, $\Gamma \vdash P \ \texttt{wf}$,

$$\llbracket \Gamma \vdash \bot \rrbracket = \emptyset$$
$$\llbracket \Gamma \vdash F(r_1, \ldots, r_n) \rrbracket = \{\eta \vDash \Gamma \mid \text{ for all } a_i \text{ in } \llbracket \Gamma \vdash r_i \rrbracket(\eta) \,.\, \langle a_1, \ldots, a_n \rangle \in F^{\mathcal{A}}\}$$
$$\llbracket \Gamma \vdash P \supset P' \rrbracket = \{\eta \vDash \Gamma \mid \eta \notin \llbracket \Gamma \vdash P \rrbracket \text{ or } \eta \in \llbracket \Gamma \vdash P' \rrbracket\}$$
$$\llbracket \Gamma \vdash \forall x : \sigma . P \rrbracket = \{\eta \vDash \Gamma \mid \text{ for all } a \text{ in } \sigma^{\mathcal{A}} \,.\, \langle \eta, a \rangle \in \llbracket \Gamma, x : \sigma \vdash P \rrbracket\}$$
$$\llbracket \Gamma \vdash r \sqsubseteq_\sigma r' \rrbracket = \{\eta \vDash \Gamma \mid \llbracket \Gamma \vdash r : \sigma \rrbracket(\eta) \supseteq \llbracket \Gamma \vdash r' : \sigma \rrbracket(\eta)\}$$

Figure 3.8: Interpretation of Well-formed Propositions

as the set, $\llbracket \Gamma \vdash P \rrbracket^{\mathcal{A}}$ of environments, $\eta \vDash^{\mathcal{A}} \Gamma$, in which P holds, though we usually drop the superscript $\mathcal{A}$. We write $\Gamma \vDash^{\mathcal{A},\eta} P$ to mean $\eta \in \llbracket \Gamma \vdash P \rrbracket^{\mathcal{A}}$. If $\mathcal{A}$ is a Henkin interpretation, we say that $\Gamma; \Delta \vDash^{\mathcal{A},\eta} P$, if for all $\eta \vDash^{\mathcal{A}} \Gamma$, if $\Gamma \vDash^{\mathcal{A},\eta} A$ for each A in Δ, then $\Gamma \vDash^{\mathcal{A},\eta} P$. We write $\Gamma; \Delta \vDash^{\mathcal{A}} P$ when $\Gamma; \Delta \vDash^{\mathcal{A},\eta} P$ for each $\eta \vDash^{\mathcal{A}} \Gamma$.

Completeness is with respect to the class of Henkin models (of the axiom system) with the factoring condition.

We extend the definition of Henkin theory to account for refinement. The idea is that a refinement is a form of existential for which we add a witness. If $\texttt{let}\ x : \tau\ \texttt{in}\ t \sqsubseteq_\sigma \texttt{let}\ x' : \tau'\ \texttt{in}\ t'$, then for all $x' : \tau$, there must exist an $x : \tau$ such that $t =_\sigma t'$. In fact, for completeness we make the stronger assumption that we can make a uniform choice of x for each x' given by a term $t'' : \tau' \to \tau$.

Definition 3.5.5 *A* first-order $\lambda_?$-Henkin theory*, T, over axiom system $\langle Sg, Ax \rangle$, in context, Γ, is a collection of propositions closed under derivation from $\langle Sg, Ax \rangle$, such that for every proposition $\exists x : \tau \,.\, P$, there is a term $\Gamma \vdash t : \tau$ such that $P[t/x]$ is in T, and for every proposition $\texttt{let}\ x : \tau\ \texttt{in}\ t \sqsubseteq_\sigma \texttt{let}\ x' : \tau'\ \texttt{in}\ t'$ there exists a term $\Gamma \vdash t'' : \tau' \to \tau$ such that $\forall x' : \tau' . t[t'' x'/x] =_\sigma t'$ is in T.*

One subtle point is that we must be sure that adding witnesses for refinements preserves consistency. This is because the refinement $\texttt{let}\ x : \tau\ \texttt{in}\ t \sqsubseteq_\sigma \texttt{let}\ x' : \tau'\ \texttt{in}\ t'$ is admissibly equivalent to $\exists f : \tau' \to \tau . \forall x' : \tau' . t[f x'/x] =_\sigma t'$

Theorem 3.5.6 *(Soundness and Completeness of logical system) Let $\langle Sg, Ax \rangle$ be a first-order $\lambda^{\times \to}$-axiom system. Then, $\langle Sg, Ax \rangle \rhd \Gamma; \Delta \vdash P$ iff $\Gamma; \Delta \vDash^{\mathcal{A}} P$ for all Henkin models $\mathcal{A}$ of $\langle Sg, Ax \rangle$.*

Proof: We modify the proof of Theorem 2.4.5. As there, soundness is straightforward to prove, and completeness is shown using a term model. We show that any consistent axiom system is satisfiable. For axiom system $\langle Sg, Ax\rangle$, we want to show that $\langle Sg, Ax\rangle \triangleright \Gamma; \Delta \vdash P$ iff $\Gamma; \Delta \models^{\mathcal{A}} P$ in all $\lambda_?$-Henkin models, $\mathcal{A}$, of $\langle Sg, Ax\rangle$. (As in Theorem 2.4.5, we do not assume that types are nonempty.)

1. Given $\langle Sg, Ax\rangle \not\triangleright \Gamma; \Delta \vdash P$ we want to find a Henkin model $\mathcal{A}$ of $\langle Sg, Ax\rangle$ and Γ-environment, η, in $\mathcal{A}$ such that $\Gamma \models^{\mathcal{A},\eta} A$ for each A in $\Delta, \neg P$.

2. Construct a maximal consistent Henkin theory Δ_∞ and infinite context Γ_∞ such that $Ax \cup \Delta \cup \{\neg P\} \subseteq \Delta_\infty$, $\Gamma \subseteq \Gamma_\infty$, and Δ_∞ is a $\lambda_?$-Henkin theory in Γ_∞.

   As in the proof of Theorem 2.4.5, we construct a first-order $\lambda_?$-Henkin theory which extends Γ and $Ax \cup \Delta \cup \{\neg P\}$, by taking the Henkin closure of $\Gamma; Ax \cup \Delta \cup \{\neg P\}$ and the limit of sets of propositions, Δ', which are well-formed consistent extensions.

3. Construct the term interpretation $\mathcal{A}$ where $\tau^{\mathcal{A}}$ is the set of equivalence classes of determined terms, where $[t] = [t']$ iff $\Gamma_\infty; \Delta_\infty \vdash t =_\sigma t'$ and show that $[\![\Gamma \vdash r : \tau]\!]^{\mathcal{A}}(\eta) = \{[t] \mid \Gamma_\infty; \Delta_\infty \vdash r[\eta/\Gamma] \sqsubseteq_\tau t\}$. This requires the generalisation of Lemma 3.3.2 to logical contexts. In contrast with Remark 3.3.4, this holds because of the construction of the Henkin theory Δ_∞.

   $\mathcal{A}$ is a Henkin interpretation with the factoring condition.

4. For all $\eta' \models \Gamma'$, prove that $\Gamma' \models^{\mathcal{A},\eta'} P$ iff $P[\eta'/\Gamma'] \in \Delta_\infty$. The crucial cases are $\exists x : \tau \,.\, P$ and $r \sqsubseteq_\tau r'$, which go through by virtue of Δ_∞ being a $\lambda_?$-Henkin theory. The $r \sqsubseteq_\tau r'$ case is proven as in Theorem 3.4.9, with the observation that the appropriate generalisation of Lemma 3.3.2 holds.

5. Hence $\mathcal{A}$ is a $\lambda_?$-Henkin model of $\langle Sg, Ax\rangle$, and for $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$, defining η to be $\langle [x_1], \ldots, [x_n]\rangle$, we have $\Gamma \models^{\mathcal{A},\eta} A$, for each A in $\Delta, \neg P$. ■

The use of canonical forms in the proof of Theorem 3.4.9 suggests that we can translate refinement into first-order logic over $\lambda^{\times\to}$, that is, just using equality of determined terms. If we let r° denote the canonical form of open term r, then if $r^\circ = \mathtt{let}\ x_1 : \sigma_1, \ldots, x_n : \sigma_n\ \mathtt{in}\ t$, and $r'^\circ = \mathtt{let}\ y_1 : \tau_1, \ldots, y_m : \tau_m\ \mathtt{in}\ t'$, define $(r \sqsubseteq_v r')^\circ$ to be $\forall y_1 : \tau_1, \ldots, y_m : \tau_m \,.\, \exists x_1 : \sigma_1, \ldots, x_n : \sigma_n . t =_v t'$.

We use the completeness results (Theorems 3.4.9 and 3.5.6) to infer that the logical system is a conservative extension of the equational system (which, in turn, is a conservative extension of the simply-typed lambda calculus). In fact, we have

Corollary 3.5.7 *Let $\langle Sg, Ax\rangle$ be a first-order $\lambda^{\times\to}$-axiom system for which all types are inhabited. Then $\langle Sg, Ax\rangle \rhd \Gamma \vdash r \sqsubseteq_\tau r'$ iff $\langle Sg, Ax\rangle \rhd \Gamma; \langle\rangle \vdash r \sqsubseteq_\tau r'$ iff $\langle Sg, Ax\rangle \rhd \Gamma; \langle\rangle \vdash (r \sqsubseteq_\tau r')^\circ$.*

Proof: Both systems are complete with respect to Henkin models (with the factoring condition), and the statements have the same interpretation. ∎

Remark 3.5.8 The above corollary suggests an alternative proof of completeness. Since $(r \sqsubseteq r')^\circ$ is a proposition in first-order logic, we can prove directly that $r \sqsubseteq r'$ (in $\lambda_?$) iff $(r \sqsubseteq r')^\circ$ (in FOL), and then use the completeness of first-order logic over the λ-calculus (Theorem 2.4.5) to deduce the completeness of first-order $\lambda_?$.

3.6 Conclusions

Though the language we have presented in this chapter is very simple, we believe that it captures an important part of program refinement. In combination with the calculus of specifications in the next chapter, this gives a calculus which is conceptually simple, but expressive enough to study program development.

Chapter 4

Refinement Types

In this chapter we develop a theory of refinement types. This is intended to give us a calculus, $\lambda_{(:)}$, for refining specifications, and for proving that programs satisfy specifications. This is a necessary part of a theory of program refinement. We will first justify our view of specification, and then outline what constructs this requires in a calculus and the associated judgement forms. We then give the calculus and illustrate its use with an example verification. In the final section we give a semantics based on Henkin models and prove the system to be sound and complete.[1]

4.1 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. Recall that 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 [Luo91] and [NPS90] for example. An integer square root function might be specified as the existential type $\exists f : \texttt{nat} \to \texttt{nat} \;.\; \Pi n : \texttt{nat} \;.\; (f\ n)^2 = n \;\vee\; (f\ n)^2 + 1 = n$, 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. A further problem is that it is not easy to combine nontermination with type theories.

[1] Earlier versions of some of the work in this chapter were presented in [Den97a] and [Den98].

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 $\forall n : \texttt{nat} \ . \ (f\ n)^2 = n \ \vee \ (f\ n)^2 + 1 = n$, where f is a free variable of type $\texttt{nat} \to \texttt{nat}$. This is the approach traditionally taken by the program refinement community. Morgan [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 it is better to put more structure on specifications.

In this thesis, 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 encode propositions as types but, rather, use them directly.

We describe a verification calculus based on the simply-typed lambda calculus with products ($\lambda^{\times\to}$) and some ground types such as `nat` and `bool`. The satisfaction of specifications by programs is axiomatised as a generalised typing relation, in a sense which we make precise below. 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.

Refinement types are constructed as combinations of types and propositions from the program logic. Types themselves are trivial refinement types, and we can restrict a refinement type to those elements for which some proposition holds. This is similar to subset types [NPS90], though not quite the same since we maintain a distinction from the types themselves. Also, it is convenient to form 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 this chapter is soundness and completeness with respect to the resulting class of models.

In Section 4.2 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 4.3. In Section 4.4 we return to the example. Section 4.5 gives the semantics and proofs of soundness and completeness. Finally, we give some conclusions in Section 4.6.

4.2 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 the constant, `natiter`, for iteration over the naturals, where `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

$$\texttt{div2} = \lambda n : \texttt{nat} \,.\, \pi_1(\texttt{div2}'\ n) : \texttt{nat} \to \texttt{nat}$$

where this uses the auxiliary function

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

Now this can be specified as

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

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

$$\texttt{div2 sat div2\_spec}$$

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 the rule that t `sat` $(x : \tau)P$ when

$t : \tau$ and $P[t/x]$. This example reduces then to proving

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

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

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

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

The dependent function $\Pi_{x:\sigma|P}(y : \tau)Q$ 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 $x : \texttt{nat} \,|\, \top$, then we can write our specification more compactly as

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

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

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

where we informally understand the sequent $n : \texttt{nat} \vdash t\ \texttt{sat}\ \phi$ to mean for all closed $t' : \texttt{nat}$ (or equivalently, for all numerals), $t[t'/n]\ \texttt{sat}\ \phi[t'/n]$. 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 `div2'` has specification

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

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 4.4.

We do not throw away the original rule that t satisfies $(x : \tau)P$ when $t : \tau$ and $P[t/x]$, however.

A significant benefit in writing specifications 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. 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.

In this small example, the disadvantage of using propositions as specification is not so obvious. However, structure is essential for large specifications so we build it into the theory. Moreover, it is natural to incorporate equality in the definition of specification, rather than express it as a separate proposition. McKinna was led to the same conclusion in [McK92].

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 inducing an equality on terms. This is a *partial equivalence relation* (per) on terms at the underlying type. A per is a symmetric and transitive relation on some set, or equivalently, an equivalence relation on a particular subset. The partiality is because not all terms (of the corresponding type) need satisfy a specification.

For example, the specification $\Pi_{l:\mathtt{nonemptylist}}(n : \mathtt{nat})\mathtt{Min}(n, l)$ where the proposition $\mathtt{Min}(n, l)$ says that n is the minimum element in list l, is a refinement type over type $\mathtt{list} \to \mathtt{nat}$. We want to regard functions $f, f' : \mathtt{list} \to \mathtt{nat}$ 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 equality at a specification, $=_{\Pi_{x:\sigma|P}(y:\tau)Q}$, where $t =_{\Pi_{x:\sigma|P}(y:\tau)Q} t'$ means

$$\forall x, x' : \sigma.(x =_\sigma x') \wedge P[x] \wedge P[x'] \supset (tx =_\tau t'x') \wedge Q[tx] \wedge Q[t'x']$$

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 $(n : \texttt{nat})\top$ in place of $\texttt{nat}$, and $\Sigma_{n:(n:\texttt{nat})\top}(b : \texttt{bool})\top$ for $\texttt{nat} \times \texttt{bool}$, where $\Sigma_{x:\phi|P}\psi$ abbreviates $\Sigma_{x:(x:\phi)P}\psi$, and similarly for Π. At this point, we must cease to regard $\Pi_{x:\sigma|P}(y : \tau)Q$ as we did before, since the above convention means that the equality is only with respect to arguments in $(x : \sigma)P$, rather than σ as for $(f : \sigma \to \tau)\ \forall x : \sigma\ .\ P \supset Q[fx/y]$.

We believe it is misleading to regard specifications a rich form of types, though, and refer to the specifications of this idealised specification language as *refinement types*. We regard types, rather, as being part of the programming language, and specifications as being constructed at a level above this.

In fact, we will take the denotation of refinement types to be a per. A specification therefore, is a refinement type, and denotes

- a set, together with
- a per over the set

We take a program in this calculus to denote

- an equivalence class of a per ϕ

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

So refinement types induce a per on the set of terms of the underlying type. The converse is not true, that is, not all pers of terms correspond to refinement types. For example, the per R on naturals, $n\ R\ n' \iff$ "both n and n' are even, or both are odd".

We use a notation for the equivalence classes of pers, by allowing refinement types on the variables in abstractions. For example, $\lambda n : \texttt{even}.n$ denotes a class in the per denoted by $\texttt{even} \to \texttt{nat}$ (functions in the corresponding set are equal if they give the same results for even arguments) but not $\texttt{nat} \to \texttt{nat}$, and $\lambda n : \texttt{nat}.n$ denotes a class in both $\texttt{even} \to \texttt{nat}$ and $\texttt{nat} \to \texttt{nat}$. The meaning of the equality $t =_\phi t'$ is that t and t' denote sets in the same equivalence class of per ϕ.

For refinement types ϕ and ϕ' over the same underlying type, we want to consider refinements $\phi \sqsubseteq_\tau \phi'$, to be thought of semantically as per inclusion (*i.e.* equality at ϕ' implies equality at ϕ). We use the square $\sqsubseteq_\tau$ symbol to indicate an information ordering — the refinement of specifications over type τ. Note that this convention for refinement is the opposite direction to the usual subtyping relation.

4.3 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.

4.3.1 Syntax

The idea is that we construct a theory of refinement types on top of an underlying $\lambda^{\times\rightarrow}$ 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 4.3.4 below.

We construct the theory of refinement types from the same basic data as the first-order theories of lambda calculus. A $\lambda_{(:)}$-*signature* is the same as a first-order $\lambda^{\times\rightarrow}$-signature. We repeat the definition.

Definition 4.3.1 *A* $\lambda_{(:)}$-signature, $Sg = \langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$, *consists of:*

- *a collection, $\mathcal{G}$, of ground types (ranged over by γ)*
- *a collection, $\mathcal{K}$, of constant symbols (ranged over by k), each of which has an arity n and sort $\tau_1, \ldots, \tau_n \rightarrow \tau$, which we write as $k : \tau_1, \ldots, \tau_n \rightarrow \tau$.*
- *a collection, $\mathcal{F}$, of predicate symbols (ranged over by F) each of which has an arity n and sort $\tau_1, \ldots, \tau_n$, which we write as $F : \mathtt{Pred}\,(\tau_1, \ldots, \tau_n)$.*

Although we do not have arbitrary refinement types as primitive in a signature, we get much the same expressiveness using predicate symbols. For example, with the primitive type `nat`, we could have predicate `Even` : `Pred` (`nat`), and write `even` for the refinement type $(n : \mathtt{nat})\,\mathtt{Even}(n)$.

Definition 4.3.2 *Let $Sg = \langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$ be a $\lambda_{(:)}$-signature. The* pre-expressions *over Sg are given by a mutual recursion over (pre-) refinement types, terms and propositions:*

$$
\begin{array}{rcl}
\phi & ::= & \mathbf{1} \mid \gamma \mid \Sigma_{x:\phi}\psi \mid \Pi_{x:\phi}\psi \mid (x:\phi)P \\
t & ::= & x \mid k(t_1, \ldots, t_n) \mid * \mid \langle t, t' \rangle \mid \lambda x : \phi.t \mid \pi_1(t) \mid \pi_2(t) \mid tt' \\
P & ::= & \bot \mid P \supset P' \mid \forall x : \phi.P \mid F(t_1, \ldots, t_n) \mid t =_\phi t' \mid \phi \sqsubseteq_\tau \phi'
\end{array}
$$

The pre-contexts *are:*

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

We say that $\Gamma \vdash U$ is a pre-expression in context *when Γ is a pre-context, U is a pre-expression, and $FV(U) \subseteq \Gamma$.*

As for the previous calculi, we assume a countably infinite set of variables. We adopt the usual abbreviations of $\phi \times \psi$ for $\Sigma_{x:\phi}\psi$ and $\phi \to \psi$ for $\Pi_{x:\phi}\psi$, when $x \notin FV(\psi)$, and use ϕ, ψ and χ as metavariables for refinement types. We sometimes abbreviate the assumption $x : (x : \phi)P$, on abstractions or in contexts, as $x : \phi \,|\, P$.

Conceptually, it is simpler not to distinguish types and refinement types as syntactic categories. In an informal sense which we will later make formal, 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, and refer to the type *underlying* a refinement type.

Here we extend the variable convention of p. 33 so that by writing $x : \phi$, where ϕ is over τ, we assume that x is drawn from the set of variables for type τ.

There is a term $*$ of unit type $\mathbf{1}$. After we introduce the equality judgement we will see that $*$ is the unique term at $\mathbf{1}$ up to equality. 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 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, $\lambda n : \mathtt{even}.n$ specifies total terms of type $\mathtt{nat} \to \mathtt{nat}$ 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 $\mathtt{even} \to \mathtt{even}$ is a refinement of the type $\mathtt{nat} \to \mathtt{nat}$, the term $\lambda n : \mathtt{even}.n$ does not itself have type $\mathtt{nat} \to \mathtt{nat}$. 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 a typed first-order predicate logic of equalities and typed refinements. In practice, we will almost always omit the subscript from $\phi \sqsubseteq_\tau \phi'$. We use classical typed first-order logic as an example of a simple expressive logic.

Remark 4.3.3 Our choice of first-order classical logic is only significant insofar as it is an example of what we might call an *extensional* logic. In type theory, a distinction is often made between extensional and intensional equality [NPS90]. Terms are extensionally equal if they have the same input-output behaviour, as given by some proposition (hence, also called ‘propositional equality’), whereas intensional equality is a definitional equality which is generally decidable.

In this sense, intensional means ‘relating to syntactic form’, but there is a more general sense in which it is with respect to richer properties than input-output behaviour, such as time-complexity, and it is this kind of predicate we want to contrast with those used here.

Since we allow the propositional equality in properties, the system only makes sense for extensional predicates. We make essential use of the fact that for all terms t, t' and propositions P, if t is extensionally equal to t', then $P[t/x]$ holds if and only if $P[t'/x]$ does.

Since the converse clearly holds, we can express this as saying that we require Leibniz (satisfaction of the same predicates) and observational (same input-output behaviour) 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.

4.3.2 Judgements

The main judgements of the calculus have the forms

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

$$\Gamma \vdash P$$

where the atomic propositions include $\Gamma \vdash t =_\phi t'$ and $\Gamma \vdash \phi \sqsubseteq_\tau \phi'$. Equality and refinement are not separate judgement classes from the other propositions. We will write $\Gamma \vdash \phi = \phi'$ for the mutual refinement $\Gamma \vdash \phi \sqsubseteq \phi'$ and $\Gamma \vdash \phi' \sqsubseteq \phi$.

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. 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. Although the extra information that ϕ is over τ is not required for the well-formedness of ϕ itself, it is used to check the well-formedness of refinements.

We use g as a metavariable for 'syntactic 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.

Remark 4.3.4 We should not think of $\texttt{Ref}\ (\tau)$ as a power type of τ, since the refinement $\tau \sqsubseteq \phi$ is not the same as $\phi : \texttt{Ref}\ (\tau)$. This is because the attribution of refinement types is not contravariant at function types. For example, since $\texttt{nat} \to \texttt{nat} \not\sqsubseteq \texttt{even} \to \texttt{nat}$ we cannot regard $\texttt{even} \to \texttt{nat}$ as a subtype of $\texttt{nat} \to \texttt{nat}$ even though it is a refinement type. The difference between power types and refinement types can be seen by considering encodings in higher-order logic. The power types of τ may be encoded as $\tau \to \texttt{Prop}$, whereas refinement types of τ would correspond to $\tau \to \tau \to \texttt{Prop}$, since specifications comprise a (partial) equality relation. The $\lambda_{(:)}$-calculus can be seen as a convenient formalism for such relations.

4.3.3 $\lambda_{(:)}$-Axiom Systems

A $\lambda_{(:)}$-axiom system consists of a collection of first-order axioms over some signature. There is a crucial difference from Definition 2.3.3, however. Since refinement types can appear in terms, well-formedness involves logical reasoning and, in particular, can depend on the axioms. If axioms themselves are to be well-formed, therefore, we must have some dependencies among the axioms. The problem is that with axiom schemas it is possible for one instance to be needed to prove the well-formedness of another. For example, an induction principle over the naturals is schematic in some proposition $P[n]$ for $n : \texttt{nat}$, but the well-formedness of P might itself require induction.

This problem is common to all logics in which well-formedness depends on provability. Rather than introduce a hierarchy of axioms, the solution we adopt here is to drop the requirement that axioms are well-formed, and instead only check for this when they are used in a proof.

It is natural to give axioms in a deliverables style [McK92]. If k is a unary constant with sort $\sigma \to \tau$, then we give axioms of the form "if the argument satisfies some specification then the result satisfies some specification". We write $k : \phi \to \psi$ to mean that if t has refinement type ϕ then $k(t)$ has refinement type ψ. In general, we allow the refinement types to be open in some context. (Note that this is not a judgement form (see below) but an axiom which we will use in side-conditions.) Thus we have the following definition.

Definition 4.3.5 *A* $\lambda_{(:)}$-axiom system, *$\langle Sg, Ax\rangle$, consists of a $\lambda_{(:)}$-signature, Sg, and a collection of axioms Ax formed from pre-contexts and pre-expressions over Sg. Axioms are of two forms:*

- *propositions in context,* $\Gamma \vdash P$
- *axioms for constants,* $\Gamma \vdash k : \phi_1, \ldots, \phi_n \to \psi$.

We do not put any well-formedness requirements on axioms, but check for well-formedness at the point of using the axiom in a proof. A similar convention is adopted by Pitts for dependently-typed algebraic theories [Pit95].

Hence, although we intend that when k has sort $\tau_1, \ldots, \tau_n \to \tau$, that we have $\Gamma \vdash \phi_i : \texttt{Ref}\ (\tau_i)$ $(i = 1, \ldots, n)$ and $\Gamma \vdash \psi : \texttt{Ref}\ (\tau)$ we do not enforce it in the axiom system. We could, for example, have required for axiom $\Gamma \vdash P$ that $FV(P) \subseteq \Gamma$ but this will follow automatically from the well-formedness check when the axiom is used, and similarly for constant axioms.

Note that the sorting $k : \tau \to \tau'$ and axiom $k : \phi \to \psi$ do not say that the 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 $k(t) : \psi$. We do, however, consider sortings as axioms.

Allowing arbitrary propositions as axioms subsumes the definition of $\lambda^{\times\to}$-axiom systems, since we can include equations in context between determined terms, $\Gamma \vdash t =_\phi t'$.

Remark 4.3.6 It is not clear that it is necessary to allow arbitrary propositions as axioms in Definition 4.3.5. We will show later (in Section 4.3.5) that induction schemas follow from the axiomatisation of the corresponding constant for recursion. This suggests that we may only need propositional axioms in order to axiomatise the predicate symbols. It seems that these axioms can always be given in the form $\Gamma \vdash F(t)$ or $\Gamma \vdash \neg F(t)$.

The axioms for constants could be given in the form $\Gamma, x_1 : \phi_1, \ldots, x_n : \phi_n \vdash k(x_1, \ldots, x_n) : \psi$, which is actually equivalent to the, perhaps more natural, general form $\Gamma \vdash k(t_1, \ldots, t_n) : \psi$.

4.3.4 Rules of the Calculus

Definition 4.3.7 *Let $\langle Sg, Ax\rangle$ be a $\lambda_{(:)}$-axiom system. We define the theorems of $\langle Sg, Ax\rangle$ to be the judgements which can be inferred using the rules of Figures 4.1 to 4.10. We write $\langle Sg, Ax\rangle \triangleright J$ to indicate that judgement J is a theorem of axiom system $\langle Sg, Ax\rangle$.*

Note that we consider judgements of well-formedness to also be theorems, since they involve logical reasoning.

The rules of the calculus are listed in Figures 4.1-4.10. 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.

In Figure 4.1 we give the rules for generating theorems from a $\lambda_{(:)}$-axiom system. It is here that we check that an axiom is well-formed before it can be used. The substitution rule is for an arbitrary basic judgement, B. The rule is quite simple because we have explicit congruence rules for the equalities. By encoding B as a proposition, and using the rules for implication and universal quantification we can actually derive a more general rule:

$$\frac{\Gamma, x : \phi, \Gamma' \vdash B \quad \Gamma \vdash t : \phi}{\Gamma, \Gamma'[t/x] \vdash B[t/x]}$$

Next 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. Figures 4.2 and 4.3 give the well-formedness rules for contexts and refinement types respectively.

The well-formedness rules for refinement types essentially involve stripping off the logic while checking that everything fits together correctly. 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. Similar conditions are made for the base cases of the other judgement classes.

It is straightforward to formulate well-formedness rules for propositions. These are given in Figure 4.4. In proving the well-formedness of the implication $P \supset P'$, we can assume the truth of P for proving the well-formedness of P'. For the equality $t =_\phi t'$ to be well-formed we require that t and t' are both well-formed and have refinement types over the same type as ϕ. We do not require that t and t' have refinement type ϕ. This allows us to express the refinement typing $t : \phi$ as the proposition $t =_\phi t'$. The appeal to refinement typing is why well-formedness

Axioms

$$\frac{\Gamma \vdash P \ \texttt{wf}}{\Gamma \vdash P} \ (\Gamma \vdash P \ \in \ Ax)$$

Weakening

$$\frac{\Gamma_1, \Gamma_2 \vdash B \quad \Gamma_1 \vdash \phi \ \texttt{wf}}{\Gamma_1, x : \phi, \Gamma_2 \vdash B}$$

Permutation

$$\frac{\Gamma_1, x_1 : \phi_1, \Gamma_2, x_2 : \phi_2, \Gamma_3 \vdash B \quad \Gamma_1 \vdash \phi_2 \ \texttt{wf}}{\Gamma_1, x_2 : \phi_2, \Gamma_2, x_1 : \phi_1, \Gamma_3 \vdash B} \ (x_1 \notin \Gamma_2, x_2 : \phi_2)$$

Substitution

$$\frac{\Gamma, x : \phi \vdash B \quad \Gamma \vdash t : \phi}{\Gamma \vdash B[t/x]}$$

Figure 4.1: Theorems Generated from a $\lambda_{(:)}$-Axiom System $\langle Sg, Ax \rangle$

Empty Context

$$\vdash \langle \rangle \ \texttt{wf}$$

Variable Assumption

$$\frac{\Gamma \vdash \phi \ \texttt{wf}}{\vdash \Gamma, x : \phi \ \texttt{wf}} \ (x \notin \Gamma)$$

Propositional Assumption

$$\frac{\Gamma \vdash P \ \texttt{wf}}{\vdash \Gamma, P \ \texttt{wf}}$$

Figure 4.2: Well-formedness of Contexts

Unit

$$\frac{\vdash \Gamma \ \texttt{wf}}{\Gamma \vdash \mathbf{1} : \ \texttt{Ref}\ (\mathbf{1})}$$

Product

$$\frac{\Gamma \vdash \phi : \ \texttt{Ref}\ (\sigma) \quad \Gamma, x : \phi \vdash \psi : \ \texttt{Ref}\ (\tau)}{\Gamma \vdash \Sigma_{x:\phi}\psi : \ \texttt{Ref}\ (\sigma \times \tau)}$$

Function

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

Refinement Types

$$\frac{\Gamma \vdash \phi : \ \texttt{Ref}\ (\tau) \quad \Gamma, x : \phi \vdash P \ \texttt{wf}}{\Gamma \vdash (x : \phi)P : \ \texttt{Ref}\ (\tau)}$$

Ground Types

$$\frac{\vdash \Gamma \ \texttt{wf}}{\Gamma \vdash \gamma : \ \texttt{Ref}\ (\gamma)}\ (\gamma \in \mathcal{G})$$

Figure 4.3: Well-formedness of Refinement Types

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.

In Figures 4.5 and 4.6 we give the refinement typing rules, which also serve as well-formedness rules for the terms. Where this differs from the simply-typed lambda calculus is in the logical reasoning which pervades the rules. This is evident in the **Constants** rule, where well-formedness uses a logical axiom. If constant symbol k has axiom $\Gamma' \vdash k : \phi_1, \ldots, \phi_n \to \psi \ \in \ Ax$, then we infer a refinement typing in the more general context Γ, where $\Gamma' \subseteq \Gamma$. If $\Gamma \vdash t_i : \phi_i$ for $i = 1, n$, then we infer $\Gamma \vdash k(t_1, \ldots, t_n) : \psi$. The reason for the more general context is so that from an axiom like $n : \mathtt{nat} \vdash k : \phi[n] \to \psi[n]$, we can infer that $x : \phi[2] \vdash k(x) : \psi[2]$, by using the context $n : \mathtt{nat}, n = 2$. (In fact, in the case of refinement typing, we can derive the more general rule for constants from a simpler one, but this is *not* the case for the rule of **Constant Equations** below so we keep both rules in the same form.)

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 rule for abstractions is

$$\frac{\Gamma, x : \phi \vdash t : \psi}{\Gamma \vdash \lambda x : \phi . t : \Pi_{x:\phi}\psi}$$

For $\lambda x : \phi . t$ to be well-formed in context Γ, it is sufficient, but not necessary that t is well-formed in context $\Gamma, x : \phi$.

This is because our notion of well-formedness is 'having some refinement type'. For example, we have $\lambda n : \mathtt{nat}.(\lambda m : \mathtt{even}.m)n : \mathtt{even} \to \mathtt{nat}$, although the body is not well-formed in the context $n : \mathtt{nat}$.

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

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

The corresponding elimination rule — concluding that $t : \phi$ from $t : (x : \phi)P$ — 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

Falsehood

$$\frac{\vdash \Gamma \ \ \texttt{wf}}{\Gamma \vdash \bot \ \ \texttt{wf}}$$

Implication

$$\frac{\Gamma \vdash P \ \ \texttt{wf} \quad \Gamma, P \vdash P' \ \ \texttt{wf}}{\Gamma \vdash P \supset P' \ \ \texttt{wf}}$$

Universal Quantification

$$\frac{\Gamma, x : \phi \vdash P \ \ \texttt{wf}}{\Gamma \vdash \forall x : \phi . P \ \ \texttt{wf}}$$

Predication

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \phi_1 \ \cdots \ \Gamma \vdash t_n : \phi_n \\ \Gamma \vdash \phi_1 : \ \texttt{Ref}\ (\tau_1) \ \cdots \ \Gamma \vdash \phi_n : \ \texttt{Ref}\ (\tau_n)\end{array}}{\Gamma \vdash F(t_1, \ldots, t_n) \ \ \texttt{wf}} \qquad (F : \ \texttt{Pred}\ (\tau_1, \ldots, \tau_n) \in \mathcal{F})$$

Equality

$$\frac{\Gamma \vdash t : \psi \qquad \Gamma \vdash t' : \psi' \quad \Gamma \vdash \psi, \psi', \phi : \ \texttt{Ref}\ (\tau)}{\Gamma \vdash t =_{\phi} t' \ \ \texttt{wf}}$$

Refinement

$$\frac{\Gamma \vdash \phi : \ \texttt{Ref}\ (\tau) \quad \Gamma \vdash \phi' : \ \texttt{Ref}\ (\tau)}{\Gamma \vdash \phi \sqsubseteq_{\tau} \phi' \ \ \texttt{wf}}$$

Figure 4.4: Well-formedness of Propositions

Variables

$$\frac{\vdash \Gamma, x:\phi, \Gamma' \ \texttt{wf}}{\Gamma, x:\phi, \Gamma' \vdash x:\phi}$$

Constants

$$\frac{\Gamma \vdash \phi_1 : \texttt{Ref}\,(\tau_1) \ \cdots \ \Gamma \vdash \phi_n : \texttt{Ref}\,(\tau_n) \quad \Gamma \vdash \psi : \texttt{Ref}\,(\tau) \quad \Gamma \vdash t_1 : \phi_1 \ \cdots \ \Gamma \vdash t_n : \phi_n}{\Gamma \vdash k(t_1, \ldots, t_n) : \psi} \qquad \begin{cases} \Gamma' \vdash k : \phi_1, \ldots, \phi_n \to \psi \ \in \ Ax\,; \\ k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K}; \\ \Gamma' \subseteq \Gamma \end{cases}$$

Unit

$$\frac{\vdash \Gamma \ \texttt{wf}}{\Gamma \vdash * : \mathbf{1}}$$

Product Terms

$$\frac{\Gamma \vdash t : \phi \quad \Gamma \vdash t' : \psi[t/x] \quad \Gamma, x:\phi \vdash \psi \ \texttt{wf}}{\Gamma \vdash \langle t, t' \rangle : \Sigma_{x:\phi}\psi}$$

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

Function Terms

$$\frac{\Gamma, x:\phi \vdash t : \psi}{\Gamma \vdash \lambda x:\phi.t : \Pi_{x:\phi}\psi}$$

$$\frac{\Gamma \vdash t : \Pi_{x:\phi}\psi \quad \Gamma \vdash t' : \phi}{\Gamma \vdash tt' : \psi[t'/x]}$$

Figure 4.5: Refinement Typings

Refinement Type Introduction

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

Equality

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

Weakening

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

Figure 4.6: Refinement Typings cont.

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 equation for abstractions is

$$\frac{\Gamma, x : \phi \vdash P \ \mathtt{wf} \quad \Gamma, x : \phi, P \vdash t =_\psi t'}{\Gamma \vdash \lambda x : (x : \phi)P.t =_{\Pi_{x:(x:\phi)P}\psi} \lambda x : \phi.t'}$$

which lets us prove that $\lambda n : \mathtt{even}.n =_{\mathtt{even} \to \mathtt{even}} \lambda n : \mathtt{nat}.n$, and so we can infer that $\lambda n : \mathtt{nat}.n : \mathtt{even} \to \mathtt{even}$. The more general inference for $\phi' \sqsubseteq \phi$, that $\lambda x : \phi.t =_{\Pi_{x:\phi}\psi} \lambda x : \phi'.t'$, is not actually sound. For example, the term $\lambda f : \mathtt{even} \to \mathtt{nat}.f$ does not have the refinement type $(\mathtt{nat} \to \mathtt{nat}) \to (\mathtt{nat} \to \mathtt{nat})$.

Figures 4.7 and 4.8 give equality rules for terms. In combination with the rule of **Equality**, the **Congruence Equations** for abstractions lets us prove (as in [Asp95]), for example, that $\lambda n : \mathtt{nat}.n : \mathtt{even} \to \mathtt{even}$ even though $\mathtt{nat} \to \mathtt{nat}$ and $\mathtt{even} \to \mathtt{even}$ are incomparable (which is an obstacle for some subtyping systems).

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

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

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.

Example 4.3.8 We use the η-equality for abstractions to infer a refinement typing.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c} f:\mathtt{nat}\to\mathtt{nat}, \forall x:\mathtt{nat}.\mathtt{Even}(fx), n:\mathtt{nat}\vdash fn:\mathtt{nat} \\ f:\mathtt{nat}\to\mathtt{nat}, \forall x:\mathtt{nat}.\mathtt{Even}(fx), n:\mathtt{nat}\vdash \mathtt{Even}(fn)\end{array}}{f:\mathtt{nat}\to\mathtt{nat}, \forall x:\mathtt{nat}.\mathtt{Even}(fx), n:\mathtt{nat}\vdash fn:\mathtt{even}}}{f:\mathtt{nat}\to\mathtt{nat}, \forall x:\mathtt{nat}.\mathtt{Even}(fx)\vdash f =_{\mathtt{nat}\to\mathtt{even}} \lambda n:\mathtt{nat}.fn}\ \textbf{Function Eqs. }(\eta)}{f:\mathtt{nat}\to\mathtt{nat}, \forall x:\mathtt{nat}.\mathtt{Even}(fx)\vdash f:\mathtt{nat}\to\mathtt{even}}$$

Similarly, we use the η rule for pairs in order to prove, for example, that

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

The well-formedness hypothesis in the rule of **Logical Congruence** is important since, with stratified equalities, we require equality at the appropriate refinement type.

Remark 4.3.9 We can define singleton types. For $t:\phi$, write $\{t\}_\phi$ for the refinement type $(x:\phi)\,x =_\phi t$. Then we have $\Gamma\vdash P[t/x]$ iff $\Gamma, x:\{t\}_\phi\vdash P$. We conjecture that the $\lambda_{(:)}$-calculus is a conservative extension of Aspinall's singleton types calculus [Asp95].

Example 4.3.10

$$\dfrac{\dfrac{\dfrac{n:\mathtt{nat},\mathtt{Even}(n)\vdash n:\mathtt{nat}}{\mathtt{nat}\sqsubseteq(n:\mathtt{nat})\mathtt{Even}(n)}\ \textbf{Ref.Types (R)}\qquad \dfrac{n:\mathtt{even}\vdash n:\mathtt{even}}{n:\mathtt{even}\vdash n =_{\mathtt{even}} n}\ \textbf{Reflexivity}}{\lambda n:\mathtt{even}.n =_{\mathtt{even}\to\mathtt{even}} \lambda n:\mathtt{nat}.n}\ \textbf{Cong.Eqs.}}{\lambda n:\mathtt{nat}.n:\mathtt{even}\to\mathtt{even}}\ \textbf{Equality}$$

Using the symmetry rule we can deduce the symmetric forms for the congruences. *e.g.* if $t =_{\Sigma_{x:\phi}\psi} t'$ then $\pi_2(t) =_{\psi[\pi_1(t')]} \pi_2(t')$.

Figure 4.9 lists the refinement rules. These are of two kinds — 'structural' and logical. The obvious structural rules are

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

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.

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

The only other refinement rule is transitivity of refinement.

Equational Reasoning

$$\frac{\Gamma \vdash t : \phi}{\Gamma \vdash t =_\phi t}$$

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

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

Function Equations

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

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

Product Equations

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

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

Unit Equation

$$\frac{\Gamma \vdash t : \mathbf{1}}{\Gamma \vdash t =_{\mathbf{1}} *}$$

Figure 4.7: Equality Rules

Constant Equations

$$\frac{\begin{array}{l}\Gamma \vdash \phi_1 : \texttt{Ref}\,(\tau_1)\ \cdots\ \Gamma \vdash \phi_n : \texttt{Ref}\,(\tau_n)\\ \Gamma \vdash \psi : \texttt{Ref}\,(\tau)\\ \Gamma \vdash t_1 =_{\phi_1} t_1'\ \cdots\ \Gamma \vdash t_n =_{\phi_n} t_n'\end{array}}{\Gamma \vdash k(t_1, \ldots, t_n) =_{\psi} k(t_1', \ldots, t_n')} \qquad \begin{cases}\Gamma' \vdash k : \phi_1, \ldots, \phi_n \to \psi\ \in\ Ax\ ;\\ k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K};\\ \Gamma' \subseteq \Gamma\end{cases}$$

Congruence Equations

$$\frac{\Gamma, x : \phi \vdash P\ \texttt{wf} \quad \Gamma, x : \phi, P \vdash t =_{\psi} t'}{\Gamma \vdash \lambda x : (x : \phi)P.t =_{\Pi_{x:(x:\phi)P}\psi} \lambda x : \phi.t'}$$

$$\frac{\Gamma \vdash t_1 =_{\Pi_{x:\phi}\psi} t_1' \quad \Gamma \vdash t_2 =_{\phi} t_2'}{\Gamma \vdash t_1 t_2 =_{\psi[t_2/x]} t_1' t_2'}$$

$$\frac{\Gamma \vdash t_1 =_{\phi} t_1' \quad \Gamma \vdash t_2 =_{\psi[t_1/x]} t_2'}{\Gamma \vdash \langle t_1, t_2 \rangle =_{\Sigma_{x:\phi}\psi} \langle t_1', t_2' \rangle}$$

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

Logical Congruence

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

Weakening

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

Figure 4.8: Equality Rules cont.

Unit

$$\frac{\vdash \Gamma \ \mathtt{wf}}{\Gamma \vdash \mathbf{1} \sqsubseteq \mathbf{1}}$$

Product

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma, x : \phi \vdash \psi \sqsubseteq \psi'}{\Gamma \vdash \Sigma_{x:\phi}\psi \sqsubseteq \Sigma_{x:\phi'}\psi'}$$

Function

$$\frac{\Gamma \vdash \phi' \sqsubseteq \phi \quad \Gamma, x : \phi \vdash \psi \sqsubseteq \psi'}{\Gamma \vdash \Pi_{x:\phi}\psi \sqsubseteq \Pi_{x:\phi'}\psi'}$$

Refinement Types

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

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

Transitivity

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma \vdash \phi' \sqsubseteq \phi''}{\Gamma \vdash \phi \sqsubseteq \phi''}$$

Figure 4.9: Refinement Rules

Conjunction

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}$$

Disjunction

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q \ \texttt{wf}}{\Gamma \vdash P \vee Q} \qquad \frac{\Gamma \vdash P \ \texttt{wf} \quad \Gamma \vdash Q}{\Gamma \vdash P \vee Q}$$

$$\frac{\Gamma \vdash P \vee Q \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R}$$

Implication

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash P \supset Q}{\Gamma \vdash Q}$$

Universal Quantification

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

Falsehood

$$\frac{\Gamma \vdash P \ \texttt{wf}}{\Gamma, \bot \vdash P} \qquad \frac{\Gamma, \neg P \vdash \bot}{\Gamma \vdash P}$$

Assumptions

$$\frac{\Gamma \vdash P \ \texttt{wf}}{\Gamma, P \vdash P}$$

Elimination Rules

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

$$\frac{\Gamma \vdash t : (x : \phi) P}{\Gamma \vdash P[t/x]}$$

Figure 4.10: Natural Deduction Rules in a Theory of Refinement Types

Finally, the rules of the logic are given in Figure 4.10. This is a natural deduction presentation of a typed classical predicate logic of equalities and refinements. Assumptions in context can be used via **Assumptions** or **Elimination Rules**.

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

Contexts differ from the usual formulations of typed lambda calculi since they contain propositions. The two forms of assumption are combined in the one context to make explicit the mutual dependencies. This is illustrated 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

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

4.3.5 Booleans and Naturals

We give axioms for booleans and naturals which combine the typing and logical rules of Chapter 2.

We give the $\lambda_{(:)}$-axiom system for booleans in Figure 4.11. There is one ground type $\texttt{bool}$, and constants $\texttt{true} : \texttt{bool}, \texttt{false} : \texttt{bool}$ and $\texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_ : \texttt{bool}, \tau, \tau \to \tau$, for each type τ. The axiom schema for conditionals is given as

$$\texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_ : P + P',\ (x : \phi)\, P \supset Q[x],\ (y : \phi)\ P' \supset Q[y] \to (z : \phi)\, Q[z]$$

where $P + P'$ abbreviates $(b : \texttt{bool})\, b = \texttt{true} \supset P \wedge b = \texttt{false} \supset P'$. The axiom says that if the truth of the boolean condition implies P, and its falsity implies P', and under these assumptions we can infer that the respective branches have refinement type $(z : \phi) Q$, then the conditional as a whole has this refinement type.

In fact, this axiom implies the others. A particular case of this axiom is

$$\frac{\Gamma, b = \texttt{true} \vdash t : \psi \quad \Gamma, b = \texttt{false} \vdash t' : \psi}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t' : \psi}$$

Sortings

$$\texttt{true} : \texttt{bool}$$

$$\texttt{false} : \texttt{bool}$$

$$\texttt{if \_ then \_ else \_} : \texttt{bool}, \tau, \tau \to \tau$$

Axioms

$$x : \phi, y : \psi \vdash \texttt{if true then } x \texttt{ else } y =_\phi x$$

$$x : \phi, y : \psi \vdash \texttt{if false then } x \texttt{ else } y =_\psi y$$

$$f : \texttt{bool} \to \psi, b : \texttt{bool} \vdash f(b) =_\psi \texttt{ if } b \texttt{ then } f(\texttt{true}) \texttt{ else } f(\texttt{false})$$

$$\texttt{if \_ then \_ else \_} : P{+}P', (x : \phi)\, P \supset Q[x], (y : \phi)\, P' \supset Q[y] \to (z : \phi)\, Q[z]$$

Figure 4.11: Axiom System for Booleans

This could be expressed more elegantly using sums, which we have not studied here.

In Figure 4.12 we give the $\lambda_{(:)}$-axiom system for naturals, making use of the singleton type notation from Remark 4.3.9. There are the zero and successor constants, and two constants for recursion. There are refinement typing and equational axioms for each form of recursion.

We give constants for both *primitive* and *well-founded* recursion. Primitive recursion enables us to write simple terminating programs which loop through a finite set of the values of some type, such that at each stage the program has access to the computation on the previous value. This is the functional equivalent of the for-loop. We use the constant `natrec` for primitive recursion over the naturals.

However, many programs are most naturally expressed by a form of recursion for which, at each stage of looping over the values of some type, the program has access to computations on all previous values. For example, the recursive call of the merge sort algorithm is not to the tail of a list, but to a sublist. What we require is the functional equivalent of the while-loop. The problem is that, in general, while loops do not terminate, and if we add a construct for full recursion to the simply-typed λ-calculus, then this results in inconsistency.[2]

[2]If we have constants $\texttt{mu}_\tau : (\tau \to \tau) \to \tau$ such that $f : \tau \to \tau \vdash f(\texttt{mu}_\tau(f)) =_\tau \texttt{mu}_\tau(f)$, then we can prove all well-typed equations. Let $\texttt{not} : \texttt{bool} \to \texttt{bool}$ and $\texttt{eq} : \texttt{bool} \to \texttt{bool} \to \texttt{bool}$ be the negation and equality functions, respectively. We can use the η-equality for conditionals to prove that $b : \texttt{bool} \vdash \texttt{eq}\ b\ b = \texttt{true}$ and $b : \texttt{bool} \vdash \texttt{eq}\ b\ \texttt{not}(b) = \texttt{false}$. By substituting $\texttt{mu}_{\texttt{bool}}(\texttt{not})$ for b, we can prove that $\texttt{true} = \texttt{false}$.

Sortings

$$\texttt{0} : \texttt{nat}$$

$$\texttt{succ} : \texttt{nat} \to \texttt{nat}$$

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

$$\texttt{natwfrec} : (\texttt{nat} \to (\texttt{nat} \to \tau) \to \tau) \to \texttt{nat} \to \tau$$

Axioms

$$n : \texttt{nat} \vdash \texttt{natrec} : \phi[0], \Pi_{n:\texttt{nat}}\phi[n] \to \phi[\texttt{succ}\ n], \{n\}_{\texttt{nat}} \to \phi[n]$$

$$\frac{\Gamma \vdash z : \phi[0] \quad \Gamma \vdash s : \Pi_{n:\texttt{nat}}\phi[n] \to \phi[\texttt{succ}\ n]}{\Gamma \vdash \texttt{natrec}\ z\ s\ 0 =_{\phi[0]} z}$$

$$\frac{\Gamma \vdash z : \phi[0] \quad \Gamma \vdash s : \Pi_{n:\texttt{nat}}\phi[n] \to \phi[\texttt{succ}\ n]}{\Gamma, n : \texttt{nat} \vdash \texttt{natrec}\ z\ s\ (\texttt{succ}\ n) =_{\phi[\texttt{succ}\ n]} s\ n\ (\texttt{natrec}\ z\ s\ n)}$$

$$\texttt{natwfrec} : (\Pi_{x:\texttt{nat}}(\Pi_{z<x}\phi[z]) \to \phi[x]) \ \to \ \Pi_{x:\texttt{nat}}\phi[x]$$

$$g : \Pi_{x:\texttt{nat}}(\Pi_{z<x}\phi[z]) \to \phi[x],\ n : \texttt{nat} \vdash \texttt{natwfrec}\ g\ n =_{\phi[n]} g\ n\ (\texttt{natwfrec}\ g)$$

Figure 4.12: Axiom System for Naturals

The solution is to restrict the recursion to loops which terminate. This can be done by defining a well-founded order on the data which the recursion is over, such that at each stage the computation on a value can only make use of computations on values lower in the order. This form of recursion is known as *well-founded recursion* [Nor88]. We will only use well-founded recursion over the naturals with the usual less-than ordering. We write $\Pi_{z<x}\phi[z]$ for $\Pi_{z:\texttt{nat}|(z<x)}\phi[z]$.

Rather than separate the proof of termination from well-formedness, however, we build it in by defining a constant, `natwfrec`, which can only construct terminating loops. The termination requirement can be expressed using refinement types.

The sorting is

$$\texttt{natwfrec} : (\texttt{nat} \to (\texttt{nat} \to \tau) \to \tau) \to \texttt{nat} \to \tau$$

and the axiom is given as a refinement typing:

$$\texttt{natwfrec} : (\Pi_{x:\texttt{nat}}(\Pi_{z<x}\phi[z]) \to \phi[x]) \ \to \ \Pi_{x:\texttt{nat}}\phi[x]$$

Recursion is formulated, without sacrificing termination, by

$$g : \Pi_{x:\mathtt{nat}}(\Pi_{z<x}\phi[z]) \to \phi[x],\ n : \mathtt{nat} \vdash \mathtt{natwfrec}\ g\ n =_{\phi[n]} g\ n\ (\mathtt{natwfrec}\ g)$$

If $f : \mathtt{nat} \to \tau$ is given recursively as $f(n) = t[n, f]$ where the body of the loop, $t[n, f]$, is such that f is applied to values smaller than n, then we can define f as $\mathtt{natwfrec}\ (\lambda n.\lambda f.t)$. Then,

$$\begin{array}{rcl} f\ n & \equiv & \mathtt{natwfrec}\ (\lambda n.\lambda f.t)\ n \\ & = & t[n, \mathtt{natwfrec}\ (\lambda n.\lambda f.t)\,] \\ & \equiv & t[n, f] \end{array}$$

so that $f\ n$ can be thought of as looping to $t[n, f]$.

The induction rules of Chapter 2 can be derived. For $n : \mathtt{nat} \vdash P[n]\ \mathtt{wf}$ we derive the rule of mathematical induction:

$$\dfrac{\dfrac{\dfrac{P[0]}{* : (z : \mathbf{1})P[0]} \quad \dfrac{\forall n : \mathtt{nat}\,.\,P[n] \supset P[\mathtt{succ}\,n]}{\lambda n : \mathtt{nat}.\lambda z : \mathbf{1}.*\ :\ \Pi_{n:\mathtt{nat}}(z : \mathbf{1})P[n] \to (z : \mathbf{1})P[\mathtt{succ}\,n]}}{n : \mathtt{nat} \vdash \mathtt{natrec}\ *\ (\lambda n : \mathtt{nat}.\lambda z : \mathbf{1}.*)\ n\ :\ (z : \mathbf{1})P[n]}}{\forall n : \mathtt{nat}.P[n]}$$

Similarly, we can derive well-founded induction, and computational induction for $\mathtt{natwfrec}$: for $g : \mathtt{nat} \to (\mathtt{nat} \to \tau) \to \tau$ and $x : \tau, n : \mathtt{nat} \vdash P[x, n]\ \mathtt{wf}$,

$$\dfrac{\forall n : \mathtt{nat}.(\forall n' < n.P[\mathtt{natwfrec}\ g\ n']) \supset P[g\ (\mathtt{natwfrec}\ g\ n)]}{\forall n : \mathtt{nat}.P[\mathtt{natwfrec}\ g\ n]}$$

In fact, as noted on p. 47, well-founded induction follows from mathematical induction which, in turn, can be derived from the axiom for $\mathtt{natrec}$.

Although constants are added in the simple type theory, axioms are given using refinement types. For example, although $\mathtt{natwfrec}\ g$ is always defined for g of appropriate type, the recursion equation for $\mathtt{natwfrec}$ only holds when g has the appropriate refinement type. Semantically, $\mathtt{natwfrec}$ is interpreted as a map from the set of the underlying type, but the interpretation is only constrained on the refinement type.

4.3.6 Metatheory

We prove a few syntactic results about the calculus. Some of the following results will be needed in Section 4.5 for proving completeness with respect to the semantics. Other standard metatheoretic results (not listed here) can be deduced from completeness.

Lemma 4.3.11 *The following is derivable:*

$$\frac{\Gamma \vdash t : \phi \quad \Gamma, x : \phi \vdash P}{\Gamma \vdash t : (x : \phi)P}$$

Proof:

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

∎

We use this to show that the rule of **Refinement Type Introduction** is derivable.

Lemma 4.3.12 *The rule*

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

is derivable.

Proof:

$$\frac{\dfrac{\dfrac{\Gamma \vdash t : \phi}{\Gamma \vdash t : \{t\}_\phi} \quad \dfrac{\Gamma \vdash P[t/x]}{\Gamma, x : \{t\}_\phi \vdash P}}{\Gamma \vdash t : (x : \{t\}_\phi)P} \qquad \Gamma \vdash \phi \sqsubseteq \{t\}_\phi}{\Gamma \vdash t : (x : \phi)P}$$

∎

It is an easy proof to show that for well-formed refinement types $\Gamma \vdash \phi$, we have reflexivity of refinement $\Gamma \vdash \phi \sqsubseteq \phi$. We now give some other derived refinements.

Lemma 4.3.13 *The following rules can be derived:*

1. $$\frac{\Gamma, x : \phi \vdash P' \supset P \quad \Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash (x : \phi)P \sqsubseteq (x : \phi')P'}$$

2. $$\frac{\Gamma, P \vdash \phi \ \texttt{wf}}{\Gamma, P \vdash (x : \phi)P = \phi}$$

3. $$\frac{\Gamma \vdash \phi \ \texttt{wf}}{\Gamma \vdash (x : \phi)\top = \phi}$$

4. $$\frac{\Gamma \vdash \phi \ \texttt{wf} \quad \Gamma, x : \phi \vdash \psi \ \texttt{wf} \quad \Gamma, x : \phi, y : \psi \vdash Q \ \texttt{wf}}{\Gamma \vdash \Pi_{x:(x:\phi)P}(y : \psi)Q \sqsubseteq (f : \Pi_{x:\phi}\psi)\, \forall x : \phi.P \supset Q[fx/y]}$$

5. $$\frac{\Gamma \vdash \phi \ \texttt{wf} \quad \Gamma, x : \phi \vdash \psi \ \texttt{wf} \quad \Gamma, x : \phi, y : \psi \vdash Q \ \texttt{wf}}{\Gamma \vdash \Pi_{x:\phi}(y : \psi)Q = (f : \Pi_{x:\phi}\psi)\, \forall x : \phi.Q[fx/y]}$$

$$6.\quad \frac{\Gamma \vdash \phi \ \mathtt{wf} \quad \Gamma, x:\phi \vdash P \ \mathtt{wf} \quad \Gamma, x:\phi \vdash \psi \ \mathtt{wf} \quad \Gamma, x:\phi, y:\psi \vdash Q \ \mathtt{wf}}{\Gamma \vdash \Sigma_{x:(x:\phi)P}(y:\psi)Q = (z:\Sigma_{x:\phi}\psi)\, P[\pi_1 z/x] \wedge Q[\pi_1 z/x, \pi_2 z/y]}$$

$$7.\quad \frac{\Gamma \vdash \phi \ \mathtt{wf} \quad \Gamma, y:\phi \vdash P \ \mathtt{wf} \quad \Gamma, x:\phi, P[x/y] \vdash Q \ \mathtt{wf}}{\Gamma \vdash (x:(y:\phi)P)Q = (z:\phi)P[z/y] \wedge Q[z/x]}$$

In Remark 4.3.4 we said that refinement types correspond to a relation over a type rather than a subset. One consequence of this is that we cannot give a canonical form for refinement types simply using a type and a proposition as $(x:\tau)P$. This is in contrast to in [NPS90], where subset types are given meaning via a translation into the underlying basic type theory (see Section 5.4.3).

Instead, we introduce the notion of *pseudotype*, which we will use in the proof of completeness. These have some of the properties of types. For example, pseudotypes have no logical import, in the sense that they are inhabited if and only if the underlying type is. Moreover, we can express all refinement types in a canonical form as a propositional 'subset' of pseudotypes.

Definition 4.3.14 *The* pseudotypes *for a given $\lambda_{(:)}$-signature are given by the grammar:*

$$\kappa ::= \mathbf{1} \ \mid \ \gamma \ \mid \ \Sigma_{x:\kappa}\kappa' \ \mid \ \Pi_{x:\phi}\kappa$$

To see that we must keep the dependent constructors consider, for example, the refinement type $\Sigma_{n:\mathtt{nat}}\{n\}_{\mathtt{nat}} \to \mathtt{nat}$. This cannot be expressed in the form $\phi \times \phi'$ for any ϕ and ϕ'.

Lemma 4.3.15 *For all $\Gamma \vdash \phi$ $\mathtt{wf}$, there exists a pseudotype κ and proposition P such that, $\Gamma \vdash \phi = (x:\kappa)P$.*

Proof: Use the rules in Lemma 4.3.13. ∎

Lemma 4.3.16 *If $\bot, \Gamma \vdash \phi, \phi' : \mathtt{Ref}\ (\tau)$, then $\bot, \Gamma \vdash \phi \sqsubseteq_\tau \phi'$.*

Proof: By Lemma 4.3.15, we can assume, without loss of generality, that ϕ and ϕ' are in canonical form. The proof is a straightforward induction over τ. ∎

The evident generalisation to arbitrary propositions (if $\bot, \Gamma \vdash P$ $\mathtt{wf}$ then $\bot, \Gamma \vdash P$) holds also.

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

Proposition 4.3.17 *Given preterm t in context Γ, the following are equivalent:*

1. $\texttt{Typify}(t)$ *is well-typed in typing context* $\texttt{Typify}(\Gamma)$*, where* $\texttt{Typify}$ *replaces each occurrence of a refinement type with its underlying type and removes propositions from the context.*

2. *There exists a refinement type* ϕ *such that* $\bot, \Gamma \vdash t : \phi$*.*

3. *There exists a type* τ*, such that for all* $\bot, \Gamma \vdash \phi : \texttt{Ref}\ (\tau)$*, we can prove* $\bot, \Gamma \vdash t : \phi$*.*

Proof: Clearly (3) implies (2) and, by Lemma 4.3.16, (2) implies (3).

We induct over preterms to show that for each $\Gamma \vdash t$, (1) is equivalent to (2,3). We write $\overline{U}$ for $\texttt{Typify}(U)$. We just consider two cases.

(applications) If $\bot, \Gamma \vdash tt' : \phi$ then there exists a ψ such that $\bot, \Gamma \vdash t : \psi \to \phi$ and $\bot, \Gamma \vdash t : \psi$, so by induction, $\overline{\Gamma} \vdash \overline{t} : \sigma \to \tau$ and $\overline{\Gamma} \vdash \overline{t'} : \sigma$ for some σ and τ, and so $\overline{\Gamma} \vdash \overline{tt'} : \tau$.

Conversely, if $\overline{\Gamma} \vdash \overline{tt'} : \tau$, then $\overline{\Gamma} \vdash \overline{t} : \sigma \to \tau$ and $\overline{\Gamma} \vdash \overline{t'} : \sigma$ so, by induction, $\bot, \Gamma \vdash t : \phi \to \psi$ and $\bot, \Gamma \vdash t' : \phi$, and so $\bot, \Gamma \vdash tt' : \psi$.

(abstractions) If $\bot, \Gamma \vdash \lambda x : \phi.t : \chi$ then $\bot, \Gamma, x : \phi, P \vdash t : \chi'$ for some P and χ', so $\overline{\Gamma}, x : \sigma \vdash t : \tau$ and then $\overline{\Gamma} \vdash \lambda x : \sigma.\overline{t} : \sigma \to \tau$, that is, $\overline{\Gamma} \vdash \overline{\lambda x : \phi.t} : \sigma \to \tau$.

Conversely, if $\overline{\Gamma} \vdash \overline{\lambda x : \phi.t} : \sigma \to \tau$, then $\overline{\Gamma}, x : \sigma \vdash \overline{t} : \tau$ so, by induction, $\bot, \Gamma, x : \phi \vdash t : \psi$ for some ψ, and so, $\bot, \Gamma \vdash \lambda x : \phi.t : \Pi_{x:\phi}\psi$. ∎

Definition 4.3.18 *Let* $\Gamma \vdash t$ *be a preterm in context. If the conditions of Proposition 4.3.17 hold, then we say that* $\Gamma \vdash t$ *is* well-structured.

We extend the definition of well-structuredness to arbitrary pre-expressions, and write $\Gamma \vdash U\ \texttt{ws}$ *for* $\bot, \Gamma \vdash U\ \texttt{wf}$*.*

We will sometimes say, informally, that an expression U is well-structured. Note that if Γ is well-formed, then $\Gamma \vdash U\ \texttt{wf}$ iff $\Gamma, \bot \vdash U\ \texttt{wf}$.

There are two levels of 'well-formedness' therefore. What we have called well-structured corresponds to terms being put together correctly, irrespective of logical annotation, whereas being well-formed, as such, means that the logic is respected. In contrast to well-formedness, Proposition 4.3.17 shows that well-structuredness is decidable.

This distinction is reminiscent of the *rough types* of Sannella and Aspinall, which are like the type underlying a refinement type.

We will see in Section 4.5 that we only give a semantic interpretation to well-structured terms.

Proposition 4.3.19 *If* $\Gamma \vdash \phi, \phi' : \mathtt{Ref}\ (\tau)$*, then* $\Gamma \vdash \phi \sqsubseteq (x : \phi')\perp$.

Proof: Since $\mathtt{Typify}(\Gamma), x : \tau \ \vdash \ x \ : \ \tau$, by Proposition 4.3.17 we infer that $\Gamma, x : \phi', \perp \vdash x : \phi$, and so **Refinement Types (R)** gives $\Gamma \vdash \phi \sqsubseteq (x : \phi')\perp$. ∎

Remark 4.3.20 Refinement is a definitional extension of refinement typing in the sense that $\Gamma, x : \phi' \vdash x : \phi$ if and only if $\Gamma \vdash \phi \sqsubseteq \phi'$.

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \phi' \vdash x : \phi}{\Gamma, x : \phi', \top \vdash x : \phi}}{\Gamma \vdash \phi \sqsubseteq (x : \phi')\top}\ \textbf{Ref.Types}\ (\mathrm{R}) \qquad \dfrac{\Gamma \vdash \phi' \sqsubseteq \phi' \quad \Gamma, x : \phi' \vdash \top}{\Gamma \vdash (x : \phi')\top \sqsubseteq \phi'}\ \textbf{Ref.Types}\ (\mathrm{L})}{\Gamma \vdash \phi \sqsubseteq \phi'}$$

In fact, this can be strengthened by showing that for all $\Gamma \vdash \phi\ \mathtt{wf}$, we can prove $\Gamma \vdash \phi \sqsubseteq \phi' \equiv \forall x, y : \phi' \,.\, x =_{\phi'} y \supset x =_{\phi} y$.

A natural question, then, is can we eliminate the refinement relation and treat it as syntactic sugar? As the calculus stands, the **Weakening** rule for refinement typing has a refinement as hypothesis and so we cannot naively treat $\phi \sqsubseteq \phi'$ as syntactic sugar for $\Gamma, x : \phi' \vdash x : \phi$.

Although the system could be reformulated, we believe it is more insightful to have an explicit definition of refinement (as in Chapter 3). In practice, when applying the rules of Figure 4.9 backwards to find a proof of $\phi \sqsubseteq \phi'$, the refinement type rules are only used in **Refinement Types (R)**, when ϕ' is of the form $(x : \psi)Q$.

Remark 4.3.21 Although we have emphasised the per intuition for the $\lambda_{(:)}$-calculus, the match is not perfect. For example, we might expect equality at $(\mathtt{even} \to \mathtt{nat}) \to \mathtt{nat}$ to mean that if arguments are equal at $\mathtt{even} \to \mathtt{nat}$ then the results are equal (at $\mathtt{nat}$), but this is not so. For example, $\lambda f : \mathtt{nat} \to \mathtt{nat} \,.\, 3$ does not equal $\lambda f : \mathtt{even} \to \mathtt{nat} \,.\, 3$ at $(\mathtt{even} \to \mathtt{nat}) \to \mathtt{nat}$ since the term $\lambda f : \mathtt{nat} \to \mathtt{nat} \,.\, 3$ does not have the refinement type $(\mathtt{even} \to \mathtt{nat}) \to \mathtt{nat}$.

In Section 4.5, we will define relations on terms, $\simeq_{\phi}$, such that these two terms are related by $\simeq_{(\mathtt{even} \to \mathtt{nat}) \to \mathtt{nat}}$. Then we can think of terms of the calculus as uniquely specifying total terms up to $\simeq_{\phi}$ for some refinement type ϕ.

However, this is more a mismatch of refinement typing than equality, for if $t, t' : \phi$, then $t \simeq_{\phi} t'$ implies $t =_{\phi} t'$.

Remark 4.3.22 We can define a form of annotation, $P \to \phi$, for proposition, P, and refinement type, ϕ. We will use this notion in the completeness proof below.

The definition is:

$$\begin{array}{rcl} P \to \mathbf{1} & \equiv & \mathbf{1} \\ P \to \gamma & \equiv & \gamma \\ P \to \Sigma_{x:\phi}\psi & \equiv & \Sigma_{x:(P\to\phi)}(P \to \psi) \\ P \to \Pi_{x:\phi}\psi & \equiv & \Pi_{x:(P\to\phi)}(P \to \psi) \\ P \to (x:\phi)Q & \equiv & (x:(P\to\phi))P \supset Q \end{array}$$

We state the following two properties, for each P and ϕ:

1. If $\Gamma, P \vdash \phi$ $\mathtt{wf}$ then $\Gamma \vdash (P \to \phi)$ $\mathtt{wf}$.
2. $\Gamma, P \vdash \phi = (P \to \phi)$

4.4 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 Section 4.2 again. Recall that we define iteration from the more general recursion, as

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

where $x \notin FV(t')$.

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

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

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

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

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

The proof continues with

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

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

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

4.5 Models

In contrast with the previous two chapters we will not interpret derivations of judgements, but rather 'pre-judgements'. This is because we do not have unique refinement typings, or even unique derivations of particular refinement typings. To show that a semantics based on derivations gives unique interpretations would require an analysis of coherence which we avoid. This is not quite what we might call a 'Curry-style' interpretation, however, since we do not erase the refinement types from terms.

In Chapter 2, we used Henkin models to interpret the simply-typed lambda calculus and first-order logic. Here we will extend this, and interpret terms as sets (their 'total realizers'). Refinement types over type σ are interpreted as pers over $\sigma^{\mathcal{A}}$.

Definition 4.5.1 *A* Henkin interpretation *of a $\lambda_{(:)}$-signature is the same as a Henkin interpretation of a first-order $\lambda^{\times\rightarrow}$-signature.*

Although the raw data of $\lambda_{(:)}$- and first-order $\lambda^{\times\rightarrow}$-interpretations (Definition 2.4.1) are the same, the induced semantics are different.

We assume some Henkin interpretation below when we write $[\![\;]\!]$. A Henkin interpretation $\mathcal{A}$ models an axiom system when all constants and predicates are given an interpretation, and each axiom is true in the interpretation, as defined below. Although the environment model condition is only given for $\lambda^{\times\rightarrow}$, the interpretation of $\lambda_{(:)}$-terms is well-defined.

Strictly speaking, the meaning function is a partial mapping from pre-expressions to meanings. We will show later that it is total for the well-structured terms.

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

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

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

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

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

Figure 4.13: Interpretation of Refinement Types

Now, expressions are all interpreted in context, so for context Γ we must first define Γ-environments, η, in interpretation $\mathcal{A}$, written $\eta \vDash^{\mathcal{A}} \Gamma$ (dropping the $\mathcal{A}$ when not significant), where η 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 indicate that a is in the domain of R, *i.e.* $a \; R \; a$.

We first define the notion of equality of environments, in the obvious way, as simultaneous equality of elements in the corresponding per, written $\eta \,[\![\Gamma]\!]\, \eta'$.

$$\langle\rangle \,[\![\langle\rangle]\!]\, \langle\rangle$$

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

$$\eta \,[\![\Gamma, P]\!]\, \eta' \text{ when } \eta \,[\![\Gamma]\!]\, \eta' \text{ and } \eta, \eta' \in [\![\Gamma \vdash P]\!]$$

Then we define $\eta \vDash \Gamma$ to mean $\eta \,[\![\Gamma]\!]\, \eta$.

To avoid questions of coherence, we interpret pre-terms, and so pre-propositions and pre-refinement types too.

Now as mentioned above, refinement types are interpreted as pers. The interpretation is given in Figure 4.13 where we adopt the convention that the pers are over the set corresponding to the underlying type. 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. It is easy to see that all types are interpreted as identities.

There is an apparent asymmetry in the definition of the product per for $\Sigma_{x:\phi}\psi$, but in fact, if ϕ is a well-formed refinement type in context Γ, then the soundness result below states that if $\eta\ [\![\Gamma]\!]\ \eta'$ we have $[\![\Gamma \vdash \phi]\!](\eta) = [\![\Gamma \vdash \phi]\!](\eta')$.

In Figure 4.14, The pre-term in context $\Gamma \vdash t$ is interpreted in environment $g \vDash \Gamma$ as a subset (its 'total realizers') of $\sigma^{\mathcal{A}}$, where σ is the type underlying t. The types are implicit in the interpretation. An alternative would be to make them explicit by giving an interpretation over well-structuredness judgements. This is the approach taken in [Asp97], for example.

$$\frac{[\![\Gamma \vdash \phi]\!] = R}{[\![\Gamma, x : \phi, \Gamma' \vdash x]\!]\langle \eta, a, \eta' \rangle = \{a' \mid a'\ R(\eta)\ a\}}$$

$$\frac{[\![\Gamma \vdash t_1]\!] = m_1 \ \cdots \ [\![\Gamma \vdash t_n]\!] = m_n}{[\![\Gamma \vdash k(t_1, \ldots, t_n)]\!](\eta) = \{k^{\mathcal{A}}(a_1, \ldots, a_n) \mid a_i \in m_i(\eta)\}}$$

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

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

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

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

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

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

Figure 4.14: Interpretation of Terms

It is because of the refinement type in abstractions that we interpret terms as sets rather than as single elements. For example, $\lambda n : \mathtt{even}.n$ is interpreted as the set of elements in $(\mathtt{nat} \to \mathtt{nat})^{\mathcal{A}}$ which are the identity for even arguments. In Figure 4.15 we give the interpretation of propositions. We interpret a pre-proposition in context $\Gamma \vdash P$ as the set of environments $\eta \vDash \Gamma$ in which P holds.

As we mentioned above, although the interpretation function is partial, well-structured terms are always given an interpretation. For example, $(\lambda n : \mathtt{even}.n)*$

does not have a well-defined interpretation, but $(\lambda n : \texttt{even}.n)3$ does, even though it is not syntactically well-formed.

Proposition 4.5.2 *If* $\Gamma \vdash U$ $\texttt{ws}$ *and* $\eta \vDash \Gamma$, *then* $[\![\Gamma \vdash U]\!](\eta)$ *is defined.*

Proof: We induct over pre-expressions, and consider two cases.

(applications) If $\Gamma \vdash tt'$ $\texttt{ws}$, then $\overline{\Gamma} \vdash \overline{tt'} : \tau$, so $\overline{\Gamma} \vdash \overline{t} : \sigma \to \tau$ and $\overline{\Gamma} \vdash \overline{t'} : \sigma$, so by the inductive hypothesis, $[\![\Gamma \vdash t]\!](\eta)$ is a well-defined subset of $(\sigma \to \tau)^{\mathcal{A}}$, and $[\![\Gamma \vdash t']\!](\eta)$ is a well-defined subset of $\sigma^{\mathcal{A}}$. Hence, $[\![\Gamma \vdash tt']\!](\eta)$ is a well-defined subset of $\tau^{\mathcal{A}}$.

(abstractions) If $\Gamma \vdash \lambda x : \phi.t$ $\texttt{ws}$ then $\overline{\Gamma} \vdash \lambda x : \sigma.\overline{t} : \sigma \to \tau$, so $\overline{\Gamma}, x : \sigma \vdash \overline{t} : \tau$ and, by induction, if $\eta \vDash \Gamma$ and $a \in [\![\Gamma \vdash \phi]\!](\eta)$, then $[\![\Gamma, x : \phi \vdash t]\!](\langle \eta, a\rangle) \subseteq \tau^{\mathcal{A}}$ is well-defined.

∎

$$[\![\Gamma \vdash \bot]\!] = \emptyset$$
$$[\![\Gamma \vdash P \supset P']\!] = \{\eta \vDash \Gamma \mid \eta \notin [\![\Gamma \vdash P]\!] \text{ or } \eta \in [\![\Gamma \vdash P']\!]\}$$
$$[\![\Gamma \vdash \forall x : \phi.P]\!] = \{\eta \vDash \Gamma \mid \forall a \in [\![\Gamma \vdash \phi]\!](\eta) \;.\; \langle \eta, a\rangle \in [\![\Gamma, x : \phi \vdash P]\!]\}$$
$$\frac{[\![\Gamma \vdash t_1]\!] = m_1 \quad \cdots \quad [\![\Gamma \vdash t_n]\!] = m_n}{[\![\Gamma \vdash F(t_1, \ldots, t_n)]\!] = \{\eta \vDash \Gamma \mid \forall a_i \in m_i\eta \;.\; \langle a_1, \ldots, a_n\rangle \in F^{\mathcal{A}}\}}$$
$$\frac{[\![\Gamma \vdash t]\!] = m \quad [\![\Gamma \vdash t']\!] = m' \quad [\![\Gamma \vdash \phi]\!] = R}{[\![\Gamma \vdash t =_\phi t']\!] = \{\eta \vDash \Gamma \mid \forall a \in m(\eta) \;.\; \forall a' \in m'(\eta) \;.\; a\ R(\eta)\ a'\}}$$
$$\frac{[\![\Gamma \vdash \phi]\!] = R \quad [\![\Gamma \vdash \phi']\!] = R'}{[\![\Gamma \vdash \phi \sqsubseteq \phi']\!] = \{\eta \vDash \Gamma \mid R(\eta) \supseteq R'(\eta)\}}$$

Figure 4.15: Interpretation of Propositions

We may now say what the semantic analogues of the judgements are. Let $\mathcal{A}$ be a $\lambda_{(:)}$-Henkin interpretation, and assume that $\eta \vDash^{\mathcal{A}} \Gamma$. Define $\Gamma \vDash^{\mathcal{A},\eta} t : \phi$ when for all $\eta\ [\![\Gamma]\!]\ \eta'$, for all $a \in [\![\Gamma \vdash t]\!](\eta)$ and $a' \in [\![\Gamma \vdash t]\!](\eta')$, we have $a\ [\![\Gamma \vdash \phi]\!](\eta)\ a'$. In other words, the interpretation is unique up to the equality of the per. We say that $\Gamma \vDash^{\mathcal{A},\eta} P$ when $\eta \in [\![\Gamma \vdash P]\!]$. In particular, the refinement $\Gamma \vDash^{\mathcal{A},\eta} \phi \sqsubseteq \phi'$ is true when there is an inclusion of pers $[\![\Gamma \vdash \phi']\!](\eta) \subseteq [\![\Gamma \vdash \phi]\!](\eta)$. We define $\Gamma \vDash^{\mathcal{A},\eta} \phi$ $\texttt{wf}$ to mean: for all $\eta\ [\![\Gamma]\!]\ \eta'$, $[\![\Gamma \vdash \phi]\!](\eta) = [\![\Gamma \vdash \phi]\!](\eta')$, and $\Gamma \vDash^{\mathcal{A},\eta} P$ $\texttt{wf}$ to mean: for all $\eta\ [\![\Gamma]\!]\ \eta'$, $\eta \in [\![\Gamma \vdash P]\!] \iff \eta' \in [\![\Gamma \vdash P]\!]$. We define validity of a basic judgement, B, to be its truth in all environments, that is, $\Gamma \vDash^{\mathcal{A}} B$ means: for all $\eta \vDash^{\mathcal{A}} \Gamma$, $\Gamma \vDash^{\mathcal{A},\eta} B$.

In defining when an interpretation models an axiom system we only require the well-formed axioms to hold.

Definition 4.5.3 *Let $\langle Sg, Ax\rangle$ be a $\lambda_{(:)}$-axiom system. A Henkin interpretation $\mathcal{A}$ of Sg is a* model *of $\langle Sg, Ax\rangle$ when*

- *for each $\Gamma \vdash P \;\in\; Ax$, if $\langle Sg, Ax\rangle \rhd \Gamma \vdash P$ $\mathtt{wf}$ then $\Gamma \models^{\mathcal{A}} P$.*
- *for each $\Gamma \vdash k : \phi_1, \ldots, \phi_n \to \psi \;\in\; Ax$ such that $Sg \rhd k : \sigma_1, \ldots, \sigma_n \to \tau$, $\langle Sg, Ax\rangle \rhd \Gamma \vdash \phi_i :$ $\mathtt{Ref}$ (σ_i), $\langle Sg, Ax\rangle \rhd \Gamma \vdash \psi :$ $\mathtt{Ref}$ (τ),*
  for all $\eta \models^{\mathcal{A}} \Gamma$, if $a_i \; [\![\Gamma \vdash \phi_i]\!]^{\mathcal{A}}(\eta) \; a'_i$ for each $i = 1, \ldots, n$, then

  $$k^{\mathcal{A}}(a_1, \ldots, a_n) \;\; [\![\Gamma \vdash \psi]\!]^{\mathcal{A}}(\eta) \;\; k^{\mathcal{A}}(a'_1, \ldots, a'_n)$$

  We write this as $\Gamma \models^{\mathcal{A}} k : \phi_1, \ldots, \phi_n \to \psi$.

First we give a substitution lemma.

Lemma 4.5.4 *(Substitution Lemma) If $\Gamma \models^{\mathcal{A},\eta} t_i : \phi_i$ $(i = 1, \ldots, n)$, then*

$$[\![x_1 : \phi_1, \ldots, x_n : \phi_n \vdash U]\!]^{\mathcal{A}} \; \langle a_1, \ldots, a_n\rangle = [\![\Gamma \vdash U[t_i/x_i]]\!]^{\mathcal{A}}(\eta)$$

where $a_i \in [\![\Gamma \vdash t_i]\!]^{\mathcal{A}}(\eta)$ (so $\langle a_1, \ldots, a_n\rangle \models^{\mathcal{A}} x_1 : \phi_1, \ldots, x_n : \phi_n$).

Proof: Induction over $x_1 : \phi_1, \ldots, x_n : \phi_n \vdash U$. ∎

We need the condition that $\Gamma \models^{\mathcal{A},\eta} t_i : \phi_i$. The weaker requirement that for $a_i \in [\![\Gamma \vdash t_i]\!]^{\mathcal{A}}(\eta)$, $a_i \in [\![\Gamma \vdash \phi_i]\!]^{\mathcal{A}}(\eta)$ is not sufficient. For example, in any model $\lambda n : \mathtt{even}.n \not\models \mathtt{nat} \to \mathtt{nat}$, and for each $a \in [\![\lambda n : \mathtt{even}.n]\!]$ we have $a \models \mathtt{nat} \to \mathtt{nat}$, but $[\![f : \mathtt{nat} \to \mathtt{nat} \vdash f]\!](a) \neq [\![\lambda n : \mathtt{even}.n]\!]$.

Note also that the substitution pre-expression $\Gamma \vdash U[t_i/x_i]$ might not be well-formed. It is, however, if $\Gamma \vdash t_i : \phi_i$ (for $i = 1, n$).

A consequence of the substitution lemma is that for $x : \phi \vdash U$ $\mathtt{wf}$, and $\Gamma \models^{\mathcal{A},\eta} t : \phi$, we can unambiguously use the notation $[\![x : \phi \vdash U]\!] \; ([\![\Gamma \vdash t]\!](\eta))$ (dropping the $\mathcal{A}$) to mean $[\![x : \phi \vdash U]\!](a)$ for any $a \in [\![\Gamma \vdash t]\!](\eta)$. We can then express (an instance of) Lemma 4.5.4 as

$$[\![x : \phi \vdash U]\!] \; ([\![\Gamma \vdash t]\!](\eta)) = [\![\Gamma \vdash U[t/x]]\!](\eta)$$

More generally,

$$[\![\Gamma, x : \phi, \Gamma' \vdash U]\!]\langle \eta, [\![\Gamma \vdash t]\!](\eta), \eta'\rangle = [\![\Gamma, \Gamma'[t/x] \vdash U[t/x]]\!]\langle \eta, \eta'\rangle$$

Lemma 4.5.5 *If $\models t =_{\phi} t'$ and $x : \phi \models U$ $\mathtt{wf}$, then $[\![U[t/x]]\!] = [\![U[t'/x]]\!]$.*

Proof: Let $a \in [\![t]\!]$, $a' \in [\![t']\!]$. Then $a \; [\![\phi]\!] \; a'$, and $[\![U[t/x]]\!] = [\![x : \phi \vdash U]\!](a) = [\![x : \phi \vdash U]\!](a') = [\![U[t'/x]]\!]$. ∎

Lemma 4.5.6 *For* $\vDash t : \phi$*, and* $x : \phi \vDash P$ $\texttt{wf}$*, if* $[\![t]\!] \subseteq [\![x : \phi \vdash P]\!]$ *then* $\vDash P[t]$*.*

Proof: This is an immediate consequence of Lemma 4.5.4. ∎

We now verify that interpretations respect the rules of the calculus, that is, the calculus is sound with respect to models of $\lambda_{(:)}$-axiom systems. A consequence of this is that the axiom system for booleans and naturals is consistent since we can give nontrivial models.

Theorem 4.5.7 *(Soundness) Let* $\mathcal{A}$ *be a Henkin model of* $\lambda_{(:)}$*-axiom system* $\langle Sg, Ax\rangle$*. Then if* $\langle Sg, Ax\rangle \rhd \Gamma \vdash t : \phi$ *then* $\Gamma \vDash^{\mathcal{A}} t : \phi$*, if* $\langle Sg, Ax\rangle \rhd \Gamma \vdash \phi$ $\texttt{wf}$ *then* $\Gamma \vDash^{\mathcal{A}} \phi$ $\texttt{wf}$*, if* $\langle Sg, Ax\rangle \rhd \Gamma \vdash P$ $\texttt{wf}$ *then* $\Gamma \vDash^{\mathcal{A}} P$ $\texttt{wf}$*, and if* $\langle Sg, Ax\rangle \rhd \Gamma \vdash P$ *then* $\Gamma \vDash^{\mathcal{A}} P$*.*

Proof: Simultaneous induction over all derivations. The soundness of β-equality for abstractions follows from Lemma 4.5.4. Assume that $\Gamma, x : \phi \vDash t : \psi$, $\Gamma \vDash t' : \phi$, and let $\eta\ [\![\Gamma]\!]\ \eta'$, $a \in [\![\Gamma \vdash (\lambda x : \phi.t)t']\!](\eta)$, $a' \in [\![\Gamma \vdash t[t'/x]]\!](\eta')$. Then $a\ [\![\Gamma \vdash \psi[t'/x]]\!](\eta)\ a'$. ∎

Although we interpret a term in an environment as a set, the soundness theorem shows that since contexts can be seen as pers, the interpretation of a term gives rise to a morphism of pers, that is, a map of equivalence classes. For example, we can think of the interpretation of a variable as a map from an element to its equivalence class (in the relevant refinement type). These informal remarks will be further clarified in the next chapter.

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 some axiom system then it is provable. The 'ideal' completeness theorem would be (for refinement typings) that if $\Gamma \vDash t : \phi$ then $\Gamma \vdash t : \phi$. Unfortunately, this fails for two reasons. Firstly, due to the way in which well-formedness is combined with satisfying logical properties, we must assume that the judgement is well-formed, by which we mean the well-formedness of its component expressions. 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 $\Gamma \vDash t =_\phi t'$. In other words, t and t' correspond to the same equivalence class of ϕ. We will give a syntactic characterisation of this.

As in the previous two chapters, we prove completeness using the notion of Henkin theory, suitably extended. We will regard theories as infinite contexts, Γ, rooted on the left, for which $\Gamma \vdash P$ iff $P \in \Gamma$. We say that an infinite context is well-formed when every finite prefix is well-formed.

Definition 4.5.8 *Let $\langle Sg, Ax\rangle$ be a $\lambda_{(:)}$-axiom system. A $\lambda_{(:)}$-*Henkin theory *over $\langle Sg, Ax\rangle$, is a well-formed infinite context, Γ, closed under derivation from $\langle Sg, Ax\rangle$ such that if $\exists x : \phi.P$ is in Γ, then there is a term $\Gamma \vdash t : \phi$ such that $P[t/x]$ is in Γ.*

The completeness proof rests on the construction of a term model, formed from a suitable Henkin theory.

There are actually a number of possibilities which, *a priori*, we can consider for the class of terms in the set $\tau^{\mathcal{A}}$. Firstly, there is a choice between total terms of $\lambda^{\times\rightarrow}$ and arbitrary terms of $\lambda_{(:)}$. Another choice is between well-formed terms — either terms with type τ, or with any refinement type ϕ such that $\phi : \mathtt{Ref}\,(\tau)$ — or all well-structured terms over τ.

We rule out total terms at types because, with such an interpretation, it is not immediately obvious how to construct an environment in the proof of completeness. For example, if $\mathtt{halt} : \mathtt{Ref}(\mathtt{nat})$ is the refinement type of encodings of programs which halt, and $x : \mathtt{halt} \to \mathtt{nat}$ is in the context, then there would be no term in $\mathtt{nat}^{\mathcal{A}}$ (*i.e.* no term of type $\mathtt{nat}$) which equals x.

We do not use arbitrary terms at refinement types because, as pointed out in the discussion after Lemma 4.5.4, this would lead to substitutions not being well-formed. For example,

$$[\![h : (\mathtt{even} \to \mathtt{nat}) \to \mathtt{nat} \vdash (\lambda h' : (\mathtt{even} \to \mathtt{nat}) \to \mathtt{nat}.h')h]\!]\ [\lambda f : \mathtt{nat} \to \mathtt{nat}.3]$$

would contain a pre-term which is not well-formed.

Thus, we will use the well-structured terms of $\lambda^{\times\rightarrow}$. We use the following definition to characterise the term model.

Definition 4.5.9 *We define $\tilde{P}$ for each proposition, P by replacing each equality, $t =_\phi t'$, with $t \simeq_\phi t'$, where we write:*

- $t_1 \simeq_{\mathbf{1}} t_2$ *for* $t_1 =_{\mathbf{1}} t_2$

- $t_1 \simeq_\gamma t_2$ *for* $t_1 =_\gamma t_2$
- $t_1 \simeq_{\Sigma_{x:\phi}\psi} t_2$ *for* $\pi_1(t_1) \simeq_\phi \pi_1(t_2) \wedge \pi_2(t_1) \simeq_{\psi[\pi_1(t_1)]} \pi_2(t_2)$
- $t_1 \simeq_{\Pi_{x:\phi}\psi} t_2$ *for* $\forall x : \tau.\forall x' : \tau \,.\, x \simeq_\phi x' \supset t_1 x \simeq_\psi t_2 x'$ *(where* $\phi :$ $\mathtt{Ref}\,(\tau)$*)*
- $t_1 \simeq_{(x:\phi)P} t_2$ *for* $t_1 \simeq_\phi t_2 \wedge \tilde{P}[t_1] \wedge \tilde{P}[t_2]$

Lemma 4.5.10 *If* $\Gamma \vdash t \simeq_{\phi'} t'$ *and* $\Gamma \vdash \phi \sqsubseteq \phi'$ *then* $\Gamma \vdash t \simeq_\phi t'$.

Proof: Induction over $\Gamma \vdash \phi \sqsubseteq \phi'$. ∎

Lemma 4.5.11 *Suppose* $x \notin FV(\psi)$. *If* $\Gamma, x : \phi \vdash t : \psi$ *and* $\Gamma \vdash t_1 \simeq_\phi t_2$, *then* $\Gamma \vdash t[t_1/x] \simeq_\psi t[t_2/x]$.

Lemma 4.5.12 *If* $\Gamma \vdash t \simeq_\phi t$ *then there exists a* t' *such that* $\Gamma \vdash t \simeq_\phi t'$ *and* $\Gamma \vdash t' : \phi$.

Proof: Induction over ϕ. ∎

Definition 4.5.13 *Let* u *and* t *be well-formed terms with* u *total. We define* $u\ \mathtt{sat}^\Gamma\ t$ *to mean: for all* ϕ, *if* $\Gamma \vdash t : \phi$ *then* $\Gamma \vdash u \simeq_\phi t$.

Although this definition would make sense for arbitrary well-formed terms, the idea is that it formalises when total u is a realizer of t. We superscript the context, Γ, rather than writing $\Gamma \vdash u\ \mathtt{sat}\ t$, so that when we use infinite contexts, Γ_∞, this will not clash with our convention of writing $\Gamma_\infty \vdash B$ to mean $\Gamma \vdash B$ for Γ 'some sub-context' of Γ_∞.

Definition 4.5.14 *We define* $\Gamma \vdash t \sim \phi$ *to mean: there exists a term* t' *such that* $\Gamma \vdash t' : \phi$ *and* $\Gamma \vdash t \simeq_\phi t'$.

In Definition 2.4.4, we defined the Henkin closure of a collection of first-order $\lambda^{\times\rightarrow}$-propositions by adding variables for every nonempty type and propositions stating that all existentials have witnesses. The analogous definition here would be to repeat this for all refinement types (over nonempty types), ϕ, and propositions, $\exists x : \phi.P$, but there are several problems with this.

One problem is that we cannot just assume some variable $x : \phi$, because this has some logical import which might give a contradiction in the current context. We will see that it suffices, in fact, to work with pseudotypes (Definition 4.3.14), since for pseudotype, κ, the assumption $x : \kappa$ has no logical import.

Another problem is that an arbitrary (well-structured) proposition or pseudo-type need not be well-formed in the given context. An added complication is that the order in which assumptions are listed might be significant. We can get round these problems by the following trick. For any well-structured expression, U, we can give a well-formed proposition, $\mathtt{wf}(U)$, which says that U is well-formed. Then, for example, the proposition $\mathtt{wf}(P) \supset P$ is always well-formed, and when P is actually well-formed, is equivalent to P. Similarly, for any well-structured pre-refinement type, ϕ, the refinement type $\mathtt{wf}(\phi) \to \phi$ (using the notation of Remark 4.3.22) is always well-formed.

Thus, the only order that matters in the Henkin theory is that variables precede any expressions in which they appear. This is similar to Definition 2.4.4 for first-order $\lambda^{\times\to}$.

Definition 4.5.15 *Let* Γ *be a* $\lambda_{(:)}$*-context. We define the* Henkin closure *of* Γ *by the following procedure.*

1. *Iterate through the well-structured* κ *deciding which are inhabited (as in Definition 2.4.4),*
2. *List all well-structured propositions of the form* $\exists x : \kappa_n.P_n$ *for inhabited* κ_n*.*
3. *Make a list of variables* $\{y_n : \kappa_n\}$ *such that* $y_n \notin P_{n'}, \kappa_{n'}$*, for* $n' \leq n$*, and* $FV(\exists x : \kappa_n.P_n) \subseteq \Gamma, y_1 : \kappa_1, \ldots, y_n : \kappa_n$*.*
4. *Let* $Q_n \equiv \mathtt{wf}(\exists x : \kappa_n.P_n) \supset \exists x : \kappa_n.P_n \supset P_n[y_n/x]$*, and* $\kappa'_n \equiv \mathtt{wf}(\kappa_n) \to \kappa_n$*. Define the Henkin closure,* Γ_H*, as* $\Gamma, y_1 : \kappa'_1, Q_1, y_2 : \kappa'_2, Q_2, \ldots$

As was the case with first-order $\lambda^{\times\to}$ and $\lambda_?$, although we do not have minimal term models (due to having propositional assumptions), we can still use a term model construction to prove completeness. We use a slight generalisation of the standard 'consistency implies satisfiability' argument.

First we generalise consistency and satisfiability from sets of closed propositions to arbitrary contexts. We say that context Γ is *consistent*, when $\Gamma \nvdash \bot$, and *satisfiable*, when there exists a model $\mathcal{A}$ and Γ-environment η in $\mathcal{A}$, that is, $\eta \vDash^{\mathcal{A}} \Gamma$. In the case that Γ is a context of closed propositions, these reduce to the usual definitions of consistency and satisfiability. Let us write $\Gamma \vDash P$ to mean, informally, that $\Gamma \vDash^{\mathcal{A}} P$ for every model $\mathcal{A}$ of some axiom system. 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 $\Gamma \nvDash P$. The situation for the other judgement form, namely refinement typings, can be reduced to that of propositions.

Theorem 4.5.16 *(Completeness) Let $\langle Sg, Ax\rangle$ be a $\lambda_{(:)}$-axiom system. For $\langle Sg, Ax\rangle \rhd \Gamma \vdash P$ $\mathtt{wf}$, if $\Gamma \models^{\mathcal{A}} P$ for all Henkin models $\mathcal{A}$ of $\langle Sg, Ax\rangle$ then $\langle Sg, Ax\rangle \rhd \Gamma \vdash P$. For $\langle Sg, Ax\rangle \rhd \Gamma \vdash t$ $\mathtt{wf}$ and $\langle Sg, Ax\rangle \rhd \Gamma \vdash \phi$ $\mathtt{wf}$, if $\Gamma \models^{\mathcal{A}} t : \phi$ for all models $\mathcal{A}$ of $\langle Sg, Ax\rangle$ then $\langle Sg, Ax\rangle \rhd \Gamma \vdash t \sim \phi$.*

Proof:

Let Γ be a $\lambda_{(:)}$-context such that $\tilde{\Gamma}$ is consistent. First we sketch the construction of a particular model $\mathcal{T}$ and environment η such that $\eta \models^{\mathcal{T}} \Gamma$ (steps 1-5), and then use this to deduce completeness (step 6).

1. Construct a maximal consistent Henkin theory Γ_∞ such that $\{P \mid \tilde{\Gamma} \vdash P\} \subseteq \Gamma_\infty$.

   Let Γ_H be the Henkin closure of $\widetilde{\Gamma \cup Ax}$. We apply the $\widetilde{(\_)}$ operation to the axioms so that the interpretation will be a model.

   First we consider sets of propositions, Δ, with the property that Δ can be 'inserted' into Γ_H giving a consistent extension; that is, there exists a context Γ' such that Γ is a subcontext of Γ', and Γ' consists of Γ plus the propositions in Δ, in some order. We form the partial order of such sets, ordered by subsetting. This is clearly nonempty. It is also closed under unions of chains. To see why, let us formalise the insertion of a set, Δ_n, as a mapping $i_n : \Delta_n \to \mathcal{N}$. Then, by always adding new elements 'to the right', it is possible to insert Δ_n in Γ_H in such a way that all supersets can be inserted in a way which extends this insertion, that is, such that $i_{n+1} \restriction \Delta_n = i_n$. The limit, i_∞, is not necessarily an embedding of the union, since it might try to insert all the set at one place (if $\iota(n) = n$, for example). We must be careful to 'spread' the set throughout Γ_H. This can be achieved by inserting each element at double the index of the naive embedding so we define $i(n) = 2 \times i_\infty(n)$.

   Hence each chain has an upper bound and so, by Zorn's Lemma, the collection has a maximal element, Δ_∞. We define Γ_∞ to be any insertion of Δ_∞ into Γ_H. Clearly Γ_∞ is a theory, that is, it is well-formed and closed under deduction.

2. We define a relation $=$ on well-structured terms in Γ_∞. For $\Gamma_\infty \vdash u, u'$ $\mathtt{ws}$, we define $u = u'$ to mean: if $\Gamma_\infty \vdash u, u'$ $\mathtt{wf}$ then u $\mathtt{sat}^{\Gamma_\infty}$ u' and u' $\mathtt{sat}^{\Gamma_\infty}$ u; otherwise, if neither is well-formed then $\top$, else $\bot$. The intuition behind this definition is that terms which are not well-formed correspond to the set of *all* well-structured terms and so should be equal. Define $\sigma^{\mathcal{T}}$ as the set of

=-equivalence classes of well-structured terms of $\lambda^{\times\to}$, open with respect to Γ_∞, and over the type σ, that is,

$$\{u \mid \Gamma_\infty, \perp \vdash u : \sigma\}$$

The position of $\perp$ in the context does not matter since Γ_∞ is well-formed. We write $[u]$ for the equivalence class of u.

We construct a Henkin interpretation, $\mathcal{T}$, by interpreting constants syntactically. For constant symbol $k : \tau_1, \ldots, \tau_n \to \tau$, define $k^{\mathcal{T}} : \tau_1^{\mathcal{T}} \times \cdots \times \tau_n^{\mathcal{T}} \to \tau^{\mathcal{T}}$ as $k^{\mathcal{T}}([u_1], \ldots, [u_n]) = [k(u_1, \ldots, u_n)]$. Since these terms are well-structured, the interpretation is well-defined.

For predicate symbol $F :$ `Pred` $(\tau_1, \ldots, \tau_n)$, define $F^{\mathcal{T}} \subseteq \tau_1^{\mathcal{T}} \times \cdots \times \tau_n^{\mathcal{T}}$ as $\{\langle [u_1], \ldots, [u_n] \rangle \mid F(u_1, \ldots, u_n) \in \Gamma_\infty\}$.

We can show that $\mathcal{T}$ is extensional, and since the environment model condition clearly holds, $\mathcal{T}$ is an interpretation.

3. We must characterise the interpretation of terms and refinement types in the term interpretation, $\mathcal{T}$.

   We need two cases, if $\eta' \vDash^{\mathcal{T}} \Gamma'$:

   - for $\Gamma_\infty \vdash \tilde{P}[\eta'/\Gamma']$ `wf`, prove that $\Gamma' \vDash^{\mathcal{T},\eta'} P \iff \Gamma_\infty \vdash \tilde{P}[\eta'/\Gamma']$
   - for $\Gamma_\infty \vdash \tilde{\phi}[\eta'/\Gamma']$ `wf`, $[u]\ \ [\![\Gamma' \vdash \phi]\!]^{\mathcal{T}}(\eta')\ [u'] \iff \Gamma_\infty \vdash u \simeq_{\tilde{\phi}[\eta'/\Gamma']} u'$

   This is carried out in Lemma 4.5.17 below, after the sketch of completeness.

4. The interpretation, $\mathcal{T}$, is a model of the axioms. For each axiom of the form $\Gamma' \vdash P$, we can define a closed equivalent which we write as $\forall\Gamma' . P$. This has the obvious inductive definition: $\forall\langle\rangle . P \equiv P$, $\forall(\Gamma', x : \phi) . P \equiv \forall\Gamma' . \forall x : \phi.P$, and $\forall(\Gamma', P') . P \equiv \forall\Gamma' . P' \supset P$.

   Then since $\widetilde{\forall\Gamma'.P}$ is well-formed and $\Gamma_\infty \vdash \widetilde{\forall\Gamma'.P}$, by the previous step we get $\vDash^{\mathcal{T}} \forall\Gamma'.P$ and so $\Gamma' \vDash^{\mathcal{T}} P$.

   We must show that for each axiom $\Gamma' \vdash k : \phi \to \psi$ (without loss of generality, we just consider unary constants), for all $\eta' \vDash^{\mathcal{T}} \Gamma'$, for all $a\ [\![\Gamma' \vdash \phi]\!](\eta)\ a'$, that $k^{\mathcal{T}}(a)\ [\![\Gamma' \vdash \psi]\!](\eta)\ k^{\mathcal{T}}(a')$. By step 3, this is if when $\Gamma_\infty \vdash \eta' \simeq_{\Gamma'} \eta'$ and $\Gamma_\infty \vdash u \simeq_{\phi[\eta'/\Gamma']} u'$, then $\Gamma_\infty \vdash k(u) \simeq_{\psi[\eta'/\Gamma']} k(u')$.

We use singleton types (as discussed on p. 106) to derive this:

$$\dfrac{\dfrac{\dfrac{\Gamma_\infty \vdash u \simeq_{\phi[\eta'/\Gamma']} u'}{\Gamma_\infty, \Gamma', \eta' \simeq_{\Gamma'} \eta' \vdash u \simeq_\phi u'}}{\Gamma_\infty, \Gamma', \eta' \simeq_{\Gamma'} \eta' \vdash k(u) \simeq_\psi k(u')}\ \text{(Lemma 4.5.11)}}{\Gamma_\infty \vdash k(u) \simeq_{\psi[\eta'/\Gamma']} k(u')}$$

Hence $\mathcal{T}$ models $\langle Sg, Ax\rangle$.

5. As in the proof of completeness for first-order $\lambda^{\times\rightarrow}$, if $x_1 : \phi_1, \ldots, x_n : \phi_n$ are the variables in Γ, then we define the Γ-environment, η, as $\langle [x_1], \ldots, [x_n]\rangle$. We can show that $\eta \models^{\mathcal{T}} \Gamma$, by induction over subcontexts of Γ.

   Thus for an arbitrary context, Γ, if $\tilde{\Gamma}$ is consistent then Γ is satisfiable.

6. Finally, we show that if $\Gamma \models^{\mathcal{T}} B$ then $\tilde{\Gamma} \vdash \tilde{B}$, for B a proposition or refinement typing (writing $\widetilde{t : \phi}$ for $t \sim \phi$). Thus if $\Gamma \models B$ we have $\tilde{\Gamma} \vdash \tilde{B}$.

   Suppose $\tilde{\Gamma} \nvdash \tilde{P}$. Then $\tilde{\Gamma}, \neg\tilde{P}$ is consistent, so by the previous step, there is an environment, η, such that $\eta \models^{\mathcal{T}} \Gamma, \neg P$, so $\Gamma \not\models^{\mathcal{T},\eta} P$, and $\Gamma \not\models P$.

   The situation for refinement typings can be reduced to that of propositions, since $\Gamma' \models^{\mathcal{T},\eta'} t : \phi$ is equivalent to $\Gamma' \models^{\mathcal{T},\eta'} t =_\phi t$. The crucial point is that the permissive well-formedness rule for equalities (**Equality**) means that $t =_\phi t'$ is well-formed even though t and t' need not have refinement type ϕ.

   Then, $\Gamma \models t =_\phi t$ implies $\tilde{\Gamma} \vdash t \simeq_\phi t$ so, by Lemma 4.5.12, $\tilde{\Gamma} \vdash t \sim \phi$.

∎

In order to prove the equivalence in step 3, we need to characterise the interpretation of expressions in the term model, $\mathcal{T}$. Because of the mutual recursion between terms, refinement types and propositions, we must carry out the proof for each syntactic category simultaneously.

For the reasons given in Remark 4.3.21, the pers, $[\![\phi]\!]^{\mathcal{T}}$, do not correspond exactly to the equalities, $=_\phi$.

Lemma 4.5.17 *Let Γ be a $\lambda_{(:)}$-context and define Γ_∞ as in the proof of Theorem 4.5.16. Then, for $\eta \models^{\mathcal{T}} \Gamma$:*

1. *For $\Gamma_\infty \vdash \tilde{\phi}[\eta/\Gamma]$ $\texttt{wf}$, $[u]\ \ [\![\Gamma \vdash \phi]\!]^{\mathcal{T}}(\eta')\ [u'] \iff \Gamma_\infty \vdash u \simeq_{\tilde{\phi}[\eta/\Gamma]} u'$*
2. *For $\Gamma_\infty \vdash \tilde{t}[\eta/\Gamma]$ $\texttt{wf}$, $[\![\Gamma \vdash t]\!]^{\mathcal{T}}(\eta) = \{[u] \mid u\ \texttt{sat}^{\Gamma_\infty}\ \tilde{t}[\eta/\Gamma]\}$*
3. *For $\Gamma_\infty \vdash \tilde{P}[\eta/\Gamma]$ $\texttt{wf}$, $\Gamma \models^{\mathcal{T},\eta} P$ iff $\Gamma_\infty \vdash \tilde{P}[\eta/\Gamma]$*

Proof: Simultaneous induction over all expressions, unpacking the definition in the term model. The inductive ordering is

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

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

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

$$t < F(t)$$

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

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

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

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

The interesting cases are for propositions so we prove these in detail. To save on symbols, we will write $\overline{U}$ for $\tilde{U}[\eta/\Gamma]$.

- $\Gamma \not\models^{\mathcal{T},\eta} \bot$, and $\Gamma_\infty \nvdash \bot$ by construction of Γ_∞.
- $\Gamma \models^{\mathcal{T},\eta} P \supset Q \iff \Gamma \not\models^{\mathcal{T},\eta} P$ or $\Gamma \models^{\mathcal{T},\eta} Q \iff \Gamma_\infty \vdash \overline{\neg P}$ or $\Gamma_\infty \vdash \overline{Q} \iff \Gamma_\infty \vdash \overline{P \supset Q}$.
- $\Gamma \models^{\mathcal{T},\eta} \forall x : \phi.P$ when for all $a \in [\![\Gamma \vdash \phi]\!](\eta)$, $\langle \eta, a\rangle \in [\![\Gamma, x : \phi \vdash P]\!]$. Now if $\Gamma_\infty \vdash t : \overline{\phi}$, by the inductive hypothesis on ϕ we have $\Gamma \models^{\mathcal{T},\eta} t : \phi$, and so for each $a \in [\![\Gamma \vdash t]\!](\eta)$, $\langle \eta, a\rangle \in [\![\Gamma, x : \phi \vdash P]\!]$. Then, by Lemma 4.5.6, $\Gamma \models^{\mathcal{T},\eta} P[t/x]$, and by the inductive hypothesis on $P[t/x]$, $\Gamma_\infty \vdash \bar{P}[t/x]$. In other words, for all $\Gamma_\infty \vdash t : \overline{\phi}$ we have $\Gamma_\infty \vdash \overline{P}[t/x]$. Thus $\Gamma_\infty \vdash \overline{\forall x : \phi.P}$, for if not, by maximality we would have $\Gamma_\infty \vdash \overline{\exists x : \phi.\neg P}$, and since Γ_∞ is a Henkin theory, $\Gamma_\infty \vdash \neg\overline{P}$ contradicting the above.

  Conversely, suppose $\Gamma_\infty \vdash \overline{\forall x : \phi.P}$. Now let $[u] \in [\![\Gamma \vdash \phi]\!](\eta)$. By the inductive hypothesis and Lemma 4.5.12, $\Gamma_\infty \vdash u \sim \overline{\phi}$, that is, there exists $t : \overline{\phi}$ such that $u \simeq_{\overline{\phi}} t$. Now by **Quantification**, $\Gamma_\infty \vdash \overline{P}[t/x]$ so by the inductive hypothesis, $\Gamma \models^{\mathcal{T},\eta} P[t/x]$, so $\langle \eta, u\rangle \in [\![\Gamma \vdash P]\!]$. Hence $\Gamma \models^{\mathcal{T},\eta} \forall x : \phi.P$.
- $\Gamma \models^{\mathcal{T},\eta} F(t)$ when $[\![\Gamma \vdash t]\!](\eta) \subseteq F^{\mathcal{T}}$. Since t is well-formed we have $\Gamma \vdash t : \phi$.

  Define the maximal refinement type over τ, $max(\tau)$, as:

  $$max(\mathbf{1}) = \mathbf{1}$$

$$max(\gamma) = \gamma$$
$$max(\sigma \times \tau) = max(\sigma) \times max(\tau)$$
$$max(\sigma \to \tau) = (x : \sigma)\bot \to \tau$$

Then all terms over τ have refinement type $max(\tau)$. Let χ be $max(\tau)$, where $\phi :$ `Ref` (τ). Then we have $x : \chi \vdash F(x)$ **`wf`** by the permissive well-formedness rule for predications. Let $[u] \in [\![\Gamma \vdash t]\!](\eta)$, so by induction on t, $u\ \mathtt{sat}^{\Gamma_\infty}\ \overline{t}$, and since $u : \chi$, and $u =_\chi \overline{t}$, by **Elimination Rules** we have $\Gamma_\infty \vdash F(\overline{t})$.

Conversely, if $\Gamma_\infty \vdash F(\overline{t})$, then $\Gamma_\infty \vdash t : (x : \phi)F(x)$ for some ϕ, so by the definition of `sat`, $\Gamma_\infty \vdash F(u)$ for all $u\ \mathtt{sat}^{\Gamma_\infty}\ \overline{t}$, and so $[\![\Gamma \vdash t]\!](\eta) \subseteq F^{\mathcal{T}}$.

- $\Gamma \vDash^{\mathcal{T},\eta} t =_\phi t'$ is: for all $u\ \mathtt{sat}^{\Gamma_\infty}\ \overline{t}$ and $u'\ \mathtt{sat}^{\Gamma_\infty}\ \overline{t'}$, we have $\Gamma_\infty \vdash u \simeq_{\overline{\phi}} u'$. We can show that this is equivalent to $\Gamma_\infty \vdash \overline{t} \simeq_{\overline{\phi}} \overline{t'}$, by induction over $\overline{\phi}$.

- Assume that $\Gamma \vDash^{\mathcal{T},\eta} \phi \sqsubseteq \phi'$. By induction on ϕ, ϕ', this is: for all u, u', if $u \simeq_{\overline{\phi'}} u'$ then $u \simeq_{\overline{\phi}} u'$.

  If $x : \overline{\phi'} \in \Gamma_\infty$ then $\Gamma_\infty \vdash x : \overline{\phi}$ and we can deduce (see Remark 4.3.20) that $\Gamma_\infty \vdash \overline{\phi} \sqsubseteq \overline{\phi'}$.

  If $\overline{\phi'} \notin \Gamma_\infty$, then we can prove $\Gamma_\infty \vdash \overline{\phi} \sqsubseteq \overline{\phi'}$ directly. Since $\overline{\phi'} \notin \Gamma_\infty$ there does not exist a term $\Gamma_\infty \vdash t : \overline{\phi'}$. Suppose $\exists x : \overline{\phi'} \,.\, \top \in \Gamma_\infty$. Then since Γ_∞ is a $\lambda_{(:)}$-Henkin theory we must have $\Gamma_\infty \vdash t' : \overline{\phi'}$, for some t'. We deduce that there exists a $\Gamma_\infty \vdash t' : \overline{\phi'}$, a contradiction. Hence, since Γ_∞ is maximally consistent, we must have $\neg\exists x : \overline{\phi'} \,.\, \top \in \Gamma_\infty$, and so $\Gamma_\infty, x : \overline{\phi'} \vdash \bot$. Hence by **Refinement Types (L)**, $\Gamma_\infty \vdash (x : \overline{\phi'})\bot \sqsubseteq \overline{\phi'}$. Now Proposition 4.3.19 gives us $\Gamma_\infty \vdash \overline{\phi} \sqsubseteq (x : \overline{\phi'})\bot$ and so using **Transitivity**, $\Gamma_\infty \vdash \overline{\phi} \sqsubseteq \overline{\phi'}$.

  Conversely, suppose $\Gamma_\infty \vdash \overline{\phi} \sqsubseteq \overline{\phi'}$, then by Lemma 4.5.10 and the inductive hypothesis on $\overline{\phi}$ and $\overline{\phi'}$, we get $\Gamma \vDash^{\mathcal{T},\eta} \overline{\phi} \sqsubseteq \overline{\phi'}$.

∎

Now since first-order logic, the first-order logic of simply-typed lambda calculus, and the refinement types calculus are all complete for the class of Henkin models (without the assumption of nonemptiness), we have:

Corollary 4.5.18 *The calculus is a conservative extension of the first-order logic of $\lambda^{\times\to}$: If $\Gamma \vdash t =_\tau t'$ is a well-formed equation in $\lambda^{\times\to}$, then it is provable in $\lambda^{\times\to}$, if and only if it is provable in the calculus of refinement types.*

Corollary 4.5.19 *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.

4.6 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.

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.

In the proof that `div2` satisfied its specification we used the proof for `div2'`. There is an implicit element of refinement on terms here. This is made explicit in the next chapter.

Chapter 5

Refinement Calculus

In this chapter, we present the full refinement calculus, $\lambda_{\sqsubseteq}$. This is a calculus in which the stepwise refinement of logical specifications into programs, and the correctness of partially developed programs can be formalised. The calculus combines the refinement terms and refinement types calculi of the previous two chapters.

We define a notion of refinement axiom system and a corresponding class of Henkin models with 'logical factoring'. We prove soundness of the calculus in these models, and prove completeness for a restricted fragment.

5.1 Introduction

Much of the intuition for specification and refinement has been presented in the previous two chapters. Let us recall the scenario in which we are studying refinement. We have a programming language and a program logic. In Chapter 3, we showed how to internalise a simple notion of partial development in a programming language, the terms of which, *refinement terms*, are a record of the stage of development towards a program. In Chapter 4, we studied how to construct a specification language from a program logic, the specifications being given as *refinement types*. These are orthogonal extensions to the programming language, which in our case is the simply-typed lambda calculus. We now combine these calculi to give a refinement calculus for the stepwise refinement of logical specifications into programs. This claim of orthogonality will be backed up in Section 5.5 below. We now discuss how the features of the subcalculi are combined. The main issue is combining the logic with refinement.

The central language construct is the logical stub. We write $?_\phi$, where ϕ is a refinement type, to denote some unknown program with refinement type ϕ, and combine such unknowns with the other language features as in Chapter 3. As in the simpler system, $\lambda_?$, we will refer to such terms as refinement terms. If ϕ expresses the properties of interest then a refinement will begin with $?_\phi$. In general, though, it can be useful to specify with a mixture of logic and program code. Of course, we are now at liberty to 'overspecify', and can write specifications which cannot be implemented, even when all types are inhabited. This is in contrast to $\lambda_?$, and has a bearing on the refinement rules of $\lambda_\sqsubseteq$.

One of the slogans of the refinement methodology is that "refinement is correctness-preserving". To make this clear we must have a notion of partially developed program, that is, refinement term, satisfying a property. In fact, one of the main reasons for keeping an explicit record of the stage of development is that we can draw inferences about partially developed programs. During the course of development, questions might arise of the form, "given that certain implementation steps have now been made, is the final program guaranteed to have a certain property?"

In Chapter 4, we formulated the satisfaction of specifications as programs having a refinement type. We generalise these rules from programs to partially developed programs, that is, we generalise the rules for proving $r : \tau$ and $t : \phi$ to proving $r : \phi$. The main technical problem here is combining underdetermined terms with the logic. Since we cannot substitute arbitrary terms for variables we cannot infer that $r : (x : \phi)P$ from $r : \phi$ and $P[r/x]$. Intuitively, we must show that $r : \phi$ and that every determined term t to which r refines, $P[t/x]$ holds. Let us write this second fact as "$\forall x \in r.P$". It is cumbersome to prove quantifications like this, however, so observe that we can preserve the truth of such quantifications with rules like

$$\frac{\forall x \in r.P \quad \forall y \in r'.Q}{\forall z \in \langle r, r' \rangle.P[\pi_1 z/x] \wedge Q[\pi_2 z/y]}$$

In other words, we can use the refinement type methodology for proving that an arbitrary r has refinement type ϕ, without ever substituting r directly in a proposition.

In Chapter 3, refinement was the decomposition of stubs and their replacement with code. In Chapter 4, the idea was that the refinement of refinement types formalised logical manipulations of specifications. In the combined refinement calculus we combine these two distinct aspects of refinement, by adding a rule that if ϕ refines to ϕ' then $?_\phi$ refines to $?_{\phi'}$. Moreover, just as equality in $\lambda_{(:)}$ is defined with respect to a refinement type, we now extend this idea to refinement at a refinement type, and write the refinement of r to r' at ϕ as $r \sqsubseteq_\phi r'$.

Hence, there are two complementary aspects to refinement in $\lambda_\sqsubseteq$, corresponding to the forms of refinement in the two subcalculi. We can manipulate specifications and replace them with something more specific. This corresponds to the refinement of refinement types in $\lambda_{(:)}$, $\phi \sqsubseteq_\tau \phi'$. We can also decompose specifications and replace 'holes' in refinement terms with code, and this is formalised by the refinement of refinement terms, $r \sqsubseteq_\phi r'$. One difference from the $\lambda_?$-calculus is that in $\lambda_\sqsubseteq$ refinement is under a context with logical assumptions, as given in $\lambda_{(:)}$.

Sometimes the particular ϕ at which the refinement is carried out is not important, so we define a notion of 'nonlogical refinement' (on refinement terms), $r \sqsubseteq r'$, meaning: for all ϕ, if $r : \phi$ then $r \sqsubseteq_\phi r'$. We will use this notion in the formal system. One of the main results of this chapter will be the factorisation of $\sqsubseteq_\phi$ into $\sqsubseteq$ and $=_\phi$.

In Chapter 3, we saw that refinement was a sequence of implementation steps terminating in a program, *i.e.* a term of $\lambda^{\times\to}$. In the refinement calculus with refinement types, it is more natural to refine to a term of $\lambda_{(:)}$. In particular, we take $\lambda x : \phi.t$ to be determined. This is a natural choice when we only consider terms modulo some refinement type.

Although our notion of program now is not some unique term of $\lambda^{\times\to}$, it *is* unique up to the equality of some refinement type. It is in the spirit of refinement to only refine as far as is necessary. We can always give a term of $\lambda^{\times\to}$ by replacing the refinement types in a $\lambda_{(:)}$-term by their underlying types.

5.2 The Calculus

Following the pattern of previous chapters, we give the syntax of the calculus, and the judgement classes. Next we define the notion of $\lambda_\sqsubseteq$-axiom system, and give the rules of the calculus.

5.2.1 Syntax

We define an applied refinement calculus by first giving a signature of ground types, constants and predicate symbols. The terms are generated from the same basic signature as in the $\lambda_{(:)}$-calculus.

Definition 5.2.1 *A* $\lambda_\sqsubseteq$-signature $Sg = \langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$ *consists of:*

- *a collection, $\mathcal{G}$, of ground types (ranged over by γ)*
- *a collection, $\mathcal{K}$, of constants (ranged over by k), each of which has an arity n and sort $\tau_1, \ldots, \tau_n \to \tau$, which we write as $k : \tau_1, \ldots, \tau_n \to \tau$.*

- *a collection, $\mathcal{F}$, of predicate symbols (ranged over by F) each of which has an arity n and sort $\tau_1, \ldots, \tau_n$, which we write as $F : \mathtt{Pred}\,(\tau_1, \ldots, \tau_n)$.*

Definition 5.2.2 *Let $Sg = \langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$ be a $\lambda_{\sqsubseteq}$-signature. The pre-expressions over Sg are generated by the grammar:*

$$
\begin{array}{rcl}
\phi & ::= & \mathbf{1} \mid \gamma \mid \Sigma_{x:\phi}\psi \mid \Pi_{x:\phi}\psi \mid (x:\phi)P \\
r & ::= & x \mid k(r_1, \ldots, r_n) \mid * \mid \langle r, r' \rangle \mid \lambda x:\phi.r \mid ?_\phi \mid \pi_1(r) \mid \pi_2(r) \mid \\
 & & rr' \mid \mathtt{let}\ x:\phi\ \mathtt{be}\ r\ \mathtt{in}\ r' \\
P & ::= & \bot \mid P \supset P' \mid \forall x:\phi.P \mid F(r_1, \ldots, r_n) \mid r \sqsubseteq_\phi r'
\end{array}
$$

The pre-contexts are:

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

As in Chapter 4, we write $\phi \times \psi$ and $\phi \to \psi$ for $\Sigma_{x:\phi}\psi$ and $\Pi_{x:\phi}\psi$, respectively, when $x \notin FV(\psi)$. We also abbreviate the assumption $x : (x:\phi)P$ as $x : \phi | P$.

Refinement types have the same meaning as in $\lambda_{(:)}$, and correspond to a partial equality over an underlying type.

The specification construct is the logical stub, $?_\phi$, for each refinement type, ϕ, meaning 'some unknown of refinement type ϕ'. The stub also carries the data of when concrete implementations are to be regarded as equal (that is, up to ϕ). This will be made clearer in Section 5.6. In general, $\lambda_{\sqsubseteq}$-terms can be thought of as specifying a collection of programs, up to some equivalence.

We say that a term is *determined* if it contains no stubs, and otherwise is *underdetermined*. We use the metavariable t to range over determined terms, and r over arbitrary refinement terms.

Refinement types also appear in the two binding constructs — abstractions and let-terms. This is useful for specification and refinement. When refining the body of the abstraction, $\lambda x:\phi.r$, the information in ϕ can be used. The term itself can be thought of as a specification of programs which only constrains the result for arguments in ϕ. We regard $\lambda x:\phi.t$ as being determined even though, in general, it does not uniquely specify a program in $\lambda^{\times\to}$. These programs are unique up to the equality of some refinement type, however, and we can always give a canonical example by replacing refinement types with the underlying types.

The let-term $\mathtt{let}\ x:\phi\ \mathtt{be}\ r\ \mathtt{in}\ r'$ is a description of some y in $r'[x]$ for some x in r, where y is only specified up to ϕ. For example, the term

$$\mathtt{let}\ f:\mathtt{even} \to \mathtt{nat}\ \mathtt{be}\ \lambda n:\mathtt{nat}.n\ \mathtt{in}\ f$$

specifies the $\mathtt{even} \to \mathtt{nat}$ class which contains $\lambda n:\mathtt{nat}.n$.

The other terms have much the same meaning as in Chapter 3. As in Chapter 4, the interaction between well-formedness and logical reasoning means that we cannot define well-formedness until we give the rules of the calculus, and that axioms are not assumed to be well-formed until their use in a proof.

5.2.2 Judgements

The refinement calculus, $\lambda_{\sqsubseteq}$, consists of two basic judgements.

$$\Gamma \vdash r : \phi$$

$$\Gamma \vdash P$$

where the propositions include refinement of terms, $\Gamma \vdash r \sqsubseteq_{\phi} r'$, and of refinement types, $\Gamma \vdash \phi \sqsubseteq_{\tau} \phi'$. There are also well-formedness judgements

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

$$\Gamma \vdash \phi : \ \texttt{Ref}\,(\tau)$$

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

The judgements extend those of $\lambda_{(:)}$ in Chapter 4, and have similar intuitive readings. As there, we will use g as a metavariable for syntactic environments, but we use tuples of *determined* terms, as in $\lambda_{?}$.

5.2.3 $\lambda_{\sqsubseteq}$-Axiom Systems

We adopt the same definition of axiom system as in the $\lambda_{(:)}$-calculus.

Definition 5.2.3 *A* $\lambda_{\sqsubseteq}$-axiom system *consists of a* $\lambda_{\sqsubseteq}$*-signature* Sg *and a collection of* $\lambda_{(:)}$*-axioms* Ax *formed from pre-contexts and pre-expressions in* Sg*. Axioms are of two forms:*

- *propositions in context,* $\Gamma \vdash P$
- *axioms for constants,* $\Gamma \vdash k : \phi_1, \ldots, \phi_n \to \psi$*.*

The comments following Definition 4.3.5 for $\lambda_{(:)}$ are relevant here too. The restriction of axioms to the $\lambda_{(:)}$-fragment is a natural restriction to disallow refinements as axioms. Moreover, this ensures certain metatheoretic properties.

5.2.4 Rules of the Calculus

In Figure 5.1 we summarise the different forms of judgement in the $\lambda_{\sqsubseteq}$-calculus. We can make a basic division into judgements of well-formedness for each syntactic category, and judgements of truth. The division into well-formedness and truth is somewhat arbitrary as all judgements involve logical reasoning, and the refinement typings formalise both well-formedness of terms and the satisfaction of specifications. The upward arrow in Figure 5.1 indicates inclusion of rules. We do not make refinement a separate judgement class from the other propositions. Similarly, the equality rules are just mutual refinements.

Definition 5.2.4 *Let $\langle Sg, Ax\rangle$ be a $\lambda_{\sqsubseteq}$-axiom system. We define the theorems of $\langle Sg, Ax\rangle$ to be the judgements which can be inferred from the rules in Figures 4.2, 4.3, and 5.2 to 5.12. We write $\langle Sg, Ax\rangle \triangleright J$ when the judgement J is provable from the $\lambda_{\sqsubseteq}$-axiom system $\langle Sg, Ax\rangle$. We drop the Sg and Ax when they are obvious and just write J, meaning 'J is provable'.*

As for the $\lambda_{(:)}$-calculus, we consider the provable well-formedness judgements to be theorems too.

The well-formedness rules for contexts and refinement types are given in Figures 4.2 and 4.3 in Chapter 4. The well-formedness rules for propositions are the natural extensions of those in Chapter 4, with the additional rule that the well-formedness of the refinement $r \sqsubseteq_{\phi} r'$ requires that r, r' and ϕ have the same underlying type; they are given in Figure 5.3.

The refinement typing rules in Figures 5.4 and 5.5 are the obvious generalisations of those in Chapter 4, with side-conditions on the elimination rules to ensure that we do not substitute underdetermined terms in refinement types (see Remark 5.2.5 below). For example, the elimination rule for **Function Terms** has the hypothesis $\Gamma \vdash r : \phi \to \psi$ which abbreviates $\Gamma \vdash r : \Pi_{x:\phi}\psi$ with the side condition that $x \notin FV(\psi)$.

There are also rules for logical stubs and let-terms.

A special case of the introduction rule for **Product Terms** is:

$$\frac{\Gamma \vdash r : \phi \quad \Gamma \vdash r' : \psi}{\Gamma \vdash \langle r, r'\rangle : \phi \times \psi}$$

The connection between the logic and refinement typing lies in the two rules for **Refinement Type Introduction** in Figure 5.5. The first rule is actually derivable (as in Chapter 4) but is natural to include. It does not generalise to a rule for arbitrary underdetermined terms, however, since in general, r having

Well-formedness	$\vdash \Gamma\ \mathtt{wf}$	Figure 4.2	
	$\Gamma \vdash P\ \mathtt{wf}$	5.3	
	$\Gamma \vdash \phi : \mathtt{Ref}\ (\tau)$	4.3	
	$\Gamma \vdash r : \phi$	5.4, 5.5	Truth
	$\Gamma \vdash P$	5.12	
	$\uparrow$		
Refinement	$\Gamma \vdash \phi \sqsubseteq_\tau \phi'$	5.11	
	$\Gamma \vdash r \sqsubseteq_\phi r'$	5.9, 5.10	
	$\uparrow$		
	$\Gamma \vdash t =_\phi t'$	5.6	Equality
	$\Gamma \vdash r =_\phi r'$	5.7, 5.8	

Figure 5.1: Summary of Judgements in the Refinement Calculus

refinement type $(x : \phi)P$ can not be encoded as the proposition $P[r/x]$, as discussed in Section 5.1. This is the case, though, for determined terms and for predications.

The **Refinement Elimination** rule is the generalisation of the **Equality** rule on p. 105. This is so that refinements can be used to infer refinement typings.

The related rule

$$\frac{\Gamma \vdash r \sqsubseteq_\phi r'}{\Gamma \vdash r' : \phi}$$

is admissible (being a special case of Subject Refinement, Lemma 5.5.2). It would be unnatural to take this rule as primitive and use it in proving refinement typings because this would require guessing the term r.

It is to make Subject Refinement admissible that the $\lambda_\sqsubseteq$-calculus has stronger rules for abstractions and let-terms than might be expected:

$$\frac{\Gamma, x : \phi, P \vdash r : \psi}{\Gamma \vdash \lambda x : \phi . r : \Pi_{x:\phi|P}\psi}$$

$$\frac{\Gamma \vdash r : (x : \phi)P \quad \Gamma, x : \phi, P \vdash r' : \chi}{\Gamma \vdash \mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ r' : \chi}\ (x \notin FV(\chi))$$

We can derive the obvious simpler forms by letting P be $\top$. We use $(x : \phi)P$ rather than the general $\phi \sqsubseteq \phi'$ for the same reason as in $\lambda_{(:)}$ (see p. 105). The corresponding rule for abstractions in $\lambda_{(:)}$ (inferring that $\lambda x : \phi.t : \Pi_{x:\phi|P'}\psi$) follows from the rule for equality elimination in $\lambda_{(:)}$.

The final rule for refinement typing is a **Weakening** rule. We need to add this because the axioms are not necessarily closed under weakening.

Figures 5.6 to 5.10 formalise refinement of terms. This includes the equality rules for determined terms in Figure 5.6, and the equality rules for let-terms in Figures 5.7 and 5.8, which are a straightforward extension of those in Chapter 3, replacing arbitrary types with arbitrary refinement types.

The rule for **Abstractions** is most conveniently given using nonlogical equality, that is, using mutual $\sqsubseteq$ (see p. 137). As mentioned on p. 137, $r \sqsubseteq r'$ is not a new judgement, as such, but rather a meta-judgement with the meaning: if r has refinement type χ, then r refines to r' at χ and r' has refinement type χ. We use different symbols to distinguish the abbreviation, $\hat{\sqsubseteq}$, from the meta-judgement, $\sqsubseteq$, for clarity's sake.

Formally, we can write $r \hat{\sqsubseteq} r'$ in the conclusions of rules, where

$$\frac{\Gamma \vdash J}{\Gamma \vdash r \hat{\sqsubseteq} r'}$$

abbreviates the schemas (for well-formed ϕ):

$$\frac{\Gamma \vdash J \quad \Gamma \vdash r : \phi}{\Gamma \vdash r \sqsubseteq_\phi r'}$$

$$\frac{\Gamma \vdash J \quad \Gamma \vdash r : \phi}{\Gamma \vdash r' : \phi}$$

Superficially, the **Abstractions** rule is stronger than the form with refinement types:

$$\frac{\Gamma, x : \phi, y : \psi \vdash r[x,y] : \psi'}{\begin{array}{c}\Gamma \vdash \texttt{let } z : \phi \to \psi \texttt{ be } ?_{\phi\to\psi} \texttt{ in } \lambda x : \phi.r[x, zx] \\ =_{\phi\to\psi'} \ \lambda x : \phi.(\texttt{let } y : \psi \texttt{ be } ?_\psi \texttt{ in } r[x,y])\end{array}} \ (x \notin FV(\psi))$$

though they may be equivalent. In the absence of a proof of equivalence we adopt the former for technical reasons. (We will need this for the Generation Lemma below.)

There is no primitive rule for stubs (though see Chapter 6). We can derive:

$$\texttt{let } x : \phi \texttt{ in } ?_{(y:\tau)\ Q[x,y]} = ?_{(y:\tau)\ \exists x:\phi.Q[x,y]}$$

The other refinement rules for terms are listed in Figures 5.9 and 5.10. It is the rule for **Stubs** that allows the refinement of refinement types inside terms, and formalises the interaction between the two forms of refinement:

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash ?_{\phi} \sqsubseteq_{\phi} ?_{\phi'}}$$

There is a weakening rule, **Refinement Weakening**, and a strengthening rule, **Logical Congruence**.

Finally, Figure 5.11 lists the rules for refinement of refinement types, and Figure 5.12 extends the first-order logic of Chapter 4 with one additional rule of **Predicates**:

$$\frac{\Gamma, x : \phi \vdash F(x) \quad \Gamma \vdash r : \phi}{\Gamma \vdash F(r)}$$

This lets us substitute an arbitrary r directly into a predication so we can derive

$$\frac{\Gamma \vdash r : (x : \phi)F(x)}{\Gamma \vdash F(r)}$$

We have a **Refinement Type Introduction** rule in $\lambda_{\sqsubseteq}$ for the converse of this:

$$\frac{\Gamma \vdash r : \phi \quad \Gamma \vdash F(r)}{\Gamma \vdash r : (x : \phi)F(x)}$$

In addition, we must add a rule for a limited form of subject refinement:

$$\frac{\Gamma \vdash r : (x : \phi)F(x) \quad \Gamma \vdash r \sqsubseteq_{\phi} r'}{\Gamma \vdash r' : (x : \phi)F(x)}$$

We would have expected this rule to at least be admissible, but if $F(r)$ is assumed in the context there seems no other way to conclude $F(r')$. However, it seems that if we can prove $F(r)$ directly, then we can prove $F(r')$ without using this rule, so the rule is only necessary in this one case.

Hence $\lambda_{\sqsubseteq}$ has two refinement type introduction rules and two elimination rules for arbitrary underdetermined terms. The introduction and elimination rules for determined terms can be derived.

Remark 5.2.5 In Chapter 3 we introduced let-terms since we cannot substitute arbitrary terms for variables in terms. Similarly, we cannot substitute arbitrary terms for variables in refinement types. If $r : \Pi_{x:\phi}\psi$ and $r' : \phi$ then it is not the case that $rr' : \psi[r/x]$. However, instead of carrying out a similar extension for refinement types here, we make a restriction on the elimination rules for function and product terms so that this problem does not arise. This is discussed further in Chapter 6.

Axioms

$$\frac{\Gamma \vdash P \ \texttt{wf}}{\Gamma \vdash P} \ (\Gamma \vdash P \ \in \ Ax)$$

Weakening

$$\frac{\Gamma_1, \Gamma_2 \vdash B \quad \Gamma_1 \vdash \phi \ \texttt{wf}}{\Gamma_1, x : \phi, \Gamma_2 \vdash B}$$

Permutation

$$\frac{\Gamma_1, x_1 : \phi_1, \Gamma_2, x_2 : \phi_2, \Gamma_3 \vdash B \quad \Gamma_1 \vdash \phi_2 \ \texttt{wf}}{\Gamma_1, x_2 : \phi_2, \Gamma_2, x_1 : \phi_1, \Gamma_3 \vdash B} \ (x_1 \notin \Gamma_2, x_2 : \phi_2)$$

Substitution

$$\frac{\Gamma, x : \phi \vdash B \quad \Gamma \vdash t : \phi}{\Gamma \vdash B[t/x]}$$

Figure 5.2: Theorems Generated from a $\lambda_{\sqsubseteq}$-Axiom System $\langle Sg, Ax \rangle$

Example 5.2.6 The $\lambda_{(:)}$-axiom systems for booleans and naturals from Section 4.3.5 serve also as $\lambda_{\sqsubseteq}$-axiom systems so we do not repeat them here. The main point to be made here is (as for $\lambda_{?}$) that we do not need special refinement rules for particular constants. This is important because it means that if we extend the theory with new constants, we need only add equations for determined terms; refinement rules will be automatic from the general rules already in the calculus.

For constant $k : \phi_1, \ldots, \phi_n \to \psi$, there is one refinement rule (omitting the well-formedness hypotheses):

$$\Gamma \vdash ?_\psi \sqsubseteq_\psi k(?_{\phi_1}, \ldots, ?_{\phi_n})$$

For example, since

$$\texttt{if \_ then \_ else \_} : P + P',\ (x : \phi)\, P \supset Q[x],\ (y : \phi)\ P' \supset Q[y] \to (z : \phi)\, Q[z]$$

we have

$$?_{(z:\phi) Q} \sqsubseteq \texttt{if } ?_{P+P'} \texttt{ then } ?_{(x:\phi)\, P \supset Q[x]} \texttt{ else } ?_{(y:\phi)\ P' \supset Q[y]}$$

Now, if $\Gamma, P \vdash ?_{(x:\phi)\, Q[x]} \sqsubseteq r$, then $\Gamma \vdash ?_{(x:\phi)\, P \supset Q[x]} \sqsubseteq r$, so we have an admissible rule for refining to conditionals:

$$\frac{\Gamma, P \vdash ?_{(x:\phi)\, Q[x]} \sqsubseteq r \quad \Gamma, P \vdash ?_{(y:\phi)\, Q[y]} \sqsubseteq r'}{\Gamma \vdash ?_{(z:\phi) Q} \sqsubseteq \texttt{if } ?_{P+P'} \texttt{ then } r \texttt{ else } r'}$$

Falsehood

$$\frac{\vdash \Gamma\ \texttt{wf}}{\Gamma \vdash \bot\ \texttt{wf}}$$

Implication

$$\frac{\Gamma \vdash P\ \texttt{wf} \quad \Gamma \vdash P'\ \texttt{wf}}{\Gamma \vdash P \supset P'\ \texttt{wf}}$$

Universal Quantification

$$\frac{\Gamma, x : \phi \vdash P\ \texttt{wf}}{\Gamma \vdash \forall x : \phi . P\ \texttt{wf}}$$

Predication

$$\frac{\begin{array}{l}\Gamma \vdash r_1 : \phi_1\ \cdots\ \Gamma \vdash r_n : \phi_n \\ \Gamma \vdash \phi_1 :\ \texttt{Ref}\,(\tau_1)\ \cdots\ \Gamma \vdash \phi_n :\ \texttt{Ref}\,(\tau_n)\end{array}}{\Gamma \vdash F(r_1, \ldots, r_n)\ \texttt{wf}} \qquad (F :\ \texttt{Pred}\,(\tau_1, \ldots, \tau_n) \in \mathcal{F})$$

Refinement

$$\frac{\Gamma \vdash r : \psi \quad \Gamma \vdash r' : \psi' \quad \Gamma \vdash \phi, \psi, \psi' :\ \texttt{Ref}\,(\tau)}{\Gamma \vdash r \sqsubseteq_\phi r'\ \texttt{wf}}$$

$$\frac{\Gamma \vdash \phi :\ \texttt{Ref}\,(\tau) \quad \Gamma \vdash \phi' :\ \texttt{Ref}\,(\tau)}{\Gamma \vdash \phi \sqsubseteq_\tau \phi'\ \texttt{wf}}$$

Figure 5.3: Well-formedness of Propositions

Variables

$$\frac{\vdash \Gamma, x : \phi, \Gamma' \ \mathtt{wf}}{\Gamma, x : \phi, \Gamma' \vdash x : \phi}$$

Constants

$$\frac{\begin{array}{l}\Gamma \vdash \phi_1 : \ \mathtt{Ref}\,(\tau_1) \ \cdots \ \Gamma \vdash \phi_n : \ \mathtt{Ref}\,(\tau_n) \\ \Gamma \vdash \psi : \ \mathtt{Ref}\,(\tau) \\ \Gamma \vdash r_1 : \phi_1 \ \cdots \ \Gamma \vdash r_n : \phi_n\end{array}}{\Gamma \vdash k(r_1, \ldots, r_n) : \psi} \quad \begin{cases}\Gamma' \vdash k : \phi_1, \ldots, \phi_n \to \psi \ \in \ Ax; \\ k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K}; \\ \Gamma' \subseteq \Gamma\end{cases}$$

Unit

$$\frac{\vdash \Gamma \ \mathtt{wf}}{\Gamma \vdash * : \mathbf{1}}$$

Stubs

$$\frac{\Gamma \vdash \phi \ \mathtt{wf}}{\Gamma \vdash ?_\phi : \phi}$$

Product Terms

$$\frac{\Gamma \vdash r : \phi \quad \Gamma, x : \phi \vdash r' : \psi}{\Gamma \vdash \mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ \langle x, r' \rangle : \Sigma_{x:\phi}\psi}$$

$$\frac{\Gamma \vdash r : \phi \times \psi}{\Gamma \vdash \pi_1(r) : \phi} \qquad \frac{\Gamma \vdash r : \phi \times \psi}{\Gamma \vdash \pi_2(r) : \psi}$$

Function Terms

$$\frac{\Gamma, x : \phi, P \vdash r : \psi}{\Gamma \vdash \lambda x : \phi . r : \Pi_{x:\phi|P}\psi}$$

$$\frac{\Gamma \vdash r : \phi \to \psi \quad \Gamma \vdash r' : \phi}{\Gamma \vdash r r' : \psi}$$

Let Terms

$$\frac{\Gamma \vdash r : (x : \phi)P \quad \Gamma, x : \phi, P \vdash r' : \psi}{\Gamma \vdash \mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ r' : \psi} \ (x \notin FV(\psi))$$

Figure 5.4: Refinement Typings

Refinement Type Introduction

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

$$\frac{\Gamma \vdash r : \phi \quad \Gamma \vdash F(r)}{\Gamma \vdash r : (x : \phi)F(x)}$$

$$\frac{\Gamma \vdash r : (x : \phi)F(x) \quad \Gamma \vdash r \sqsubseteq_{\phi} r'}{\Gamma \vdash r' : (x : \phi)F(x)}$$

Refinement Elimination

$$\frac{\Gamma \vdash r \sqsubseteq_{\phi} r'}{\Gamma \vdash r : \phi}$$

Weakening

$$\frac{\Gamma \vdash r : \phi' \quad \Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash r : \phi}$$

Figure 5.5: Refinement Typings cont.

The derived refinement rule for `natwfrec` (the constant for well-founded recursion) is particularly interesting, as rules for 'recursive refinement' are central to many refinement calculi (see, for example, [Bun97], p. 46). The axiom is

$$\texttt{natwfrec} : (\Pi_{x:\texttt{nat}}(\Pi_{z<x}\phi[z]) \to \phi[x]) \;\to\; \Pi_{x:\texttt{nat}}\phi[x]$$

so, after simplifying the well-formedness hypotheses, the refinement rule is

$$\frac{\Gamma, x : \texttt{nat} \vdash \phi[x] \;\; \texttt{wf}}{\Gamma \vdash ?_{\Pi_{x:\texttt{nat}}\phi[x]} \sqsubseteq \texttt{natwfrec}\ (?_{\Pi_{x:\texttt{nat}}(\Pi_{z<x}\phi[z]) \to \phi[x]})}$$

We can then derive the rule of recursive (or 'circular') refinement:

$$\frac{\Gamma, n : \texttt{nat} \vdash ?_{\phi[n]} \sqsubseteq_{\phi[n]} t[n, \lambda m : \texttt{nat} \,|\, m < n.?_{\phi[m]}]}{\Gamma \vdash ?_{\Pi_{n:\texttt{nat}}\phi[n]} \sqsubseteq_{\Pi_{n:\texttt{nat}}\phi[n]} \texttt{natwfrec}\ (\lambda n.\lambda f.t[n, f])}$$

Compare this with the discussion of equality on p. 113. This rule is equivalent to (Intro. rec. func.) on p. 46 of [Bun97].

Remark 5.2.7 As mentioned in Remark 3.2.13, a general form of β-equality does not hold in the $\lambda_{\sqsubseteq}$-calculus; instead we have β-equality only for determined

Constant Equations

$$\frac{\begin{array}{l}\Gamma \vdash \phi_1 : \texttt{Ref}\,(\tau_1) \;\cdots\; \Gamma \vdash \phi_n : \texttt{Ref}\,(\tau_n) \\ \Gamma \vdash \psi : \texttt{Ref}\,(\tau) \\ \Gamma \vdash t_1 =_{\phi_1} t'_1 \;\cdots\; \Gamma \vdash t_n =_{\phi_n} t'_n\end{array}}{\Gamma \vdash k(t_1, \ldots, t_n) =_{\psi} k(t'_1, \ldots, t'_n)} \quad \begin{cases}\Gamma' \vdash k : \phi_1, \ldots, \phi_n \to \psi \;\in\; Ax\,; \\ k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K}; \\ \Gamma' \subseteq \Gamma\end{cases}$$

Function Equations

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

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

$$\frac{\Gamma, x : \phi \vdash P \ \texttt{wf} \quad \Gamma, x : \phi, P \vdash t =_{\psi} t'}{\Gamma \vdash \lambda x : (x : \phi)P.t =_{\Pi_{x:\phi|P}\psi} \lambda x : \phi.t'} \qquad (\xi)$$

Product Equations

$$\frac{\Gamma \vdash t_1 : \phi_1 \quad \Gamma \vdash t_2 : \phi_2}{\Gamma \vdash \pi_1\langle t_1, t_2\rangle =_{\phi_1} t_1} \qquad \frac{\Gamma \vdash t_1 : \phi_1 \quad \Gamma \vdash t_2 : \phi_2}{\Gamma \vdash \pi_2\langle t_1, t_2\rangle =_{\phi_2} t_2} \qquad (\beta)$$

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

Unit Equation

$$\frac{\Gamma \vdash t : \mathbf{1}}{\Gamma \vdash t =_{\mathbf{1}} *}$$

Figure 5.6: Equality Rules for Determined Terms

Let Beta

$$\frac{\Gamma \vdash t : \phi \quad \Gamma, x : \phi \vdash r : \psi}{\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ t\ \texttt{in}\ r =_{\psi[t/x]} r[t/x]}$$

Let Eta

$$\frac{\Gamma \vdash r : \phi}{\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ x =_{\phi} r}$$

Let Associativity

$$\frac{\Gamma \vdash r : \phi \quad \Gamma, y : \phi \vdash r' : \psi \quad \Gamma, x : \psi \vdash r'' : \psi'}{\begin{array}{l}\Gamma \vdash \texttt{let}\ x : \psi\ \texttt{be}\ (\texttt{let}\ y : \phi\ \texttt{be}\ r\ \texttt{in}\ r'[y])\ \texttt{in}\ r''[x] \\ \qquad =_{\psi'} \texttt{let}\ y : \phi\ \texttt{be}\ r\ \texttt{in}\ (\texttt{let}\ x : \psi\ \texttt{be}\ r'[y]\ \texttt{in}\ r''[x])\end{array}}$$

Figure 5.7: Let Term Equalities

arguments and bodies. This is unlike refinement calculi based on nondeterminism (such as [Bun97] and [Mor94]). To illustrate this, let $n : \texttt{nat} \vdash \texttt{Fermat}(n) : \texttt{Ref}\,(\texttt{nat} \times \texttt{nat} \times \texttt{nat})$ be the specification of solutions to Fermat's Last Theorem at index n (*i.e.* tuples $\langle x, y, z\rangle$ such that $x^n + y^n = z^n$). Then, in contrast to $\lambda_{\sqsubseteq}$, [Bun97] and [Mor94] both have:

$$(\lambda n : \texttt{nat}.?_{\texttt{Fermat}(n)})2 = \texttt{Fermat}(2)$$

In $\lambda_{\sqsubseteq}$, the left hand side is unsatisfiable, whereas the right hand side is satisfiable.

This is similar to the situation with ASL and Extended ML. As observed in [SST92] (Section 4.3), the "principle of modular decomposition" means that if a module is decomposed into the application of a parameterised module to some other module, the parameterised module must be implemented for arbitrary arguments so as to be implementation independent, and *not* make use of particular properties of the actual argument. A similar point was made in Example 3.2.4. We discuss EML in more detail in Section 5.4.1 below.

Although the general β-equality does not hold in $\lambda_{\sqsubseteq}$, we do have the inequality, $r[t/x] \sqsubseteq (\lambda x : \phi.r)t$, for underdetermined r. This means that we can use the common programming technique of refining by first abstracting from a specific t, and then implementing recursively for a general argument. For example, $?_{\phi[t]} \sqsubseteq (\lambda x.?_{\phi[x]})t$, and we could then use recursive refinement to get $\texttt{natwfrec}\ (\lambda n.\lambda f.u[n, f])\ t$ for some u.

Constants

$$\frac{\Gamma \vdash r_1 : \phi_1 \ \cdots \ \Gamma \vdash r_n : \phi_n}{\begin{array}{l}\Gamma \vdash \texttt{let}\ x_1 : \phi_1, \ldots, x_n : \phi_n\ \texttt{be}\ r_1, \ldots, r_n \\ \qquad \texttt{in}\ k(x_1, \ldots, x_n) =_\psi k(r_1, \ldots, r_n)\end{array}} \ (\Gamma \vdash k : \phi_1, \ldots, \phi_n \to \psi \ \in \ Ax)$$

Applications

$$\frac{\Gamma \vdash r : \phi \to \psi \quad \Gamma \vdash r' : \phi}{\Gamma \vdash \texttt{let}\ x : \phi \to \psi, x' : \phi\ \texttt{be}\ r, r'\ \texttt{in}\ xx' =_\psi rr'}$$

Pairs

$$\frac{\Gamma \vdash r : \phi \quad \Gamma \vdash r' : \psi}{\Gamma \vdash \texttt{let}\ x : \phi, x' : \psi\ \texttt{be}\ r, r'\ \texttt{in}\ \langle x, x' \rangle =_{\phi \times \psi} \langle r, r' \rangle}$$

Projections

$$\frac{\Gamma \vdash r : \phi_1 \times \phi_2}{\Gamma \vdash \texttt{let}\ x : \phi_1 \times \phi_2\ \texttt{be}\ r\ \texttt{in}\ \pi_i(x) =_{\phi_i} \pi_i(r)}$$

Abstractions

$$\frac{\Gamma, x : \phi, y : \psi \vdash r[x, y] : \psi'}{\begin{array}{l}\Gamma \vdash \texttt{let}\ z : \Pi_{x:\phi}\psi\ \texttt{be}\ ?_{\Pi_{x:\phi}\psi}\ \texttt{in}\ \lambda x : \phi.r[x, zx] \\ \qquad \hat{=}\ \lambda x : \phi.(\texttt{let}\ y : \psi\ \texttt{be}\ ?_\psi\ \texttt{in}\ r[x, y])\end{array}}$$

Figure 5.8: Let Term Equalities cont.

Variables

$$\frac{\vdash \Gamma, x : \phi, \Gamma' \ \texttt{wf}}{\Gamma, x : \phi, \Gamma' \vdash ?_\phi \sqsubseteq_\phi x}$$

Constants

$$\frac{\Gamma \vdash \phi_1 : \ \texttt{Ref}\,(\tau_1) \ \cdots \ \Gamma \vdash \phi_n : \ \texttt{Ref}\,(\tau_n) \quad \Gamma \vdash \psi : \ \texttt{Ref}\,(\tau)}{\Gamma \vdash ?_\psi \sqsubseteq_\psi k(?_{\phi_1}, \ldots, ?_{\phi_n})} \quad \begin{cases} \Gamma' \vdash k : \phi_1, \ldots, \phi_n \to \psi \ \in \ Ax \ ; \\ k : \tau_1, \ldots, \tau_n \to \tau \in \mathcal{K}; \\ \Gamma' \subseteq \Gamma \end{cases}$$

Unit

$$\frac{\vdash \Gamma \ \texttt{wf}}{\Gamma \vdash ?_{\mathbf{1}} \sqsubseteq_{\mathbf{1}} *}$$

Pairs

$$\frac{\Gamma \vdash \phi \times \psi \ \texttt{wf}}{\Gamma \vdash ?_{\phi\times\psi} \sqsubseteq_{\phi\times\psi} \langle ?_\phi, ?_\psi \rangle}$$

Abstractions

$$\frac{\Gamma \vdash \Pi_{x:\phi}\psi \ \texttt{wf}}{\Gamma \vdash ?_{\Pi_{x:\phi}\psi} \sqsubseteq_{\Pi_{x:\phi}\psi} \lambda x : \phi.?_\psi}$$

Figure 5.9: Refinement Rules

Stubs

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash ?_\phi \sqsubseteq_\phi ?_{\phi'}}$$

Congruence

$$\frac{\Gamma, x : \phi \vdash P' \supset P \quad \Gamma, x : \phi, P' \vdash r \sqsubseteq_\psi r'}{\Gamma \vdash \lambda x : (x : \phi)P.r \sqsubseteq_{\Pi_{x:\phi|P'}\psi} \lambda x : \phi.r'}$$

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma \vdash r_1 \sqsubseteq_{\phi'} r_1' \quad \Gamma, x : \phi \vdash r_2 \sqsubseteq_\psi r_2'}{\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ r_1\ \texttt{in}\ r_2 \sqsubseteq_\psi \texttt{let}\ x : \phi'\ \texttt{be}\ r_1'\ \texttt{in}\ r_2'}\ (x \notin FV(\psi))$$

$$\frac{\Gamma, x : \phi \vdash P\ \texttt{wf} \quad \Gamma \vdash r_1 =_{(x:\phi)P} r_1' \quad \Gamma, x : \phi, P \vdash r_2 =_\psi r_2'}{\Gamma \vdash \texttt{let}\ x : (x : \phi)P\ \texttt{be}\ r_1\ \texttt{in}\ r_2 =_\psi \texttt{let}\ x : \phi\ \texttt{be}\ r_1'\ \texttt{in}\ r_2'}\ (x \notin FV(\psi))$$

Logical Congruence

$$\frac{\Gamma \vdash r \sqsubseteq_\phi r' \quad \Gamma \vdash r : (x : \phi)P}{\Gamma \vdash r \sqsubseteq_{(x:\phi)P} r'}$$

Reflexivity

$$\frac{\Gamma \vdash r : \phi}{\Gamma \vdash r \sqsubseteq_\phi r}$$

Transitivity

$$\frac{\Gamma \vdash r \sqsubseteq_\phi r' \quad \Gamma \vdash r' \sqsubseteq_\phi r''}{\Gamma \vdash r \sqsubseteq_\phi r''}$$

Let Weakening

$$\frac{\Gamma \vdash r' : \psi \quad \Gamma \vdash r : \phi}{\Gamma \vdash r' \sqsubseteq_\psi \texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ r'}\ (x \notin FV(r'))$$

Refinement Weakening

$$\frac{\Gamma \vdash r \sqsubseteq_{\phi'} r' \quad \Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash r \sqsubseteq_\phi r'}$$

Figure 5.10: Refinement Rules cont.

Unit

$$\frac{\vdash \Gamma\ \texttt{wf}}{\Gamma \vdash \mathbf{1} \sqsubseteq \mathbf{1}}$$

Product

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma, x : \phi \vdash \psi \sqsubseteq \psi'}{\Gamma \vdash \Sigma_{x:\phi}\psi \sqsubseteq \Sigma_{x:\phi'}\psi'}$$

Function

$$\frac{\Gamma \vdash \phi' \sqsubseteq \phi \quad \Gamma, x : \phi \vdash \psi \sqsubseteq \psi'}{\Gamma \vdash \Pi_{x:\phi}\psi \sqsubseteq \Pi_{x:\phi'}\psi'}$$

Refinement Types

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

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

Transitivity

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma \vdash \phi' \sqsubseteq \phi''}{\Gamma \vdash \phi \sqsubseteq \phi''}$$

Figure 5.11: Refinements on Refinement Types

Predicates

$$\frac{\Gamma, x : \phi \vdash F(x) \quad \Gamma \vdash r : \phi}{\Gamma \vdash F(r)}$$

Together with the rules in Figure 4.10, Chapter 4.

Figure 5.12: First-order Logic for the Refinement Calculus

Finally, it is interesting, in retrospect, that we *do* have the general β-equality for the $\lambda_?$ fragment, as discussed on p. 67. This means that, to a certain extent, we can evaluate such terms as though they were programs.

One other illustration of the difference between $\lambda_\sqsubseteq$ and two refinement calculi based on nondeterminism, [Bun97] and [War94], is that, as for call-by-value nondeterminism, arbitrary abstractions are considered to be values, *i.e.* nondeterministic functions.

5.3 An Example of Refinement

The example we will consider involves sorting association lists of keys and complex values. The idea behind association lists is that values of a complex datatype can be manipulated efficiently by pairing them with keys that encode some useful information.

We will develop two programs: first a sorting function and, then, a function which determines whether or not two lists are permutations. We use the axiom system for naturals and booleans given in Example 5.2.6 and extend it with an axiom system for keys, values and association lists (though we will only give some of the axioms).

The axiom system $\langle Sg_{\mathtt{Assoc}}, Ax_{\mathtt{Assoc}}\rangle$ is defined as follows. Let $Sg_{\mathtt{Assoc}} = \langle \mathcal{G}_{\mathtt{Assoc}}, \mathcal{K}_{\mathtt{Assoc}}, \mathcal{F}_{\mathtt{Assoc}}\rangle$, where

$$\mathcal{G}_{\mathtt{Assoc}} = \{\mathtt{key}, \mathtt{value}, \mathtt{assoclist}\}$$

$$\mathcal{K}_{\mathtt{Assoc}} = \{\mathtt{eq\_key}, \mathtt{compare\_key}, \mathtt{nil}, \mathtt{cons}, \mathtt{head}, \mathtt{tail}, \mathtt{remove}, \mathtt{listrec}\}$$

$$\mathcal{F}_{\mathtt{Assoc}} = \{\mathtt{Ordered}, \mathtt{In}, \mathtt{Sublist}\}$$

The ground types are `key`, `value` and `assoclist`. We define association pairs as $\mathtt{assocpair} = \mathtt{key} \times \mathtt{value}$.

There are efficient equality and comparison functions on keys:

$$\mathtt{compare\_key} : \mathtt{key}, \mathtt{key} \to \mathtt{bool}$$

$$\mathtt{eq\_key} : \mathtt{key}, \mathtt{key} \to \mathtt{bool}$$

The signature for lists is:

$$\mathtt{nil} : \mathtt{assoclist}$$

$$\mathtt{cons} : \mathtt{assocpair}, \mathtt{assoclist} \to \mathtt{assoclist}$$

$$\mathtt{head} : \mathtt{assoclist} \to \mathtt{assocpair}$$

$$\texttt{tail} : \texttt{assoclist} \to \texttt{assoclist}$$

$$\texttt{remove} : \texttt{assocpair}, \texttt{assoclist} \to \texttt{assoclist}$$

$$\texttt{listrec} : \tau, (\texttt{assoclist} \to \texttt{assocpair} \to \tau \to \tau), \texttt{assoclist} \to \tau$$

The constants `head` and `tail` are defined for all lists, including `nil`. However, the axioms do not say what the values at `nil` are. We use `listrec`, which is primitive recursion over lists (*fold left* in functional programming).

The axioms include:

$$\texttt{head} : (l : \texttt{assoclist})\, l \neq \texttt{nil} \;\to\; \texttt{assocpair}$$

$$\texttt{tail} : (l : \texttt{assoclist})\, l \neq \texttt{nil} \;\to\; \texttt{assoclist}$$

$$\texttt{listrec}\ (t, f, \texttt{nil}) = t$$

$$\texttt{listrec}\ (t, f, \texttt{cons}(x, xs)) = f\ xs\ x\ (\texttt{listrec}\ (t, f, xs))$$

$$l : \texttt{assoclist} \vdash$$
$$\texttt{listrec} : \phi[\texttt{nil}], \Pi_{l':\texttt{assoclist}}\Pi_{p:\texttt{assocpair}}\Pi_{x:\phi[l']}\phi[\texttt{cons}(p, l')], \{l\}_{\texttt{assoclist}} \;\to\; \phi[l]$$

We will use the predicate symbols:

$$\texttt{Ordered} :\ \texttt{Pred}\,(\texttt{assoclist}, \texttt{assocpair} \to \texttt{assocpair} \to \texttt{bool})$$

$$\texttt{In} :\ \texttt{Pred}\,(\texttt{assocpair}, \texttt{assoclist})$$

$$\texttt{Sublist} :\ \texttt{Pred}\,(\texttt{assoclist}, \texttt{assoclist})$$

with the axioms

$$\forall p : \texttt{assocpair}.\neg\texttt{In}(p, \texttt{nil}) \wedge \forall p.\forall x.\forall xs.\texttt{In}(p, x :: xs) \iff p = x \wedge \texttt{In}(p, xs)$$

$$\forall l'\,.\,\texttt{Sublist}(\texttt{nil}, l')$$

$$\forall x.\forall l.\forall l'\,.\,\texttt{Sublist}(\texttt{cons}(x, l), l') \iff \texttt{In}(x, l') \wedge \texttt{Sublist}(l, \texttt{remove}(x, l'))$$

$$\texttt{Ordered}(\texttt{nil}, <) \;\wedge\; x' < x \;\wedge\; \texttt{Ordered}(x :: xs, <) \supset \texttt{Ordered}(x' :: x :: xs, <)$$

where $\texttt{Ordered}(l, <)$ holds when the list l is sorted relative to ordering $<$, and $\texttt{In}(p, l)$ holds when the pair p is in the list l, that is, mathematically in, rather than in terms of the key.

For $l, l' : \texttt{assoclist}$, the proposition $\texttt{Perm}[l, l']$ is defined as:

$$\texttt{Perm}[l, l'] \equiv \texttt{Sublist}(l, l') \wedge \texttt{Sublist}(l', l)$$

We specify the ordering on association pairs as

$$\texttt{compare} : \texttt{assocpair} \to \texttt{assocpair} \to \texttt{bool}$$

$$\forall k, k' : \mathtt{key}.\forall v, v' : \mathtt{value}\,.$$
$$\mathtt{compare\_key}\ (k, k') = \mathtt{true} \supset \mathtt{compare}\ \langle k, v\rangle\ \langle k', v'\rangle = \mathtt{true}$$

For example,

$$\mathtt{compare} = \lambda a : \mathtt{assoclist}.\lambda a' : \mathtt{assoclist}.\mathtt{compare\_key}(\pi_1(a), \pi_1(a'))$$

and the sorting function as

$$\mathtt{sort\_spec}\ :\ \mathtt{Ref}\,(\mathtt{assoclist} \to \mathtt{assoclist})$$

$$\mathtt{sort\_spec} \equiv \Pi_{l:\mathtt{assoclist}}(l' : \mathtt{assoclist})\,\mathtt{Ordered}(l', \mathtt{compare})\ \wedge\ \mathtt{Perm}[l, l']$$

Note that this specification does not say what should happen when two values have the same key, but this does not matter for now. We remark that the semantics of $?_{\mathtt{sort\_spec}}$ is truly underdetermined.

Define $\phi[l]$ to be the specification 'is a sorting of l':

$$(l' : \mathtt{assoclist})\,\mathtt{Ordered}(l', \mathtt{compare})\ \wedge\ \mathtt{Perm}[l, l']\ :\ \mathtt{Ref}\,(\mathtt{assoclist})$$

The refinement begins as:

$$\begin{array}{l} ?_{\mathtt{sort\_spec}} \\ \quad \sqsubseteq\ \ \lambda l : \mathtt{assoclist}\,.\,?_{(l':\mathtt{assoclist})\,\mathtt{Ordered}(l',\mathtt{compare})\,\wedge\,\mathtt{Perm}[l,l']} \\ \quad \sqsubseteq\ \ \lambda l : \mathtt{assoclist}\,.\,\mathtt{listrec}\,(?_{\phi[\mathtt{nil}]}, ?_{\Pi_{l':\mathtt{assoclist}}\Pi_{p:\mathtt{assocpair}}\Pi_{x:\phi[l']}\phi[\mathtt{cons}(p,l')]}, ?_{\{l\}_{\mathtt{assoclist}}}) \\ \quad \sqsubseteq\ \ \lambda l : \mathtt{assoclist}\,.\,\mathtt{listrec}\,(\mathtt{nil}, ?_{\Pi_{l':\mathtt{assoclist}}\Pi_{p:\mathtt{assocpair}}\Pi_{x:\phi[l']}\phi[\mathtt{cons}(p,l')]}, l) \end{array}$$

We now plan the next stage of the implementation. One possibility is to construct a new list by systematically removing elements, and inserting them in the correct position in a new list. This is an *insertion sort*. We need to implement the specification

$$\Pi_{l':\mathtt{assoclist}}\Pi_{p:\mathtt{assocpair}}\Pi_{x:\phi[l']}\phi[\mathtt{cons}(p, l')]\ :\ \mathtt{Ref}(\mathtt{assoclist} \to \mathtt{assocpair} \to \tau \to \tau)$$

In fact, we do not need to use the first argument. The specification of an insertion function is (for $l' : \mathtt{assoclist}$):

$$\begin{array}{l} \mathtt{insert\_spec} \equiv \Pi_{p:\mathtt{assocpair}}\Pi_{x:\phi[l']}\phi[\mathtt{cons}(p, l')] \\ \qquad\qquad\qquad :\ \mathtt{Ref}\,(\mathtt{assocpair} \to \mathtt{assoclist} \to \mathtt{assoclist}) \end{array}$$

which says 'given $p : \mathtt{assocpair}$ and x a sorting of l', return a sorting of $\mathtt{cons}(p, l')$', that is, 'insert p in the correct position in the sorted list x'. Thus,

$$?_{\Pi_{l':\mathtt{assoclist}}\Pi_{p:\mathtt{assocpair}}\Pi_{x:\phi[l']}\phi[\mathtt{cons}(p,l')]} \sqsubseteq \lambda l' : \mathtt{assoclist}.?_{\mathtt{insert\_spec}}$$

We use the refinement rule for `listrec` again to refine `insert_spec`. For $m : \texttt{assoclist}$, let $\psi[m] \equiv \phi[\texttt{cons}(p, m)]$. We have

$$
\begin{array}{l}
?_{\texttt{insert\_spec}} \\
\ \sqsubseteq \quad \lambda p : \texttt{assocpair}.\lambda x : \phi[l'].?_{\psi[l']} \\
\ \sqsubseteq \quad \lambda p : \texttt{assocpair}.\lambda x : \phi[l'].\texttt{listrec}\ (?_{\psi[\texttt{nil}]}, ?_{\Pi_{m:\texttt{assoclist}}\Pi_{p':\texttt{assocpair}}\Pi_{y:\psi[m]}\psi[\texttt{cons}(p',m)]}, x) \\
\ \sqsubseteq \quad \lambda p : \texttt{assocpair}.\lambda x : \phi[l'].\texttt{listrec}\ (\texttt{cons}(p, \texttt{nil}), ?_{\Pi_{m:\texttt{assoclist}}\Pi_{p':\texttt{assocpair}}\Pi_{y:\psi[m]}\psi[\texttt{cons}(p',m)]}, x)
\end{array}
$$

Then since we can prove

$$
\begin{array}{l}
p : \texttt{assocpair}, x : \phi[l'], m : \texttt{assoclist}, p' : \texttt{assocpair}, y : \psi[m] \vdash \\
\qquad \texttt{if compare}\ p'\ p\ \texttt{then cons}(p', y)\ \texttt{else cons}(p, \texttt{cons}(p', m)) : \psi[\texttt{cons}(p', m)]
\end{array}
$$

we have

$$
\begin{array}{l}
p : \texttt{assocpair} \vdash ?_{\Pi_{m:\texttt{assoclist}}\Pi_{p':\texttt{assocpair}}\Pi_{y:\psi[m]}\psi[\texttt{cons}(p,m)]} \sqsubseteq \\
\qquad\qquad \lambda m : \texttt{assoclist}.\lambda p' : \texttt{assocpair}.\lambda y : \psi[m]\,. \\
\qquad\qquad\qquad\qquad \texttt{if compare}\ p'\ p\ \texttt{then cons}(p', y)\ \texttt{else cons}(p, \texttt{cons}(p', m))
\end{array}
$$

This is the only step of the refinement that generates a proof obligation.

We can now give the code for the sorting algorithm:

$$
\texttt{sort} = \lambda l : \texttt{assoclist}\,.\,\texttt{listrec}\ (\texttt{nil}, \lambda l' : \texttt{assoclist}.\texttt{insert}, l)
$$

where `insert` is

$$
\begin{array}{l}
\lambda p : \texttt{assocpair}.\lambda x : \phi[l']. \\
\qquad\qquad \texttt{listrec}\ ([p], \\
\qquad\qquad\qquad \lambda m : \texttt{assoclist}.\lambda p' : \texttt{assocpair}.\lambda y : \psi[m]\,. \\
\qquad\qquad\qquad\qquad \texttt{if compare}\ p'\ p\ \texttt{then cons}(p', y)\ \texttt{else cons}(p, \texttt{cons}(p', m)), x)
\end{array}
$$

We are guaranteed (by subject refinement; see Lemma 5.5.2) that $\texttt{sort} : \texttt{sort\_spec}$.

Now suppose we want to write a function to test whether two association lists are (true) permutations of each other, and so implement `Perm`. We can sort the lists using `sort` and compare corresponding entries using `compare_key`. This is not quite right, though, since as pointed out above, distinct values may have the same key and so end up with different relative orderings in separate lists. We would like to specify that our sorting is, in some sense, 'context independent', in the sense that two pairs in a list will always be sorted in the same order, no matter where they appear in a list. This can be done by strengthening the specification on `insert` so that in the case when two values have the same key, the insert function makes some predetermined choice dependent only on the value. We require

$$
\forall v, v' : \texttt{value}\ .\ v \neq v' \supset \mathit{choose}(v, v') \iff \neg\mathit{choose}(v', v)
$$

We specify

$$\texttt{insert}' : \texttt{assocpair} \to \texttt{assoclist} \to \texttt{assoclist}$$

$$\exists choose : \texttt{value} \to \texttt{value} \to \texttt{bool} \,.\, \forall p : \texttt{assocpair} \,.\, \forall l : \texttt{assoclist} \,. \\ \texttt{Ordered}(\texttt{insert}'\ p\ l, \texttt{compare}') \wedge \texttt{Perm}[\texttt{insert}\ p\ l, \texttt{cons}(p, l)]$$

where $\texttt{compare}'$ is a lexicographic ordering that first orders on the key, then on *choose*:

$$\begin{array}{l} \texttt{compare}'\ \langle k, v\rangle\ \langle k', v'\rangle = \\ \quad \texttt{if}\ (\texttt{compare\_key}(k, k') = \texttt{true})\ \texttt{then}\ \texttt{true}\ \texttt{else} \\ \qquad \texttt{if}\ (\texttt{eq\_key}(k, k') = \texttt{true})\ \texttt{then}\ choose\ v\ v'\ \texttt{else}\ \texttt{false} \end{array}$$

Thus, we leave it up to the implementer of $\texttt{insert}'$ to find some injective ordering, perhaps by exploiting implementation details of the values.

We can now define the permutation function as

$$\texttt{perm} : \texttt{assoclist} \to \texttt{assoclist} \to \texttt{bool}$$

$$\texttt{perm}\ l_1\ l_2 = \texttt{eqlist}\ (\texttt{sort}'\ l_1)\ (\texttt{sort}'\ l_2)$$

where `eqlist` is defined using `listrec`, and $\texttt{sort}'$ uses $\texttt{insert}'$.

Remark 5.3.1

The functions `head` and `tail` are *partial* in the sense that their results are not defined for the argument `nil`. In [vL90], various ways of accounting for partiality in algebraic specification are considered. There are two ways in which the simple approach of total algebras can be extended. On the one hand, we can extend the syntax of specifications; on the other, the algebra semantics.

With *error algebras*, the specification is augmented with error values at each type, together with predicates to distinguish between error and non-error values, and axioms to explain how errors are propagated. This complicates specifications considerably, but terms can still be interpreted as total functions.

An extension of this idea is to use *monotonic* (or *continuous*) algebras, where each type is axiomatised as a poset (or cpo). Again, terms are interpreted as total functions, but must be monotonic (or continuous) with respect to the orderings.

An alternative approach is to just change the definition of algebras, rather than the specifications. In a *partial algebra*, terms are interpreted as partial functions over the carrier sets. This necessitates a change in the definition of homomorphism, satisfaction, and so on.

An approach which alters both the notion of specification and algebra is *order-sorted algebras*. This is a form of subtype polymorphism with an ordering defined on the sorts. The idea is to give terms more specific sorts so they become total. For example, we can define $\texttt{Nonemptylist} \leq \texttt{List}$ and then have $\texttt{tail} : \texttt{Nonemptylist} \to \texttt{List}$.

The approach we have used here differs from each of these ideas. Although we have a form of subtyping, this is not used to constrain the domains of primitive functions. Instead, by regarding terms as representatives of equivalence classes we can abstract away irrelevant details. This seems a natural approach because, at the end of the day, we write total programs in the underlying programming language. We might say that such functions are *computationally* total but *specificationally* partial.

5.4 Comparisons

We compare $\lambda_{\sqsubseteq}$ with some related approaches to program development. Extended ML and Lego are based on the same notion of refinement as $\lambda_{\sqsubseteq}$. Although the calculi of Morgan *et.al.* are also refinement calculi, they are based on nondeterminism. We show how $\lambda_{\sqsubseteq}$ can be used as a metalanguage for studying and comparing program development methodologies. We also compare with two alternative approaches to program development, based on type theory.

5.4.1 Extended ML

The Extended ML language (EML) is similar to spirit to our approach in that it takes an existing language, in this case Standard ML, and conservatively extends it with specification constructs to give a wide-spectrum development language. Moreover, the constructs added – placeholders ("question marks") and axioms – corresponds exactly to those of our modular analysis here. There is a well-defined semantics [KST97] and methodology [San91].

The semantics is separated into static, dynamic and verification parts. The static semantics is analogous to finding the underlying type of a term, which we have not formalised directly. Refinement typing corresponds to both static and verification semantics. One difference is that a verification checks that an abstract program is well-annotated with respect to a particular interpretation, rather than showing that a particular property holds for all interpretations as we do. The dynamic semantics formalises the evaluation of terms, whenever possible, in order to 'experiment' with abstract programs. Although we have not formalised this, we suggest how this could be considered here on p. 67.

The intention in EML is to formalise the specification language in terms of an arbitrary logic (or rather, an institution). As here, the specification style is *property-oriented.*

Terms are interpreted with respect to (amongst other things) a particular "question mark interpretation". This is a syntactic mapping of ?'s to *arbitrary* expressions.

There are a number of other differences. The question marks can replace arbitrary expressions and so, in particular, types. Booleans and propositions are combined. Satisfaction of properties is up to behavioural equivalence.

There is no proof theory. Rather, three general forms of refinement rule are given, any particular application of which generates proof obligations which must be verified with respect to the semantics. In contrast, our rules are low-level and have been proven sound (and complete). The rules of [San91] (in a suitably translated form) are admissible in $\lambda_{\sqsubseteq}$.

To take a simple example, if we model functors

$$\texttt{functor}\ F(X0 : SIG0) : SIG0' = exp[X0]$$

as abstractions $\lambda X0 : SIG0.exp[X0]$, then the *coding* rule of [San91] can be derived. The rule becomes

$$\lambda X : SIG.?_{SIG'} \sqsubseteq \lambda X : SIG.r$$

when "$SIG \cup r \vDash SIG'$", that is,

$$X : SIG \vdash r : SIG'$$

This follows since it is admissible that if $r : \phi$ then $?_{\phi} \sqsubseteq r$.

5.4.2 Aspinall's λ_{ASL+}

In his thesis [Asp97], Aspinall presents a number of lambda-calculus based calculi for program development. In the same spirit as our work, he constructs his main calculus from a number of subcalculi which he studies separately.

The development methodology is based on the "specification as type, elementhood as satisfaction, subtyping as refinement" idea, but the specification language is parameterised with respect to an arbitrary *institution*. The underlying type theory is not used as a specification language, however, but gives a type structure to the specification building operations of the institution.

The two subcalculi, $\lambda_{\leq\{\}}$ and λ_{Power}, are extensions of the dependently-typed lambda calculus with singleton and power types respectively. The singleton types are a simple form of specification (independent of the institution) while the power types allow parameterisation of specifications over arbitrary specifications. We have not studied parameterised specifications however.

There are significant similarities between $\lambda_{\leq\{\}}$ and $\lambda_{(:)}$. Although Aspinall's intention was to provide a framework in which modular specification constructors could be studied independently of any particular logic, since specifications of functions are a simple form of 'specification of parameterised program', this gives a specification language anyway. The notion of refinement defined in $\lambda_{\leq\{\}}$ only accounts for singleton types, and not general propositions as in $\lambda_{(:)}$. Rather, propositions are added on top with the institution.

Aspinall's calculi are parameterised by a signature, and a consequence relation over that signature which satisfies certain properties. In contrast, the axiom systems used here are defined as a signature and an explicit set of axioms which are then used with the inference rules.

Although specifications are treated as types, he has a notion of *rough type* (originally due to Sannella) which is analogous to the underlying types here.

Another similarity is that he also uses a per semantics, interpreting specifications as pers over the underlying type. However, he interprets terms as elements of pers whereas we interpret them as equivalence classes. This is evident in the interpretation of abstractions, $\lambda x : \phi.t$. Aspinall does not take account of the ϕ but, rather, uses the (rough) type of ϕ. However, he does not have any completeness results.

5.4.3 Type Theory

There are two general approaches to using type theory for program derivation. On the one hand, there is the subtyping approach, as exemplified by Sannella and Tarlecki [ST87], and Aspinall. There, specifications are formalised as types, and the refinement of specifications is formalised as a subtyping $spec \leq spec'$. Refinement continues until it is obvious that some program satisfies the specification, that is, inhabits the type.

The other approach exploits the constructive nature of type theory via the Curry-Howard isomorphism. A specification is phrased as a theorem so that the proof of this theorem in the constructive logic of type theory automatically gives a program which satisfies a specification, via some extraction mechanism.

In [NPS90], Nordström, Petersson and Smith, present Martin-Löf's type theory as a unified formalism for specification and programming based on the extraction style. The derivation methodology is based on the idea that the typing rules can be read as goal-directed tactics.

There are two levels to Martin-Löf type theory: the basic type theory of dependent types, and on top of this, a theory with subset types. A subset type, $\{x : \tau \mid P\}$, consists of a type τ and propositional function, P and types those terms with type τ for which P is true.

This split into two levels is similar to that of types and refinement types here, but there is an important difference in that refinement types correspond to relations over types rather than subsets. We could regard $\lambda_{(:)}$ as formalising an alternative interpretation of the subset theory.

Some consequences of using this type-theoretic formalism for specification and programming are that the logic is intuitionistic and all programs terminate. With subset types, $\{x : \tau \,|\, P\}$, the proposition P is translated into the underlying type theory, and so must be intuitionistic.

The interpretation of the subset theory in the basic theory is given as a translation of a type into basic types and propositional functions. For example, $\mathtt{even} \to \mathtt{even}$ is translated to the type $\mathtt{nat} \to \mathtt{nat}$ and the propositional function $\forall x : \mathtt{nat} \,.\, \mathtt{Even}(x) \supset \mathtt{Even}(fx)$ in f. The typing rules can be translated in this way because they use Curry style rules where abstractions are not labelled with types.

Program refinement, as conceived in this thesis, has similarities with both of the type-theoretic approaches. The refinement relation of $\lambda_?$ corresponds to the program extraction approach (where refinement is often implicit), whereas that of $\lambda_{(:)}$ corresponds to the subtyping approach.

5.4.4 Lego

Lego [LP92] is an example of a proof assistant which implements the extraction style of type-theoretic development. There is no subtyping, but instead an explicit notion of refinement based on *existential variables*.

At any stage during a refinement in Lego, the user is presented with a proof state consisting of a context of assumptions $x_1 : \phi_1, \ldots, x_n : \phi_n$, and a number of goals $?1 : \psi_1, \ldots ?m : \psi_m$. There is also a stored representation of the proof so far, which is hidden from the user, and any other goals which are out of context. Naively, we might represent this state as the refinement term $\lambda x_1 : \phi_1, \ldots, x_n : \phi_n \,.\, \mathtt{let}\ y_1 : \psi_1, \ldots, y_m : \psi_m\ \mathtt{in}\ t$ where t is a translation of the proof so far. To see how goals out of context arise, suppose the first goal is $?1 : \psi_1 \to \psi_1'$ and that we refine this. The resulting proof state contains assumptions $x_1 : \phi_1, \ldots, x_n : \phi_n, z : \psi_1$ and the single goal $?m+1 : \psi_1'$. The other goals are hidden since we cannot use the assumption $z : \psi_1$ to refine them. In fact, the proof states correspond to arbitrary refinement terms. To a certain extent, the user is able to manipulate terms which contain existential variables $?n : \psi_n$ corresponding to refinement terms.

This is more sophisticated than the approach of [NPS90] since it incorporates existential variables and an explicit notion of refinement, all of which is implicit in the straightforward type-theoretic approach.

Existential variables correspond to stubs rather than free variables. In fact, our refinement calculus may be viewed as an explicit formalisation of Lego's refinement process. Conversely, Lego may be viewed as a tool for performing refinement. Although a closer comparison would be between Lego and version of $\lambda_?$ for the calculus of constructions, we consider underdeterminism to be orthogonal to the type theory.

The basic commands in Lego are setting a goal, claiming a lemma, making a local definition, refining the current goal, and changing the order of goals. Each of these commands corresponds naturally to a refinement step in our calculus.

The first step in a development, setting a goal ϕ, introduces the refinement term $\texttt{let}\ x : \phi\ \texttt{in}\ x$; claiming a lemma is refinement by `let`-weakening; while making a local definition $x = t$ is also a `let`-weakening, though t must be determined (*i.e.* not contain any existential variables).

Refinement of goals in Lego is performed by directly solving a goal, unifying it with another goal, or using some library function $f : \psi_1, \psi_2 \to \phi$, so that $?1 : \phi$ is refined to $?2 : \psi_1, ?3 : \psi_2$. We translate this as

$$\frac{f : \psi_1 \times \psi_2 \to \phi}{\texttt{let}\ x : \phi\ \texttt{in}\ t[x] \sqsubseteq \texttt{let}\ y_1 : \psi_1, y_2 : \psi_2\ \texttt{in}\ t[f(y_1, y_2)]}$$

Rearrangement of goals corresponds to the commutativity of `let`-terms. We could regard this as a nontrivial justification for the reordering of goals in Lego, though for the simpler expressions arising here though, it is more obviously sound! Lego has some ability to perform automatic unification during refinement. Such steps are derived from more basic ones.

It would be interesting to formally compare the rules of Lego with those of $\lambda_{\sqsubseteq}$. This would let us apply some of the metatheoretic results here to Lego. For example, if Lego has all the rules of $\lambda_?$ we could conclude that claims are unnecessary.

5.4.5 Refinement Calculus of Back, Morgan and Morris

We compare our calculus with the imperative refinement calculus of Back, Morgan and Morris. This version is taken from [Mor94]. A simplified grammar of the language is:

$$C ::= \texttt{skip} \mid \texttt{abort} \mid x := E \mid \texttt{if}\ P_1 \to C_1 \mathbin{[\!]} \ldots \mathbin{[\!]} P_n \to C_n\ \texttt{fi} \mid \texttt{re}\ x.C\ \texttt{er} \mid$$

$$\texttt{if}\ P\ \texttt{then}\ C\ \texttt{else}\ C' \mid \vec{x} : [P, P'] \mid \texttt{var}\ x : \sigma \bullet C \mid \texttt{con}\ x : \sigma \bullet C$$

$$E ::= \text{expressions}$$

$$P ::= \text{first-order logic plus arithmetic } \textit{etc.}$$

There is also a notation for procedures, which we do not consider here.

Only commands can be specified, and not expressions. There are two specification constructs. The notation $\vec{x} : [P, P']$ is a specification of a command which with precondition P of the state, results in postcondition P', but only altering variables in the 'frame' $\vec{x}$.

The alternation construct, $\texttt{if} \ldots [\!] \ldots \texttt{fi}$, is a *nondeterministic* choice between commands C_i whose 'guard' P_i is true. If none of the P_i are true then the command is unsatisfiable.

The declaration of logical constants $\texttt{con}\ x : \sigma \bullet C$, is not program code, but an abbreviation introduced during development that must ultimately be refined into code.

Annotations – 'assumptions' and 'coercions' – are defined as commands using specifications (unlike in Remark 4.3.22). The assumption $\{pre\} = \langle\rangle : [pre, \top]$, and the coercion $[post] = \langle\rangle : [\top, post]$. In fact, abort and skip can be defined using specifications.

The distinction between imperative and functional languages seems (theoretically) irrelevant for our study of refinement. We can translate the imperative features into our calculus in the style of Idealised Algol. For example, we add a primitive type state, and define a translation $(\_)^\circ$ of terms into our calculus:

$$(\texttt{com})^\circ = \texttt{state} \to \texttt{state}$$

$$(x := E)^\circ = \texttt{assign}(x, E^\circ)$$

$$(\texttt{var}\ x : \sigma \bullet\ C)^\circ = \texttt{new}(\lambda x : \sigma . C^\circ)$$

In order to avoid considerations of nontermination, we could assume that all recursion is terminating, and so could be encoded using primitive recursion (say).

We are more interested here though in translating the specificational features into our calculus. Logical constants can be translated as:

$$(\texttt{con}\ x : \sigma \bullet C)^\circ = \texttt{let}\ x : \sigma\ \texttt{in}\ C^\circ$$

Although the imperative refinement calculus does not have choice for expressions, most other authors do, and it is useful in our translation. We can define this as

$$(r \| r')^{\circ} = \texttt{let}\ b : \texttt{bool}\ \texttt{in}\ (\texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ r')$$

The connection between propositions and booleans is usually not satisfactorily accounted for. It is commonplace (and useful) to write propositions in place of booleans, but not explained how they might ultimately be refined into booleans. We do not study this, but can associate boolean term t with proposition P, by using sum types and asserting that $t : \mathbf{1} \mid P + \mathbf{1} \mid \neg P$. We use the notation $\texttt{if}\ P\ \texttt{then}\ P \to r\ \texttt{else}\ \neg P \to r'$ to mean

$$\texttt{case}\ b : \mathbf{1} | P + \mathbf{1} | \neg P\ \texttt{of}\ \lambda z : \mathbf{1} | P.r,\ \lambda z : \mathbf{1} | \neg P.r$$

so may use the assumptions P, $\neg P$ when reasoning about r and r' respectively. We will write propositions with this convention.

$$\begin{aligned}
(\texttt{if}\ b \to r\ \|\ b' \to r'\ \texttt{fi})^{\circ} = \\
&\texttt{if}\ (b\ \texttt{and}\ b')\ \texttt{then}\ r \| r' \\
&\texttt{else if}\ b\ \texttt{then}\ r \\
&\texttt{else if}\ b'\ \texttt{then}\ r' \\
&\texttt{else}\ ?_{(z:\tau)\ b = \texttt{true}\ \vee\ b' = \texttt{true}}
\end{aligned}$$

The final branch is intended to mean that if both b and b' are false, then the term is unsatisfiable. If we had put $?_{(z:\tau)\ \perp}$ then this would force the whole term to be unsatisfiable.

Logically, pre and postconditions are just a particular form of property. We define the frame proposition $\texttt{Fr}_{\Gamma}(s, s')$ to mean that states s and s' can only differ on the variables in Γ. We use the propositions *pre* and *post* as properties over $\texttt{state}$. Let $\texttt{Fr}_{\Gamma}(s, s') \equiv \forall x : \texttt{var}\,.\, sx \neq s'x \supset x \in \Gamma$.

$$(\Gamma : [pre, post])^{\circ} = ?_{\Pi_{s:pre}(s':post)\texttt{Fr}_{\Gamma}(s,s')}$$

We can show that

$$\{pre\}^{\circ} = \lambda s : pre.s$$

Without some form of annotation, coercions do not have such a neat representation though.

We assume a Hoare logic of commands is given schematically. For example, for all propositions P, terms $e : \sigma$ and variables $x : \sigma$, we have $\texttt{assign}_{\sigma}(x, e)$:

$P[e/x] \to P$. With a sufficiently powerful type theory, and object level substitutions, this definition could be internalised.

Morgan introduces a large number of refinement laws, though they are not arranged into a complete system of refinement rules, and there is no logic of refinement terms. We now consider two laws presented in [Mor94].

Absorb assumption $\{pre'\}; (\Gamma : [pre, post]) = \Gamma : [pre' \wedge pre, post]$

This does not hold in our calculus. To see why, observe that in general we do not have

$$(\lambda x : \phi \,|\, P.x); ?_{\Pi_{x:\phi}\psi} = ?_{\Pi_{x:\phi|P}\psi}$$

since although we might be able to satisfy ψ for every ϕ such that P, so the second expression is satisfiable, we might not be able to do this for every ϕ, so the first expression is unsatisfiable. Now, the translation of the left hand side of the law is

$$(\lambda s : pre'.s); ?_{\Pi_{s:pre}(s':post)\mathtt{Fr}_\Gamma(s,s')}$$

The failure of this equivalence does not mean that in our calculus it is impossible to use external assumptions when reasoning about specifications. Indeed, this nonlocality of satisfiability leads to complications. In order to understand one subspecification, the entire system needs to be considered. Rather, we believe that satisfiability of specifications should be local, and that assumptions be made contextually – that is, in an explicit global or local context. It seems unlikely that an expression of the form $\{pre'\}; \Gamma : [pre, post]$ would actually arise during refinement anyway.

The contrasting status of the law in $\lambda_{\sqsubseteq}$ and [Mor94] is indicative of the difference between underdeterminism and nondeterminism (see Remark 5.2.7).

Alternation If $pre \supset P_1 \vee P_2$, then $\Gamma : [pre, post]$ refines to

$\quad$ `if` $P_1 \to \Gamma : [pre \wedge P_1, post]$ $[\!]$
$\quad\quad$ $P_2 \to \Gamma : [pre \wedge P_2, post]$ `fi`

Let $r_1 \equiv ?_{\Pi_{s:pre \wedge P_1}(s':post)\mathtt{Fr}_\Gamma(s,s')}$ and $r_2 \equiv ?_{\Pi_{s:pre \wedge P_2}(s':post)\mathtt{Fr}_\Gamma(s,s')}$. We show that

$$\begin{aligned} r \equiv\ & ?_{\Pi_{s:pre}(s':post)\mathtt{Fr}_\Gamma(s,s')} \\ & \sqsubseteq \mathtt{if}\ P_1 \wedge P_2\ \mathtt{then}\ r_1 \| r_2 \\ & \mathtt{else\ if}\ P_1\ \mathtt{then}\ r_1 \\ & \mathtt{else\ if}\ P_2\ \mathtt{then}\ r_2 \\ & \mathtt{else}\ ?_{(c:\mathtt{com})\ P_1 \vee P_2} \end{aligned}$$

We need two auxiliary results

1. $r \sqsubseteq \texttt{if } P \texttt{ then } P \rightarrow r \texttt{ else } \neg P \rightarrow r$

2. If $r \sqsubseteq r_1$ and $r \sqsubseteq r_2$, then $r \sqsubseteq r_1 \| r_2$

So by 1,

$$r \sqsubseteq \texttt{if } P_1 \wedge P_2 \texttt{ then } P_1 \wedge P_2 \rightarrow r \texttt{ else } \neg(P_1 \wedge P_2) \rightarrow r$$

and by 1 again, refine the second branch to get

$$\begin{array}{l}\texttt{if } P_1 \wedge P_2 \texttt{ then } P_1 \wedge P_2 \rightarrow r \\ \texttt{else if } \neg(P_1 \wedge P_2) \wedge P_1 \texttt{ then } P_1 \rightarrow r \\ \texttt{else } \neg(P_1 \wedge P_2) \wedge \neg P_1 \rightarrow r\end{array}$$

and then

$$\begin{array}{l}\texttt{if } P_1 \wedge P_2 \texttt{ then } P_1 \wedge P_2 \rightarrow r \\ \texttt{else if } \neg(P_1 \wedge P_2) \wedge P_1 \texttt{ then } P_1 \rightarrow r \\ \texttt{else if } P_2 \texttt{ then} \neg(P_1 \wedge P_2) \wedge \neg P_1 \rightarrow r \\ \texttt{else } \neg(\neg(P_1 \wedge P_2) \wedge \neg P_1) \rightarrow r\end{array}$$

which refines to

$$\begin{array}{l}\texttt{if } P_1 \wedge P_2 \texttt{ then } P_1 \wedge P_2 \rightarrow r \\ \texttt{else if } \neg(P_1 \wedge P_2) \wedge P_1 \texttt{ then } P_1 \rightarrow r \\ \texttt{else } \neg(P_1 \wedge P_2) \wedge \neg P_1 \rightarrow r\end{array}$$

and then

$$\begin{array}{l}\texttt{if } P_1 \wedge P_2 \texttt{ then } P_1 \wedge P_2 \rightarrow r \\ \texttt{else if } P_1 \texttt{ then } P_1 \rightarrow r \\ \texttt{else if } P_2 \texttt{ then } P_2 \rightarrow r \\ \texttt{else } \neg(P_1 \vee P_2) \rightarrow r\end{array}$$

Now $P_1 \vdash r \sqsubseteq r_1$ and $P_2 \vdash r \sqsubseteq r_2$, so by 2, $P_1 \wedge P_2 \vdash r \sqsubseteq r_1 \| r_2$. For the final branch, if $\neg(P_1 \vee P_2)$ then r is unsatisfiable so we refine it to $?_{(c:\texttt{com})\ P_1 \vee P_2}$. Hence the term refines to

$$\begin{array}{l}\texttt{if } P_1 \wedge P_2 \texttt{ then } r_1 \| r_2 \\ \texttt{else if } P_1 \texttt{ then } r_1 \\ \texttt{else if } P_2 \texttt{ then } r_2 \\ \texttt{else } ?_{(c:\texttt{com}) P_1 \vee P_2}\end{array}$$

5.5 Metatheory

In this section we prove a number of proof-theoretic results about the refinement calculus. Besides being used in the completeness proof of the next section, these results are inherently interesting and provide insight into the nature of refinement.

We extend the results of Section 3.3 in which we showed that the simple refinement relation of $\lambda_?$ could be factored into 'coding' and equality. The main idea is that a refinement can be factored into a simple form of 'non-logical' refinement, and a logical equality. These relations can be seen as generalisations of the simply-typed refinement relation in $\lambda_?$ and the logical equality in $\lambda_{(:)}$ respectively. Mirroring the results for $\lambda_?$, the simple refinement can, in turn, be factored into coding and 'coercion'.

Before proving the factorisation itself, we use the characterisation of logical equality to show that refinement typings can be proven in a standard way. Such so-called *generation lemmas* are useful for metatheoretic reasoning about judgements.

The idea of the lemma is that if a term satisfies a specification, then we should be able to prove this by induction on the structure of the term. For example, if the pair $\langle r, r' \rangle$ satisfies some specification χ, then we should be able to prove something about r, something about r' and conclude from this that the pair satisfies χ. Formally, we would like so say that there are refinement types ϕ and ψ such that $r : \phi$, $r' : \psi$ and that $\chi \sqsubseteq \phi \times \psi$ (or, in general, that $x : \phi \vdash r' : \psi$ and $\chi \sqsubseteq \Sigma_{x:\phi}\psi$). This is often the case. However, it is sometimes possible to directly infer that a term satisfies a specification, if this is taken as an axiom, for example. In fact, it is the three **Refinement Type Introduction** rules which break the structural form of refinement typing, in the sense that the inferred refinement type need have no relationship to the term. Thus we formulate the Generation Lemma to account for these two possibilities.

The proof exploits the fact that **Refinement Elimination** is only useful in combination with the subset of rules corresponding to a relation 'logical eta' which we will define, and that this can be eliminated.

Lemma 5.5.1 *(Generation) If $\Gamma \vdash r : \chi$ then either this is derived using a rule of* **Refinement Type Introduction**, *followed by* **Weakening**, *or it is derived on the structure of r, as follows:*

1. *If $\Gamma \vdash x : \chi$ then there exists a ϕ such that $\Gamma \equiv \Gamma_1, x : \phi, \Gamma_2$ and $\Gamma_1 \vdash \chi \sqsubseteq \phi$.*

2. *If $\Gamma \vdash k(r_1, \ldots, r_n) : \chi$ then there is an axiom $\Gamma \vdash k : \phi_1, \ldots, \phi_n \to \psi$ such that $\Gamma \vdash r_i : \phi_i$ $(i = 1, \ldots, n)$ and $\Gamma \vdash \chi \sqsubseteq \psi$.*

3. *If $\Gamma \vdash * : \chi$ then there exists $\Gamma \vdash P$ $\mathtt{wf}$ such that $\Gamma \vdash P$ and $\Gamma \vdash \chi = (z : \mathbf{1})P$.*

4. *If $\Gamma \vdash \langle r, r' \rangle : \chi$ then there exists $\Gamma \vdash \phi, \psi$ $\mathtt{wf}$ such that $\Gamma \vdash r : \phi$, $\Gamma, x : \phi \vdash r' : \psi$ and $\Gamma \vdash \chi \sqsubseteq \Sigma_{x:\phi}\psi$.*

5. *If* $\Gamma \vdash \lambda x : \phi . r : \chi$ *then there exists* $\Gamma, x : \phi \vdash P$ `wf` *and* $\Gamma, x : \phi, P \vdash \psi$ `wf` *such that* $\Gamma, x : \phi, P \vdash r : \psi$ *and* $\Gamma \vdash \chi \sqsubseteq \Pi_{x:\phi|P}\psi$.

6. *If* $\Gamma \vdash ?_\phi : \chi$ *then* $\Gamma \vdash \chi \sqsubseteq \phi$.

7. *If* $\Gamma \vdash \pi_1(r) : \chi$ *then there exists* $\Gamma \vdash \psi$ `wf` *such that* $\Gamma \vdash r : \chi \times \psi$.

8. *If* $\Gamma \vdash \pi_2(r) : \chi$ *then there exists* $\Gamma \vdash \phi$ `wf` *such that* $\Gamma \vdash r : \phi \times \chi$.

9. *If* $\Gamma \vdash rr' : \chi$ *then there exists* $\Gamma \vdash \phi$ `wf` *and* $\Gamma \vdash \psi$ `wf` *such that* $\Gamma \vdash r : \phi \to \chi$ *and* $\Gamma \vdash r' : \phi$.

10. *If* $\Gamma \vdash \mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ r' : \chi$ *then there exists* $\Gamma, x : \phi \vdash P$ `wf` *such that* $\Gamma \vdash r : (x : \phi)P$ *and* $\Gamma, x : \phi, P \vdash r' : \chi$.

Proof: We first show that we can eliminate 'nonessential' uses of the **Refinement Elimination** rule. This rule is only useful in combination with the refinements given by the logical equality rules of Figure 5.13. We induct over these rules to show that in each case we can replace the use of the rule followed by a refinement elimination with a single derived (or basic) rule whose hypotheses and conclusions are all refinement typings (or well-formedness conditions). For example, in place of using **Function Equations (**ξ**)** in

$$\dfrac{\dfrac{\Gamma, x : \phi \vdash P\ \mathtt{wf} \quad \Gamma, x : \phi, P \vdash t' : \psi}{\Gamma \vdash \lambda x : (x : \phi)P.t =_{\Pi_{x:\phi|P}\psi} \lambda x : \phi . t'}\ \textbf{Func. Eqs. }(\xi)}{\Gamma \vdash \lambda x : \phi . t' : \Pi_{x:\phi|P}\psi}\ \textbf{Ref. Elim.}$$

we have the rule

$$\dfrac{\Gamma, x : \phi \vdash P\ \mathtt{wf} \quad \Gamma, x : \phi, P \vdash t' : \psi}{\Gamma \vdash \lambda x : \phi . t' : \Pi_{x:\phi|P}\psi}$$

Note that we are not eliminating uses of **Refinement Elimination** here. This derived rule still makes use of it. The rule **Let Eta** would be used in

$$\dfrac{\dfrac{\Gamma \vdash r : \phi}{\Gamma \vdash \mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ x =_\phi r}\ \textbf{Let Eta}}{\Gamma \vdash \mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ x : \phi}\ \textbf{Ref. Elim.}$$

but this can be proven directly as

$$\dfrac{\Gamma \vdash r : \phi \quad \Gamma, x : \phi \vdash x : \phi}{\Gamma \vdash \mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ x : \phi}$$

Similar analyses hold for the other rules of Figure 5.13.

Thus we can assume, without loss of generality, that if $\Gamma \vdash r : \phi$ is provable and **Ref.Type Intro.** is not used, then it has been inferred from the refinement typing rules of Figure 5.4, together with the derived rules above and **Weakening** from Figure 5.5. Now we need just show that each of these rules preserves the conditions of the lemma, in the sense that if the hypotheses of a rule can be derived in the standard way (described by the lemma), then the conclusion can be inferred in the standard way. In fact, this is immediate for the refinement typing rules of Figure 5.4 and the derived rules. We need just check the case of **Weakening**. For example, suppose

$$\frac{\Gamma \vdash \langle r, r' \rangle : \chi' \quad \Gamma \vdash \chi \sqsubseteq \chi'}{\Gamma \vdash \langle r, r' \rangle : \chi}$$

By the inductive hypothesis, either there exists ϕ, ψ such that $\Gamma \vdash r : \phi, \Gamma, x : \phi \vdash r' : \psi$ and $\Gamma \vdash \chi' \sqsubseteq \Sigma_{x:\phi}\psi$, and so $\Gamma \vdash \chi \sqsubseteq \Sigma_{x:\phi}\psi$. ■

Cases 7-9 show that nondependent hypotheses suffice for the elimination rules. Then, from the point of view of completeness, the restriction on the rules for **Product Terms** and **Function Terms** in Figure 5.4 is not a problem.

We remark that the proof of Lemma 5.5.1 does not depend on the factorisation result which we give below.

Lemma 5.5.2 *(Subject Refinement) If* $\Gamma \vdash r : (x : \phi)P$ *and* $\Gamma \vdash r \sqsubseteq_\phi r'$*, then* $\Gamma \vdash r' : (x : \phi)P$.

Proof: We use induction over $\Gamma \vdash r \sqsubseteq_\phi r'$ and the Generation Lemma. For example, if $\Gamma \vdash ?_{\psi \times \psi'} : (x : \phi)P$ then, either this follows from **Ref. Type Intro.** followed by **Weakening** (in which case P must be a predicate symbol and we can use **Ref.Type Intro.**) or $\Gamma \vdash (x : \phi)P \sqsubseteq \psi \times \psi'$. So, $\Gamma \vdash ?_{\psi \times \psi'} \sqsubseteq_{\psi \times \psi'} \langle ?_\psi, ?_{\psi'} \rangle$ and $\Gamma \vdash \langle ?_\psi, ?_{\psi'} \rangle : \psi \times \psi'$ so by **Weakening**, $\Gamma \vdash \langle ?_\psi, ?_{\psi'} \rangle : (x : \phi)P$. ■

We want to split a refinement $r \sqsubseteq_\phi r'$ into an equality at ϕ and some form of 'nonlogical' refinement independent of any refinement type. However, it is not immediately clear how make such a definition, because we have only defined refinement at specific refinement types. Some refinements are provable at every refinement type (of the term to be refined), though, and this will be our definition. For example, $?_{\mathtt{even} \to \mathtt{nat}}$ refines to $\lambda n : \mathtt{even}.n$ at every refinement type of $?_{\mathtt{even} \to \mathtt{nat}}$.

Definition 5.5.3 *We define a form of untyped refinement,* $\sqsubseteq$*, between terms. We say that* $\Gamma \vdash r \sqsubseteq r'$ *holds when for all provable* $\Gamma \vdash \phi$ `wf`*, if* $\Gamma \vdash r : \phi$ *is provable then so is* $\Gamma \vdash r \sqsubseteq_\phi r'$.

In order to prove the factorisation theorem we need the fact that the axioms can all be factorised. The easiest way of doing this is to assume that the axioms are in $\lambda_{(:)}$ (which we assumed in Definition 5.2.3).

Theorem 5.5.4 *(Factorisation) If $\Gamma \vdash r \sqsubseteq_\phi r'$ then there exists a term r'' such that $\Gamma \vdash r \sqsubseteq r''$ and $\Gamma \vdash r'' =_\phi r'$, and a term r''' such that $\Gamma \vdash r =_\phi r'''$ and $\Gamma \vdash r''' \sqsubseteq r'$.*

Proof: We give a sketch of the proof. The central idea is to partition the rules into what we call logical eta, simple refinement, and computation, by defining relations $=^\eta_\phi$, $\sqsubseteq^s$ and $=^{\beta\mathtt{let}}$, given in Figures 5.13, 5.14 and 5.15 respectively. Note that the decomposition rules (top-down refinement rules of Figure 5.9) are derivable for $\sqsubseteq^s$.

To a certain extent, the definitions of these relations are arbitrarily made to get the proof to go through. For example, we include the **Eta** rule in $\sqsubseteq^s$ simply because it is not clear whether it commutes with the other rules in $\sqsubseteq^s$ (this fact being needed for the proof).

Then, by combining computation with simple refinement and logical eta, respectively, we get nonlogical refinement and logical equality. Specifically, we define $='_\phi$ as the reflexive symmetric transitive closure of $=^\eta_\phi$ and $=^{\beta\mathtt{let}}$; and define $\sqsubseteq'$ as the reflexive transitive closure of $\sqsubseteq^s$ and $=^{\beta\mathtt{let}}$. (We will show that $='_\phi$ is contained in $=_\phi$ and $\sqsubseteq'$ is contained in $\sqsubseteq$.)

1. Refinement rules are of two kinds: axioms, that is, those whose hypotheses do not contain refinements; and the congruence rules.

   Prove that all axioms factor into $='_\phi; \sqsubseteq'$ and $\sqsubseteq'; ='_\phi$, and that congruence rules and **Substitution** preserve factorisations. For **Disjunction**, if assuming P the refinement factors through r_P and assuming Q it factors through r_Q, then assuming $P \vee Q$, it factors through $P \to r_P \parallel Q \to r_Q$ (defined using annotations and choice). The only rules which are not exclusively $=^{\beta\mathtt{let}}$, $=^\eta_\phi$ or $\sqsubseteq^s$ are the stubs refinement rule, and the three 'complex' congruence rules.

2. Induct over $R \in \sqsubseteq^s$ to show that

$$R; ='_\phi \;\Rightarrow\; ='_\phi; \sqsubseteq^s$$

$$='_\phi; R \;\Rightarrow\; \sqsubseteq^s; ='_\phi$$

3. Since this clearly holds for $R \in =^{\beta\mathtt{let}}$ we conclude that $\sqsubseteq'$ (on ϕ) commutes with $=_\phi$, and hence that $\sqsubseteq_\phi$ factors into $\sqsubseteq'$ and $='_\phi$.

4. Show that $='_\phi \subseteq =_\phi$. Hence, $\sqsubseteq_\phi$ factorises into $\sqsubseteq'$ and $=_\phi$.

∎

Corollary 5.5.5 *The refinement relation, $\sqsubseteq_\phi$, factorises into $\sqsubseteq$ and $=_\phi$; that is, if $\Gamma \vdash r \sqsubseteq_\phi r'$ then*

Proof: By Theorem 5.5.4, $\sqsubseteq_\phi$ factorises into $\sqsubseteq'$ and $=_\phi$. The result follows on using the Generation Lemma to show that $\sqsubseteq' \subseteq \sqsubseteq$. ∎

Corollary 5.5.6 *If $\Gamma \vdash r \sqsubseteq_\phi t$, then there exists a term t' such that $\Gamma \vdash r \sqsubseteq t'$ and $\Gamma \vdash t' =_\phi t$.*

Proof: Suppose $\Gamma \vdash r \sqsubseteq_\phi t$. By Theorem 5.5.4, there exists a term r' such that $\Gamma \vdash r \sqsubseteq r'$ and $\Gamma \vdash r' =_\phi t$. Now, it is not necessarily the case that r' is determined. However, if we construct the term t' by replacing each stub in r' with a determined refinement, then $\Gamma \vdash r \sqsubseteq r' \sqsubseteq t'$ and $\Gamma \vdash t' =_\phi t$. ∎

We now generalise the canonical forms lemma of Chapter 3 and show that each term has a canonical form to which it is equal at all its refinement types.

Lemma 5.5.7 *(Canonical Forms) For all terms in context $\Gamma \vdash r$, there exists a context $x_1 : \phi_1, \ldots, x_n : \phi_n$ and a determined term $\Gamma, x_1 : \phi_1, \ldots, x_n : \phi_n \vdash t$ `wf` such that each x_i appears exactly once in t, and*

$$\Gamma \vdash (\texttt{let}\ x_1 : \phi_1, \ldots, x_n : \phi_n\ \texttt{be}\ ?_{\phi_1}, \ldots, ?_{\phi_n}\ \texttt{in}\ t) = r$$

Proof: Use $=^{\beta\texttt{let}}$ rules of Figure 5.15 to move the underdeterminism outwards. ∎

We have shown that an arbitrary refinement, $r \sqsubseteq_\phi r'$, factorises into $\sqsubseteq$ (or $\sqsubseteq'$) and $=_\phi$. We now show that further factorisations can be made when refinement is to a determined term. We defined $\sqsubseteq'$ as the reflexive transitive closure of $\sqsubseteq^s$ and $=^{\beta\texttt{let}}$. In fact, it factorises in the following way:

Lemma 5.5.8 *If $\Gamma \vdash r \sqsubseteq' t$ then there exists a t' such that $\Gamma \vdash r \sqsubseteq^s t'$ and $\Gamma \vdash t' =^{\beta\texttt{let}} t$.*

Proof: We can show that $=^{\beta\texttt{let}}$ commutes with $\sqsubseteq^s$, in the direction: if $\Gamma \vdash r =^{\beta\texttt{let}}; \sqsubseteq^s r'$ then $\Gamma \vdash r \sqsubseteq^s; =^{\beta\texttt{let}} r'$, and the result follows. ∎

Eta

Function Equations (η) (Fig. 5.6)
Let Eta (Fig. 5.7)

Axioms

Axioms of the form $\Gamma \vdash t =_\phi t'$

Weakening

$$\frac{\Gamma \vdash r =_{\phi'} r' \quad \Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash r =_\phi r'}$$

Strengthening

$$\frac{\Gamma \vdash r =_\phi r' \quad \Gamma \vdash r : (x : \phi)P}{\Gamma \vdash r =_{(x:\phi)P} r'}$$

Congruence

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

$$\frac{\Gamma, x : \phi \vdash r =_\psi r'}{\Gamma \vdash \lambda x : \phi.r =_{\Pi_{x:\phi}\psi} \lambda x : \phi.r'}$$

$$\frac{\Gamma, x : \phi \vdash P \ \mathtt{wf} \quad \Gamma \vdash r_1 =_{(x:\phi)P} r_1' \quad \Gamma, x : \phi, P \vdash r_2 =_\psi r_2'}{\Gamma \vdash \mathtt{let}\ x : (x : \phi)P\ \mathtt{be}\ r_1\ \mathtt{in}\ r_2 =_\psi \mathtt{let}\ x : \phi\ \mathtt{be}\ r_1'\ \mathtt{in}\ r_2'} \ (x \notin FV(\psi))$$

congruence rules for constants, pairs, applications, projections

Figure 5.13: Logical Eta: $=_\eta$

Coding

$$\frac{\Gamma \vdash t : \phi}{\Gamma \vdash ?_\phi \sqsubseteq^s t}$$

Let Weakening

$$\frac{\Gamma \vdash r' : \psi \quad \Gamma \vdash r : \phi}{\Gamma \vdash r' \sqsubseteq^s \texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ r'}\ (x \notin FV(r'))$$

Stubs

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi'}{\Gamma \vdash ?_\phi \sqsubseteq^s ?_{\phi'}}$$

Eta

$$\frac{\Gamma \vdash r : \phi}{\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ x \sqsubseteq^s r}$$

Congruence

$$\frac{\Gamma \vdash r \sqsubseteq^s r'}{\Gamma \vdash C[r] \sqsubseteq^s C[r']}$$

$$\frac{\Gamma, x : \phi, P \vdash r \sqsubseteq^s r'}{\Gamma \vdash \lambda x : \phi \,|\, P.r \sqsubseteq^s \lambda x : \phi.r'}$$

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma \vdash r_1 \sqsubseteq^s r_1'}{\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ r_1\ \texttt{in}\ r_2 \sqsubseteq^s \texttt{let}\ x : \phi'\ \texttt{be}\ r_1'\ \texttt{in}\ r_2}$$

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma, x : \phi \vdash r_2 \sqsubseteq^s r_2' \quad \Gamma \vdash r_1 : \phi'}{\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ r_1\ \texttt{in}\ r_2 \sqsubseteq^s \texttt{let}\ x : \phi'\ \texttt{be}\ r_1\ \texttt{in}\ r_2'}$$

Figure 5.14: Simple Refinement: $\sqsubseteq^s$

Decomposition

$$\frac{\Gamma \vdash \phi \times \psi \ \ \texttt{wf}}{\Gamma \vdash ?_{\phi \times \psi} = \langle ?_\phi, ?_\psi \rangle}$$

$$\frac{\Gamma \vdash \Pi_{x:\phi}\psi \ \ \texttt{wf}}{\Gamma \vdash ?_{\Pi_{x:\phi}\psi} = \lambda x : \phi . ?_\psi}$$

Beta

Function Equations (β) (Fig. 5.6)
Product Equations (β) (Fig. 5.6)

Eta

Product Equations (η) (Fig. 5.6)
Unit Equation (Fig. 5.6)

Let Equalities

Figures 5.7 and 5.8 (except **Let Eta**)

Congruence

$$\frac{\Gamma, x : \phi \vdash r = r'}{\Gamma \vdash \lambda x : \phi . r = \lambda x : \phi . r'}$$

$$\frac{\Gamma \vdash r_1 = r_1' \quad \Gamma, x : \phi \vdash r_2 = r_2'}{\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ r_1\ \texttt{in}\ r_2 = \texttt{let}\ x : \phi\ \texttt{be}\ r_1'\ \texttt{in}\ r_2'}$$

Figure 5.15: Computation: $=_{\beta\texttt{let}}$

In Chapter 3, we defined a coding relation, $\rightsquigarrow$, and we now extend the definition to $\lambda_{\sqsubseteq}$ in the obvious way.

Definition 5.5.9 *We define the* coding *relation on well-formed terms,* $\Gamma \vdash r \rightsquigarrow r'$*, as the reflexive, transitive, congruence closure of the following one-step relation:*

$$\frac{\Gamma \vdash t : \phi}{\Gamma \vdash ?_{\phi} \rightsquigarrow t}$$

It is also possible to refine terms simply by weakening the refinement types on binders. We define a relation, $\leq$, for this notion of 'coercion'.

Definition 5.5.10 *We define the* coercion *relation on well-formed terms,* $\Gamma \vdash r \leq r'$*, to be the reflexive, transitive, congruence closure of the following one-step relation:*

$$\frac{\Gamma, x : \phi \vdash r \ \texttt{wf} \quad \Gamma, x : \phi \vdash P \ \texttt{wf}}{\Gamma \vdash \lambda x : \phi | P.r \leq \lambda x : \phi.r}$$

$$\frac{\Gamma \vdash \phi \sqsubseteq \phi' \quad \Gamma \vdash r : \phi' \quad \Gamma, x : \phi \vdash r' \ \texttt{wf}}{\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \leq \texttt{let}\ x : \phi'\ \texttt{be}\ r\ \texttt{in}\ r'}$$

Now, in Chapter 3 we showed that refinement to a determined term in $\lambda_?$ could be factored into coding and equality (Lemma 3.3.2). The generalisation of this lemma to $\lambda_{\sqsubseteq}$ is:

Lemma 5.5.11 *If* $\Gamma \vdash r \sqsubseteq^s t$ *then* $\Gamma \vdash r \rightsquigarrow; \leq t$ *and* $\Gamma \vdash r \leq; \rightsquigarrow t$*.*

Proof: We first show that all rules of $\sqsubseteq^s$ factor into $\rightsquigarrow$ and $\leq$. Clearly $\rightsquigarrow$ and $\leq$ commute. All we need show, then, is that the $\sqsubseteq^s$ rules of **Let Weakening** and **Eta** can be eliminated:

(Let Weakening) Suppose $r' \sqsubseteq^s \texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \rightsquigarrow \texttt{let}\ x : \phi\ \texttt{be}\ t\ \texttt{in}\ t'$. Then, clearly, $r' \rightsquigarrow t'[t/x]$.

(Eta) Suppose $\texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ x \sqsubseteq^s r \rightsquigarrow t$. Then $\texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ x \rightsquigarrow \texttt{let}\ x : \phi\ \texttt{be}\ t\ \texttt{in}\ x$. ∎

We will use the following consequences in the completeness proof:

Lemma 5.5.12 *1. If* $\Gamma \vdash \texttt{let}\ z : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \sqsubseteq' t'$ *then there exists a term* $\Gamma \vdash t : \phi$ *such that* $\Gamma \vdash r \sqsubseteq' t$ *and* $\Gamma \vdash r'[t/z] \sqsubseteq' t'$*.*

2. If $\Gamma \vdash \texttt{let}\ z : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \sqsubseteq_{\psi} t'$ *then there exists a term* $\Gamma \vdash t : \phi$ *such that* $\Gamma \vdash r \sqsubseteq_{\phi} t$ *and* $\Gamma \vdash r'[t/z] \sqsubseteq_{\psi} t'$*.*

Proof:

1. Suppose $\Gamma \vdash \texttt{let}\ z : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \sqsubseteq' t'$. Then, by Lemma 5.5.8, there exists a term t'' such that $\Gamma \vdash \texttt{let}\ z : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \sqsubseteq^s t''$, and $\Gamma \vdash t'' =^{\beta\texttt{let}} t'$, so by Lemma 5.5.11, $\Gamma \vdash \texttt{let}\ z : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \rightsquigarrow; \leq t''$, that is, there exists determined u, u' such that $\Gamma \vdash r \rightsquigarrow u$ and $\Gamma \vdash r' \rightsquigarrow u'$ and $\texttt{let}\ z : \phi\ \texttt{be}\ u\ \texttt{in}\ u' \leq t''$. Hence, $\Gamma \vdash r \sqsubseteq' u$ and $\Gamma \vdash r'[u/z] \sqsubseteq' u'[u/z] \sqsubseteq' t'' =^{\beta\texttt{let}} t'$.

2. If $\Gamma \vdash \texttt{let}\ z : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \sqsubseteq_\psi t'$, then by Theorem 5.5.4, there exists a t'' such that $\Gamma \vdash \texttt{let}\ z : \phi\ \texttt{be}\ r\ \texttt{in}\ r' \sqsubseteq' t'' =_\psi t'$. By part 1 of this lemma, there exists a term $\Gamma \vdash t : \phi$ such that $\Gamma \vdash r \sqsubseteq' t$, so $\Gamma \vdash r \sqsubseteq_\phi t$, and $\Gamma \vdash r'[t/z] \sqsubseteq' t''$, so $\Gamma \vdash r'[t/z] \sqsubseteq_\psi t'$.

∎

Lemma 5.5.13 *(Completeness of refinement to programs) If* $\Gamma \vdash t : \phi$ *then* $\Gamma \vdash ?_\phi \sqsubseteq_\phi t$.

Proof: This follows directly using **Substitution**. ∎

Definition 5.5.14 *For* $\Gamma \vdash r, r'\ \texttt{wf}$*, define* $r \lesssim^\Gamma_\phi r'$ *to mean: for all* $\Gamma' \supseteq \Gamma$*, for all determined* t'*, if* $\Gamma' \vdash r' \sqsubseteq_\phi t'$*, then* $\Gamma' \vdash r \sqsubseteq_\phi t'$*.*

Lemma 5.5.15 *(Refinement Mappings) If*

$$\texttt{let}\ x : \phi\ \texttt{in}\ t_1 \ \lesssim^\Gamma_\chi\ \texttt{let}\ y : \psi\ \texttt{in}\ t_2$$

then there exists $\Gamma, y : \psi \vdash t : \phi$*, such that* $\Gamma, y : \psi \vdash t_1[t/x] =_\chi t_2$*.*

Proof: Since $\Gamma, y : \psi \vdash \texttt{let}\ y : \psi\ \texttt{in}\ t_1 \sqsubseteq_\chi t_2[y]$, by the definition of $\lesssim^\Gamma_\chi$ we have $\Gamma, y : \psi \vdash \texttt{let}\ x : \phi\ \texttt{in}\ t_1 \sqsubseteq_\chi t_2[y]$. By Lemma 5.5.12 (2) this means there exists a term $\Gamma, y : \psi \vdash t : \phi$ such that $\Gamma, y : \psi \vdash t_1[t/x] =_\chi t_2$. ∎

5.6 Models

We first motivate the semantics for the $\lambda_\sqsubseteq$-calculus in Section 5.6.1 before giving the details in Section 5.6.2, and proving soundness and completeness.

5.6.1 Discussion

We discuss the properties we would like the semantics of the refinement calculus to have and give intuitive meanings to refinement typing and refinement in the $\lambda_{\sqsubseteq}$-calculus. It is our intention to model the calculus using a form of Henkin interpretation, and so enable comparison with the models of the subcalculi.

In Chapter 3, a refinement term r was thought of as denoting a set of values, or realizers, corresponding to the programs which satisfy the specification. In Chapter 4, determined terms t were also seen as denoting sets of total realizers. Perhaps unexpectedly, we cannot think of terms in the full refinement calculus as denoting sets.

To see this, we must consider what the meaning of the refinement typing, $r : \phi$, should be. A first approximation to what this means is 'every realizer of r is in ϕ'. This would be wrong, however, as we do not want $\lambda n : \texttt{even}.n : \texttt{nat} \to \texttt{nat}$ to be true, yet every realizer of $\lambda n : \texttt{even}.n$ is certainly in $\texttt{nat} \to \texttt{nat}$. The problem is that the interpretation of $\lambda n : \texttt{even}.n$ as a set is losing the information that its realizers are only determined up to $\texttt{even} \to \texttt{nat}$.

This is analogous to the distinction between $\texttt{even} \to \texttt{even}$ and $(f : \texttt{nat} \to \texttt{nat})\, \forall n : \texttt{even}\,.\, \texttt{Even}(fn)$. Although these refinement types correspond to the same sets of total terms, (we could say they have the same 'extension'), they represent different equalities. We can recast this example as the distinction between the refinement terms $\lambda n : \texttt{even}.n$ and $?_{(f:\texttt{nat}\to\texttt{nat})\, \forall x:\texttt{nat}\,.\,\texttt{Even}(x) \supset fx=x}$.

The problem is that our interpretation of refinement types uses pers, rather than just types. Somehow we need to involve pers in the interpretation of terms as well.

In Chapter 4, we said that $t : \phi$ is true when all realizers of t are equal at ϕ. This will certainly prevent $\lambda n : \texttt{even}.n$ having refinement type $\texttt{nat} \to \texttt{nat}$, but then $?_{\texttt{nat}\to\texttt{nat}} : \texttt{nat} \to \texttt{nat}$ will not be true either.

The solution is to think of refinement terms as sets of equivalence classes of some per. In the case of $\lambda n : \texttt{even}.n$, we should interpret this as a single class in the per $\texttt{even} \to \texttt{nat}$. The term $?_{(f:\texttt{nat}\to\texttt{nat})\, \forall n:\texttt{even}.fn=n}$ is interpreted as all the classes in the per $(f : \texttt{nat} \to \texttt{nat})\, \forall n : \texttt{even}.fn = n$.

However, since we interpret preterms, it is not immediately obvious which per the equivalence classes should be from, but since a set of equivalence classes of some per is itself just a per, we simply interpret refinement terms as pers.

For example, the refinement term $?_\phi$ will be interpreted as the same per as the refinement type ϕ. Then we can succinctly express the semantic meaning of refinement typing: $r : \phi$ is true when (the meaning of) r is a subper of (the meaning of) ϕ.

We will show in the next section that determined terms (*i.e.* terms in the $\lambda_{(:)}$-calculus) are interpreted as a single equivalence class. Thus we can regain the set-theoretic intuition that an underdetermined term r corresponds to a set of realizers — now the realizers can be thought of as determined terms, corresponding to the equivalence classes of r. This means that $r : \phi$ can be thought of as "for all $r \sqsubseteq_\phi t$, we have $t : \phi$". Semantically, the final stage of a refinement is a single equivalence class.

Given this intuition of a refinement term as a per, how are we to think of refinement? Semantically, there are two forms of refinement: reducing the number of classes (restriction) and reducing the size of the classes (quotienting). For example, $?_{\texttt{even}\to\texttt{nat}}$ can be quotiented to $?_{\texttt{nat}\to\texttt{nat}}$, and restricted to $\lambda n : \texttt{even}.n$. Both operations give subpers.

In practice, refinement is more likely to consist of progressive restrictions (in this semantic sense) on a specification towards a program. Non-discrete equivalences can only arise through refinement types on abstractions. Quotienting would correspond to a relaxing of these assumptions, thus increasing the domain of definition.

We think of refinement at ϕ, then, as being a combination of reducing the number of ϕ-classes, and of making the classes finer. These two relations can be combined by saying that $r \sqsubseteq_\phi r'$ is true when (using capitals for the meaning of expressions) R' is contained in the Φ-closure of R, that is, R' is a subper of $\Phi; R; \Phi$. In fact, we also require that R is a subper of Φ.

For example, $\lambda n : \texttt{even}.n \sqsubseteq_{\texttt{nat}\to\texttt{nat}} \lambda n : \texttt{even}.n$ is not true, since the single class denoted by $\lambda n : \texttt{even}.n$ (in $\texttt{even} \to \texttt{nat}$) is not equal to itself at $\texttt{nat} \to \texttt{nat}$. As pointed out above, $\lambda n : \texttt{even}.n$ does not have refinement type $\texttt{nat} \to \texttt{nat}$. Similarly, we do not have $?_{\texttt{nat}\to\texttt{nat}} \sqsubseteq_{\texttt{nat}\to\texttt{nat}} \lambda n : \texttt{even}.n$.

Since the goal of refinement is to reach a term which represents a single equivalence class, syntactically it culminates in a determined term, and not necessarily as a term in $\lambda^{\times\to}$. Hence, from a semantic standpoint, we are consistent in continuing to use t as a metavariable for determined terms.

5.6.2 $\lambda_{\sqsubseteq}$-Henkin Models

As suggested for $\lambda_?$ in Remark 3.4.12, there are two possible approaches to giving a semantics. One possibility is to interpret $\Gamma \vdash r$ as a per. Here we interpret the calculus in Henkin models with the additional structure introduced to model the subcalculi in Sections 3.4 and 4.5, namely factoring and per structure. The only difference is that the factoring condition must be strengthened to account for logical structure.

We follow the pattern of previous chapters, by first giving the interpretation of pre-expressions, and then defining when an interpretation of a signature models an axiom system.

We define the notion of Γ-environment as in Chapter 4, and write $\eta \models^{\mathcal{A}} \Gamma$ when η is a Γ-environment in $\lambda_{\sqsubseteq}$-Henkin interpretation, $\mathcal{A}$. Then the pre-expression in context, $\Gamma \vdash U$, is interpreted in a Γ-environment, η. We interpret pre-expressions so as to avoid the need for establishing coherence (as for the $\lambda_{(:)}$-calculus).

Since the basic data of $\lambda_{\sqsubseteq}$-axiom systems is the same as for first-order $\lambda^{\times\rightarrow}$- and $\lambda_{(:)}$-axiom systems, we give the interpretation of pre-expressions in a first-order $\lambda^{\times\rightarrow}$-Henkin interpretation in Γ-environment, η, in Figures 5.16 to 5.18. Figure 5.16 gives the interpretation of refinement types as pers over the set corresponding to the underlying type. This is the same interpretation for the $\lambda_{(:)}$-calculus but we repeat it here. Figure 5.17 gives the interpretation of refinement terms, also as pers. For a an element of per R, we use the notation $\{a\}_R$ for the singleton per consisting of the class of a.

As for $\lambda_{(:)}$-models, the soundness theorem for $\lambda_{\sqsubseteq}$ will imply that the choice of a in the semantics of abstractions is not important; similarly for the other binding expressions.

We define some of the pers as sets of pairs. We explain the cases for abstractions and pairs. The abstraction, $\lambda x : \phi . r$, denotes a per which relates two functions if for all arguments related at ϕ, the results are related by r. The let-expression, $\texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ r'$, is similar, but relates two individuals if there exists a pair related by r. Since $r : \phi$, soundness implies that the choice does not matter.

The interpretation of pre-propositions, $\Gamma \vdash P$, is given in Figure 5.18 as the set of Γ-environments in which P is true. It is convenient to write $[\![\Gamma \vdash P]\!](\eta)$ for the truth or falsehood of $\eta \in [\![\Gamma \vdash P]\!]$.

Having given the interpretation of pre-expressions we can make the following definitions.

Definition 5.6.1 *Let $\mathcal{A}$ be a first-order $\lambda^{\times\rightarrow}$-Henkin interpretation. We say that $\mathcal{A}$ satisfies the* logical factoring condition *when for each $f' \in A^{\tau\rightarrow\tau'}$, $f \in A^{\sigma\rightarrow\tau'}$, such that $f' \in [\![\psi \rightarrow \psi']\!]^{\mathcal{A}}$, $f \in [\![\phi \rightarrow \psi']\!]^{\mathcal{A}}$, we require that if there exists $h \in A^{\tau} \rightarrow A^{\sigma}$ such that $\bar{f}' = h; \bar{f}$ then there exists $g \in [\![\psi \rightarrow \phi]\!]^{\mathcal{A}}$ such that $\bar{f}' = \bar{g}; \bar{f}$.*

Definition 5.6.2 *Let $Sg = \langle \mathcal{G}, \mathcal{K}, \mathcal{F} \rangle$ be a $\lambda_{\sqsubseteq}$-signature. A* $\lambda_{\sqsubseteq}$-Henkin interpretation *of Sg is a first-order $\lambda^{\times\rightarrow}$-Henkin interpretation of Sg which satisfies the*

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

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

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

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

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

Figure 5.16: Interpretation of Refinement Types

logical factoring condition.

Remark 5.6.3 We make the obvious extension of Definition 4.3.18 (well-structured expressions) to $\lambda_{\sqsubseteq}$. As for the $\lambda_{(:)}$-calculus, it turns out that well-structured expressions have a well-defined interpretation. In fact, some other expressions have an interpretation too, for example: $(\lambda n : \mathtt{nat}.n)?_{(b:\mathtt{bool})\perp}$ is interpreted as the empty per.

For $\mathcal{A}$ a $\lambda_{\sqsubseteq}$-Henkin interpretation we define $\Gamma \vDash^{\mathcal{A}} U \;\; \mathtt{wf}$ when $\eta \; [\![\Gamma]\!] \; \eta'$ implies $[\![\Gamma \vdash U]\!] \; (\eta) = [\![\Gamma \vdash U]\!] \; (\eta')$. That is, well-formedness is interpreted semantically as equal environments giving equal interpretations. For $\eta \vDash^{\mathcal{A}} \Gamma$ we define $\Gamma \vDash^{\mathcal{A},\eta} r : \phi$ to mean: $[\![\Gamma \vdash r]\!](\eta)$ is a subper of $[\![\Gamma \vdash \phi]\!](\eta)$ and $\Gamma \vDash^{\mathcal{A},\eta} P$ to mean: $\eta \vDash \Gamma \Rightarrow \eta \in [\![\Gamma \vdash P]\!]$. In particular, then, $\Gamma \vDash^{\mathcal{A},\eta} r \sqsubseteq_{\phi} r'$ when $R \subseteq \Phi$ and $R' \subseteq \Phi; R; \Phi$, where $R = [\![\Gamma \vdash r]\!]^{\mathcal{A}}(\eta)$, $R' = [\![\Gamma \vdash r']\!]^{\mathcal{A}}(\eta)$, and $\Phi = [\![\Gamma \vdash \phi]\!]^{\mathcal{A}}(\eta)$. We will see below that this 'asymmetric' meaning of refinement has a more symmetric formulation.

For judgement $\Gamma \vdash J$, we write $\Gamma \vDash^{\mathcal{A}} J$ when $\Gamma \vDash^{\mathcal{A},\eta} J$ for all $\eta \vDash^{\mathcal{A}} \Gamma$.

Definition 5.6.4 *Let $\langle Sg, Ax\rangle$ be a $\lambda_{\sqsubseteq}$-axiom system, and let $\mathcal{A}$ be a $\lambda_{\sqsubseteq}$-Henkin interpretation of signature Sg. We say that $\mathcal{A}$ is a* model *of $\langle Sg, Ax\rangle$ when*

- *for each well-formed axiom $\Gamma \vdash P$, for all $\eta \in [\![\Gamma]\!]^{\mathcal{A}}$, $\eta \in [\![\Gamma \vdash P]\!]^{\mathcal{A}}$. We write this as $\Gamma \vDash^{\mathcal{A}} P$.*

$$b \ [\![\Gamma, x:\phi, \Gamma' \vdash x]\!]\langle \eta, a, \eta' \rangle \ b' \iff b \ \{a\}_{[\![\Gamma \vdash \phi]\!](\eta)} \ b'$$

$$\frac{[\![\Gamma \vdash r_1]\!] = m_1 \ \cdots \ [\![\Gamma \vdash r_n]\!] = m_n}{b \ [\![\Gamma \vdash k(r_1, \ldots, r_n)]\!](\eta) \ b' \iff \begin{array}{r} \exists a_i \ m_i(\eta) \ a'_i \ . \ b = k^{\mathcal{A}}(a_1, \ldots, a_n) \text{ and} \\ b' = k^{\mathcal{A}}(a'_1, \ldots, a'_n) \end{array}}$$

$$b \ [\![\Gamma \vdash *]\!](\eta) \ b' \iff b, b' \in \mathcal{A}^{\mathbf{1}}$$

$$\frac{[\![\Gamma \vdash r]\!] = m \quad [\![\Gamma \vdash r']\!] = m'}{b \ [\![\Gamma \vdash \langle r, r' \rangle]\!](\eta) \ b' \iff \begin{array}{l} \mathtt{Proj}_1^{\sigma,\tau}(b) \ m(\eta) \ \mathtt{Proj}_1^{\sigma,\tau}(b') \text{ and} \\ \mathtt{Proj}_2^{\sigma,\tau}(b) \ m'(\eta) \ \mathtt{Proj}_2^{\sigma,\tau}(b') \end{array}}$$

$$\frac{[\![\Gamma, x:\phi \vdash r]\!] = m}{f \ [\![\Gamma \vdash \lambda x:\phi.r]\!](\eta) \ f' \iff \forall a \ [\![\Gamma \vdash \phi]\!](\eta) \ a' \ . \ \mathtt{App}(f, a) \ m\langle \eta, a \rangle \ \mathtt{App}(f', a')}$$

$$\frac{[\![\Gamma \vdash \phi]\!] = R}{[\![\Gamma \vdash ?_\phi]\!](\eta) = R(\eta)}$$

$$\frac{[\![\Gamma \vdash r]\!] = m}{[\![\Gamma \vdash \pi_1(r)]\!](\eta) = \{(\mathtt{Proj}_1^{\sigma,\tau}(a), \mathtt{Proj}_1^{\sigma,\tau}(a')) \mid a \ m(\eta) \ a'\}}$$

$$\frac{[\![\Gamma \vdash r]\!] = m}{[\![\Gamma \vdash \pi_2(r)]\!](\eta) = \{(\mathtt{Proj}_2^{\sigma,\tau}(a), \mathtt{Proj}_2^{\sigma,\tau}(a')) \mid a \ m(\eta) \ a'\}}$$

$$\frac{[\![\Gamma \vdash r]\!] = m \quad [\![\Gamma \vdash r']\!] = m'}{[\![\Gamma \vdash rr']\!](\eta) = \{(\ \mathtt{App}(f, a), \mathtt{App}(f', a') \ ) \mid f \ m(\eta) \ f', a \ m'(\eta) \ a'\}}$$

$$\frac{[\![\Gamma \vdash r]\!] = m \quad [\![\Gamma, x:\phi \vdash r']\!] = m'}{b \ [\![\Gamma \vdash \mathtt{let} \ x:\phi \ \mathtt{be} \ r \ \mathtt{in} \ r']\!](\eta) \ b' \iff \exists a \ m(\eta) \ a' \ . \ b \ m'\langle \eta, a \rangle \ b'}$$

Figure 5.17: Interpretation of Refinement Terms

$$\llbracket \Gamma \vdash \bot \rrbracket = \emptyset$$

$$\llbracket \Gamma \vdash P \supset P' \rrbracket = \{\eta \vDash \Gamma \mid \eta \notin \llbracket \Gamma \vdash P \rrbracket \text{ or } \eta \in \llbracket \Gamma \vdash P' \rrbracket\}$$

$$\llbracket \Gamma \vdash \forall x : \phi . P \rrbracket = \{\eta \vDash \Gamma \mid \forall a \in \llbracket \Gamma \vdash \phi \rrbracket(\eta) \, . \, \langle \eta, a \rangle \in \llbracket \Gamma, x : \phi \vdash P \rrbracket\}$$

$$\frac{\llbracket \Gamma \vdash r_1 \rrbracket = R_1 \quad \cdots \quad \llbracket \Gamma \vdash r_n \rrbracket = R_n}{\llbracket \Gamma \vdash F(r_1, \ldots, r_n) \rrbracket = \{\eta \vDash \Gamma \mid \forall a_i \in R_i(\eta) \, . \, \langle a_1, \ldots, a_n \rangle \subseteq F^{\mathcal{A}}\}}$$

$$\frac{\llbracket \Gamma \vdash r \rrbracket = R \quad \llbracket \Gamma \vdash r' \rrbracket = R' \quad \llbracket \Gamma \vdash \phi \rrbracket = \Phi}{\llbracket \Gamma \vdash r \sqsubseteq_\phi r' \rrbracket = \{\eta \vDash \Gamma \mid R(\eta) \subseteq \Phi(\eta) \ \wedge \ R'(\eta) \subseteq \Phi(\eta); R(\eta); \Phi(\eta)\}}$$

$$\frac{\llbracket \Gamma \vdash \phi \rrbracket = \Phi \quad \llbracket \Gamma \vdash \phi' \rrbracket = \Phi'}{\llbracket \Gamma \vdash \phi \sqsubseteq \phi' \rrbracket = \{\eta \vDash \Gamma \mid \Phi(\eta) \supseteq \Phi'(\eta)\}}$$

Figure 5.18: Interpretation of Propositions

- *for each well-formed axiom* $\Gamma \vdash k : \phi_1, \ldots, \phi_n \to \psi$, *for all* $\eta \vDash^{\mathcal{A}} \Gamma$, *if* $a_i \llbracket \Gamma \vdash \phi_i \rrbracket^{\mathcal{A}}(\eta) \, a'_i$ $(i = 1, n)$, *then* $k^{\mathcal{A}}(a_1, \ldots, a_n) \ \llbracket \Gamma \vdash \psi \rrbracket^{\mathcal{A}}(\eta) \, k^{\mathcal{A}}(a'_1, \ldots, a'_n)$. *We write this as* $\Gamma \vDash^{\mathcal{A}} k : \phi_1, \ldots, \phi_n \to \psi$

The meanings of the judgements may be equivalently expressed in terms of equivalence classes. We will use cl and cl' as metavariables for equivalence classes, and by writing $cl \in R$, we mean that cl is a class of R, rather than a value.

Lemma 5.6.5

1. $\Gamma \vDash^\eta r : \phi$ *when* $\forall cl \in \llbracket \Gamma \vdash r \rrbracket(\eta) \, . \, \forall x \in cl \, . \, \forall x' \in cl \, . \, x \, \llbracket \Gamma \vdash \phi \rrbracket(\eta) \, x'$.
2. $\Gamma \vDash^\eta r \sqsubseteq_\phi r'$ *when* $\Gamma \vDash^\eta r : \phi$ *and* $\forall cl' \in \llbracket \Gamma \vdash r' \rrbracket(\eta) \, . \, \exists cl \in \llbracket \Gamma \vdash r \rrbracket(\eta) \, . \, \forall x \in cl \, . \forall x' \in cl' \, . \, x \, \llbracket \Gamma \vdash \phi \rrbracket(\eta) \, x'$.

Proof: We prove part (2). Suppose $\Gamma \vDash^\eta r \sqsubseteq_\phi r'$. The literal reading is that there is are inclusions of pers $R' \subseteq \Phi; R; \Phi$ and $R \subseteq \Phi$, where $R = \llbracket \Gamma \vdash r \rrbracket(\eta)$, $R' = \llbracket \Gamma \vdash r' \rrbracket(\eta)$ and $\Phi = \llbracket \Gamma \vdash \phi \rrbracket(\eta)$. Let $cl' \in R'$. Then $cl' \in \Phi; R; \Phi$. So for every $x' \in cl'$ there exists $x_1, x_2 \in R$ such that $x' \; \Phi \; x_1 \; R \; x_2 \; \Phi \; x'$. Let cl be the set of such x_1. Then we have $cl \; \Phi \; cl'$.

Conversely, suppose $\forall cl' \in R' \, . \, \exists cl \in R \, . \, cl \; \Phi \; cl'$. Let $x'_1 \; R' \; x'_2$. Then $x'_1, x'_2 \in cl'$ for some $cl' \in R'$, and so there exists a $cl \in R$ such that $cl \; \Phi \; cl'$. Choose any $x \in cl$. Then $x'_1 \; \Phi \; x \; R \; x \; \Phi \; x'_2$. ∎

Lemma 5.6.6 *For all determined terms* t, *the per* $\llbracket \Gamma \vdash t \rrbracket(\eta)$ *is a singleton class.*

Proof: Induction over preterms $\Gamma \vdash t$. ∎

We can formalise a sense in which $\lambda_{\sqsubseteq}$-interpretations generalise the semantics of the two subcalculi. First observe that $\lambda_?$- and $\lambda_{(:)}$-axiom systems are also $\lambda_{\sqsubseteq}$-axiom systems. Now $\lambda_?$- and $\lambda_{(:)}$-Henkin interpretations also give rise to $\lambda_{\sqsubseteq}$-Henkin interpretations, and similarly for environments, though the interpretation functions are different. Terms from $\lambda_?$ and $\lambda_{(:)}$ are interpreted as sets, but terms in $\lambda_{\sqsubseteq}$ are interpreted as pers. Now the $\lambda_{\sqsubseteq}$-interpretation of terms from the subcalculi is a special kind of per. In particular, types and terms from $\lambda_?$ are interpreted as discrete pers. Terms from $\lambda_{(:)}$ (*i.e.* determined terms) are interpreted as indiscrete pers. We will subscript interpretations with the calculus.

Define two mappings ι and κ from sets to pers. Let S be a set. Then we define pers ιS and κS as:

$$\begin{array}{rcll} x\ \iota S\ y & \iff & x, y \in S \text{ and } x = y & \quad \text{discrete per} \\ x\ \kappa S\ y & \iff & x, y \in S & \quad \text{indiscrete per} \end{array}$$

Proposition 5.6.7 *Let r and τ be a well-formed term and type in $\lambda_?$. Then:*

- $[\![\Gamma \vdash r]\!]^{\mathcal{A}}_{\lambda_{\sqsubseteq}}(\eta) = \iota\,([\![\Gamma \vdash r : \tau]\!]^{\mathcal{A}}_{\lambda_?}(\eta))$
- $[\![\Gamma \vdash \tau]\!]^{\mathcal{A}}_{\lambda_{\sqsubseteq}}(\eta) = \iota\,(\tau^{\mathcal{A}})$

Let t, ϕ and P be a pre-term, -refinement type and -proposition in $\lambda_{(:)}$. Then:

- $[\![\Gamma \vdash t]\!]^{\mathcal{A}}_{\lambda_{\sqsubseteq}}(\eta) = \kappa\,([\![\Gamma \vdash t]\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta))$
- $[\![\Gamma \vdash \phi]\!]^{\mathcal{A}}_{\lambda_{\sqsubseteq}}(\eta) = [\![\Gamma \vdash \phi]\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta)$
- $[\![\Gamma \vdash P]\!]^{\mathcal{A}}_{\lambda_{\sqsubseteq}}(\eta) = [\![\Gamma \vdash P]\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta)$

Proof: Induction over $\lambda_?$- and $\lambda_{(:)}$-expressions. ∎

Proposition 5.6.8 *The $\lambda_{\sqsubseteq}$-calculus is a conservative extension of $\lambda_?$ and $\lambda_{(:)}$ in the following (semantic) sense:*

- *Let $\langle Sg, Ax\rangle$ be a $\lambda_?$-axiom system, and $\mathcal{A}$ a $\lambda_?$-Henkin model of $\langle Sg, Ax\rangle$. Let $\eta \models^{\mathcal{A}} \Gamma$ and suppose that r and r' are terms of $\lambda_?$. Then,*

  $$\Gamma \models^{\mathcal{A},\eta}_{\lambda_{\sqsubseteq}} r \sqsubseteq_\tau r' \iff \Gamma \models^{\mathcal{A},\eta}_{\lambda_?} r \sqsubseteq_\tau r'$$

- *Let $\langle Sg, Ax\rangle$ be a $\lambda_{(:)}$-axiom system, and $\mathcal{A}$ a $\lambda_{(:)}$-Henkin model of $\langle Sg, Ax\rangle$. Let $\eta \models^{\mathcal{A}} \Gamma$ and suppose that t and t' are terms of $\lambda_{(:)}$. Then,*

  $$\Gamma \models^{\mathcal{A},\eta}_{\lambda_{\sqsubseteq}} t =_\phi t' \iff \Gamma \models^{\mathcal{A},\eta}_{\lambda_{(:)}} t =_\phi t'$$

Proof:

$$
\begin{aligned}
\Gamma \vDash^{\mathcal{A},\eta}_{\lambda_\sqsubseteq} r \sqsubseteq_\tau r' &\iff [\![\Gamma \vdash r']\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \subseteq [\![\Gamma \vdash \tau]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta); [\![\Gamma \vdash r]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta); [\![\Gamma \vdash \tau]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \\
&\iff \iota\left([\![\Gamma \vdash r']\!]^{\mathcal{A}}_{\lambda_?}(\eta)\right) \subseteq \iota\left(\tau^{\mathcal{A}}\right); \iota\left([\![\Gamma \vdash r]\!]^{\mathcal{A}}_{\lambda_?}(\eta)\right); \iota\left(\tau^{\mathcal{A}}\right) \\
&\iff [\![\Gamma \vdash r' : \tau]\!]^{\mathcal{A}}_{\lambda_?}(\eta) \subseteq [\![\Gamma \vdash r : \tau]\!]^{\mathcal{A}}_{\lambda_?}(\eta) \\
&\iff \Gamma \vDash^{\mathcal{A},\eta}_{\lambda_?} r \sqsubseteq_\tau r'
\end{aligned}
$$

$$
\begin{aligned}
\Gamma \vDash^{\mathcal{A},\eta}_{\lambda_\sqsubseteq} t =_\phi t' &\iff \forall cl \in [\![\Gamma \vdash t]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \,.\, \exists cl' \in [\![\Gamma \vdash t']\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \,.\, cl\, [\![\Gamma \vdash \phi]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta)\, cl' \wedge \\
&\qquad \forall cl' \in [\![\Gamma \vdash t']\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \,.\, \exists cl \in [\![\Gamma \vdash t]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \,.\, cl\, [\![\Gamma \vdash \phi]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta)\, cl' \\
&\iff \text{the class } [\![\Gamma \vdash t]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \quad [\![\Gamma \vdash \phi]\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \quad \text{the class } [\![\Gamma \vdash t']\!]^{\mathcal{A}}_{\lambda_\sqsubseteq}(\eta) \\
&\iff \kappa\left([\![\Gamma \vdash t]\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta)\right) \quad [\![\Gamma \vdash \phi]\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta) \quad \kappa\left([\![\Gamma \vdash t']\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta)\right) \\
&\iff \forall a \in [\![\Gamma \vdash t]\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta) . \forall a' \in [\![\Gamma \vdash t']\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta) \;.\; a\, [\![\Gamma \vdash \phi]\!]^{\mathcal{A}}_{\lambda_{(:)}}(\eta)\, a'
\end{aligned}
$$

∎

We will need the generalisations of some lemmas used in the soundness and completeness proofs of the two subcalculi.

Lemma 5.6.9 *(Substitution Lemma) If $\Gamma \vDash^{\mathcal{A},\eta} t_i : \phi_i$ $(i = 1, \ldots, n)$, then*

$$[\![x_1 : \phi_1, \ldots, x_n : \phi_n \vdash U]\!]^{\mathcal{A}} \langle a_1, \ldots, a_n \rangle = [\![\Gamma \vdash U[t_i/x_i]]\!]^{\mathcal{A}}(\eta)$$

where $a_i \in [\![\Gamma \vdash t_i]\!]^{\mathcal{A}}(\eta)$ (so $\langle a_1, \ldots, a_n \rangle \vDash^{\mathcal{A}} x_1 : \phi_1, \ldots, x_n : \phi_n$).

Proof: Induction over $x_1 : \phi_1, \ldots, x_n : \phi_n \vdash U$. ∎

Although this is written the same as the substitution lemma for $\lambda_{(:)}$ (Lemma 4.5.4), there it is stated using sets, whereas here we use pers. The analogues of Lemmas 4.5.5 and 4.5.6 follow similarly.

Theorem 5.6.10 *(Soundness) Let $\mathcal{A}$ be a Henkin Model of axiom system $\langle Sg, Ax \rangle$. If $\langle Sg, Ax \rangle \rhd \Gamma \vdash B$ (where B ranges over basic judgements) then $\Gamma \vDash^{\mathcal{A}} B$. In particular, if $\langle Sg, Ax \rangle \rhd \Gamma \vdash \phi$ `wf` then $\Gamma \vDash^{\mathcal{A}} \phi$ `wf`, if $\langle Sg, Ax \rangle \rhd \Gamma \vdash P$ `wf` then $\Gamma \vDash^{\mathcal{A}} P$ `wf`, if $\langle Sg, Ax \rangle \rhd \Gamma \vdash r : \phi$ then $\Gamma \vDash^{\mathcal{A}} r : \phi$, and if $\langle Sg, Ax \rangle \rhd \Gamma \vdash P$ then $\Gamma \vDash^{\mathcal{A}} P$.*

Proof: Simultaneous induction over derivations of all judgements. We can simplify the proof of soundness of $r \sqsubseteq_\phi r'$ by observing that, for those rules which are $\sqsubseteq$, that $[\![r]\!] \supseteq [\![r']\!] \Rightarrow \vDash r \sqsubseteq r'$. Hence, we need only show that $R' \subseteq R$, from which $R' \subseteq \Phi; R; \Phi$ follows. We work through some key cases (omitting the proofs that $R \subseteq \Phi$).

- **(Variables)** Suppose $\vDash \Gamma, x : \phi, \Gamma'$ $\texttt{wf}$. Let $\langle \eta, a, \eta' \rangle \vDash \Gamma, x : \phi, \Gamma'$ and suppose that $b \; [\![\Gamma, x : \phi, \Gamma']\!]\langle \eta, a, \eta' \rangle \; b'$. Then, $b \; \{a\}_{[\![\Gamma \vdash \phi]\!](\eta)} \; b'$, so $b \; [\![\Gamma \vdash \phi]\!](\eta) \; b'$, and $b \; [\![\Gamma, x : \phi, \Gamma' \vdash \phi]\!]\langle \eta, a, \eta' \rangle \; b'$.

- **(Constants)** Fix $\eta \vDash \Gamma$. Let $b \; [\![\Gamma \vdash k(r)]\!](\eta) \; b'$, so $b = k^{\mathcal{A}}(a)$, $b' = k^{\mathcal{A}}(a')$ for some $a \; [\![\Gamma \vdash r]\!](\eta) \; a'$. By the soundness of $\Gamma \vdash r : \phi$ we have $a \; [\![\Gamma \vdash \phi]\!](\eta) \; a'$. Now $\Gamma' \subseteq \Gamma$ so suppose $\Gamma \equiv \Gamma_1, \Gamma', \Gamma_2$, $\eta \equiv \langle \eta_1, \eta', \eta_2 \rangle$. Then $[\![\Gamma \vdash \phi]\!](\eta) = [\![\Gamma' \vdash \phi]\!](\eta')$, so $a \; [\![\Gamma' \vdash \phi]\!](\eta') \; a'$, and since $\mathcal{A}$ models the axiom, $k^{\mathcal{A}}(a) \; [\![\Gamma' \vdash \psi]\!](\eta') \; k^{\mathcal{A}}(a')$. Hence $b \; [\![\Gamma \vdash \phi]\!](\eta) \; b'$.

- (**Function Equations (β)**) Suppose $\Gamma, x : \phi \vDash t : \psi$ and $\Gamma \vDash t' : \phi$. Then $b \; [\![\Gamma \vdash (\lambda x : \phi . t)t']\!](\eta) \; b'$ iff there exists elements f, f', a, a' such that $f \; [\![\Gamma \vdash \lambda x : \phi . t]\!](\eta) \; f'$, $a \; [\![\Gamma \vdash t']\!](\eta) \; a'$ and $b = \texttt{App}(f, a)$, $b' = \texttt{App}(f', a')$. This holds iff $b \;\; [\![\Gamma, x : \phi \;\vdash\; t]\!](\langle \eta, a \rangle) \;\; b'$ for $a \in [\![\Gamma \vdash t]\!](\eta)$, iff $b \; [\![\Gamma \vdash t[t'/x]]\!](\eta) \; b'$ (substitution lemma).

- (**Let Beta)** Suppose $\eta \vDash \Gamma$ and $b \; [\![\Gamma \vdash \texttt{let } x : \phi \texttt{ be } t \texttt{ in } r]\!](\eta) \; b'$. This is the same as $b \; [\![\Gamma, x : \phi \vdash r]\!]\langle \eta, a \rangle \; b'$ for some $a \in [\![\Gamma \vdash t']\!](\eta)$. By the Substitution Lemma, this is the same as $b \; [\![\Gamma \vdash r[t/x]]\!](\eta) \; b'$.

- Let Term Equalities:

  - **(Projections)** Suppose $b \; [\![\Gamma \vdash \texttt{let } x : \phi_1 \times \phi_2 \texttt{ be } r \texttt{ in } \pi_1(x)]\!](\eta) \; b'$. This is when there exists $a \in [\![\Gamma \vdash r]\!](\eta)$ such that $b \; [\![\Gamma, x : \phi_1 \times \phi_n \vdash \pi_1(x)]\!]\langle \eta, a \rangle \; b'$. Hence $b \; [\![\Gamma \vdash \pi_1(r)]\!](\eta) \; b'$.

  - **(Abstractions)** This is similar to the proof in Chapter 3, but uses the stronger factoring condition. For the sake of simplicity, we will consider closed terms. We must show that

    $$[\![\texttt{let } z : \Pi_{x:\phi}\psi \texttt{ in } \lambda x : \phi . t[zx/y]]\!] = [\![\lambda x : \phi . (\texttt{let } y : \psi \texttt{ in } t)]\!]$$

    Now $f \in [\![\texttt{let } z : \Pi_{x:\phi}\psi \texttt{ in } \lambda x : \phi . t[zx/y]]\!]$ when

    $$\exists a \in [\![\Pi_{x:\phi}\psi]\!] \; . \; \forall b \in [\![\phi]\!] \; . \; fb \in [\![z : \Pi_{x:\phi}\psi, x : \phi \vdash t[zx/y]]\!]\langle a, b \rangle \quad (5.1)$$

    and $f \in [\![\lambda x : \phi . (\texttt{let } y : \psi \texttt{ in } t)]\!]$ when

    $$\forall b \in [\![\phi]\!] \; . \; \exists a_b \in [\![x : \phi \vdash \psi]\!](b) \; . \; fb \in [\![x : \phi, y : \psi \vdash t]\!]\langle b, a_b \rangle \qquad (5.2)$$

    We follow the same line of reasoning as in Chapter 3, p. 80, to prove these two statements equivalent. The interesting direction is showing that (5.2) implies (5.1). Define $h : \sigma^{\mathcal{A}} \to (\sigma \times \tau)^{\mathcal{A}}$ as $(b \in \sigma^{\mathcal{A}} \mapsto \langle b, a_b \rangle)$ and f' as $[\![\lambda p : \Sigma_{x:\phi}\psi \vdash t[\pi_1 p/x, \pi_2 p/y]]\!]$.

Then, since (5.2) says that $\bar{f} = h; \bar{f}'$, by the logical factoring condition, there exists $g \in [\![\phi \to \Sigma_{x:\phi}\psi]\!]$ such that $\bar{g}; \bar{f}' = \bar{f}$.

The remainder of the proof follows Chapter 3.

- **(Logical Congruence)** Suppose $\Gamma \vDash r \sqsubseteq_\phi r'$ and $\Gamma \vDash r : (x : \phi)P$. Let $\eta \vDash \Gamma$, and suppose $b\ R'(\eta)\ b'$. Then there exists b_1, b_2 such that $b\ \Phi(\eta)\ b_1\ R(\eta)\ b_2\ \Phi(\eta)\ b'$. Now $b_1\ R(\eta)\ b_2$ implies that $b_1\ [\![\Gamma \vdash (x : \phi)P]\!](\eta)\ b_2$, so $\Gamma, x : \phi \vDash^{\eta, b_1} P$, and since $b_2\ [\![\Gamma \vdash \phi]\!](\eta)\ b'$, we have $\Gamma, x : \phi \vDash_{\langle \eta, b' \rangle} P$, and so $b_2\ [\![\Gamma \vdash (x : \phi)P]\!](\eta)\ b'$. Similarly, $b\ [\![\Gamma \vdash (x : \phi)P]\!](\eta)\ b_1$, and so $b\ [\![\Gamma \vdash (x : \phi)P]\!](\eta)\ R(\eta)\ [\![\Gamma \vdash (x : \phi)P]\!](\eta)\ b'$.
- Refinement Rules: The soundness of $r \sqsubseteq_\phi r'$ (Figures 5.7, 5.8, 5.9 and 5.10) follows the corresponding proofs in Chapter 3.

∎

Since $\eta\ [\![\Gamma]\!]\ \eta'$ implies $[\![\Gamma \vdash r]\!](\eta) = [\![\Gamma \vdash r]\!](\eta')$ we can give the semantics as a mapping from $[\![\Gamma]\!]$ classes. Sometimes we write an environment as $[\![t]\!]$, meaning any member of the class of $[\![t]\!]$.

In Chapter 4 we had the problem of formulating a completeness result, since the interpretation of refinement types as pers did not correspond exactly to the rules of the calculus as they currently stand.

There were two kinds of mismatch. On the one hand, a term could be 'well-formed' in the semantics, by virtue of having a unique interpretation, yet not be syntactically well-formed, an example being $(\lambda n : \texttt{even}.*)3$. The other problem arose with higher-order terms, and was 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 refinement type $(\texttt{even} \to \texttt{nat}) \to \texttt{nat}$ in the model but not in the calculus.

We got round this in Chapter 4 by defining a contextual equivalence $\simeq_\phi$ on terms such that if $\vDash t : \phi$ then there was a t' such that $t' \simeq_\phi t$ and $t' : \phi$. For the above two examples, we have $(\lambda n : \texttt{even}.*)3 \simeq_{\mathbf{1}} *$ and $\lambda f : \texttt{nat} \to \texttt{nat}\,.\,3 \simeq_{\texttt{even} \to \texttt{nat}} \lambda f : \texttt{even} \to \texttt{nat}\,.\,3$.

Another possibility is to restrict the statement of completeness to avoid these classes of terms, and this is what we do in this chapter. This has the virtue of being simpler, and it also makes it easier to extend the completeness theorem using suggestions in Chapter 6.

Suppose some judgement is true in all $\lambda_{\sqsubseteq}$-models of the relevant axiom system. In order to show that the judgement is provable we first assume that it is well-formed, where well-formedness of the judgement $\Gamma \vdash r : \phi$ means that $\Gamma \vdash \phi$ $\mathtt{wf}$ and $\Gamma \vdash r$ $\mathtt{wf}$, that is, $\Gamma \vdash r : \phi'$ for *some* ϕ'. We then make the additional assumption that the judgement is of *rank* less than or equal to 1, where the rank is defined recursively for each syntactic category, the idea being to exclude any higher-order refinement types. For example, $\mathtt{Rank}\ (\mathtt{nat} \times \mathtt{bool} \to \mathtt{nat}) = 1$, $\mathtt{Rank}\ ((\mathtt{even} \to \mathtt{nat}) \to \mathtt{nat}) = 2$.

The completeness proof has the same pattern as in previous chapters. We construct a term model from an appropriate notion of Henkin theory. As with Definition 4.5.8 in the $\lambda_{(:)}$-calculus, we regard theories as infinite contexts, rooted on the left.

Definition 5.6.11 *Let $\langle Sg, Ax\rangle$ be a $\lambda_{\sqsubseteq}$-axiom system. A* $\lambda_{\sqsubseteq}$-Henkin theory over *$\langle Sg, Ax\rangle$ is a well-formed infinite context, Γ, closed under derivation from $\langle Sg, Ax\rangle$ such that:*

- *if $\exists x : \phi . P \in \Gamma$ then for some term $\Gamma \vdash t : \phi$, $P[t/x] \in \Gamma$*
- *if $(\mathtt{let}\ x : \phi\ \mathtt{in}\ t \sqsubseteq_{\chi} \mathtt{let}\ y : \psi\ \mathtt{in}\ t') \in \Gamma$, then there is a determined term $\Gamma \vdash f : \psi \to \phi$, such that $(\forall y : \psi\ .\ t[fy/x] =_{\chi} t') \in \Gamma$*

Theorem 5.6.12 *(Completeness) Let $\langle Sg, Ax\rangle$ be a $\lambda_{\sqsubseteq}$-axiom system. For $\langle Sg, Ax\rangle \rhd \Gamma \vdash B$ $\mathtt{wf}$ and $\mathtt{Rank}\ (\Gamma \vdash B) \leq 1$, if $\Gamma \models^{\mathcal{A}} B$ for all $\lambda_{\sqsubseteq}$-Henkin Models, $\mathcal{A}$, of $\langle Sg, Ax\rangle$, then $\langle Sg, Ax\rangle \rhd \Gamma \vdash B$. In particular, assuming rank < 1, for $\langle Sg, Ax\rangle \rhd \Gamma \vdash P$ $\mathtt{wf}$, if $\Gamma \models^{\mathcal{A}} P$ for all models, $\mathcal{A}$, of $\langle Sg, Ax\rangle$ then $\langle Sg, Ax\rangle \rhd \Gamma \vdash P$, and for $\langle Sg, Ax\rangle \rhd \Gamma \vdash \phi$ $\mathtt{wf}$ and $\langle Sg, Ax\rangle \rhd \Gamma \vdash r$ $\mathtt{wf}$, if $\Gamma \models^{\mathcal{A}} r : \phi$ for all models, $\mathcal{A}$, of $\langle Sg, Ax\rangle$ then $\langle Sg, Ax\rangle \rhd \Gamma \vdash r : \phi$.*

Proof: Let Γ be a consistent context. We sketch the construction of a model $\mathcal{A}$ and environment $\eta \models^{\mathcal{A}} \Gamma$ below, and use this to derive completeness.

1. Construct a maximal consistent $\lambda_{\sqsubseteq}$-Henkin theory Γ_∞ such that $\{P \mid \Gamma \vdash P\} \subseteq \Gamma_\infty$.

   The construction follows that of Theorem 4.5.16.

2. Construct the term model from open terms. Define $\tau^{\mathcal{A}}$ as the set of equivalence classes of well-structured open terms of $\lambda^{\times\to}$, $\{u \mid \Gamma_\infty, \bot \vdash u : \phi \wedge \Gamma_\infty \vdash \phi : \mathtt{Ref}\ (\tau)\}$ with respect to the same equivalence as for $\lambda_{(:)}$.

We construct a $\lambda_{\sqsubseteq}$-Henkin interpretation $\mathcal{A}$, by interpreting constant and predicate symbols syntactically. Prove that $\mathcal{A}$ satisfies the factoring condition, and so is a well-defined interpretation.

3. For $\eta' \models^{\mathcal{A}} \Gamma'$ and $\Gamma_\infty \vdash B[\eta'/\Gamma']$ `wf`, prove that $\Gamma' \models^{\mathcal{A},\eta'} B \iff \Gamma_\infty \vdash B[\eta'/\Gamma']$. This uses the characterisation of expressions in the term model given in Lemma 5.6.15 below.

4. $\mathcal{A}$ is a model of the axioms, by reasoning similar to the step on p. 130

5. For $x_1 : \phi_1, \ldots, x_n : \phi_n$ the variables in Γ, we define the Γ-environment, η, as $\langle [x_1], \ldots, [x_n] \rangle$. We can show that $\eta \models^{\mathcal{A}} \Gamma$. Thus, we have shown that an arbitrary consistent context is satisfiable.

6. The final step is to show that if $\Gamma \models^{\mathcal{A}} B$ then $\Gamma \vdash B$. This is just as for $\lambda_{(:)}$.

   Suppose $\Gamma \nvdash P$. Then $\Gamma, \neg P$ is consistent, so by the previous steps, there is an environment, η, such that $\eta \models^{\mathcal{A}} \Gamma, \neg P$, so $\Gamma \not\models^{\mathcal{A},\eta} P$, and $\Gamma \not\models P$.

   The situation for refinement typings can be reduced to that of propositions, since $\Gamma \models^{\mathcal{A},\eta} r : \phi$ is equivalent to $\Gamma \models^{\mathcal{A},\eta} r \sqsubseteq_\phi r$. The crucial point is that the permissive well-formedness rule for refinements (**Refinement**) means that $r \sqsubseteq_\phi r$ is well-formed even though r need not have refinement type ϕ.

   Thus, $\Gamma \models r \sqsubseteq_\phi r$ implies $\Gamma \vdash r \sqsubseteq_\phi r$ so, by **Refinement Elimination**, $\Gamma \vdash r : \phi$.

In order to prove step 3, we use some lemmas.

First we characterise the interpretation of refinement terms and types in the term model, $\mathcal{A}$. In the following, the semantic interpretation $[\![\_]\!]$ is to be understood as being in $\mathcal{A}$ and Γ_∞ is fixed. As discussed above, we make restrictions so that the pers correspond to equality.

For the completeness of $\lambda_{(:)}$, we used an implicit definition of `sat`. Here we will use an explicit definition.

Definition 5.6.13 *For* $\Gamma_\infty \vdash r, t$ `wf`*, define* t `sat` r *to mean* $\Gamma_\infty \vdash r \sqsubseteq' t$.

Refinement types are interpreted as pers whose equivalence classes are in one-to-one correspondence with determined terms.

Lemma 5.6.14

$$[u]\ [\![\Gamma \vdash \phi]\!](\eta)\ [u'] \iff [u]\ [\![\Gamma \vdash t]\!](\eta)\ [u'],\ \textit{for some}\ \Gamma \models^{\eta} t : \phi$$

Proof: Define a term $\Gamma \vdash t_\phi$ by induction on ϕ for each $\Gamma \vdash \phi$. Let $t_{\mathbf{1}} = *$, $t_\gamma = u$, $t_{\Sigma_{x:\phi}\psi} = \langle t_\phi, t_\psi[t_\phi/x]\rangle$, $t_{\Pi_{x:\phi}\psi} = \lambda x : \phi . t_\psi$, and $t_{(x:\phi)P} = t_\phi$.

We prove, by induction over ϕ, that $[u]$ $[\![\Gamma \vdash \phi]\!](\eta)$ $[u']$ iff $[u]$ $[\![\Gamma \vdash t_\phi]\!](\eta)$ $[u']$ and $\Gamma \vDash^{\mathcal{A},\eta} t_\phi : \phi$. The interesting cases are γ and $(x : \phi)P$. Clearly, we have $[u]$ $[\![\Gamma \vdash \gamma]\!](\eta)$ $[u']$ iff $[u]$ $[\![\Gamma \vdash u]\!](\eta)$ $[u']$. For the $(x : \phi)P$ case, if $[u]$ $[\![\Gamma \vdash (x : \phi)P]\!](\eta)$ $[u']$ then $[u] \in [\![\Gamma, x : \phi \vdash P]\!](\eta)$, so we deduce that $\Gamma \vDash^{\mathcal{A},\eta} t_\phi : (x : \phi)P$. ∎

Lemma 5.6.15 *With the rank restriction: Let $\eta \vDash^{\mathcal{A}} \Gamma$.*

1. $[u]$ $[\![\Gamma \vdash \phi]\!](\eta)$ $[u'] \iff \Gamma_\infty \vdash u =_{\phi[\eta/\Gamma]} u'$
2. *For* $\Gamma \vdash r$ `wf`, $[u]$ $[\![\Gamma \vdash r]\!](\eta)$ $[u'] \iff [u]$ $[\![t]\!]$ $[u']$ *for some* t `sat` $r[\eta/\Gamma]$
3. *For* $\Gamma_\infty \vdash P[\eta/\Gamma]$ `wf`, $\Gamma \vDash^{\mathcal{A},\eta} P \iff \Gamma_\infty \vdash P[\eta/\Gamma]$

Proof: Simultaneous induction over expressions. The inductive ordering is

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

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

$$\phi < r \sqsubseteq_\phi r'$$

$$r < F(r)$$

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

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

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

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

The proof for `let` $x : \phi$ `be` r `in` r' uses Lemma 5.5.12 (1). We prove the cases for propositions, writing $\overline{U}$ for $U[\eta/\Gamma]$.

- The $\perp$, $P \supset P'$, $\forall x : \phi . P$ and $\phi \sqsubseteq \phi'$ cases are proven as in Chapter 4.
- $\Gamma \vDash^{\mathcal{A},\eta} F(r)$ means (using the inductive hypothesis on r): for all t, t `sat` $\overline{r}$ implies $\Gamma_\infty \vdash F(t)$. Since $\Gamma_\infty \vdash F(\overline{r})$ `wf`, we must have $\Gamma_\infty \vdash \overline{r} : \phi$ for some ϕ. We assume, without loss of generality, that F : `Pred` (τ) and r has the canonical form `let` $x : \psi$ `in` t'. We must show that $\Gamma \vDash^{\mathcal{A},\eta} F(r)$ is equivalent to $\Gamma_\infty \vdash F(\overline{r})$.

  Suppose $\Gamma \vDash^{\mathcal{A},\eta} F(r)$. If $\exists x : \overline{\psi}.\top \in \Gamma_\infty$ ($\overline{\psi}$ is inhabited) then $t'[x]$ `sat` $\overline{r}$, so $\Gamma_\infty, x : \overline{\psi} \vdash F(t')$. If $\exists x : \overline{\psi}.\top \notin \Gamma_\infty$ then by maximality, $\forall x : \overline{\psi}.\perp \in \Gamma_\infty$

and by consistency, we also infer that $\Gamma_\infty, x : \overline{\psi} \vdash F(t')$. Then, by **Refinement Type Introduction**, $\Gamma_\infty, x : \overline{\psi} \vdash t' : (y : \phi)F(y)$, so by **Let Terms** $\Gamma_\infty \vdash \texttt{let}\ x : \overline{\psi}\ \texttt{in}\ \overline{t'} : (y : \phi)F(y)$, and using subject refinement $\Gamma_\infty \vdash \overline{r} : (y : \phi)F(y)$. Hence, using **Predicates**, $\Gamma_\infty \vdash F(\overline{r})$.

Conversely, if $\Gamma_\infty \vdash F(\overline{r})$ then, by **Refinement Type Introduction**, $\Gamma_\infty \vdash \overline{r} : (y : \phi)F(y)$, so by the definition of $\texttt{sat}$ and using subject refinement, if $t\ \texttt{sat}\ \overline{r}$ then $\Gamma_\infty \vdash t : (y : \phi)F(y)$, and so $\Gamma_\infty \vdash F(t)$.

- $\Gamma \vDash^{\mathcal{A},\eta} r \sqsubseteq_\phi r'$: We show that this is equivalent to $\overline{r} \lesssim_{\overline{\phi}} \overline{r'}$ which, in turn, is equivalent to $\Gamma_\infty \vdash \overline{r} \sqsubseteq_{\overline{\phi}} \overline{r'}$.

  Suppose $\Gamma \vDash^{\mathcal{A},\eta} r \sqsubseteq_\phi r'$. By Lemma 5.6.5, for all $t'\ \texttt{sat}\ \overline{r'}$ there exists $t\ \texttt{sat}\ \overline{r}$ such that $t\ [\![\Gamma \vdash \phi]\!](\eta)\ t'$ (where t and t' are representatives of equivalence classes). By the inductive hypotheses, and reasoning as in Chapter 4, p. 133, we have $\Gamma_\infty \vdash t =_{\overline{\phi}} t'$. Now suppose $\Gamma_\infty \vdash \overline{r'} \sqsubseteq_{\overline{\phi}} t''$. By Corollary 5.5.6, there exists a (determined) term t' such that $\Gamma_\infty \vdash t =_{\overline{\phi}} t''$ and $t'\ \texttt{sat}\ \overline{r'}$. Then there exists a t such that $t\ \texttt{sat}\ \overline{r}$ and $\Gamma_\infty \vdash t =_{\overline{\phi}} t'$. By the definition of $\sqsubseteq$, we have $\Gamma_\infty \vdash \overline{r} \sqsubseteq_{\overline{\phi}} t''$, and so $\Gamma_\infty \vdash \overline{r} \sqsubseteq_{\overline{\phi}} t'' =_{\overline{\phi}} t =_{\overline{\phi}} t'$, and hence $\Gamma_\infty \vdash \overline{r} \sqsubseteq_{\overline{\phi}} t'$. Thus, $\overline{r} \lesssim_{\overline{\phi}} \overline{r'}$.

  Conversely, suppose that $\overline{r} \lesssim_{\overline{\phi}} \overline{r'}$ and $t'\ \texttt{sat}\ \overline{r'}$. Then $\Gamma_\infty \vdash \overline{r'} \sqsubseteq_{\overline{\phi}} t'$ so $\Gamma_\infty \vdash \overline{r} \sqsubseteq_{\overline{\phi}} t'$. By Corollary 5.5.6, there exists a t such that $t\ \texttt{sat}\ \overline{r}$ and $t =_\phi t'$, so by Lemma 5.6.5, $\Gamma \vDash^{\mathcal{A},\eta} r \sqsubseteq_\phi r'$.

  Now, we show that $\overline{r} \lesssim_{\overline{\phi}} \overline{r'} \iff \Gamma_\infty \vdash \overline{r} \sqsubseteq_{\overline{\phi}} \overline{r'}$.

  The reasoning is the same as the corresponding step in the completeness proof for the $\lambda_?$-calculus. We can assume, without loss of generality, that the terms are in canonical form, and since $\overline{r}, \overline{r'} : \overline{\phi}$ by the assumption of well-formedness, Lemma 5.5.7 gives canonical forms which are equal to the terms at $\overline{\phi}$. Hence we have

  $$\texttt{let}\ x : \psi_1\ \texttt{in}\ t_1 \lesssim_{\overline{\phi}}^{\Gamma_\infty} \texttt{let}\ y : \psi_2\ \texttt{in}\ t_2$$

  If $\psi_2 \in \Gamma_\infty$ then, by Lemma 5.5.15, there exists a term $\Gamma_\infty, y : \psi_2 \vdash t : \psi_1$ such that $\Gamma_\infty, y : \psi_2 \vdash t_1[t] =_\phi t_2$.

  Now suppose ψ_2 is not inhabited. Let t be any term in the type below $\overline{\phi}$. Such a term exists because of the assumption that all types are inhabited. Then we have, $\Gamma_\infty, y : \psi_2 \vdash \bot$, so Lemma 4.3.17 implies that $\Gamma_\infty, y : \psi_2 \vdash t : \phi$. Similarly, $\Gamma_\infty, y : \psi_2 \vdash t_1[t/x] =_\chi t_2\ \texttt{wf}$, and $\Gamma_\infty, y : \psi_2 \vdash t_1[t/x] =_\chi t_2$.

Then, in the context Γ_∞, the term $\texttt{let } x : \psi_1 \texttt{ in } t_1$ refines (at ϕ) to $\texttt{let } y : \psi_2 \texttt{ in } (\texttt{let } x : \psi_1 \texttt{ in } t_1)$, which refines to $\texttt{let } y : \psi_2 \texttt{ in } t_1[t[y]]$, and this equals $\texttt{let } y : \psi_2 \texttt{ in } t_2[y]$.

Hence $\Gamma_\infty \vdash \overline{r} \sqsubseteq_{\overline{\phi}} \overline{r'}$.

∎

The characterisation of the interpretation of expressions in Lemma 5.6.15 is specific to the term model. For example, in the term model, $[\![?_{\texttt{nat}\to\texttt{nat}}]\!]$ consists of equivalence classes of terms of type $\texttt{nat} \to \texttt{nat}$, but in the full set-theoretic function hierarchy (Definition 3.4.3), the classes consist of arbitrary functions.

In Lemma 5.6.6, we showed that determined terms are interpreted as a single equivalence class. We can give a simple characterisation of that class in the term model.

Lemma 5.6.16 *With the rank restriction:*

$$[u_1]\ [\![\Gamma \vdash t]\!](\eta)\ [u_2] \iff u_1 \ \texttt{sat}^{\Gamma_\infty}\ t[\eta/\Gamma] \text{ and } u_2 \ \texttt{sat}^{\Gamma_\infty}\ t[\eta/\Gamma]$$

Proof: Follows from Lemmas 5.6.14 and 5.6.15. ∎

By Lemma 5.6.15, we see that in the term model, arbitrary refinement terms are interpreted as pers consisting of classes of this form.

Corollary 5.6.17 *With the rank restriction, if* $\Gamma \vdash r : \phi$*, then:*
$\Gamma \vdash r : (x : \phi)P$ *iff for all determined* t*, if* $\Gamma \vdash r \sqsubseteq_\phi t$ *then* $\Gamma \vdash P[t/x]$

Proof: Both statements have the same interpretation, so the result follows from soundness and completeness. ∎

Remark 5.6.18 The $r : \phi$ step in the proof of completeness suggests a general strategy for proving that $r : (x : \phi)P$. First express r in canonical form as $\texttt{let } y : \psi \texttt{ in } t$. Then prove that $\forall y : \psi \,.\, P[t/x]$. We know that if the refinement typing is true then, by the completeness of the propositional fragment, this proposition is provable. Hence, we conclude that $\texttt{let } y : \psi \texttt{ in } t \ :\ (x : \phi)P$ and so, by subject refinement, $r : (x : \phi)P$.

5.7 Conclusion

In this chapter we presented the $\lambda_\sqsubseteq$-calculus, a refinement calculus based on the notions of refinement term and refinement type. In the next chapter we discuss how this could give a basis for a more comprehensive theory of software development.

Chapter 6

Conclusions and Further Work

6.1 Conclusions

In this thesis we constructed a canonical refinement calculus based on the lambda calculus and classical first-order predicate logic, and studied its proof theory and semantics. Let us summarise the main points of this formalisation:

Formalisation of Refinement

programs	—	lambda terms
abstract programs	—	refinement terms
specifications	—	refinement types

We gave a set-theoretic semantics based on Henkin models for which the calculus was proven sound and complete. As far as we know, this is the first proof of completeness of any refinement calculus.

A key feature of this approach was the construction of the refinement calculus in a modular fashion, as the combination of two orthogonal extensions to the underlying programming language (in this case, the simply-typed lambda calculus). These subcalculi are interesting in their own right as they provide separate analyses of structured specifications and non-logical decomposition. 'Full' refinement, then, can be factored into logical equational reasoning and simple decomposition. We used a two-level formalisation of specifications, consisting of an underlying level of program types, and a more expressive level of program properties.

We now discuss how the issues raised in the introductory chapter have been addressed.

We set ourselves the task of investigating the logic and semantics of refinement calculi, and saw that it is possible to induce a refinement calculus from an extensional program logic and the equational theory of a programming language (in the canonical case of the simply-typed lambda calculus). We construct structured specifications — refinement types — and have a notion of equality at a specification. The syntactic category of refinement terms consists of combinations of specifications and programs.

First-order logic and the simply-typed lambda calculus can be modelled using Henkin interpretations, and given models of particular lambda theories and logics, we can form a model of the corresponding refinement calculus.

The refinement calculus is completely characterised by the underlying theories in the sense that it is complete (given certain restrictions) with respect to the class of models induced from the models of the underlying theories. Moreover, we can use the completeness theorems to deduce (under these restrictions) the conservativity of refinement calculi over program logics and (equational theories of) programming languages.

The refinement calculus can be thought of as being constructed from two subcalculi — a calculus of refinement terms and a calculus of refinement types. These calculi are useful in their own right. For example, we saw that terms of the $\lambda_?$-calculus can be evaluated in a program-like fashion. A factorisation theorem justifies us in regarding the subcalculi as being orthogonal extensions to the programming language and program logic.

The factorisation suggests an interesting possibility for the construction of a modular refinement tool, in which checking program correctness is a combination of type checking and theorem proving. The modularity would come from constructing a verifier, or 'specification checker', from an existing theorem prover (which we can think of as an oracle) and a type checker, for a program logic and programming language respectively. Then we would write a separate program to handle simple refinement and combine the two to get the refinement tool.

Although this thesis has presented a simple calculus, we believe that we have motivated the general methodology of inducing a refinement calculus from a programming language together with some logic, rather than constructing a development methodology from scratch. We believe that, from a theoretical standpoint, this approach is more likely to be useful for formal methods. It seems rather naive to expect programmers to treat "programming as a mathematical activity", working directly in some refinement calculus. Indeed, this is a dangerous viewpoint insofar as it leads to taking the mathematical formalisation as the primary object of interest, thus distancing theory from actual programming practice. While formalisation of the relevant concepts is indeed desirable, it is more realistic to provide a theoretical underpinning for tool support. The intention here is not that the calculus should actually be used directly, but that it serves as an underlying theory.

We believe that factoring a complicated calculus into two subcalculi has proven its worth as a research methodology. Many of the extensions suggested below could also be first studied as extensions to each of the two subcalculi.

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. The calculus could provide a foundation for other specification based formalisms, and we will make some specific suggestions below.

6.1.1 Refinement Terms

We would be interested to see how this calculus might be usefully combined with work on logical frameworks [Pfe96]. The use of logical variables there is an example of underdeterminism.

Although Lego has a richer type system than those studied here, a fragment of it could be studied using $\lambda_?$ as a metalanguage. It would be interesting to use $\lambda_?$ as a metalanguage for giving a semantics to Lego and to prove some metatheoretic results.

That the concept of underdeterminism arises both in computing science and in linguistics strengthens our belief that it is an important concept in the study of general informatics.

We could annotate the types with simple properties, such as whether or not an exception is raised at some point. This would be a useful intermediate stage between the calculus of refinement terms and the full refinement calculus.

6.1.2 Refinement Types

A number of systems have intersection and union types. This can not always be expressed in our system. For example, no refinement type corresponds to $\mathtt{even} \to \mathtt{even} \wedge \mathtt{odd} \to \mathtt{odd}$. Hayashi's [Hay94b] intersection and union are even more powerful. It is not clear whether the degree of expressivity in this system is particularly useful (but see the comments about quotients below).

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, where type-checking can generate proof obligations.

6.2 Technical Extensions and Conjectures

We make some suggestions for various technical extensions and results for the calculi. The first two extensions, in particular, are aimed at tidying up the two main loose ends in this work — the restrictions on completeness in Chapter 4 and 5, and the restrictions on dependent refinement types in Chapter 5.

The final ideas (11, 12 and 13) are suggestions for reformulations of the calculi.

1. We had to place restrictions on the statements of completeness for $\lambda_{(:)}$ and $\lambda_{\sqsubseteq}$ due to the underlying mismatch between the refinement type $\phi \to \psi$ and its per semantics. For example, we could not prove that $\lambda f : \texttt{nat} \to \texttt{nat} \,.\, f2$ has refinement type $(\texttt{even} \to \texttt{nat}) \to \texttt{nat}$. The problem is that as the rules stand, for the abstraction $\lambda x : \phi'.t$ to have refinement type $\phi \to \psi$ we require $\phi' \sqsubseteq \phi$ but, in this case, the refinement goes in the opposite direction, $\texttt{nat} \to \texttt{nat} \sqsubseteq \texttt{even} \to \texttt{nat}$. It is not sound, in general, to say that $\lambda x : \phi'.t : \phi \to \psi$ when $\phi \sqsubseteq \phi'$ (and the other conditions). However, the following rule does appear to be sound. Define $\texttt{dom}\ \phi$ to be the set of terms with refinement type ϕ. Then,

   $$\frac{\Gamma, x : \phi' \vdash t : \psi \quad \Gamma \vdash \phi' \sqsubseteq \phi}{\Gamma \vdash \lambda x : \phi.t : \phi' \to \psi} \ (\texttt{dom}\ \phi' \subseteq \texttt{dom}\ \phi)$$

   The combination of $\phi' \sqsubseteq \phi$ and $\texttt{dom}\ \phi' \subseteq \texttt{dom}\ \phi$ (*i.e.* $x\ \phi'\ x \Rightarrow x\ \phi\ x$) means that $\texttt{dom}\ \phi' = \texttt{dom}\ \phi$, and ϕ' is a *quotient* of ϕ. The rule would let us prove

   $$\lambda f : \texttt{nat} \to \texttt{nat} \,.\, f2 \ : \ (\texttt{even} \to \texttt{nat}) \to \texttt{nat}$$

   and may be enough to strengthen Theorems 4.5.16 and 5.6.12 (completeness of $\lambda_{(:)}$ and $\lambda_{\sqsubseteq}$) to unrestricted versions.

   Similarly, this rule is sound, and may be admissible

   $$\frac{\phi' \text{ a quotient of } \phi}{?_{\phi} \sqsubseteq_{\phi'} \phi'}$$

A provable consequence of the rule is $?_{\mathtt{nat}\to\mathtt{nat}} \sqsubseteq_{\mathtt{even}\to\mathtt{nat}} ?_{\mathtt{even}\to\mathtt{nat}}$.

The combination of quotienting and the subset type-like refinement types we use might be interesting for specification. Many specification formalisms use some form of quotienting and it would be interesting to see it arise naturally here in order to get completeness.

2. In Chapter 5, we used nondependent refinement types to avoid the combination of underdetermined terms with refinement types. The same problem arose in Chapter 3 at the level of terms and led us to introduce let-terms. We could introduce a notion of let-types, therefore, where $\mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ \psi$ has the obvious meaning. The rule of introduction would be

$$\frac{\Gamma \vdash r : \Pi_{x:\phi}\psi \quad \Gamma \vdash r' : \phi}{\Gamma \vdash rr' : \mathtt{let}\ x : \phi\ \mathtt{be}\ r'\ \mathtt{in}\ \psi}$$

We would have a special rule for stubs.

$$\frac{\Gamma \vdash r : \phi \quad \Gamma, x : \phi \vdash \psi\ \mathtt{wf}}{\Gamma \vdash \mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ ?_\psi = ?_{\mathtt{let}\ x:\phi\ \mathtt{be}\ r\ \mathtt{in}\ \psi}}$$

The refinement rules for let-types would be analogous to the equalities for let-terms. To give these, it is convenient to extend the singleton type notation of Remark 4.3.9 to arbitrary refinement terms: we write $\{r\}_\phi$ for the refinement type $(x : \phi)r \sqsubseteq_\phi x$. For example:

$$\mathtt{let}\ x : \phi\ \mathtt{be}\ t\ \mathtt{in}\ \psi = \psi[t/x]$$

$$\mathtt{let}\ x : \phi\ \mathtt{be}\ r\ \mathtt{in}\ \{x\}_\phi = \{r\}_\phi$$

$$\mathtt{let}\ z : \Sigma_{x:\phi}\psi\ \mathtt{be}\ r\ \mathtt{in}\ \{\pi_1 z\}_\phi = \{\pi_1 r\}_\phi$$

3. It may be that any maximal first-order $\lambda^{\times\to}$-Henkin theory is a first-order $\lambda_?$-Henkin theory. That is, the witness condition for refinements may follow from the condition for existentials.

4. We conjecture that the $\lambda_?$ factoring condition is equivalent to the satisfaction of a choice axiom (or skolemisation):

$$\models \forall x : \tau.\exists y : \sigma.P \ \supset\ \exists f : \tau \to \sigma.\forall x : \tau.P[fx/y]$$

for all P.

5. We conjecture that the $\lambda_{(:)}$-calculus is a conservative extension of Aspinall's $\lambda_{\leq\{\}}$ calculus [Asp95].

6. If we define a notion of 'strong' well-formedness for $\lambda_{(:)}$ and $\lambda_{\sqsubseteq}$, which requires the appropriate refinement typings for equalities and predications (*i.e.* $t =_\phi t'$ when $t, t' : \phi$), then it should be that for a strongly well-formed P, if $\Gamma \models P$ then $\Gamma \vdash P$.

   We should formulate the connection between the different restrictions for completeness of $\lambda_{(:)}$ and $\lambda_{\sqsubseteq}$.

7. In Remark 4.3.6 we suggested that axioms in $\lambda_{(:)}$ (and $\lambda_{\sqsubseteq}$) could be given in a particular form and this should be investigated further.

8. We conjecture that the Let Term Equality **Abstractions** in Chapter 5 is equivalent to the typed form, and so we could avoid the meta-judgement in the formal system.

9. A satisfactory account of the subrelations in Section 5.5 remains to be given. For example, does **Eta** commute with the other rules in $\sqsubseteq^s$?

10. We have taken refinements (of both kinds) to be atomic propositions rather than separate judgement classes. We conjecture that restricted calculi in which the atomic propositions are equalities, and the refinements are separate judgement classes would also be complete. If so, the full systems would be conservative extensions.

    This could be considered a more natural approach, as specification using refinement itself is more complex than just using the underlying program logic.

11. We conjecture that $\lambda_?$ and $\lambda_{\sqsubseteq}$ are complete for the alternative semantics of Remark 3.4.12. This would let us avoid using the factoring conditions.

12. We could use the suggestive notation $\langle r, (x : \phi) r' \rangle$ for $\texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ \langle x, r' \rangle$. A dependent form of the refinement rule **Pairs** would be:

$$?_{\Sigma_{x:\phi}\psi} \sqsubseteq \langle ?_\phi, (x : \phi) ?_\psi \rangle$$

13. There is some overlap between the refinement rules for terms and refinement types. We could combine the two judgements into the form $r : \phi \sqsubseteq r' : \phi'$,

meaning $r \sqsubseteq_\phi r'$ and $\phi \sqsubseteq \phi'$. Some natural rules would be:

$$\frac{\Gamma \vdash \phi \sqsubseteq \mathbf{1}}{\Gamma \vdash ?_\phi : \phi \sqsubseteq * : \mathbf{1}}$$

$$\frac{\Gamma \vdash \chi \sqsubseteq \Pi_\phi \psi}{\Gamma \vdash ?_\chi : \chi \;\sqsubseteq\; \lambda y : \phi . ?_\psi : \Pi_{x:\phi}\psi}$$

Using the notation introduced above we have:

$$\frac{\Gamma \vdash \chi \sqsubseteq \Sigma_{x:\phi}\psi}{\Gamma \vdash ?_\chi : \chi \;\sqsubseteq\; \langle ?_\phi, (x:\phi)?_\psi \rangle : \Sigma_{x:\phi}\psi}$$

The overlap is clear when giving the rules for let-types. We could have, for example:

$$\frac{\Gamma, x:\phi \vdash r:\psi \quad \Gamma \vdash t:\phi}{\Gamma \vdash \texttt{let}\ x:\phi\ \texttt{be}\ t\ \texttt{in}\ r : \texttt{let}\ x:\phi\ \texttt{be}\ t\ \texttt{in}\ \psi \;=\; r[t/x] : \psi[t/x]}$$

14. We could base the refinement calculus on primitive definitions of $\sqsubseteq$ and $=_\phi$ rather than $\sqsubseteq_\phi$. This might be more natural, as we usually omit the subscripted ϕ anyway.

6.3 Operational Semantics

As for the denotational semantics, we can give a modular operational semantics, by first giving a semantics to the subcalculi. Here we will just outline how to do this for the subcalculi. We restrict ourselves to the specific axiom systems of booleans and naturals.

6.3.1 Refinement Terms

Because terms of the calculus are a mixture of specification and program, we do not inherit a notion of reduction from the lambda-calculus, but we can give an operational semantics based on satisfaction of terms and properties by canonical terms.

The canonical terms are the closed terms of the form:

$$c ::= \texttt{b} \;\mid\; \texttt{n} \;\mid\; * \;\mid\; \langle c, c' \rangle \;\mid\; \lambda x : \sigma . t$$

where t is an arbitrary determined term, `b` is one of the booleans `true` and `false`, and `n` is a numeral. The operational semantics is given in Figures 6.1 to 6.4, and consists of an evaluation relation on determined terms $t \Downarrow c$, together with three

$$* \Downarrow * \quad \mathtt{n} \Downarrow \mathtt{n} \quad \mathtt{b} \Downarrow \mathtt{b}$$

$$\frac{t \Downarrow c \quad t' \Downarrow c'}{\langle t, t' \rangle \Downarrow \langle c, c' \rangle}$$

$$\lambda x : \sigma.t \Downarrow \lambda x : \sigma.t$$

$$\frac{t \Downarrow \langle c, c' \rangle}{\pi_1(t) \Downarrow c} \qquad \frac{t \Downarrow \langle c, c' \rangle}{\pi_2(t) \Downarrow c'}$$

$$\frac{t \Downarrow \lambda x : \sigma.t'' \quad t''[t'/x] \Downarrow c}{tt' \Downarrow c}$$

$$\frac{t'[t/x] \Downarrow c}{\mathtt{let}\ x : \sigma\ \mathtt{be}\ t\ \mathtt{in}\ t' \Downarrow c}$$

Figure 6.1: Evaluation

mutually recursive relations: an extensional equality on canonical terms $c =_\sigma c'$; a satisfaction relation between canonical and underdetermined terms, $c \vDash r$; and the validity of propositions $\vDash P$.

Since we want equality and refinement to be extensional for determined terms, but not for arbitrary underdetermined terms we first define a typed extensional equality on canonical terms, $=_\sigma$, by induction on σ.

The second component of the operational semantics is a satisfaction relation, $c \vDash r$. For example, $\lambda x : \sigma.t \vDash \lambda x : \sigma.r$ when for all $c : \sigma$, for all $d : \sigma$, $d \vDash t[c] \Rightarrow d \vDash r[c]$. In fact, since there is a unique canonical form equal to $t[c]$, we could have written $t[c] \vDash r[c]$, but we do not assume the uniqueness here.

Now we can define $\vDash r : \sigma$ as: for all $c \vDash r$, $c : \sigma$. Next, for $\vDash r : \sigma$, $\vDash r' : \sigma$, we say define validity of refinement, $\vDash r \sqsubseteq_\sigma r'$ as: for all $c' \vDash r'$, there exists $c \vDash r$ such that $c =_\sigma c'$.

Remark 6.3.1 We say that a $\lambda_?$-axiom system is *operationally complete* if whenever a refinement is operationally valid, then it is provable. An example of a signature which is not operationally complete was given in Example 3.4.4. We have $\vDash \lambda n : \mathtt{nat}.\mathtt{let}\ y : \mathtt{nat}\ \mathtt{in}\ \mathtt{succ}\ y \sqsubseteq \mathtt{id}_{\mathtt{pos}}$, where $\mathtt{id}_{\mathtt{pos}}$ is the identity on positive naturals $\lambda n : \mathtt{nat}.\mathtt{cond}(\mathtt{eq}(n, \mathtt{0}), \mathtt{1}, n)$, but without any form of recursion we cannot define a predecessor term and actually prove the refinement. It is important in practice to ensure that we only use operationally complete signatures, so as to avoid writing specifications which can not be implemented, yet are intuitively implementable.

We saw that $\lambda_?$-axiom systems are (denotationally) complete with respect to a class of Henkin models with a factoring condition. It might be that operationally

- $c \vDash *$ when $c \equiv *$
- $c \vDash \langle r_1, r_2 \rangle$ when $c \equiv \langle c_1, c_2 \rangle$ and $c_1 \vDash r_1, c_2 \vDash r_2$
- $c \vDash \lambda x : \sigma.r$ when $c \equiv \lambda x : \sigma.t$, for all $c' : \sigma$, $t[c'] \Downarrow d$ and $d \vDash r[c']$
- $c \vDash ?_\sigma$ when $c : \sigma$
- $c \vDash \pi_1(r)$ when there exists c' such that $\langle c, c' \rangle \vDash r$
- $c \vDash \pi_2(r)$ when there exists c' such that $\langle c', c \rangle \vDash r$
- $c \vDash r_1 r_2$ when there exists $\lambda x : \sigma.t \vDash r_1$, $c_2 \vDash r_2$, such that $c \vDash t[c_2/x]$
- $c \vDash \texttt{let } x : \sigma \texttt{ be } r \texttt{ in } r'$ when there exists $c' \vDash r$ such that $c \vDash r'[c'/x]$

Figure 6.2: Satisfaction

$$* =_{\mathbf{1}} *$$
$$\langle c_1, c_2 \rangle =_{\sigma \times \tau} \langle c_1', c_2' \rangle \text{ when } c_1 =_\sigma c_1' \text{ and } c_2 =_\tau c_2'$$
$$\lambda x : \sigma.t =_{\sigma \to \tau} \lambda x : \sigma.t' \text{ when for all } c : \sigma, \text{ for all } d : \tau, \ d \vDash t[c/x] \text{ iff } d \vDash t'[c/x]$$

Figure 6.3: Equality of Canonical Terms

- $\vDash \bot$ never
- $\vDash P \supset P'$ when $\nvDash P$ or $\vDash P'$
- $\vDash \forall x : \sigma.P$ when for all $c : \sigma$, $\vDash P[c/x]$
- $\vDash r \sqsubseteq_\sigma r'$ when for all $c' \vDash r'$, there exists $c \vDash r$ such that $c =_\sigma c'$.

Figure 6.4: Validity of Propositions

complete theories are complete for arbitrary Henkin models.

6.3.2 Refinement Types

We can also give an operational semantics to the $\lambda_{(:)}$-calculus based on the satisfaction of terms and properties by canonical terms. There are three mutually recursive components to the operational semantics: an extensional equality on canonical terms, $c =_\phi c'$; the satisfaction of underdetermined terms by canonical terms, $c \vDash t$; and the validity of propositions, $\vDash P$.

We define extensional equality, $c =_\phi c'$, by induction on ϕ. Then the satisfaction of refinement types by canonical terms, $c \vDash \phi$, can be defined as $c =_\phi c$. Now we can define $\vDash t : \phi$ as for all $c \vDash t$, $c \vDash \phi$. Finally, we say that $t =_\phi t'$ is operationally valid when for all $c \vDash t$, and for all $c' \vDash t'$, $c =_\phi c'$.

6.4 Annotations

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.

Annotating program text with propositions was first suggested by Floyd [Flo67] and is now used in many refinement calculi (*e.g.* [Bun97]) to facilitate reasoning and to express local assumptions.

Extending the type system of $\lambda^{\times\rightarrow}$ to refinement types gives a simple notion of program annotation, where variables on abstractions are labelled with logical information. Although we do not have explicit annotations in our calculus, we can define certain forms. For $r : \phi$, define

$$\begin{array}{ll} \text{assertion} & r \mid (x : \phi)P = ?_{(x:\phi)\ r \sqsubseteq_\phi x\ \wedge\ P} \\ \text{guard} & (x : \phi)P \to r = ?_{(x:\phi)\ P\ \supset\ r \sqsubseteq_\phi x} \end{array}$$

As we showed in Remark 4.3.22, we can also combine guards with the refinement types. For example:

$$P \to (x : \phi)Q \equiv (x : (P \to \phi))P \supset Q$$

$$P \to (\phi \to \psi) \equiv (P \to \phi) \to (P \to \psi)$$

Assertions could be treated similarly.

We could, however, consider a calculus with true annotations. One possibility would be to extend the simply-typed lambda calculus with terms of the form $P \to t$ and $t\,|\,P$ where P is a proposition, and operational meanings

$$c \vDash P \to t \text{ when } \vDash P \text{ implies } c \vDash t$$

$$c \vDash t\,|\,P \text{ when } \vDash P \text{ and } c \vDash t$$

Thus if $\nvDash P$, any c (of appropriate type) will satisfy $P \to t$.

We could define a (meaning preserving?) translation $(\_)^\circ$ from a subset of $\lambda_{(:)}$ to the annotation calculus, with

$$(\lambda x : \sigma\,|\,P.t)^\circ = \lambda x : \sigma.P \to t^\circ$$

An alternative formulation of the refinement calculus would be to take annotations as primitive. Then we could define 'set-theoretic' (as opposed to per-theoretic) specifications as

$$?_{(x:\tau)P} = \texttt{let } x : \tau \texttt{ in } (x\,|\,P)$$

6.5 Search Calculi

The thesis of Pym [Pym90] presents a theory of *proof search*. One idea developed there (and also in [PW90]) is to give a hierarchy of calculi each of which can be regarded as the metatheory of the next and in which the search space for proofs is increasingly constrained. This idea could be applied to the present work on refinement.

Refinement is traditionally formulated as a generalised equality and, as pointed out after Lemma 3.2.17, this is also the style of the $\lambda_?$- and $\lambda_\sqsubseteq$-calculi. The refinement of specification ϕ to program t, $\phi \sqsubseteq t$, is thought of as "t is less than ϕ in the refinement ordering". The calculi do not contain rules for directing a search. Now we commented in Section 1.4 that we could consider a search-oriented (rather than equational) refinement calculus. Then t is seen as a solution to the search for a program to satisfy ϕ.

In Section 5.5 we defined a number of auxiliary relations. The relationship between $\sqsubseteq_\phi$ and $\rightsquigarrow$ is interesting because it mirrors the difference between refinement and search. Following [PW90], we could consider a hierarchy of subsystems of the full equational refinement theory. For example:

$$\Gamma, x : \sigma \Rightarrow \sigma \qquad\qquad \Gamma \Rightarrow \mathbf{1}$$

$$\frac{\Gamma \Rightarrow \sigma \quad \Gamma \Rightarrow \tau}{\Gamma \Rightarrow \sigma \times \tau} \qquad\qquad \frac{\Gamma, y : \sigma \Rightarrow \tau}{\Gamma \Rightarrow \sigma \to \tau}$$

$$\frac{\Gamma, x : \tau, y : \tau' \Rightarrow \sigma}{\Gamma, z : \tau \times \tau' \Rightarrow \sigma}$$

$$\frac{\Gamma, x : \tau' \Rightarrow \sigma}{\Gamma, y : \tau \to \tau', z : \tau \Rightarrow \sigma}$$

Figure 6.5: Type Inhabitation

1. $\Gamma \Rightarrow \sigma$ (type inhabitation)

2. $\Gamma \vdash r \rightsquigarrow t$ (coding)

3. $\Gamma \vdash r \sqsubseteq^s t$ (simple refinement)

4. $\Gamma \vdash r \sqsubseteq t$ (nonlogical refinement)

We consider first the simple task of finding a program to inhabit a type. Figure 6.5 presents backward oriented rules which may be used to automate such a search. These rules can be viewed as reformulations of refinements of the form $\Gamma \vdash ?_\sigma \sqsubseteq_\sigma t$, where t could be read off the rules, or included as a labelled deduction.

We can then consider rules for proving inhabitation of an arbitrary r. This coding can be thought of as a 'big step' refinement. The rules for equality can be omitted from $\sqsubseteq^s$ as equality is orthogonal to refinement in the sense that if $r \sqsubseteq t$, then there exists t' such that $r \sqsubseteq^s t'$, and $t = t'$.

This system can then be embedded in a more general system for proving refinements of the form $r \sqsubseteq t$, where instead of thinking of refinement as a search for inhabitation of a specification, we think of it as a generalised equality. Indeed, we could view the refinement calculus as being a means of representing stages in the search for inhabitants of specifications, and so as a metatheory for a search calculus.

We could also study the difference between equational and search-oriented refinement from a semantic viewpoint. Perhaps a search calculus could be modelled using a possible worlds semantics.

6.6 Logical Variables

A related idea to the distinction between equational and search-oriented refinement is presenting the refinement calculus in both natural deduction and sequent calculus styles. We chose what is essentially a sequent calculus presentation of natural deduction because it is better suited to proof search. However, as pointed out in the previous section, the calculus is not search-oriented anyway. A natural deduction style presentation would be clearer, though. Hence, it might be best to give the equational theory in a natural deduction style, translate a search-oriented calculus into the sequent calculus style, and then prove them equivalent. We could present the refinement calculus in true sequent style using logical variables.

This would effectively be a unification of the two paradigms of refinement, in the sense that any search theory gives rise to an associated equational theory, and the rules of any equational theory can be restricted to a search-oriented subset.

6.7 Second Order: Data Refinement

An extension to the second order (polymorphic) lambda calculus offers hope of combining program and data refinement in one formalism, as well as allowing specification by observational equivalence.

The account of specifications in Chapter 4 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. Moreover, the combination of refinement types and existential variables would be a natural way of augmenting the work in [MP88] with equations.

It will be interesting to see how the calculi can be extended to the second order. This should reduce the number of rules through the impredicative encodings of unit and product (as well as sum). More importantly, the calculus would then be able to express inductive types and iteration. We ought to get derived refinement rules for data types like `nat` and `list[X]`.

We can define abstract data types using existential types. The question of the connection between this view of data refinement and that of the methodologists (such as Back and Morgan) and the categorical studies of Hoare [Hoa87] and Tennent *et.al.* [KOP+97] then arises. The use of parametric polymorphism might throw some light on the use of relations in model-oriented data refinement.

6.8 Full Recursion

The introduction of nontermination (using say, Plotkin's computational metalanguage) will raise particular issues. However, it should be possible to integrate nontermination smoothly. Our modular approach should help in tackling this problem.

Traditional type-theoretic approaches (such as [NPS90]) cannot handle nontermination since all terms terminate. For example, naively adding full recursion to the simply-typed lambda calculus results in inconsistency (see p. 112). However, $\lambda_{(:)}$ is 'type-theoretic' without maintaining a Curry-Howard isomorphism.

Many complications arise with nontermination when underdeterminism is modelled as nondeterminism. This is most clearly seen from a semantic point of view. Such models are based on powerdomains. However, simple sets (of interpretations of determined terms, possibly nonterminating) would seem to suffice here. This reflects the intuition that underdeterminism is something 'above' computation and does not 'interact' with it.

By contrast, the powerdomain approaches raise a myriad questions concerning how exactly the ? and $\perp$ interact. For example, some authors have made a distinction between erratic, demonic and angelic nondeterminism, but it is not clear that these notions transfer to underdeterminism.

As for extending $\lambda_{(:)}$ to nontermination, the distinction between partial and total correctness then arises (and similarly for refinement in $\lambda_{\sqsubseteq}$). Although we could still model specifications with pers, we might possibly want to add some condition such as downward closure. This does not contradict the comment above that nontermination should not interact with underdeterminism because (presumably) we would still model $\lambda_{\sqsubseteq}$-terms as sets of equivalence classes.

Some other questions we might address are the interaction between refinement types and recursion (*e.g.* whether we should have $\mu x : \phi.r : \mu x : \phi.\psi[x]$), and how do we ensure progress during recursive refinement (*i.e.* avoiding refining to $\perp$). Recursion at the level of specifications is a separate issue. It is possible that work on subtyping systems for recursive types would be useful there.

6.9 Program Transformation

To many people, "program refinement" and "program transformation" are synonyms. While the basic idea of either can be generalised to include the other, it is useful to draw a distinction between refinement of a logical specification into concrete code, and transformation of concrete program code into 'better' code. This separation of concerns corresponds to the idea that a (particular) program development calculus based on the simply-typed lambda calculus can be factored into two orthogonal extensions to the lambda calculus, where program development consists of two stages: developing functionally correct code which satisfies an extensional specification, and then the application of optimising transformations to produce efficient code.

Reconciling these two approaches would address the common concern that an emphasis on methodologies for developing structured programs can result in inefficient code, while optimal programs tend to be hard to understand. Instead, both approaches can be combined: programs can be constructed by refinement from a specification, and then optimised using transformations. As the transformations, and indeed the refinements, are recorded, it is possible to view a program at various levels of abstraction, from non-optimal but clear code to logical specification.

Rather than just allow arbitrary, and possibly incorrect, transformations, we could give a transformation calculus, where an applied theory consists of a particular intensional feature, such as time complexity, and a collection of atomic transformations which respect this. The intensional feature is incorporated as an extended type and transformation rules are generated from the atomic rules.

An interesting variety of intensional features could be incorporated into this framework. For example, work on formalising program style could perhaps be recast in this way.

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 $\mathtt{nat} \to \mathtt{nat}$, 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 $\mathtt{even} \to \mathtt{nat}$.

6.10 Abstract Viewpoint

The semantics of refinement calculi could be extended to a more general categorical framework. Previous work has tended to characterise refinement in terms of either inclusion of models, or of preservation of properties, from which proof rules are then derived. We took the opposite approach by giving an explicit axiomatisation of refinement in order to get a tractable syntactic definition.

Nevertheless, there is the question of an abstract characterisation in a general semantic framework. For example, the factorisation of refinement in $\lambda_{\sqsubseteq}$ should have some semantic counterpart. A significant motivation for carrying this out is that a general notion of model offers some hope of making connections between refinement and popular development methodologies.

Hermida [Her93] uses fibrations to model predicates over $\lambda^{\times\to}$ (but not for any more complicated type theories). He uses fibrations with indeterminates to model parameterisation. Models of many calculi can be presented as fibrations. For example, a dependently-typed calculus could be modelled by two fibrations — one to handle the dependency and the other the logic. Underdeterminism should be a separate feature on top of this set-up.

We should be able to characterise $\sqsubseteq_\phi$ semantically, using the specialisation order for example, as in [Fio94]. Whether derived or assumed, there is a poset-enriched structure where the ordering of morphisms corresponds to refinement.

More generally, we could envisage a 2-categorical structure, where the 2-cells correspond to *proof* of refinement. The `let` congruence rules give a 2-categorical structure. If we interpret $\Gamma \vdash r \sqsubseteq_\phi r'$ as a 2-cell, α, from $[\![\Gamma \vdash r : \phi]\!]$ to $[\![\Gamma \vdash r' : \phi]\!]$, and $\beta = [\![\Gamma \vdash r' \sqsubseteq_\phi r'']\!]$, $\gamma = [\![x : \phi \vdash s \sqsubseteq s']\!]$, $\delta = [\![x : \phi \vdash s \sqsubseteq s'']\!]$, then we can define $\gamma \star \alpha$ to be

$$[\![\Gamma \vdash \texttt{let}\ x : \phi\ \texttt{be}\ r\ \texttt{in}\ s \sqsubseteq \texttt{let}\ x : \phi\ \texttt{be}\ r'\ \texttt{in}\ s']\!]$$

Because of the congruence rules for `let`, this is a valid refinement. The interchange law, that $(\delta \star \beta) \circ (\gamma \star \alpha) = (\delta \circ \gamma) \star (\beta \circ \alpha)$, then follows since both terms correspond to the same refinement.

6.11 Aspects of the Software Life-cycle

Most theoretical work on formal methods has been on program verification and development. However, the software life-cycle consists of many other activities, and a comprehensive theory should include these. We make some suggestions for how our work on refinement could be extended to some other related areas.

6.11.1 Prototyping

The calculus formalises partially developed programs as combinations of specification and code, and induces a logic for them from a logic on the underlying programming language. Hence it is possible to reason about partially developed programs as 'first-class' artifacts even though, in general, it is not possible to evaluate arbitrary combinations of specification and program code. It is possible, however, to give an operational semantics to the language so that terms can be evaluated in certain situations as if they were programs.

This idea was the thinking behind the dynamic semantics of Extended ML (section 5.4.1), so it would be interesting to investigate this simple notion of prototyping further, and see what possibilities this offers for specification testing.

6.11.2 Maintenance

The problem of software maintenance is to modify legacy code which performs some function, so that it performs some related function. This can be formulated using the language of refinement. Given a program t which satisfies specification ϕ, and another specification ϕ', which bears some relationship to ϕ, the problem is to construct a program which satisfies ϕ'.

By an appropriate formalisation of how ϕ' relates to ϕ, it should be possible to automatically construct a partially developed program, r, which is t with the code that needs to be rewritten replaced by the appropriate specifications. More ambitiously, it should be possible to reuse the refinement itself, so that part of the refinement of ϕ to t can be used in constructing a refinement from ϕ' to r.

This idea of extracting part of a term based on a specification is similar to program slicing, and we could investigate connections with this view of maintenance.

6.11.3 Reverse Engineering

The general idea of reverse engineering is to recover a high-level description from an actual implementation. This is useful for both comprehension and maintenance. Some researchers have considered the problem of reverse engineering a program t to specification ϕ [War88]. This appears, at first sight anyway, to be dual to the problem of refinement from ϕ to t so perhaps some formal connections can be made.

Appendix A

Notation

Refinement Terms	r
Determined Terms	t
Total Terms	u
Individual Variables	x, y, z
Function Variables	f, g, h
Boolean Variables	b
Natural Number Variables	n
Variable Contexts	Γ
Propositional Contexts	Δ
Propositions	P, Q, R
Ground Types	γ
Types	σ, τ, υ
Refinement Types	ϕ, ψ, χ
Pseudotypes	κ
Basic Judgements	B
Judgements	J
Expressions	U, V
Constant Symbols	k
Predicate Symbols	F
Henkin Interpretations, Models	$\mathcal{A}, \mathcal{T}$
Sets	A
Individuals	a, b
Mappings	m
Syntactic Environments	g
Environments	η
Relations	R, Φ
Theories	T

Bibliography

[AC96] David Aspinall and Adriana Compagnoni. Subtyping dependent types. In *Proceedings of the eleventh IEEE Symposium on Logic in Computer Science*, 1996.

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

[Asp97] David Aspinall. *Type Systems for Modular Programs and Specifications.* PhD thesis, Department of Computer Science, University of Edinburgh, 1997.

[Bac88] R. J. R Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[BB95] M. A. Bednarczyk and T. Borzyszkowski. Towards program development, specification and verification with Isabelle. In *Isabelle Users Workshop*, University of Cambridge, 1995.

[BM92] R. Burstall and J. McKinna. Deliverables: A categorical approach to program development in type theory. In *Mathematical Foundations of Computer Science: 18th International Symposium*, volume 711 of *Lecture Notes in Computer Science*, pages 32–67, 1992. An earlier version appeared as LFCS Technical Report ECS-LFCS-92-242.

[Bos95] J. Bos. Predicate logic unplugged. In *Tenth Amsterdam Colloquium*, 1995.

[Bun97] Alex Bunkenburg. *Expression Refinement.* PhD thesis, Department of Computing Science, University of Glasgow, 1997.

[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.

[CDG96] Mario Coppo, Ferruccio Damiani, and Paola Giannini. Refinement types for program analysis. In *Static analysis: third International Symposium, SAS '96*, volume 1145 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 1996.

[Cro93] Roy L. Crole. *Categories for Types.* Cambridge Mathematical Textbooks. Cambridge University Press, 1993.

[Den97a] Ewen Denney. Refining Refinement Types. In *Informal Proceedings of Types Workshop on Subtyping, Inheritance and Modular Development of Proofs*, University of Durham, 1997.

[Den97b] Ewen Denney. Simply-typed Underdeterminism. In *EU KIT/ IOS International Workshop on Formal Models of Programming and their Applications*, Institute of Software, Beijing, 1997. To appear in special issue of *Journal of Computer Science and Technology.*

[Den98] Ewen Denney. Refinement Types for Specification. In David Gries and Willem-Paul de Roever, editors, *IFIP Working Conference on Programming Concepts and Methods (PROCOMET '98), Shelter Island, New York, USA*, pages 148–166. Chapman and Hall, 1998.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[Fef85] Solomon Feferman. A theory of variable types. In *Proceedings of the Fifth Latin American Symposium on Mathematical Logic*, volume 19 of *Revista Colombiana de Matemáticas*, 1985.

[Fio94] Marcelo Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* PhD thesis, Department of Computer Science, University of Edinburgh, 1994.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwarz, editor, *Proc. Symp. in Applied Mathematics*, pages 19–32, 1967.

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

[Har79] David Harel. *First-Order Dynamic Logic.* Lecture Notes in Computer Science. Springer-Verlag, 1979.

[Har80] David Harel. Proving the correctness of regular deterministic programs: A unifying survey using dynamic logic. *Theoretical Computer Science*, 12:61–81, 1980.

[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.

[Her93] Claudio Hermida. *Fibrations, Logical Predicates and Indeterminates*. PhD thesis, Department of Computer Science, University of Edinburgh, 1993.

[HJ95] Claudio Hermida and Bart Jacobs. Fibrations with indeterminates: Contextual and functional completeness for polymorphic lambda calculi. *Mathematical Structures in Computer Science*, 5(4), 1995.

[Hoa87] C. A. R. Hoare. Data refinement in a categorical setting. Unpublished manuscript, 1987.

[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.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 1990.

[JS91] Geraint Jones and Mary Sheeran. Relations and refinement in circuit design. In Carroll Morgan and Jim Woodcock, editors, *3rd Refinement Workshop 1990*, Springer Workshops in Computing, 1991.

[KOP$^+$97] Y. Kinoshita, P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. An axiomatic approach to binary logical relations with applications to data refinement. *Lecture Notes in Computer Science*, 1281, 1997.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[Lei69] A. C Leisenring. *Mathematical Logic and Hilbert's ϵ-Symbol.* University Mathematical Series. MacDonald Technical and Scientific, 1969.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.

[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.

[Mar96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[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.

[MM91] J. C. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51:99–124, 1991. Preliminary Version in *Proc. IEEE Symposium on Logic in Computer Science*, 1987, pages 303-314.

[MMMS87] A. R. Meyer, J. C. Mitchell, E. Moggi, and R. Statman. Empty types in polymorphic lambda calculus. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, pages 253–262, 1987. Reprinted with minor revisions in *Logical Foundations of Functional Programming*, ed. G. Huet, Addison-Wesley (1990), pages 273-284.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 1, 1991.

[Mor87] J. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

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

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[NH92] T. S. Norvell and E. C. R. Hehner. Logical specifications for functional programs. In *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, 1992.

[Nie96] Flemming Nielson. Annotated type and effect systems. *ACM Computing Surveys*, 28(2):344–345, June 1996.

[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.

[Nor88] Bengt Nordström. Terminating general recursion. *Bit*, 28:605–619, 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.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, volume 1059 of *Lecture Notes in Computer Science*, pages 119–134, 1996. Invited talk.

[Pit95] A. M. Pitts. Categorical logic. Technical Report 367, University of Cambridge Computer Laboratory, May 1995. 94 pages.

[PW90] David Pym and Lincoln Wallen. Investigations into proof-search in a system of first-order dependent function types. In *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 236–250. Springer-Verlag, 1990. Also University of Edinburgh LFCS Report ECS-LFCS-90-111.

[Pym90] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, Department of Computer Science, University of Edinburgh, 1990.

[Ros39] B. Rosser. On the Consistency of Quine's New Foundations for Mathematical Logic. *Journal of Symbolic Logic*, 4, 1939.

[San91] Donald Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, Workshops in Computing, pages 99–130. Springer, 1991.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2nd edition, 1992.

[SS83] William L. Scherlis and Dana S. Scott. First steps towards inferential programming. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Computer Congress*, 1983.

[SST92] Donald Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29(8):689–736, 1992.

[ST87] Don Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. In *Proc. Joint Conf. on Theory and Practice of Software Development*, volume 249 of *LNCS*, pages 96–110. Springer, 1987. Extended abstract.

[Tal90] Carolyn Talcott. A theory for program and data type specification. *Theoretical Computer Science*, 1990. Disco90 special issue.

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

[vL90] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier: MIT Press, 1990.

[War88] Martin Ward. Transforming a program into a specification. Technical Report TR-88, Centre for Software Maintenance, University of Durham, January 1988.

[War94] Nigel Ward. *A Refinement Calculus for Nondeterministic Expressions*. PhD thesis, University of Queensland, 1994.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

Index