MAC ISO Term 1 - “Intersection Types In Practice”

David P. Cunningham
Supervisor: Steffen van Bakel

February 14, 2005

Abstract

Intersection type systems are introduced as a solution to limitations of the Curry Type System for the lambda calculus. It is summarised how Forsythe uses an intersection type system effectively, for the purposes of typing combinations of expressive language features that are used to represent conventional language features. We study some intersection type systems with type variables, which are generalisations of the Curry Type System, explain how they are expressive, but how undecidability renders them unsuitable for practical use in their unrestricted form.

Keywords:

- Forsythe
- Intersection Types
- Programming Language
- Type Systems / Disciplines

Contents

1 Introduction to Intersection Types **2**
- 1.1 Some Typical Features and Motivations of Type Systems . . . 2
- 1.2 The Curry Type System for the Lambda Calculus . . . 3
- 1.3 Limitations of Curry Type System . . . 5
- 1.4 Intersection Types . . . 8

2 Forsythe - A Case Study **10**
- 2.1 Introduction . . . 10
- 2.2 Primitive Data Types . . . 11
- 2.3 Intersections . . . 11
- 2.4 Arithmetic . . . 12
- 2.5 Imperative Commands . . . 12
- 2.6 Arrows . . . 13
- 2.7 Examples of Variables and Assignment . . . 13
- 2.8 Decidable Type Inference Algorithm . . . 15
- 2.9 Objects . . . 15
- 2.10 Arrays . . . 18
- 2.11 Conclusion . . . 19

3 Intersection Type Systems with Type Variables **21**
- 3.1 Essential Intersection Type System . . . 23
- 3.2 Terms Typable With This System . . . 24
- 3.3 Principal Typings . . . 24
- 3.4 Operations On Typings . . . 25
  - 3.4.1 Substitution . . . 25
  - 3.4.2 Lifting . . . 26
  - 3.4.3 Expansion . . . 26
  - 3.4.4 Comments . . . 29
- 3.5 Unification . . . 29
- 3.6 Undecidability . . . 32
- 3.7 Conclusion . . . 33

Chapter 1

Introduction to Intersection Types

Traditional and well understood type systems like the Curry Type System, or the ML Type System, have limitations in the scope of correct programs that they can type. Since our interest is motivated by shortcomings in other systems, it is of interest to quickly review both the purpose of type systems in general, and the limitations of "conventional" type systems. Then it will be shown how extending the Curry Type System with intersection types provides a type system that does not exhibit these limitations.

1.1 Some Typical Features and Motivations of Type Systems

The motivation for type systems stems from the need for static reasoning about programs, particularly to determine correctness, but also to allow certain optimisations in the compilation process. Some syntactically valid programs can be thought of as "semantically invalid" - for example if their meaning is not specified by the language definition. A suitable type system can be used to detect semantically invalid programs automatically. Type systems are necessarily designed to support the syntax and uphold the defined semantics of a *specific* language. Despite this, we can make a few general and typical observations about them.

Types can be thought of in terms of the set of constant values that agree with them. Typically the types of literals and such are defined in keeping with the implementation's representation of these values. To type an *expression*, a syntax which may *evaluate* during the course of program execution, it is usual that the expression agrees with a type if all the possible evaluations of the expression agree with that type, not making any unwarranted assumptions about the values of any variable data that might be referenced by the expression.

This is useful because it is likely that a language allows an expression to be used in place of a literal, specifying that the expression is evaluated to produce a concrete value, at runtime, when such a value is required by the evaluation of its context. Naturally then, we will allow any expression to form the sub-syntax in this context, if we can guarantee that the expression will evaluate to an allowed value, so thus we require the expression to agree with a certain type. This places a constraint on what expressions are allowed as sub-syntax. This property of a type system, where an expression's hypothetical evaluation result will always agree with the expression's type, is called the "subject reduction property" and is highly desirable in a type

system for any language with the feature described.

Traditionally, the type system has not evaluated the expression (in the conventional sense) to determine the set of possible results, but has inferred some conservative estimate of agreement using the syntax of the expression, and a predefined set of syntax directed rules. The set of rules is compatible with the semantics of the language. This results in a general assertion about the character of the expression's behaviour, and its invariance to runtime state gives us the universal certainty we desire:

"For all possible inputs and executions, we know this about the run-time behaviour: ..."

Because the rules in a type system are simple and syntax directed, the type inference (the determining of a type that code agrees with) is efficient and decidable, and can thus be performed autonomously by a compiler. Inventing a type system is a creative process, and there might be many type systems that are consistent for a language. Some of them might not be able to type some important programs, and also some might not have the useful properties we require. In particular, it is a challenge to define a type system with enough scope and useful features without losing *decidability*, which is essential if the type inference mechanism is to be autonomous within a compiler.

There is little more that can be said about type systems in general, so we shall visit the Curry Type System for the lambda calculus, as it the foundation for the Intersection Type System.

1.2 The Curry Type System for the Lambda Calculus

Here we demonstrate the notation used in this document, and introduce the type system whose limitations will be overcome when we extend it with intersection types.

The system is simple, well-known and well-understood, and despite being defined for the lambda calculus, can be extended to form the type systems that are used in practical programming languages today. A lot of the information in this section is summarised from the course "Type Systems for Programming Languages", and its lecture notes, [10].

Note that the original motivations given for type systems (detecting erroneous programs and efficiency) do not apply here, since there are no "stuck" lambda terms (assuming infinite evaluation is valid), and thus a type system should type all lambda terms. Also, efficiency is irrelevant when dealing with a mathematical formalisation. Nevertheless, it is interesting to study type systems for the lambda calculus because of its simple definition, and the fact that many features of conventional programming languages have analogies within the lambda calculus. This gives us confidence that our results here will scale to more complex languages.

Definition 1.1. *Syntax of the lambda calculus:* $E ::= x \mid (E_1 E_2) \mid (\lambda x.E)$

The categories are named 'term variable' 'application' and 'abstraction' respectively. To improve readability of the concrete syntax, the following conventions are used: Application binds to the left and binds tighter than abstraction, consecutive abstractions are merged together, and redundant brackets are removed.

Definition 1.2. *Semantics of the lambda calculus:* $(\lambda x.M)N \longrightarrow_\beta M[N/x]$

This definition of reduction models computation and is Turing-complete. $M[N/x]$ stands for the syntax of M but for each term variable x in M, N is substituted in its place. The syntax $(\lambda x.M)N$ is called a redex, and where multiple redexes exist as sub-terms within an expression, any can be reduced to form the reduction of that expression. A detailed account of reduction in the lambda calculus is beyond the scope of this document, but information can be found in [10].

Definition 1.3. *The set of types in the Curry system is defined to be:*

$$\tau ::= \varphi \mid \tau \to \sigma$$

Where σ and τ range over types, and φ ranges over type *variables*. Note that an arbitrary type is therefore a tree structure, the nodes being arrows and the leaves being variables. In extensions of the system, the set of types is more complex, but for the simple lambda calculus, just variables and arrows are used. Values (closed normalised terms with predefined meaning), for example the church numerals, are represented as abstractions in the lambda calculus, and thus their types are expressed as arrows.

Definition 1.4. *Curry type assignment natural deduction rules:*

$$(Ax)\ \frac{}{B \vdash_C x : \sigma}\ (x : \sigma \in B)$$

$$(\to I)\ \frac{B, x : \sigma \vdash_C M : \tau}{B \vdash_C \lambda x.M : \sigma \to \tau}$$

$$(\to E)\ \frac{B \vdash_C M_1 : \sigma \to \tau \qquad B \vdash_C M_2 : \sigma}{B \vdash_C M_1 M_2 : \tau}$$

In general, typing rules in the definition of a language's type system are short implications that define certain syntactic categories to agree with certain types as long as certain conditions, typically relating to the types of the sub-terms of those syntactic categories, are met. These implications can be chained together to form a derivation for a complex term, by applying the appropriate rule for the term's top-level structure and systematically breaking the term down into its sub-terms and typing those as well.

The judgement $B \vdash_C M : \sigma$ is an infix notation that means "M can be typed with the *typing* $\langle B, \sigma \rangle$", where B is a 'basis' - a set of predicates, one predicate for each unbound term variable stating that it agrees with a type, and where σ is a type with which M agrees under these conditions. The basis can be thought of as requirements on the types with which the free term variables of a term must agree. Another way of looking at the basis, is as a set of preconditions that the type system can use to prove a type σ for M.

Where a judgement appears above a line in a rule, that is one of the premises of the implication, and where a judgement appears below a line, that is a conclusion that can be drawn.

Conditions to the right hand side of the rule are additional conditions that must be true before the rule can be applied.

It is possible to convert a valid typing for a given lambda term, to a new, different, but still valid typing, by replacing a given type variable with a given type, consistently throughout the types in the typing. This is called *substitution*. Substitution is called an *operation* on typings, it can be thought of as making a consistent change to the derivation tree for the first typing, in order to produce a different derivation that is also valid, and thus a new, valid, typing for the original lambda term.

For every term, there exists a typing from which all other valid typings can be derived with some substitution operations, this typing is called the *principal typing*. The typings that are the result of operations on the principal typing are called instances of the principal typing.

In addition to this, it is possible to derive the principal typing itself with the rules of the Curry Type System, therefore the system has the "principal typing property". Some general remarks on principal typings and this property can be found in [11].

Since we are concerned with the practical issues of type systems, it is interesting to note here the practical benefits of this property. [2] describes how they are useful for efficient separate compilation and recompilation of a module (if the principal typing is known, the type for a term need never be re-inferred), and accurate error messages (since we can detect errors in the course of unifying two principal typings rather than through attempting to type-check one sub-term against another erroneous sub-term that has an erroneous type).

There exists a principal typing algorithm for the Curry Type System, which can generate the principal typing for a given term. Its definition is not covered here but of interest is its auxiliary algorithm *unify*. This algorithm takes two types and determines the sequence of operations required to convert both of them into the same (new) type, an instance of both of the initial types. This is essential for solving the constraints placed upon sub-typings on both sides of an application, when building the principal typing for that application. Unification will be the subject of much more detailed discussion in section 3.5. Some Curry types cannot be instantiated into the same thing through substitution operations, and unification returns an exception in these cases. When this happens, the principal typing algorithm declares that there is no typing for the term.

1.3 Limitations of Curry Type System

A large class of lambda terms are not typeable. Consider self-application xx, here the type of x has to be both a function, and also the argument of that function, which is not possible within our inductively-defined type syntax. The principal typing algorithm will throw an exception with this term as described above.

The fix combinator cannot, therefore, be typed, and this is a very useful term for programming, since it allows us to write programs that do not have an upper bound on the time they take to execute, e.g. the algorithm to calculate a factorial. Also, some strongly normalisable terms, and even terms that are normal forms cannot be typed. The lack of fix would make the system impractical for programming languages, but as it turns out it is quite easy to sidestep

this issue, for the majority of practical requirements, by extending the syntax and semantics of the language to include an operator that behaves as the fix combinator does, but has a special typing rule, as is done with the language ML. Note that ML's solution while being decidable, cannot type all programs, specifically not those using polymorphic recursion. This technique is sometimes desirable, as shown in [1]. There is a trade off in this area between decidability and the scope of the type system.

A useful property that the Curry Type System *does* have, is the subject reduction property, as discussed earlier, this can be seen by showing that $B \vdash (\lambda x.M)N : \tau \Longrightarrow B \vdash M[N/x] : \tau$, but the converse (subject expansion) does not hold. This is because terms can 'forget' their sub-terms as they reduce, e.g. M in the term $(\lambda x.y)M$ is forgotten, but these sub-terms are not forgotten by the type inference algorithm. The beta-forgotten terms might force certain basis types to be instances of what they really need to be, which will affect the resulting typing. Also the beta-forgotten sub-terms might not be typeable at all in the system, so beta-expansions of typeable terms may not be typeable.

A useful technique in software engineering is the use of modules to factor out duplicated code, in a program, to ease the task of maintanance. To correctly type this circumstance, we need to ensure that the typing required by the context of each reference to the factored-out code is a valid typing for the factored-out code itself. This can be done by by successfully typing these references with *instances* of the principal typing of the factored-out code. This, however presents a problem with the Curry Type System in the lambda calculus. Factoring out code in the lambda calculus can be represented by coding a redex into the program.

Example 1.5. *A program in the lambda calculus, typed with the Curry Type System:*

$$\{\} \vdash (\lambda x.x)(\lambda x.x) : \varphi \to \varphi$$

We can factor the $(\lambda x.x)$ out to produce a redex, this can be seen as referencing the definition of the identity abstraction $(\lambda x.x)$ multiple times with the term variable I, but the resulting program cannot be typed, we have self-application:

Example 1.6. *The same program, defined in a modular way, untypeable with the Curry Type System:*

$$\{\} \vdash (\lambda I.\ II)\ \ (\lambda x.x) :?$$

The problem is that we need to express that each of the two "I" term variables has a different type, according to its context, and that each I type is a valid type for the factored out code with respect to the same basis. This cannot be expressed without extending the type system, since the two "I" variables are forced to have the same type in the basis of II. There is no "subsumption" when typing term variables that reference polymorphic code.

When factored-out code can be referenced in a variety of typing environments, it is called polymorphic. Type polymorphism allows us to type such code in a manner that allows the typing of references to the factored-out code despite them being in different type environments.

Subsumption is the act of typing a term with one type, on the premise that it already has the type of another different type, this can be formalised with a rule like $B \vdash M : \tau \Longrightarrow B \vdash M : \sigma$.

This means that even though we are given that M has type τ, we can use it in a context that requires type σ, with the same basis. This act of subsumption is really the key to bridging the typing "abstraction gap" that arises when code is factored out of a program. If we only need polymorphism for our term variables, M can be x, in the above definition, and thus squashed into the (Ax) rule.

Note how the basis remains the same during subsumption, this has to be true since the typing of all the instances is bound by the basis of their contexts. Of course, the basis contains the link to the more general type of the polymorphic code being referenced, and cannot be disturbed. If the subsumption is to be the operation of substitution, as seems natural in the Curry System, i.e. if $\sigma = S(\tau)$, then clearly the variable being substituted must not exist in the basis, i.e. $B = S(B)$. This will never be true with the Curry System since ultimately the type for the polymorphic term variable must be extracted from this very same basis, so the type variable in the substitution will always exist in there. Thus the Curry Type System does not admit a simple subsumption rule that uses substitution.

In [10], a type system is presented for combinator systems where the principal typings for each combinator are stored in an "environment". When the combinators come to be used, the equivalent (Ax) rule for the combinator terms allows the type in the environment to be instantiated to whatever is required by the context of the combinator term. This is a form of polymorphism but does not extend to term variables in general, only combinator terms. It does, however, avoid the problem of damaging the basis during instantiation by storing the principal typings (whose type variables are disjoint to those present in the basis) for the combinators outside of the basis.

In ML, there is a `let` syntax construct which is semantically equivalent to the redex mechanism of factoring out code that is discussed above. It is typed in a different manner to an application of an abstraction however, since it first asserts a typing for the factored-out code, say $\langle B, \tau \rangle$. It then "marks" each "unconstrained" type variable in τ with a quantified symbol before embedding this as a type for the polymorphic term variable in the basis for the typing of the rest of the code. (An unconstrained type variable in τ is one that does not occur anywhere in B).

When different term variables need to be typed with instantiations of this "marked" principal typing, a special rule is used to convert the "marked" type variables into whatever is required, using an extended notion of substitution designed to ignore the marked variables in the basis. This prevents the operation affecting the basis so that it can used soundly for subsumption. The ML type system is decidable, but does not type some terms, particularly uses of polymorphic recursion, and of course using a new syntax to implement the factoring out of code does not solve the problem that $\lambda x.xx$ is not typeable. Using generalisations of the ML type system that type more terms has not lead to much success, since these type systems are often complex and undecidable, or without principal typings.

The main limitations of the Curry Type System as it stands as a type system for practical programming languages, can be summarised as:

- Some terms cannot be typed (e.g. self-application)
- Fix (an essential term in practical programming) cannot be typed

- No support for polymorphism

These criticisms are linked. The second is due to the first, so by solving the first we also solve the second. If we had some support for polymorphism for general term variables, we could actually solve the first.

Both of the solutions to polymorphism referenced above, sidestep the issue of subsumption by substitution in the Curry Type System, by first calculating the principal typing and storing it separately from the other types in the basis. We shall now see how the Intersection Type System employs a similar, but somewhat opposite approach to supporting polymorphism: It collects together the requirements of the term variables in the basis, and then ensures that the polymorphic code can be typed to each of the collected types.

1.4 Intersection Types

There are many intersection type systems in existence, they are each slightly different but they share so much in common that it makes sense to discuss them in a general way in this introduction, before examining a few in detail later in this document.

Recall that we are motivated by the idea that two identical term variables in a term must both agree with different, but non-unifiable types. However, we are constrained by the fact that in the basis, the term variable can only be given a single type. A type system can overcome this by having a type constructor that groups a set of types into a single type. Thus each term variable can be typed with a different type, and the basis can presume both. The subsumption operation is then simply to reduce this set of types to the subset that is required by each context in the term. Another way of looking at this is that the basis contains a *list* of types, any of which can be required of a term.

The type syntax for representing several types as a single type is usually a binary infix conjunction, or intersection operator: &, $\wedge$, or $\cap$. This is intuitive since we need the operator to be commutative and associative, such that the order of types, or positioning of parentheses, in $\tau_1 \cap \tau_2 \cap \tau_3$ is not important. Note that it is also usually possible for a term to be assigned the empty set of types, this is sometimes denoted ω, and called the "nullary intersection". Previously this was treated as a type constant, possibly because its appearance is removed from the usual notation for intersections. Perhaps it is more intuitive to represent intersection types as a flat set, e.g. $[\![\tau_1, \tau_2, \tau_3]\!]$ thus the nullary intersection can be stated as $[\![]\!]$, and ordinary curry types as $[\![\tau]\!]$ (abbreviated to just τ), but in this document I will use the conventional binary notation and ω.

Assigning a term the nullary intersection is not particularly useful, since it allows no deduction about what types the term's value or evaluation will definitely agree with, and thus the term cannot be used in any context that requires a specific type. Intuitively, all the type ω provides us with, is the information that the term could potentially evaluate to anything, which we already knew. It is useful though, for typing terms that we do not care about, e.g. terms that are forgotten during reduction. This is the key to the property of beta expansion, held by intersection type systems.

The intuition behind the naming of intersection types is that if a value agrees with τ and σ, then it agrees with τ, and it agrees with σ, so it agrees with a hypothetical type that is represented

by an intersection of the sets of agreeing values for both types. Understanding the members of such hypothetical agreement sets for data types is trivial, but trying to understand how the value of a function might be represented is not. I find it best to think about intersection types directly in terms of the rules that define them, without using any intuitive notion of sets and set intersection.

To formalise the claims about intersection types, here is stated the type rules used to introduce and eliminate the intersection type constructor in the derivations of many intersection type systems.

Definition 1.7. *Typical rules involving intersections:*

$$(\cap I)\ \frac{B \vdash M : \tau \qquad B \vdash M : \sigma}{B \vdash M : \tau \cap \sigma}$$

$$(\cap E)\ \frac{B \vdash M : \tau \cap \sigma}{B \vdash M : \tau}$$

These are intuitive from the English definition given above - a term can be typed to an intersection of two types if it can be typed to both types, and subsumption is the removal of a type from the intersection.

Example 1.8. *To end this chapter, it is shown how the example in the previous section can be typed with intersection types: (τ stands in place of "$(\varphi \to \varphi) \cap ((\varphi \to \varphi) \to (\varphi \to \varphi))$".)*

$$\dfrac{\dfrac{\dfrac{\dfrac{(Ax)\ \dfrac{}{I : \tau \vdash I : \tau}}{(\cap E)\ \dfrac{I : \tau \vdash I : (\varphi \to \varphi) \to (\varphi \to \varphi)}{}} \quad \dfrac{(Ax)\ \dfrac{}{I : \tau \vdash I : \tau}}{(\cap E)\ \dfrac{I : \tau \vdash I : \varphi \to \varphi}{}}}{(\to E)\ \dfrac{I : \tau \vdash II : \varphi \to \varphi}{}}}{(\to I)\ \dfrac{}{\vdash \lambda I.II : \tau \to (\varphi \to \varphi)}} \quad \dfrac{\dfrac{(Ax)\ \dfrac{}{x : \varphi \vdash x : \varphi}}{(\to I)\ \dfrac{}{\vdash \lambda x.x : \varphi \to \varphi}} \quad \dfrac{(Ax)\ \dfrac{}{x : \varphi \to \varphi \vdash x : \varphi \to \varphi}}{(\to I)\ \dfrac{}{\vdash \lambda x.x : (\varphi \to \varphi) \to (\varphi \to \varphi)}}}{(\cap I)\ \dfrac{}{\vdash \lambda x.x : \tau}}}{(\to E)\ \dfrac{}{\vdash (\lambda I.II)(\lambda x.x) : \varphi \to \varphi}}$$

Note how the deduction collects together both typings for I, and then deduces them both independently for the typing of $\lambda x.x$. This is in contrast to the ML system which would first type $\lambda x.x$ and then generate a type that can subsume any valid typing for I. An interesting character of intersection types, that complements universal types (from ML) is that we can type the use-cases of the polymorphic code before the polymorphic code itself, which may prove useful in modular programming languages.

Now we leave the lambda calculus for a while to study an Algol-like language that uses intersection types extensively.

Chapter 2

Forsythe - A Case Study

Forsythe is described in detail in [4] and [5], we focus here upon its type system, but the referred papers contain information about the language as a whole.

2.1 Introduction

The design goal of Forsythe is to be as "artistically" expressive as a practical programming language, indeed more expressive than most programming languages to date, while retaining a simple type system and well formed operational semantics (like the lambda calculus). By "artistically" expressive, it is meant that Forsythe should allow typical programming code to be elegantly expressed, in a concise, readable, and maintainable fashion.

The language's design contains a few new and abstract concepts, that suffice to represent many conventional ideas, thus these ideas need not be explicitly built in to the language as rules in the type system and operational semantics. For instance; assignment, arrays and objects can all be represented in terms of more primitive forms, as shall be seen.

Typically syntax sugaring is used to translate the conventional form of code into the new representations, in order to keep the programs readable. Part of the fuel for the generalisation is Forsythe's type system, which makes creative use of intersection types to safely describe how data can be constructed from many different things combined together.

The type system of Forsythe is not an extension of the Curry Type System, since although it has $\rightarrow$, it does not have *type variables*. Thus there are no principal typings in Forsythe, and no notion of unification. Instead of type variables, there are 6 primitive type constants, and a primitive subsumption relation defined over them. Note that without type variables and substitution, we have no way of writing truly re-usable 'template' code, since the type of such code would have to be the intersection of all possible types, including types yet to be defined by the programmer.

In this document, τ, σ and ρ are metavariables that range over the various possible Forsythe types.

2.2 Primitive Data Types

The primitive data types (for terms that are, or evaluate to, a value) are **real**, **int**, **bool** and **char**. Note that the type for *variables*, in the sense of objects that can store data, is not any of these types, and will be defined later. These types are simply the types of variable evaluations, arithmetic expressions, and literals. In short, these are the objects that represent data itself, rather than data storage. Forsythe defines the reflexive, transitive subsumption relation ($\leq$) which relates two types if agreement with the type of the left-hand-side implies agreement with the type on the right-hand-side. Initially the subsumption relation is specified here over the data types, but as more features of the Forsythe type system are discussed, its definition will be extended.

Definition 2.1. *Subsumption for primitive data types:*

$$\begin{aligned} \textbf{\textit{int}} &\leq \textbf{\textit{real}} \\ \textbf{\textit{real}} &\leq \textbf{\textit{value}} \\ \textbf{\textit{bool}} &\leq \textbf{\textit{value}} \\ \textbf{\textit{char}} &\leq \textbf{\textit{value}} \end{aligned}$$

There is a subsumption *rule* that allows types to be relaxed in a derivation, according to the subsumption relation.

Definition 2.2. *Subsumption derivation rule:*

$$(\leq)\ \frac{B \vdash M : \tau}{B \vdash M : \sigma}\ (\tau \leq \sigma)$$

2.3 Intersections

Since the type system of Forsythe is an intersection type system, there is an introduction and elimination rule in the usual way:

Definition 2.3. *Rules for* ($\cap$)*:*

$$(\cap I)\ \frac{B \vdash M : \tau \qquad B \vdash M : \sigma}{B \vdash M : \tau \cap \sigma}$$

$$(\cap E)\ \frac{B \vdash M : \tau \cap \sigma}{B \vdash M : \tau}$$

$$(ns)\ \frac{}{B \vdash M : \textbf{\textit{ns}}}$$

Note that **ns** denotes the nullary intersection, or "nonsense". The subsumption relation is extended over types involving the ($\cap$) type constructor.

Definition 2.4. *Subsumption for intersections,* $\forall \sigma, \sigma', \tau, \tau' \in Types$*:*

$$\begin{array}{rcl} \sigma & \leq & \textbf{\textit{ns}} \\ \sigma & \leq & \sigma \cap \sigma \\ \sigma \cap \sigma' & \leq & \sigma' \\ \sigma \cap \sigma' & \leq & \sigma \end{array}$$

$$(\sigma' \leq \sigma) \quad \wedge \quad (\tau' \leq \tau) \quad \Longrightarrow \quad (\sigma' \cap \tau') \leq (\sigma \cap \tau)$$

Note that now the subsumption relation has lost its property of antisymmetry since, for example **int** $\leq$ (**int** $\cap$ **int**) $\leq$ **int**, so the relation is now just a pre-order. If, however we consider types modulo equivalence, where types are equivalent if they subsume each other, we squash the cycles of the relation into single nodes, and we get a partial order again. Type equivalence, as defined here, is given the operator ($\sim$). This is a byproduct of redundancy in the syntax used to express intersection types, rather than a property of the ideas the type system represents.

2.4 Arithmetic

Forsythe has syntax, semantics, and type rules for the usual binary integer / real arithmetic and for boolean logic. Overloaded syntax (such as (+) for **int** and **real**) is given a type rule for each of its meanings, thus it can be typed to the intersection of all its typings using ($\cap I$). There is also syntax and typing rules for literals such as `true`, `false`, `3`, `'a'`, etc. The precise rules are omitted here since they are not very interesting.

2.5 Imperative Commands

In Forsythe, the type **comm** is a type for syntax that is, or evaluates to a command. A command is something which causes the state to change. **comm** is therefore not a data type like **int**, because it expresses the character of code that changes state, rather than code that returns a value. Despite looking a lot like the lambda calculus at first glance, Forsythe is an imperative language with a program state that can be altered by commands. Variables can be written to, and then later on can be read to determine what was last written to them. **compl** is a special subset of commands that never return control, this could be used for exceptions, early returns, and exiting a program.

Commands can be sequentially composed with the ";" syntax, and looped with a **while** construct, as in conventional imperative languages. Both of these control flow constructs are typed with **comm** or **compl** as appropriate to their sub-terms. As will be shown, it is the prerogative of abstraction to define new variables on the implicit stack, and the operation of assignment is actually represented in terms of an application, so there is no need to type either of these imperative features to **comm**.

Definition 2.5. *Subsumption for commands and completions:*

$$\textbf{\textit{compl}} \quad \leq \quad \textbf{\textit{comm}}$$

2.6 Arrows

Forsythe type syntax has arrows, as mentioned earlier. There are syntax, semantics, and type rules for abstraction and application, and they are essentially the same as those in the lambda calculus. Abstraction is, in fact, the only way of introducing new (bound) term variables into the program code. The subsumption relation is inductively extended to cater for arrows in the following way.

This inductive definition is not present in the Curry Type System, since we have no base case for the first definition, and no intersection for the second.

Definition 2.6. *Subsumption for arrows,* $\forall \sigma, \sigma', \tau, \tau' \in Types$:

$$\begin{array}{rcl} (\sigma' \leq \sigma) \ \wedge \ (\tau' \leq \tau) \quad \Longrightarrow \quad (\sigma \to \tau') & \leq & (\sigma' \to \tau) \\ (\sigma \to \tau) \cap (\sigma \to \tau') & \sim & \sigma \to (\tau \cap \tau') \end{array}$$

2.7 Examples of Variables and Assignment

A program in Forsythe is a block of syntax, which is typed to **comm**. Below is an example, which writes the exclamation mark to standard out, but notice that it is an application, and that the predefined constant term std_out is actually of type **char** $\to$ **comm**.

Example 2.7. *Forsythe "hello world" program:*

```
std_out '!'
```

This is the first example of how a conventional language construct has been represented in terms of a simpler, more abstract one. This code could in fact have been written using some syntax sugar:

Example 2.8. *The same program using assignment:*

```
std_out := '!'
```

That is, the assignment of the character value to the term variable std_out is in fact the application of the term variable and the value. Terms that can have characters assigned to them, are actually functions from **char** to **comm**, and one applies them to a value when one wants to assign that value to the variable. The use of the word "variable" here is slightly imprecise, since we are only interested in terms that can be assigned to, whereas variables can be evaluated as well. Variables in fact have types like (**char** $\to$ **comm**) $\cap$ **char**, i.e. they can be used in contexts requiring either assignment or evaluation.

This is the second example of how a conventional idea has been represented in terms of more abstract ideas, and also an example of how intersection types have been used to join the abstract ideas together. To make the notation more symmetric and concise, the abbreviation **characc** is used instead of **char** $\to$ **comm**, and the same for the other primitive data types. Also, the type of a variable is abbreviated to **charvar**, rather than **characc** $\cap$ **char**.

Here is another Forsythe example to reinforce the above information, and also introduce some new features. For reference, the conventional imperative code listed below it has the same behaviour, but we are not interested in the way Forsythe expresses this behaviour, rather the manner in which this is typed.

Example 2.9. *Variable and procedure definition. Recall that* ***intvar*** $\rightarrow$ ***comm*** *means the same thing as* ***int*** $\cap$ (***int*** $\rightarrow$ ***comm***) $\rightarrow$ ***comm****:*

```
let  f:intvar->comm = \x.xx  in
        newintvar 0 \i.
        (
                f i
        )
```

Example 2.10. *The same program in an imperative pseudo code:*

```
declare proc f (byref int i) {
        i := i;
}

declare int i := 0;

f(i);
```

Firstly, the `let` syntax is, as in ML, to factor some code out from the rest of the program, so f is defined to be an abstraction that self-applies (self-assigns) its argument. The `let` is actually just syntax sugar that maps to an abstraction/application pair as used in Chapter 1, however the syntax of `let` allows multiple definitions to be made at the same time, which will make for a more concise and readable definition than abstraction / application would.

Since the type of f is specified in the syntax, we know that the argument of the abstraction is an integer variable, and we know that the body is supposed to be of type **comm** with basis $\{x : \textbf{intvar}\}$. The subsumption due to intersection elimination allows us to type-check the self application in the body accordingly, since **intvar** is in fact (**int** $\rightarrow$ **comm**) $\cap$ **int**, and thus the first x takes (**int** $\rightarrow$ **comm**) type, and the second x takes **int**. Intuitively, we know that x must be read from and written to, so it makes sense that it must be a variable. Clearly since f self-applies its variable argument, its behaviour is to assign the variable passed to it, to itself, which results in no change to the state.

Moving now to the code that uses f, the predefined constant term **newintvar** has type **int** $\rightarrow$ (**intvar** $\rightarrow$ **comm**) $\rightarrow$ **comm**, it is the method of allocating storage for new variables (on the implicit stack). Note that the body of the abstraction required in the second argument is the code which uses the new variable. When applied to its 2 arguments, **newintvar** can be thought of as applying its *second* argument to a hypothetical new variable, the value of which it initialises to the value gained by evaluating its *first* argument.

Thus the body of the second argument has access to the new variable in the sense that it just uses the abstraction's bound term variable to reference the new variable. When control

returns from the second argument, the application of newintvar returns control. Note how since the body of the abstraction must be of type **comm**, there is no way it can 'return' the newly allocated variable, so it cannot be accessed out of scope.

The code therefore creates a new integer with a default value of 0, and then sets this value to itself, so no change occurs. We have demonstrated how new variables are created, and how self-application makes an appearance as self-assignment in Forsythe, and can be typed accordingly.

2.8 Decidable Type Inference Algorithm

The choice of type checking versus type inference in Forsythe deserves some comment: It is known that the process of complete type inference is impractical due to undecidability, so Forsythe requires some type annotations, but in the interests of not insisting upon a type annotation unless it is absolutely necessary, the rules that define where type annotations are required are quite complex. Vaguely; certain phrases, especially abstractions, are not allowed in certain contexts unless they have explicit type information, see [5] for more details. Also, recursive definitions, either specified using the fixed point syntax, or with a recursive let definition, must be explicitly typed.

The programmer may wish to annotate types unnecessarily, to limit the code to an instance of its principal typing, or perhaps so that erroneous code can be quickly spotted. But it must be up to the programmer to decide whether annotations are appropriate. It is not clear how to balance the clutter inherent to type annotations against the benefits in the general case.

There is an interesting issue regarding type annotations of abstractions: How do we annotate x in $\lambda(x :???).(x+x)$ if we want the abstraction itself to be (**int** $\rightarrow$ **int**)$\cap$(**real** $\rightarrow$ **real**)? Forsythe has a special notation which will provide this functionality in a limited set of cases, the syntax appears as follows.

Example 2.11. *Interesting type annotation:*

```
\x:int|real . (x + x)
```

The abstraction above can be typed to be either **int** $\rightarrow$ **int** or **real** $\rightarrow$ **real**. This is not sufficient for all cases however, as is noted in [12]. That paper presents a type system in which type annotations can be made on all terms.

2.9 Objects

Now we focus on another conventional programming language feature, the struct, tuple, or object. Normally a language has a set of special syntax and type rules for composing and decomposing objects in terms of their constituents, and the types of their constituents. Forsythe on the other hand makes creative use of intersection types so that the extension to the language required to handle objects is in fact very tiny.

Forsythe has a syntax that allows sub-syntax to be given a field name. Where we would normally have an expression, e.g. 3, we can give it a name.

Example 2.12. *Syntax for naming expressions, or "object construction"*

```
name == 3
```

Note this double equality symbol is different to the single symbol used for boolean equality. This syntactic construct is to be viewed as a primitive for object construction, although it can only construct objects containing a single field. Note that since the field can be an abstraction or a variable, we can represent the conventional notion of objects with both methods and fields with this single operation. If the type of the original expression was σ, the type of the new expression will be *name* :: σ. Thus we are introducing a new type constructor, which can assign names to existing types.

On the other hand, if we have a named expression, we can use the "field selection" syntax to extract the data from that expression that has the correct name, for example the following expression reduces to 3:

Example 2.13. *Syntax for "field selection"*

```
(name==3).name
```

The type rule for that syntax ensures that the names used for the two expressions are consistent.

This by itself does not allow us to do much, but combined with another operation it is very powerful. Here are the formal type definitions for the naming operations.

Definition 2.14. *Rules for object construction and field selection, ι ranges over valid names:*

$$\frac{B \vdash M : \tau}{B \vdash \iota{==}M : \iota :: \tau}$$

$$\frac{B \vdash M : \iota :: \tau}{B \vdash M.\iota : \tau}$$

Definition 2.15. *Intuitive extension of subsumption for objects: $\forall \sigma, \sigma' \in Types, \iota \in Labels$:*

$$\begin{array}{rcl} (\iota :: \sigma) \cap (\iota :: \sigma') & \sim & \iota :: (\sigma \cap \sigma') \\ (\sigma' \leq \sigma) \implies \iota :: \sigma' & \leq & \iota :: \sigma \end{array}$$

Now we can construct singleton objects, it is possible to construct larger objects with an operation called "merge". Its syntax is just a binary operator (,). One can merge together any number of abstractions or named terms. The result of merging two expressions is a value representing both of them simultaneously. Operations on the merged data must therefore unambiguously and implicitly select a valid value from those inside. To prevent ambiguity, reduction of the merge operation is currently only defined when it occurs on the left of an application, or within a field selection operation as just discussed above.

If a merge has two identically named fields, or two abstractions, the right-most is what emerges during a field selection or application respectively. Also, if merge is applied to program fragments (p_1, p_2), p_2 must be an abstraction or a field type. Thus you can merge anything with a named expression or with an abstraction. The "anything" can be another merge of course, so this operation can be thought of as appending a field or an abstraction to the end of an object, to specify a larger object. Here are the formal reduction and type inference rules to make things more clear. Note that "if all fails", p_1 is selected regardless of what it is:

Definition 2.16. *Reduction rules for merging:*

$$\begin{array}{rcll}
(p_1, p_2)p_3 & \rightsquigarrow & p_2 p_3 & \textit{(if } p_2 \textit{ is an abstraction)} \\
(p_1, p_2).\iota & \rightsquigarrow & p_1.\iota & \textit{(if } p_2 \textit{ is an abstraction)} \\
(p_1, p_2)p_3 & \rightsquigarrow & p_1 p_3 & \textit{(if } p_2 \textit{ is a field)} \\
(p_1, p_2).\iota & \rightsquigarrow & p_1.\iota & \textit{(if } p_2 \textit{ is a non-}\iota \textit{ field)} \\
(p_1, p_2).\iota & \rightsquigarrow & p_2.\iota & \textit{(if } p_2 \textit{ is an } \iota \textit{ field)}
\end{array}$$

Definition 2.17. *Type inference rules for merging:*

$$\frac{B \vdash p_2 : \tau}{B \vdash (p_1, p_2) : \tau} \textit{(if } \tau \textit{ is an arrow or a field type)}$$

$$\frac{B \vdash p_1 : \rho}{B \vdash (p_1, p_2) : \rho} \textit{(if } \rho \textit{ is not an arrow nor a field type)}$$

$$\frac{B \vdash p_1 : \tau \to \sigma}{B \vdash (p_1, \iota{==}p_2) : \tau \to \sigma}$$

$$\frac{B \vdash p_1 : \iota :: \tau}{B \vdash (p_1, \iota'{==}p_2) : \iota :: \tau}$$

These are powerful and abstract constructs, so it helps to demonstrate how they can be used to create object-like data abstractions in a program. The following program defines and uses a co-ordinate data-type containing an *x* and *y* integer component (generally there is no need for these to be the same type). First a type is defined to represent the co-ordinate, then a function that can be used to create new coordinates, and finally code that creates a coordinate and prints out both components.

Printing an integer in decimal representation is achieved by first converting the integer to an *array* of characters, and then applying **std_out** to that array. I assume that **std_out** is capable of printing an array of characters in the same way that it can print a single character. Converting an **int** to an array of characters is done by applying the predefined constant **int_to_charseq** to the integer, and to an abstraction that will be applied to the new character array to yield a command. In this case **std_out** suffices in this role.

Example 2.18. *Using a "co-ordinate" object:*

```
%define a new type
lettype coordvar == (x::intvar & y::intvar)

%define a "constructor" function
let newcoordvar : int -> int -> (coordvar -> comm) -> comm ==
        \x_init \y_init \using_code .
                newintvar x_init (\xcomp.
                        newintvar y_init (\ycomp.
                                using_code (x==xcomp,y==ycomp)
```

```
                )
        )
in
        %instantiate a coordinate with default values, and then print
        newcoordvar 10 20 (\c.
                int_to_charseq c.x std_out;
                int_to_charseq c.y std_out
        )
```

It is of particular interest how the Object Oriented Programming notion of type polymorphism and the subsumption of parent object types by their child types due to object inheritance manifests itself in Forsythe. Since an object is a collection of fields, and it can be thought of as inheriting from a parent object with a subset of those fields (extending the parent with the additional fields), it can be shown that $\tau_{child} \leq \tau_{parent}$ with the existing definitions of $\leq$ and object type.

For instance, consider the object above, $\tau_{parent} = (x :: intvar \cap y :: intvar)$, as the parent of object $\tau_{child} = (x :: intvar \cap y :: intvar \cap z :: intvar)$. It can be shown that $\tau_{child} \leq \tau_{parent}$ since:

- $\sigma \cap \sigma' \leq \sigma$ where $\sigma = y :: intvar$ and $\sigma' = z :: intvar$.
- $(\tau' \leq \tau) \implies (\rho \cap \tau') \leq (\rho \cap \tau)$ where τ' and τ are the types related above, and ρ is $x :: intvar$.

Thus the traditional subtype relation from Object Oriented languages such as Java manifests itself in Forsythe due to the natural properties of the subsumption relation relating to intersections of field types.

2.10 Arrays

Arrays are a data type that represents an arbitrary-length sequence of identically typed data. They are often used to represent strings of text as arrays of characters. Most programming languages have features for creating arrays, and allow accessing of the elements through the use of an integer expression as an index.

Forsythe takes the simpler approach of representing arrays in terms of a merge of the function that represents the contents of the array, and a field that holds the length. The function is actually a "lookup" function of type **int** $\rightarrow \varphi$ where φ represents the type of the array elements. If this function is called with an index that is "out of bounds", an error completion is called, so array bounds checking is done at run-time.

Clearly, arrays are therefore implemented with the same abstract mechanism that implements objects, and have type (*length* :: **int**) $\cap$ (**int** $\rightarrow \varphi$). To make this representation more manageable, there is a syntax sugaring for creating a new array with a sequence of values:

Example 2.19. *Example of array construction syntax:*

```
int_to_charseq seq(0,1,4,9,16,25,36).length std_out
```

Such a **seq** expression has array type (*length* :: **int**) ∩ (**int** → **intvar**), and can be applied to an integer between 0 and 6 to obtain the variable at that index of the array, which you could then modify, since it is a variable. Also you can extract the length of the array for whatever purposes (printing, in this example) through use of the field selection operation. Of course, such notation is designed for arrays with no regular pattern to their contents. If on the other hand such a regular pattern exists, it is best to define the array with the alternative notation:

Example 2.20. *Example of explicit array function:*

```
newintvarseq 7 (\x.x*x) \arr.
        ( int_to_charseq arr.length std_out )
```

This code behaves the same way as above, but uses the predefined constant function **newintvarseq**, which has type **int** → (**int** → **int**) → ((**intvarseq** → **comm**) → **comm**). This operates in much the same spirit as other "newvar" functions in previous sections. The first argument is the length of the array, the second is an abstraction for generating the initial values for the variables in the array, and the last is an abstraction which receives the new array. Once bound to `arr`, the array then has its length converted to a string and printed, as before.

Similar predefined functions exist for generating arrays in all the primitive types, and also to allow the use of the array in abstractions that return something other than **comm**. This makes for a lengthy definition of the type of each of these identifiers.

2.11 Conclusion

This case study shows that intersection types are capable of typing powerful features that have not been used before in programming languages. Since it is not clear how to type these features without intersection types, we can believe that intersection types are crucial for the inclusion of these features.

We have seen how the use of intersection types is involved in the simpler and more flexible definition of variables (acceptors ∩ expressions), objects (intersection of fields), arrays (intersection of function and field), in terms of these powerful features. The overriding *concept* used here, is that a conventional language feature is represented with lots of other, smaller, language features all combined into the same identifier, or expression. Intersection types allow the types of these components to be "combined".

Intersection types have also been used to effect finitary type polymorphism when typing terms such as $\lambda x.x + x$, which previously could only (inflexibly) be typed with **int** → **int** or **real** → **real**. Although this code does define an ad-hoc polymorphic function, it does not define an overloaded function as used in C++ and Java, since the body of the function is the same for both cases of the type of the input variable. There seems to be no support for overloading of the user's abstractions in Forsythe in this sense. However, intersection types allow the overloading of the predefined operators to be carried through the typing of the programmer's own abstractions.

Polymorphism is also present in the definition of the subsumption relation over the primitive data-types, such as between **int** and **real**. Such polymorphism is present in many languages such as C, Haskell and Java.

Finitary polymorphism is not ideal for writing truly re-useable code, such as a linked list abstract data type, since the type assigned to such code will be a finite intersection of types, and thus never encompass all its potential uses (in this case will never represent all possible lists of various things). This is because the programmer can define their own types that are not in the intersection. What would be ideal for this is a kind of parametric polymorphism based around type variables such as used in ML. Thus a type can represent an infinity of potential uses.

One gets the impression that quantified and intersection types will complement each other quite well: Quantified types are ideal for typing re-usable factored-out code, but cannot be used to type the applications of this code until after that point. Intersection types, on the other hand, are excellent at typing the application of polymorphic code without any knowledge of what the factored-out code actually is, and the derivation for the factored-out code tends to be tailored for the set of its deployments. In other words quantified types collect together what type a re-usable subprogram will provide, but intersection types collect together what behaviour is expected from a re-usable subprogram. The next chapter therefore concerns intersection type systems with type variables, since it seems they will be useful in practice for this resason.

Chapter 3

Intersection Type Systems with Type Variables

Type variables allow the typing of truly polymoprhic code. Previously, code has been typed using finitary polymorphism, but this will not suffice for the demands of modular programming in modern software engineering. We need a method whereby code can be typed once, and instances of this type will always suffice for typing the uses of such code. The use of type variables allows for principal typings of modular code, and operations that can bridge the gap between the most abstract "principal" typing, and the concrete requirements of the context where such code is used. Thus this chapter is an account of developments in the area of intersection types with type variables, and the scope of their potential practical application.

Since we simply wish to study an intersection type system with type variables, there is no need to add support for data types, commands, and other constants like was done for the language Forsythe. Instead we can study the type system in the context of the elegant lambda calculus, since most of the properties we are interested in manifest themselves elegantly in such a language. Thus it is for the lambda calculus where most work has been done in this area.

An intersection type system for the lambda calculus with type variables and arrows, is simply an extension of the Curry Type System with the intersection type constructor, ($\cap I$), and ($\cap E$), as seen in previous chapters. There are many different systems like this. As the theory has evolved since its original conception, systems have been devised with novel properties, and it has become better understood what the essence of intersection types are, resulting in a more elegant definition. Each different system has rules with subtle differences that affect its properties. It is of interest to briefly review some of these systems, since it gives insight into the character of intersection type systems. The information about these systems is derived from [9], which speaks of each them in much more detail.

Firstly a type system was invented with the constructor $\sigma \cap \tau$ and appropriate introduction / elimination rules. Initially, intersection was not allowed to the right hand side of arrow types. This system allowed the typing of all normalised terms, since previously untypeable expressions that forced unsolvable constraints on the typings of term variables could now be typed by intersecting the types of the term variables. Terms that were not strongly normalisable could not be typed however, for example if terms contained the fix combinator $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, as a

sub-term, there would be no derivable type for the term.

This led to the introduction of the type constant ω which could be assigned to any sub-term, particularly those that could not be typed before. This gave the type system the beta expansion property, since sub-terms known to be "forgotten" during reduction could also be "forgotten" during the type assignment without affecting the derived type of the term. Now, all normalisable terms could be typed with typings that did not mention ω (although the derivations may have contained ω). This is because normalisable terms terminate through a particular choice of reductions, i.e. eventually the infinitely-reducing sub-terms are forgotten in expressions like $(\lambda y.M)((\lambda x.xx)(\lambda x.xx))$ where M does not contain y to produce a fixed point. Such forgotten terms, and y in the basis of the typing of M would be typed with ω, but the typing of the redex would be the same as the typing of M, which would not contain ω, since M would not use y.

A proof of completeness of type assignment in the intersection type system was made possible by the introduction of a complex subsumption relation and a rule that allowed types to be subsumed according to this subsumption at any place in the derivation tree. Also the type syntax was relaxed to allow intersections to the right of arrow types. Note that this system is the one that most resembles the Forsythe type system, in its use of type syntax and subsumption. This system is known as the BCD system, after its authors, and is significant in the evolution of intersection types, because it allowed the proof of these properties.

It was later shown that this system was too general. The "looseness" of the rules mean there are multiple distinct derivations of the same typing for the same term, and many types are equivalent descriptions, in the sense that they both subsume each other. This simplifies the definition of the type system, but causes a problem when trying to understand and prove properties of the type system. It is not clear which rules to choose when building a derivation, and proofs become very complex and hard to follow, since they have to account for all possibilities.

Restricting the type syntax to the original form - no intersection on the right hand side of arrows, and removing the subsumption relation from general derivations results in a much simpler system with much the same properties, but no type equivalence during η reduction / expansion. By allowing subsumption just for term variables (originally just a subset of this relation - $(\cap E)$ was allowed for term variables), the system has this property as well, but is still free from distinct derivations for the same typing. In this system, the implication $B \vdash M : \tau \Longrightarrow B \vdash M : \sigma\ (\tau \leq \sigma)$ holds, which is consistent with our intuition of subsumption. With just $(\cap E)$ in the (Ax) rule, it was impossible to derive the typing $\{x : \tau \to \sigma\} \vdash x : (\tau \cap \rho) \to \sigma$.

In this system, since there are restrictions on the right hand side of arrows, there are restrictions on the typing of the bodies of abstractions, and the result of applications. In fact the only place $(\cap I)$ is allowed is on the right hand side of an application, which is consistent with our requirement of allowing multiple types for term variables, and hence function arguments. It was also noticed that the type constant ω in the BCD and earlier systems, can be represented by the nullary intersection, and its rule can be squashed into $(\cap I)$. Since intersection is only allowed to the left of an arrow, and at the root of a typing, the logic of $(\cap I)$ was squashed into the $(\to E)$ rule, and also allowed as a special case, at the root of derivations.

The new representation of ω simplified the definition of the subsumption relation used in the (Ax) rule for term variables, but this relation is otherwise unaltered from the BCD description, except for the removal of definitions pertaining to arrow types with intersections on the right

hand side. That intersection type system is what is given here, and can be thought of as the result of this process of development. It is probably the best system to present here, because it has a useful properties, and an elegant definition that compares well with the Curry Type System.

3.1 Essential Intersection Type System

Although there is no explicit $(\cap I)$ and $(\cap E)$ in this system, it is possible to construct derivations with intersections because the "functionality" required to do this is embedded into (Ax) and $(\to E)$.

Definition 3.1. *The set of types. Here there are two levels in the definition, $\overline{\tau}$, $\overline{\sigma}$, etc are called "strict types", and range over the types that are not intersections, whereas the usual metavariables range over all types, including intersections. Note that the nullary intersection ω is represented here as $\overline{\tau}_1 \cap \ldots \cap \overline{\tau}_n \quad (n = 0)$.*

$$\begin{array}{rcl} \overline{\tau} & ::= & \varphi \mid \tau \to \overline{\sigma} \\ \tau & ::= & \overline{\tau}_1 \cap \ldots \cap \overline{\tau}_n \quad (n \geq 0) \end{array}$$

Definition 3.2. *Subsumption:* $(n \geq 0), (m \geq 1)$

$$\begin{array}{rcl} \forall\, \overline{\tau} \in \{\overline{\tau}_1 \ldots \overline{\tau}_m\} \,.\, (\overline{\tau}_1 \cap \ldots \cap \overline{\tau}_m \;\leq\; \overline{\tau}) & & \\ (\forall\, \overline{\tau} \in \{\overline{\tau}_1 \ldots \overline{\tau}_n\} \,.\, \sigma \leq \overline{\tau}) & \Longrightarrow & \sigma \leq (\overline{\tau}_1 \cap \ldots \cap \overline{\tau}_n) \\ \sigma \leq \tau \leq \rho & \Longrightarrow & \sigma \leq \rho \\ (\sigma' \leq \sigma) \;\wedge\; (\overline{\tau}' \leq \overline{\tau}) & \Longrightarrow & (\sigma \to \overline{\tau}') \leq (\sigma' \to \overline{\tau}) \end{array}$$

Definition 3.3. *Natural deduction rules for strict typings. Strict typings are not the only available typings, but they form the core of all derivations.*

$$(Ax)\ \frac{}{B \vdash_s x : \overline{\tau}}\ (x : \sigma \in B, \sigma \leq \overline{\tau})$$

$$(\to I)\ \frac{B, x : \sigma \vdash_s M : \overline{\tau}}{B \vdash_s \lambda x.M : \sigma \to \overline{\tau}}$$

$$(\to E)\ \frac{B \vdash_s M_1 : \overline{\sigma}_1 \cap \ldots \cap \overline{\sigma}_n \to \overline{\tau} \qquad B \vdash_s M_2 : \overline{\sigma}_1 \quad \ldots \quad B \vdash_s M_2 : \overline{\sigma}_n}{B \vdash_s M_1 M_2 : \overline{\tau}}\ (n \geq 0)$$

Note that the result of each of these rules is a typing with a *non-intersection* type. This means that a typing derived for an arbitrary term with the strict rules cannot be an intersection. This is not the case in other systems, because here, intersection has been squashed into the $(\to E)$ rule, and is therefore not permitted at the root of the tree. To remedy this, there is one more rule that can be appended to the root of a strict derivation to produce an "essential" derivation that allows the intersection of any number of strict typings to be a typing for a term.

Definition 3.4. *Natural deduction rule for essential typings:*

$$(Ess)\ \frac{B \vdash_s M : \overline{\tau}_1 \quad \ldots \quad B \vdash_s M : \overline{\tau}_n}{B \vdash_E M : \overline{\tau}_1 \cap \ldots \cap \overline{\tau}_n}\ (n \geq 0)$$

Thus from now on we formally consider a typing $\langle B, \tau \rangle$ to be a valid typing in the intersection type system, for a lambda term M if $B \vdash_E M : \tau$.

Note that taking $n = m = 1$ in all the above definitions produces the original Curry Type System, with the subsumption relation reduced to merely the identity relation. The presence of the nullary intersection ω is allowed through $n = 0$ in $(\rightarrow E)$, (Ess), and in the second line of the subsumption relation.

3.2 Terms Typable With This System

This, like the BCD system, can type all lambda terms in normal form without use of the nullary intersection. This can be proved by induction on terms - the interesting part is when typing an application term of the form $xM_1M_2...M_n$, the first subterm is a term variable and thus it is easy to type it as a function over the others. If the term variable is used elsewhere, we can just form the intersection in the basis. We also know that types are preserved during beta expansion, thus all normalisable terms can be typed.

Terms that do not normalise (infinitely reduce) cannot be typed except with the nullary intersection. The canonical example of an infinitely reducing term is $(\lambda x.xx)(\lambda x.xx)$. Attempting to type this without using $(Ess)(n = 0)$ requires use of $(\rightarrow E)$, and an agreement between the two $(\lambda x.xx)$. It is impossible to solve this agreement because making an appropriate change to the sub-derivation of one produces a reaction in the other, and the system never stabilises. Because there are restrictions on where intersection types may be used in a derivation, when typing such terms with ω, sometimes it is necessary to type a larger context of the term with ω, up to a $(\rightarrow E)$ rule, or maybe up to the root of the derivation.

Weakly normalisable terms such as $(\lambda x.z)((\lambda x.xx)(\lambda x.xx))$ can be typed because the non-normaliseable part is an argument that is (eventually) forgotten in the course of reduction, and thus can be typed with the nullary intersection without affecting the derivation for the body of the abstraction $\lambda x.z$. The nullary intersection is in fact not present in the eventual typing, since this typing is equivelent to the typing of the normal form of the term, and as mentioned earlier, the derivation of the normal form need not contain the nullary intersection.

3.3 Principal Typings

The principal typing property is useful for separate compilation, recompilation, and accurate error messages [2]. It is therefore of great interest to see how the principal typing property manifests itself in an intersection type system of this nature.

The definition of a principal typing algorithm as a depth-first parse of the abstract syntax of the term is plagued by the problem of knowing which sub-terms to type with the nullary intersection (because they cannot be typed otherwise). We therefore want to nullify all un-normalisable subterms, but detecting which ones they are is an instance of the halting problem, and thus undecidable.

However it is possible to define the principal typing for a term using the $\beta_=$ property, and the fact that it is possible to type normal forms with an algorithm. The way this is formalised is showing how the set of typings of "approximate normal forms" has a maximum element, or is in fact infinite. An approximate normal form of a term is a term after a certain number of reduction steps, that has had certain subterms replaced with a special symbol ($\perp$) so that it contains no redexes (hence normal form). $\perp$ is typed with the nullary intersection. In effect, the set of typings of approximate normal forms is the set of possible typings of partial reductions with various parts typed with the nullary intersection. This allows "partial" reductions to be typed in the same way that normal forms can be typed, which otherwise wouldnt be possible.

There is more complexity however, since only the right hand premise of a ($\rightarrow E$) rule, or the root of a derivation can be used to type a subterm with a nullary intersection. Therefore the terms we can replace with $\perp$ are limited to subterms on the right of applications and the root itself.

Recall that the principal typing property is present if there is a typing for each typable term, which can be converted to any other typing for the term with an operation on the typing itself. It can be proved that an arbitrary term, when typed by the above method, can be instantiated into all other terms. For the Curry Type System, this instantiation operation was substitution, and for the same reason it is clear that this concept will play a role here. There are some typings which cannot be instantiated from the principal typing with just substitution however. To prove the principal typing property requires the introduction of other methods of instantiation as will be discussed in the next section.

Although other type systems in common use do not have principal typings, it is most desirable that they do, since it brings immediate benefits to programming. Since intersection type systems have this property, this is evidence that they will be useful in practice.

3.4 Operations On Typings

Originally principality was proved for the earlier systems, such as in [7]. The required operations and the definition of the principal types are slightly different for each system, since the rules, type syntax, and hence the derivations, are different, and operations on typings work by mutating the derivation in a consistent way.

Here is given a summary of operations that apply to the essential system. It is described how the derivation is mutated, and how this modification manifests itself in the derived typing. This information is derived from [8] and [7] (the former applies better to the essential system) where formal descriptions are given. This set of operations is not the only set which is sufficient for this type system, but it is sound, so no operation can produce an invalid typing.

3.4.1 Substitution

Firstly substitution is defined differently to the Curry Type System. Although it works in the same way - by exploiting the way that the derivation rules do not tie down the typing completely. If a derivation involves a type variable, that variable can be replaced with either another type variable, or an arrow, as long as this change is repeated throughout the derivation.

There are no rules in the derivation that require a type to be a type variable, and there are no rules that forbid an arrow type, but intersection types are only allowed in very specific places. Substitution is thus only allowed to replace type variables with strict types. This requirement is also present in the type syntax, since intersections are not allowed on the right of arrows.

As a special case, it is also possible to replace a type variable with the nullary intersection. This causes more complexity since if a type variable is to the right of an arrow, and this is done, then the whole arrow is reduced to the nullary intersection. In [9] this capability is removed from substitution, in favour of a 4th operation called covering, but covering is not sound on essential typings.

3.4.2 Lifting

Liftings work by changing the amount of removal between σ and τ in an instance of the rule (Ax) where $\sigma \leq \tau$. The effect on the derived typing is that the basis can be made smaller, or the type can be made larger, in terms of $(\leq)$.

Example 3.5. *Example of lifting in the typing of a term variable x, verify with the definition of* $(\leq)$*:*

- $\langle \{x : \overline{\sigma} \cap \overline{\tau}\}, \overline{\sigma} \cap \overline{\tau} \rangle$ *will lift to*
- $\langle \{x : \overline{\rho} \cap \overline{\sigma} \cap \overline{\tau}\}, \overline{\sigma} \cap \overline{\tau} \rangle$ *will lift to*
- $\langle \{x : \overline{\rho} \cap \overline{\sigma} \cap \overline{\tau}\}, \overline{\tau} \rangle$ *will lift to*
- ...

Operations are expressed as functions from types to types, so lifting is denoted with a pair of typings, e.g. $L_{\langle B_1,\tau_1\rangle\langle B_2,\tau_2\rangle}$ where $B_1 \geq B_2$ and $\tau_1 \leq \tau_2$. When applied to a typing, if the typing is equal to the first, then it returns the second, otherwise it simply returns the typing it was given. It will never return $\langle B_1, \tau_1 \rangle$.

3.4.3 Expansion

Consider a principal derivation with an instance of the $(\rightarrow E)$ rule. It is clearly possible to duplicate one of the sub derivations that forms the intersection type on the right-hand premise, renaming all the type variables to make it distinct. In the algorithms used, it is easier to rename the original subtree as well, the type variables in this original tree will not be present in any part of the resulting typing. This change to the derivation will clearly affect the argument of the arrow on the left hand side of the rule, and thus the sub-derivation rooted there, and thus perhaps the derived basis or type below the $(\rightarrow E)$ rule. Also since the duplicated sub-derivation will need to derive new types from the basis, the basis will have to be altered to contain intersections of new and old types for all the term variables concerned.

It is also possible to expand at the level of the (Ess) rule, which will produce a very symetrical-looking typing. Here is an example of expansion. First is presented a derivation for $x(yz)$, this example is used in [9].

Example 3.6. *Proof tree that demonstrates potential for expansion.* $B = \{x : \varphi_1 \to \varphi_2,\ y : \varphi_3 \to \varphi_1,\ z : \varphi_3\}$.

$$\dfrac{\dfrac{(Ax)\ \dfrac{}{B \vdash_s x : \varphi_1 \to \varphi_2} \quad (\to E)\ \dfrac{(Ax)\ \dfrac{}{B \vdash_s y : \varphi_3 \to \varphi_1} \quad (Ax)\ \dfrac{}{B \vdash_s z : \varphi_3}}{B \vdash_s yz : \varphi_1}}{B \vdash_s x(yz) : \varphi_2}\ (\to E)}{B \vdash_E x(yz) : \varphi_2}\ (Ess)$$

There are 3 places in this derivation where an expansion can occur: the two $(\to E)$ rules, and the (Ess) rule. The simplest example of the effect of expansion is the topmost $(\to E)$ rule. Here we see the sub-tree for z is duplicated, and the effect propogates to the definition of the basis because it affects the left hand sub-tree of said $(\to E)$ rule.

Example 3.7. *Expansion of topmost* $(\to E)$ *rule.* $B = \{x : \varphi_1 \to \varphi_2,\ y : (\varphi_3 \cap \varphi_4) \to \varphi_1,\ z : \varphi_3 \cap \varphi_4\}$.

$$\dfrac{\dfrac{(Ax)\ \dfrac{}{B \vdash_s x : (\varphi_1 \cap \varphi_5) \to \varphi_2} \quad (\to E)\ \dfrac{(Ax)\ \dfrac{}{B \vdash_s y : (\varphi_3 \cap \varphi_4) \to \varphi_1} \quad (Ax)\ \dfrac{}{B \vdash_s z : \varphi_3} \quad (Ax)\ \dfrac{}{B \vdash_s z : \varphi_4}}{B \vdash_s yz : \varphi_1}}{B \vdash_s x(yz) : \varphi_2}\ (\to E)}{B \vdash_E x(yz) : \varphi_2}\ (Ess)$$

Note how the basis was affected by the expansion. If the second $(\to E)$ rule is expanded, the result is very similar.

Example 3.8. *Expansion of lower* $(\to E)$ *rule.* $B = \{x : (\varphi_1 \cap \varphi_5) \to \varphi_2,\ y : (\varphi_3 \to \varphi_1) \cap (\varphi_4 \to \varphi_5),\ z : \varphi_3 \cap \varphi_4\}$.

$$\dfrac{\dfrac{(Ax)\ \dfrac{}{B \vdash_s x : \varphi_1 \to \varphi_2} \quad (\to E)\ \dfrac{(Ax)\ \dfrac{}{B \vdash_s y : \varphi_3 \to \varphi_1} \quad (Ax)\ \dfrac{}{B \vdash_s z : \varphi_3}}{B \vdash_s yz : \varphi_1} \quad (\to E)\ \dfrac{(Ax)\ \dfrac{}{B \vdash_s y : \varphi_4 \to \varphi_5} \quad (Ax)\ \dfrac{}{B \vdash_s z : \varphi_4}}{B \vdash_s yz : \varphi_5}}{B \vdash_s x(yz) : \varphi_2}\ (\to E)}{B \vdash_E x(yz) : \varphi_2}\ (Ess)$$

The references to y and z in the basis are altered because the duplicated derivation derives renamed types from it. Also the type of x in the basis has to be changed because the argument of the arrow is now an intersection type. So far the changes to the derivation have been quite local to the point of expansion, but where the basis is actually used elsewhere in the derivation, things get more tricky. Where another axiom rule references an altered (intersected) type in the basis, do we pull out the original type (as a member of the intersection, using $(\leq)$), or the whole intersection? If the latter is chosen, we can use the operation of lifting to get the other option, but if the former is chosen, the resultant derivation is simpler.

In the essential sytem, the latter option is chosen but this causes problems where the new intersection type is derived into a place which doesnt allow an intersection type. Consider the following derivation:

Example 3.9. *Derivation that has potential for complicated expansion.* $B = \{y : \varphi_1\}$*:*

$$\dfrac{\dfrac{\dfrac{}{B \cup \{x : \varphi_1\} \vdash_s x : \varphi_1}(Ax)}{B \vdash_s \lambda x.x : \varphi_1 \to \varphi_1}(\to I) \quad \dfrac{}{B \vdash_s y : \varphi_1}(Ax)}{\dfrac{B \vdash_s (\lambda x.x)y : \varphi_1}{B \vdash_E (\lambda x.x)y : \varphi_1}(Ess)}(\to E)$$

If we expand at the $(\to E)$ rule, we induce a change in the argument of the type of $\lambda x.x$, which manifests itself in the basis in that sub-derivation, and thus causes the right hand side of the arrow to become an intersection type as well: $(\varphi_1 \cap \varphi_2) \to (\varphi_1 \cap \varphi_2)$. This right hand intersection type then becomes the typing for the term.

Example 3.10. *Illegal derivation, from expanding the above.* $B = \{y : \varphi_1 \cap \varphi_2\}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{B \cup \{x : \varphi_1 \cap \varphi_2\} \vdash_s x : \varphi_1 \cap \varphi_2}(Ax)}{B \vdash_s \lambda x.x : (\varphi_1 \cap \varphi_2) \to (\varphi_1 \cap \varphi_2)}(\to I) \quad \dfrac{}{B \vdash_s y : \varphi_1}(Ax) \quad \dfrac{}{B \vdash_s y : \varphi_2}(Ax)}{\dfrac{B \vdash_s (\lambda x.x)y : \varphi_1 \cap \varphi_2}{B \vdash_E (\lambda x.x)y : \varphi_1 \cap \varphi_2}(Ess)}(\to E)$$

This is not allowed because of the intersection type to the right of the arrow, but there is an equivelent derivation that is. Here, the expansion has been pushed down to the (Ess) rule.

Example 3.11. *Fixed derivation.* $B = \{y : \varphi_1 \cap \varphi_2\}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{B \cup \{x : \varphi_1\} \vdash_s x : \varphi_1}(Ax)}{B \vdash_s \lambda x.x : \varphi_1 \to \varphi_1}(\to I) \quad \dfrac{}{B \vdash_s y : \varphi_1}(Ax)}{B \vdash_s (\lambda x.x)y : \varphi_1}(\to E) \quad \dfrac{\dfrac{\dfrac{}{B \cup \{x : \varphi_2\} \vdash_s x : \varphi_2}(Ax)}{B \vdash_s \lambda x.x : \varphi_2 \to \varphi_2}(\to I) \quad \dfrac{}{B \vdash_s y : \varphi_2}(Ax)}{B \vdash_s (\lambda x.x)y : \varphi_2}(\to E)}{B \vdash_E (\lambda x.x)y : \varphi_1 \cap \varphi_2}(Ess)$$

These are but a few simple examples, there are other circumstances that can force the expansion to a lower point in the derivation. Already, however, we can see that expansion is a very intricate operation that affects the whole derivation tree. Since the operation must be expressed as an algorithm on a typing (without reference to a derivation), this algorithm is rather abstract and complex.

Different people have taken different approaches to specifying this algorithm, putting aside the differences between type systems. It is possible to collect together the subtypes that are affected, and then expand them as they occur in the typing starting with the biggest instances [7]. Also it is possible to collect the affected type variables together, and then expand the strict sub-types within the typing that end with one of these type variables [8].

This mechanism is promoting what was seen in the last example - that if it is unsuitable to expand at a certain point, because doing so creates an invalid type, we expand further down in the derivation tree. That is we expand a larger subtype. By also considering the sub-types of the expanded sub-types, we can capture the changes to the basis, and the changes to other types that occur as a result, throughout the derivation.

In addition to the original typing, the type to be expanded, and the number of copies to be generated, must be specified to do an expansion. Thus there are four parameters to this operation. In [7] there were only three, since the expansion caused a single duplication, the idea being the operation could be repeated if more were desired.

3.4.4 Comments

The operations defined for a system are merely chosen to show that any valid typing can be instantiated from the principal typing, to prove the principal typing property for that system. There may be different ways of formalising the operations as algorithms on the typing itself (i.e. without direct reference to a derivation). Operations can also overlap, for example a substitution will do the same thing as an expansion, assuming certain paramaters for each. It seems likely that certain representations of operations will be more intuitive for programmers, and this is certainly relevant for the practical use of intersection types.

In [3] it is noted that the operations, particularly expansion, are very complex. An alternative is presented, that extends the notion of substitution enough so that it is sufficient to instantiate the principal typing into any typing required. The intention is to make intersection types more accessible to non-theorists, which is clearly important for their practical application.

The way this extension works is for the derivation to introduce an "expansion variable" into the derivation (with an appropriate rule). This expansion variable marks the subtypes that are affected by a particular expansion. The type inference procedure builds a derivation with expansion variables in the correct places. Substitution then maps expansion variables to expansions, in addition to its normal role in mapping type variables onto types. When the substitution is applied to a type containing an expansion variable, it expands the marked subtype according to the expansion defined in the substitution, i.e. it duplicates the subtype, renames type variables to make it distinct, and forms the intersection.

3.5 Unification

In [7], [9], [8], and others, algorithms are given to find the principal typing for arbitrary lambda normal forms, and the property of $\beta_=$ is used to generalise these results to the rest of the lambda terms. It is possible to give the principal typing for an arbitrary lambda term, however, without first beta-reducing it to normal form.

Recall that computing the principal typing of an application in normal form is trivial because the term variable on the far left can be typed to a suitable arrow using the principal typings of the arguments in the application. From now on we will call the principal typings of all the sub-terms the "sub-typings". There are constraints that must be satisfied between the sub-typings in this case, but they are easy to solve since we can just *choose* any arbitrary but suitable typing for the term variable sub-typing as discussed in section 3.2.

Generalising this to arbitrary terms requires that all the sub-typings in the application be instantiated in a manner that is consistent with the constraints of the $(\rightarrow E)$ rule, since the sub-typing for left hand side will no longer be of the form $\langle\{x : \varphi\}, \varphi\rangle$. The chosen solution to

this problem of finding suitable instantiation operations is *unification*. Using unification is the traditional method of building the principal typing for a term, since the property of $\beta_=$ is not to be found in many non-intersection type systems. The first principal typing algorithm using unification for intersection types concerned the BCD system, and appeared in [6] together with a proof of correctness and a proof of completeness.

Now we describe the problem of principal typing algorithms and unification for an intersection type discipline more formally. We relate this problem to the equivelant problem in the Curry Type System. We describe how the unification algorithm proceeds, and how unification is used by the principal typing algorithm in [6].

The principal typing algorithm for arbitrary terms traverses the term top-down, recursively, in a syntax directed way, and is trivially defined for term variables and abstractions, since the principal typing of a term variable is $\langle\{x : \varphi\}, \varphi\rangle$ and it is clear how to generate the typing for an abstraction from the typing of its body. When computing the principal typing for an application, however, the typing can be derived from the sub-typings but the algorithm must invoke the unification algorithm. This determines only the *necessary* operations required to instantiate the sub-typings into forms consistent with the application rule. That unification does not give unnecessary operations, is essential if the principal typing algorithm is to actually provide the *principal* typing, rather than an instance thereof. In the Curry Type System, the constraints are as follows:

- The left hand side of an application is an arrow (of the form $\sigma \to \tau$ for some σ, τ).
- σ is the same as the type for the right hand side of the application.
- The basis (B) must be the same for both sides of the application.

The principal typing is thus defined to be $\langle B, \tau\rangle$. Unification is used to ensure these constraints are met. This is done in a rather technical way: Suppose we are trying to find the principal typing for X_1X_2, and $\langle B_1, \pi_1\rangle$ and $\langle B_2, \pi_2\rangle$ are the sub-typings respectively. By unifying π_1 and $\pi_2 \to \varphi$, operations are found that will instantiate both sub-typings into forms that satisfy the first two constraints. Taking these two instantiated sub-typings, it is possible to unify the bases term-variable for term-variable to find operations that will instantiate the sub-typings further so that the third condition is met. Then the principal typing can be found by extracting B from these last sub-typing instances (either one - it doesnt matter since they are the same) and by applying the collection of found operations to the type variable φ, which will produce τ [10].

In an intersection type system, the constraints are more relaxed. Subsumption in (Ax) means that there is no longer an identity relationship between typings of term variables, and the assumptions in the basis. Specifically we can use finitary polymorphism to list the typing requirements, so the bases can be unified simply by intersecting them, term-variable for term-variable. The other constraints still hold however. Suppose we find the principal typing for X_1X_2, and $\langle B_1, \pi_1\rangle$ and $\langle B_2, \pi_2\rangle$ are the sub-typings respectively, we must find instances of $\langle B_1, \pi_1\rangle$ and $\langle B_2, \pi_2\rangle$ that satisfy the following constraints:

- the left hand side of an application is an arrow ($\sigma \to \tau$)

- σ is the same as the type for the right hand side of the application. To make the Essential system meet with the BCD system, we can consider the right hand sub-typing to have an intersection type, and σ to be this same intersection type.

The principal typing is thus $\langle B_1 \cap B_2, \tau \rangle$. The principal typing algorithm proceeds in exactly the same way as for the Curry Type System, except there is no need to unify the bases after applying the initial operations to them, they are simply intersected, term-variable for term-variable. It is of interest now to understand that in the Curry Type System, unification fails (returns an exception) when a type variable is unified with a type that contains that variable (there is no operation that will convert both to the equivelent types). This used to happen when unifying the bases, for example when finding the principal typing of xx it was necessary to unify $x : \varphi_2$ and $x : \varphi_1 \to \varphi_2$.

It is impossible for unification to fail when unifying two disjoint principal types π_1 and $\pi_2 \to \varphi$, so the principal typing algorithm for intersection types never returns an exception. The unification algorithm actually does not return an exception at all, when no unification is possible, but instead returns a substitution that reduces both types to the nullary intersection. The theorem still holds in the form that the principal typing algorithm does not return ω as a type.

Now we review the manner of operation for the unification algorithm in [6]. Firstly its trivial operation: When unifying ω with anything, the only possibility is to return a substitution that converts everything to ω. For the unification of a type variable with any type, the variable is substituted with the type. As just mentioned above, if the variable occurs in the type, there is no choice but to return a substitution to ω. The remaining cases require recursive calls to the unification algorithm.

The case where one type is an intersection and the other is not, returns an expansion of the type that is not, so that when applied, tye type becomes an intersection. This is where the algorithm begins to lose its elegance of definition however, since the operation of expansion has more than a local effect on a typing, it is impossible to apply the expansion to the sub-type where the algorithm is currently studying. The expansion must be applied to the whole typing and the process of unification must begin again.

Thus both of the original "root" typings (including bases!) must be passed to each recursive invocation of the algorithm so that if an expansion is required, they are available. Similarly, if an expansion was performed in a recursive step, the algorithm has already been repeated from scratch, so the answer returned from that recursive step is the ultimate answer. This detail plays a part in the unification of a pair of arrows. The use of "closures" (locally defined in [6] does not affect the result of unification but is necessary because of technical intricacies in the definition of expansion in that paper. Closures are not desirable because they complicate the definition of the principal typing algorithm, and obfuscate its true nature.

The case where two arrows are unified is identical to the corresponding case in the Curry System. First a unification is found for the left side (by recursive use of the unification algorithm), these operations are then applied to the right side and then unification is used again. The two chains of operations collected form these calls are composed and returned as a single chain.

The remaining case is where two intersections need to be unified. This proceeds in a similar way to the arrow case. First the left hand side of each intersection is unified, the resulting chain

is appled to each right hand side, which are then further unified to produce another chain. Both chains are composed and returned as a single chain. This is interesting because although the arrow type constructor is not commutative, the binary intersection type constructor used in the BCD system is. This unification algorithm, however, may return different results for $\sigma \cap \tau$ and $\tau \cap \sigma$.

While this counter-intuitive property does not seem to affect the correctness or completeness of the principal typing algorithm, it is none-the-less undesirable and may show up in pragmatic extensions of intersection type systems. It is perhaps an effect of the conceptual distance between the symbolic representation of intersections (a binary operator) and their intuitive meaning (a set of types that hold). By being syntax directed, the unification algorithm treats the intersection type as a tree, rather than a set.

In [3], an alternative approach is taken to the problem of solving the constraints imposed by the $(\rightarrow E)$ rule. An entirely different type inference algorithm is defined: Firstly a deduction tree (called a "skeleton") is explicitly built from the term using a syntax directed algorithm called "skel". This skeleton has a similar structure to the (hypothetical) principal derivation, but is constructed in a "dumb" way, without satisfying the constraints at each $(\rightarrow E)$. As well as using explicit definitions of skeletons, the paper defines substitution as an operation on skeletons (explicitly), rather than on typings as seen before. Because the paper uses expansion variables, substitution is the only operation required.

Then, a set of explicit constraints, Δ, is formed by inspecting the skeleton. These take the form of equality constraints between the types at each $(\rightarrow E)$, for example:

Example 3.12. *The constraint for this part of a skeleton is* $\rho = \tau \rightarrow \sigma$.

$$(\rightarrow E)\ \frac{B \vdash M_1 : \rho \qquad B \vdash M_2 : \tau}{B \vdash M_1 M_2 : \sigma}$$

The next step - solving these constraints - is the unification algorithm. This will take Δ and produce a set of substitutions that convert the primitive skeleton onto a sound principal derivation for the original term, thus completing the type inference. The explicitness of the approach in [3] makes for an algorithm, the nature of which is clearly and elegantly presented.

3.6 Undecidability

The problem of undecidability with the intersection type system is far worse than the problems associated with the Curry Type System and other type systems in practical use. It completely prevents a type inference algorithm for an unrestricted intersection type system being used in practice. The problem is that although all terminating programs can be typed, non-terminating ones cannot. This can be demonstrated by attempting to unify $\varphi_1 \cap (\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_2$ and $\varphi_1 \cap (\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_2 \rightarrow \varphi_3$ - which is required when inferring the type of the canonical example of an infinitely-reducing lambda term $(\lambda x.xx)(\lambda x.xx)$.

Recall that the purpose of a type system is the static analysis of computer programs, at compile time. This must be autonomous, and thus if we cannot guarantee termination, we will

have a compiler that "crashes". Forsythe solved this problem by using type annotations instead of full-on type inference, and also since it did not have type variables, there was no notion of principality for typings, since there were no operations on typings. Unification was not required in the Forsythe implementation.

Many other type systems are undecidable, in fact the search for "advanced" type systems (that solve some of the limitations of currently-used practical type systems such as used in ML) has come up with many systems that are undecidable. The root of this property is that the type system is too close to the actual language it types, in the sense of representing its computational behaviour. The intersection type system as described in this chapter, and many others in the literature that are very similar, is in fact a perfect representation of the behaviour of the lambda calculus, with its "everything is a function" ethos. It is likely therefore, that restricting the type system will produce a decidable type system. Of course the useful properties of the unrestricted system may not manifest themselves in a restriction, so it is necessary to choose a good restriction.

It turns out that finite rank restrictions of an intersection type system with type variables, such as the essential system, are not only decidable, but have principal typings [3]. This is a fantastic result, since such systems can type many more terms than any other system, even systems which do not have principal typings. The type system with rank restriction n is restricted in its type syntax: Intersection types are not allowed beyond a depth of n arrows. Recall that the unification algorithm loops because expansions and substitutions create types that keep growing. By restricting the amount that types can grow to, you are effectively giving the unification algorithm a finite amount of "fuel". When it runs out of this fuel, it will return an exception denoting that the term was not typeable. This algorithm is still correct and complete when it does terminate successfully, however, and thus a decidable restriction is found.

3.7 Conclusion

We have studied the technical nature of intersection types when typing lambda calculus terms, including the properties that it can type all normalisable terms, and is closed over beta reduction / expansion. We have also seen how intersection types have great power when used within a practical language such as Forsythe, but to admit the typing of truely re-useable code, will require type variables.

We have seen how an intersection type system with variables has principal typings with but requires an extended set of operations to be complete. Although undecidability would make the type system useless for practical languages in its unrestricted form, it turns out that a simple restriction provides a decidable inference algorithm that is more powerful than any other in existence, and also has principal typings. This result is very encouraging for the use of intersection types in practice.

References

[1] Joseph J. Hallett and Assaf J. Kfoury. Programming examples needing polymorphic recursion. Technical Report BUCS-TR-2004-004, Department of Computer Science, Boston University, January 2004.

[2] Trevor Jim. What are principal typings and what are they good for? Tech. memo. MIT/LCS/TM-532, MIT, 1995.

[3] Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004.

[4] John C. Reynolds. Replacing complexity with generality: The programming language forsythe. Unpublished manuscript, but `ftp://ftp.cs.cmu.edu/user/jcr/forsytheintro.ps.gz`, 1991. Carnegie Mellon University, Computer Science Department.

[5] John C. Reynolds. Design of the programming language forsythe. *Volume 1: ALGOl-like languages*, pages 173–233, 1997.

[6] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci.*, 59(1-2):181–209, 1988.

[7] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theor. Comput. Sci.*, 28(1-2):151–169, 1984.

[8] Steffen van Bakel. Principal type schemes for the strict type assignment system. *Logic and Computation*, pages 643–670, 1993.

[9] Steffen van Bakel. Intersection type assignment systems. *Theor. Comput. Sci.*, 151(2):385–435, 1995.

[10] Steffen van Bakel. Type systems for programming languages course notes. Course website (`http://www.doc.ic.ac.uk/~svb/TSfPL/`), 2001. Taught course at Imperial College, London.

[11] J. B. Wells. The essence of principal typings. 2002.

[12] J. B. Wells and Christian Haack. Branching types. *Inform. & Comput.*, 200X.