Ein Modulsystem zur Unterstützung von subtypbasierter Überladung unter Wahrung separater Namensräume

Christian-Albrechts-Universität Kiel
Institut für Informatik und Praktische Mathematik

Diplomarbeit

Ein Modulsystem zur Unterstützung von subtypbasierter Überladung unter Wahrung separater Namensräume

Stephan Herhut

16. Februar 2005

Betreut durch Prof. Dr. Michael Hanus*
und Dr. Sven-Bodo Scholz†

*Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel
†Faculty of Engineering and Information Sciences, University of Hertfordshire, United Kingdom

Inhaltsverzeichnis

1. Einleitung **1**

2. Single Assignment C **5**
2.1. SaC 5
2.2. SaC' 5
2.2.1. Syntax 6
2.2.2. Semantik 7
2.2.3. Typen 10
2.2.4. Funktionsüberladung 12

3. Herausforderungen **17**
3.1. Namensräume vs. Überladung 17
3.2. Überladung und Rekursion über Namensraumgrenzen 22
3.3. Optimierungen und Separate Compilation 27
3.4. Funktionsspezialisierung 29
3.5. Zusammenfassung 30

4. Vorhandene Ansätze **33**
4.1. Überladung und Namensräume 33
4.2. Rekursion über Modulgrenzen 35

5. Konzeptuelle Lösung **39**
5.1. Grundlagen des Modulsystems 39
5.2. Kontrollierte Überladung über Namensraumgrenzen 40
5.2.1. Erweiterte Syntax von SaC' 43
5.2.2. Erweiterte Semantik von SaC' 46
5.2.3. Das Diamant-Import-Problem 51
5.3. Rekursion über Namensraumgrenzen 52
5.3.1. Erweiterte Semantik von SaC' 54
5.3.2. Erweitertes Diamant-Import-Problem 57
5.4. Verfügbarkeit des Quelltextes 59
5.4.1. Entkopplung des Modulsystems 61
5.5. Funktionsspezialisierung und Adaptive Module 62
5.6. Fazit 64

6. Implementierung **65**
6.1. Das compilierende System 65
6.2. Sichern des abstrakten Syntaxbaumes 67
6.2.1. Beschreibung des abstrakten Syntaxbaumes 68
6.2.2. Implementierung der Zugriffs-Schicht 70
6.2.3. Sichern des Syntaxbaumes 70
6.2.4. Einlesen des Syntaxbaumes 72
6.3. Implementierung der Erweiterung um Module 74
6.3.1. `provide` und `export` Anweisungen 74
6.3.2. `use` Anweisungen und qualifizierte Bezeichner 74
6.3.3. Erzeugen der Sichten für eine `import` Anweisung 75
6.3.4. Lösen des Diamant-Import-Problems 77
6.4. Erzeugen der Funktionsspezialisierungen 78
6.5. Fazit 78

7. Zusammenfassung und Ausblick **79**
7.1. Ausblick 80

A. Codebeispiele **83**
A.1. Überladung über Namensraumgrenzen in Haskell 83
A.1.1. Beispiel für das Einschmuggeln von Instanzen 83
A.1.2. Beispiel für überdeckende Instanzen 84

B. XML-Darstellung des abstrakten Syntaxbaumes **87**
B.1. Verwendetes XML Schema 87
B.2. Ausschnitt aus der XML Beschreibung 93
B.3. Visualisierung der XML Beschreibung 99

C. Literaturverzeichnis **101**

Abbildungsverzeichnis

2.1. Syntax der Sprache SaC' (in BNF) 6
2.2. Beispielprogramm in der Sprache SaC' 7
2.3. Syntax der Sprache $\mathcal{F}un$ (in BNF) 8
2.4. Transformationsschema der Sprache SaC' in die Sprache $\mathcal{F}un$ 9
2.5. Transformation des Beispielprogramms aus Abbildung 2.2 10
2.6. Syntax der Typen in SaC' (in BNF) 12
2.7. Subtyphierarchie in SaC' 12
2.8. Beispiel für Funktionsüberladung in SaC'. 13
2.9. Erweiterte Syntax von $\mathcal{F}un$ 14
2.10. Erweiterung des Transformationsschemas für Funktionsüberladung . . 14
2.11. Transformationsschema für das `letrec_ovld` Konstrukt 15

3.1. Überladung von Funktionen über Namensraumgrenzen 18
3.2. Graphische Darstellung lokaler Überladung 20
3.3. Graphische Darstellung globaler Überladung 21
3.4. SaC Pseudocode der schnellen Fourier Transformation 23
3.5. Verteilte Definition der Funktion `FFT` (lokale Überladung) 23
3.6. Erweiterter Dispatch der Funktion `B:FFT` 24
3.7. Angepasster rekursiver Aufruf 25
3.8. Beispiel der Fourier Transformation mit 3 Namensräumen 26
3.9. Code Beispiel für Funktionsinlining 27
3.10. Code Beispiel nach dem Funktionsinlining 28
3.11. Spezialisierung am Beispiel der Funktion `FFT` 29

4.1. Beispiel zweier Mixin Module 35
4.2. Kombination beider Mixin Module 36
4.3. Funktion mit Pattern-Matching 37
4.4. Verteilte Funktion mit Pattern-Matching 37

5.1. Um Module erweiterte Syntax von SaC' 39
5.2. Beispiel Überladung über Modulgrenzen 41
5.3. Verfeinertes Beispiel für Überladung über Modulgrenzen 42
5.4. Syntax der Schnittstellenbeschreibung in SaC' (in BNF) 43
5.5. Beispiel für `provide`, `use`, `export` und `import` 45
5.6. Erweiterte Syntax der Sprache $\mathcal{F}un$ (in BNF) 47

5.7. Erweitertes Transformationsschema von SaC' nach $\mathcal{F}un$ 48
5.8. Vereinfachter SaC'-Code für das Modul `AlgArb` 49
5.9. Ergebnis der Transformation des Beispiels aus Abbildung 5.8 50
5.10. Modulkontext in der Sprache $\mathcal{F}un$ 50
5.11. Beispiel für das Diamant-Import-Problem 51
5.12. Sicht der Funktion `FFT` im Modul **A** 53
5.13. Sicht der Funktion `FFT` im Modul **B** 53
5.14. Transformationsschema für SaC' inklusive Sichten 55
5.15. Erweiterter Modulkontext in der Sprache $\mathcal{F}un$ 56
5.16. Beispiel für das Diamant-Import-Problem im Kontext von Sichten . . 57
5.17. Schematische Darstellung des `Fundef` Knotens 60
5.18. Schematische Darstellung der Abstraktionsschicht 62

6.1. Schematische Darstellung der Übersetzungsphasen 66
6.2. Darstellung des `Fundef` Knotens in XML. 69

1. Einleitung

Die Idee, große Programme in einzelne Module zu zerlegen, hat beim Entwurf von Programmiersprachen eine lange Tradition. Bereits Sprachen wie ASSEMBLER oder C unterstützen die Möglichkeit der partiellen Übersetzung eines Programms. Allerdings ist in diesen Sprachen noch kein Konzept von Modulen vorhanden. Die partielle Übersetzung ist hier vielmehr ein Bestandteil des jeweiligen Compilers und weitestgehend von der Sprache losgelöst.

Das Sprachkonzept der Module wurde in den 70er Jahren entwickelt [Par72], zuerst nur mit dem Ziel, die Übersetzung der zunehmend komplexeren Anwendungen mit den damals beschränkten Ressourcen zu ermöglichen [Wir94]. Dazu ist es nötig, die komplexen Anwendungen in kleinere, separat übersetzbare Teile zu zerlegen und diese erst in einem zweiten Schritt zum endgültigen Programm zu binden. Diese Teile konnten durch Module erstmals in der Sprache selbst definiert werden.

Neben Beschränkungen der zur Verfügung stehenden Ressourcen gibt es noch eine zweite Motivation. Mit zunehmender Größe der erstellten Anwendungen nimmt auch die Anzahl der an einem Projekt beteiligten Programmierer zu. Die dadurch entstehenden impliziten Abhängigkeiten zwischen den von verschiedenen Programmierern erstellten Teilen einer Anwendung erschweren die Softwareentwicklung [DDH72]. Somit gilt es Verfahren zu finden, diese Abhängigkeiten zu minimieren und formal zu beschreiben. Dies führt zur Idee der Schnittstellendefinition zwischen einzelnen Modulen. Sie ermöglicht es, die eigentliche Implementierung im Modul zu verstecken und den Zugriff auf die in einem Modul enthaltene Funktionalität zu kapseln [DK75]. Diese Art der Softwareentwicklung wird auch als *modulares Softwaredesign* bezeichnet.

Eine Erweiterung des Modul-Konzepts sind die sogenannten *Namensräume*. Unter Namensraum wird eine strikt abgetrennte Menge von Bezeichnern verstanden. Dadurch, dass jedem Modul seine eigene Menge an Bezeichnern zugeordnet wird, können Namenskonflikte effektiv verhindert werden. Somit werden die impliziten Abhängigkeiten zwischen einzelnen Modulen weiter verringert. Eine der ersten Sprachen mit einem Modulsystem mit Unterstützung von Namensräumen war MODULA-2 [Wir85]. Heute findet man ähnliche Modulsysteme in den meisten Sprachen (Beispiele sind ML [MTHM97], Haskell [Jon03], Java [GJS00], C# [HWG03], etc.). Neuere Sprachentwicklungen stellen dabei zunehmend höhere Anforderungen an die von der jeweiligen Sprache unterstützten Möglichkeiten zur Modularisierung.

Eine besondere Herausforderung entsteht hierbei durch die Überladung von Funktionen. Mittels Funktionsüberladung ist es möglich, verschiedene, semantisch ähnliche Definitionen einer Funktion zu einer gemeinsamen, überladenen Funktion zusammenzufassen. Die einzelnen Funktionsdefinitionen werden dabei auch als *Instanzen* einer Funktion bezeichnet. Erste Ansätze zur Funktionsüberladung finden sich bereits in

der Sprache C. Hier sind die arithmetischen Grundfunktionen wie `+`, `-`, `*` und `/` überladen, d.h. unabhängig vom verwendeten Typ der Argumente wird immer der gleiche Funktionsbezeichner verwendet. Da in C die Typen der Argumente statisch bekannt sind, kann der Compiler einem Aufruf dieser überladenen Funktionen statisch anhand der Argumenttypen die tatsächlich zu verwendende Instanz zuordnen. Dieser Vorgang wird auch als *Dispatch* bezeichnet. Eine Erweiterung dieser überladenen Funktionen um benutzerdefinierte Funktionen ist jedoch nicht vorgesehen.

In funktionalen Sprachen wie Haskell oder Clean ist die Funktionsüberladung durch das Konzept der Typklassen [HHPW96] abgebildet. Diese ordnen einer Menge von Typen eine Menge von auf diesen Typen definierten Operationen zu. Für jeden einer Typklasse zugehörigen Typ müssen dabei für alle der Typklasse zugehörigen Operationen entsprechende Instanzen definiert sein. Insbesondere ist es in diesen Sprachen möglich, sowohl benutzerdefinierte Instanzen einer solchen Typklasse anzugeben, als auch benutzerdefinierte Typklassen selbst zu spezifizieren.

Im Gegensatz zur Überladung in C kann bei der auf Typklassen basierenden Funktionsüberladung der Dispatch in der Regel nicht statisch erfolgen [PJ93, Jon94]. Da Typklassen auch als Argumenttyp einer Funktionsdefinition verwendet werden können, ist während der Übersetzung einer solchen Definition der Dispatch für darin verwendete überladene Funktionen statisch nicht möglich. Statt dessen muss zur Laufzeit des Programms die jeweils passende Instanz gewählt werden.

Diese Problematik wird im Kontext separater Namensräume noch verschärft. Hier stellt sich die Frage, welche Instanzen zur Laufzeit für einen solchen Funktionsaufruf zu berücksichtigen sind. Die Berücksichtigung aller global innerhalb eines Programmkontextes definierter Instanzen widerspricht dem Separationsgedanken getrennter Namensräume. Eine solche Vorgehensweise würde Instanzen überladener Funktionen vollständig vom Namensraumkonzept ausnehmen und überladene Funktionen global sichtbar machen. Werden ausschließlich innerhalb eines Namensraums definierte Instanzen für eine überladene Funktionen berücksichtigt, schränkt dies die Funktionsüberladung ein und verhindert eine effektive Modularisierung. Alle Instanzen einer überladenen Funktion müssten dann innerhalb eines Moduls definiert sein.

Um die volle Flexibilität zu erhalten wird also ein Konzept benötigt, welches einerseits überladenen Funktionen ebenfalls einen festen Namensraum zuordnet, andererseits aber auch eine Verteilung der einzelnen Instanzen auf mehrere Namensräume erlaubt. Potenziell kann damit jeder Namensraum eine eigene Version einer überladenen Funktion mit einer jeweils eigenen Menge von Instanzen definieren. Dadurch ist eine überladene Funktion nicht mehr nur die Menge aller definierten Instanzen, sondern wird zu einer Entität des jeweiligen Namensraumes.

Eine weitere Schwierigkeit ergibt sich hierbei durch überlappende Instanzen innerhalb einer überladenen Funktion [PJM97]. Hier kann der bereits getroffene Dispatch durch Hinzufügen einer die gewählte Instanz teilweise überdeckenden Instanz nachträglich beeinflusst werden. Der Dispatch wird dadurch vom aufrufenden Kontext abhängig. Insbesondere bei rekursiv über den Typ ihrer Argumente definierten Funktionen müssen für rekursive Aufrufe alle für den ursprünglichen Aufruf der überladenen Funktion verwendeten Instanzen beim Dispatch berücksichtigt werden. Der

Dispatch des rekursiven Aufrufs hängt somit nicht nur vom Namensraum, in dem die Instanz definiert wurde, sondern auch vom Namensraum, in dem die entsprechende überladene Funktion definiert ist, ab. Dies stellt besondere Anforderungen an das Laufzeitsystem.

Im Rahmen dieser Arbeit wird ein Modulsystem vorgestellt, welches ein neues Konzept für die Überladung von Funktionen über Namensraumgrenzen hinweg bietet. Insbesondere ermöglicht es, die Sichtbarkeit einer Instanz präzise auf einzelne Namensräume zu beschränken, ohne dabei die Überladung per se auf einen Namensraum zu begrenzen.

Eine Sprache, welche sowohl die Überladung von Funktionen mit überlappenden Instanzen, als auch Namensräume bietet, ist das 1994 von Sven-Bodo Scholz vorgeschlagene Single Assignment C (SaC) [Sch94]. Insbesondere ist die Funktionsüberladung mit sich überlappenden Instanzen auf Basis von Subtypen [Sch01] integraler Bestandteil des Sprachkonzepts von SaC. Die beschriebenen Probleme treten hier also verstärkt zu Tage. SaC bietet somit optimale Voraussetzungen, um die genannten Probleme näher zu untersuchen und dient aufgrund dessen als Basis für die Implementierung des vorgestellten Modulsystems.

Ziel der Entwicklung von SaC ist es, die Spezifikation von numerischen Algorithmen auf einem hohen Abstraktionsniveau zu ermöglichen, ohne Abstriche bei der Laufzeiteffizienz im Vergleich zu konventionellen Sprachen wie Fortran machen zu müssen [GS00]. Die hohe Laufzeiteffizienz der durch den SaC-Compiler `sac2c` erzeugten Programme beruht dabei auf hoch entwickelten Optimierungen auf Basis partieller Auswertung [MW01]. Dies stellt weitere Anforderungen an ein Modulsystem. Insbesondere muss sicher gestellt werden, dass trotz Modularisierung und separater Übersetzung der Programme keine Verschlechterung der Laufzeiteffizienz eintritt.

Bei der hier vorgestellten Implementierung wird daher ein besonderes Augenmerk auf die partielle Auswertung über Modulgrenzen hinweg gelegt. Insbesondere erlaubt sie den Zugriff auf die Definition einer Funktion über die Grenzen eines Moduls und das Hinzufügen partieller Auswertungen zu einem Modul. Die ansonsten durch die Modularisierung zu erwartenden Einschränkungen für die Optimierungen können dadurch vollständig vermieden werden. Das vorgestellte Modulsystem verhält sich somit transparent bezüglich der zu erwartenden Laufzeiteffizienz der erzeugten Programme.

Der Rest der Arbeit gliedert sich wie folgt: In Kapitel 2 wird die Sprache SaC vorgestellt und die im Rahmen dieser Arbeit benötigten Elemente in Form der reduzierten Sprache SaC' eingeführt. Kapitel 3 beschreibt die Herausforderungen, die im Kontext der Sprache SaC an ein Modulsystem gestellt werden, näher. Insbesondere werden die durch die erwähnte Überladung auf Basis von Subtypen entstehenden Probleme diskutiert. In Kapitel 4 werden darauf aufbauend vorhandene Lösungsansätze in den Sprachen Haskell, Clean und ML vorgestellt und ihre Anwendbarkeit auf das vorgestellte Szenario überprüft. Kapitel 5 führt ein neues Konzept für die Funktionsüberladung über Modulgrenzen ein. Desweiteren werden Lösungen für die durch das compilierende System entstehenden Anforderungen vorgestellt. Ein Überblick über die Implementierung dieses Ansatzes für den SaC-Compiler `sac2c` wird in Kapitel 6 gegeben. Kapitel 7 gibt einen abschließenden Überblick.

2. Single Assignment C

Das im Rahmen dieser Arbeit vorgestellte Modulsystem wurde primär für die Sprache SaC (Single Assignment C) entwickelt. Im folgenden Kapitel werden grundlegende Ideen und Konzepte der Sprache SaC eingeführt und eine für diese Arbeit relevante Teilmenge von SaC in Form der vereinfachten Sprache SaC' vorgestellt. Eine vollständige Beschreibung der Sprache SaC ist in [Sch03] zu finden.

2.1. SaC

SaC ist eine funktionale Sprache erster Ordnung mit Unterstützung für Arrays als Sprachelemente erster Klasse. Ziel der Entwicklung von SaC ist es, numerische Algorithmen auf großen Datenmengen auf einem möglichst hohen Abstraktionsniveau spezifizieren zu können, ohne Abstriche beim Laufzeitverhalten im Vergleich zu konventionellen Sprachen wie C [KR90] oder Fortran [Weh85] machen zu müssen [GS00, GS03].

Die Syntax von SaC ist stark an die der Sprache C [KR90] angelehnt, um den Umstieg von imperativen Sprachen auf SaC leichter zu gestalten. Im Gegensatz zu C unterstützt SaC jedoch Arrays als Basistypen und die Überladung von Funktionen anhand ihrer Argumenttypen. Die Laufzeiteffizienz von SaC-Programmen trotz des hohen Abstraktionsniveaus wird durch eine Reihe von Code Optimierungen [Sch98, MW01] erreicht.

Im Kontext eines Modulsystems sind die arithmetischen Eigenschaften von SaC von untergeordneter Bedeutung, so dass an dieser Stelle auf eine Beschreibung verzichtet werden kann. Bei der im folgenden Abschnitt vorgestellten Sprache SaC' handelt es sich daher um eine Teilmenge von SaC, die um arithmetische Ausdrücke, Bedingungen, Schleifen, etc. reduziert wurde. Alle im Rahmen dieser Arbeit für SaC' gemachten Aussagen lassen sich jedoch leicht auf SaC übertragen.

2.2. SaC'

In den folgenden Abschnitten wird zunächst die Syntax von SaC' vorgestellt, um darauf aufbauend die Semantik durch ein Transformationsschema in einen angewandten λ-Kalkül formal zu beschreiben. Weitere Abschnitte geben einen Überblick über das Typsystem von SaC' und erläutern die in SaC' angewandten Prinzipien der Funktionsüberladung.

2.2.1. Syntax

Abbildung 2.1 zeigt die Syntax von SaC' in Backus-Nauer Form. Wie im vorigen Abschnitt erwähnt, lehnt sich die Syntax von SaC – und somit auch die von SaC' – eng an die Syntax der Sprache C an. Wie in C besteht ein Programm in SaC' aus einer Sequenz von Funktionsdefinitionen.

Funktionen werden mit der in C üblichen Syntax definiert. Abweichend von C kann eine SaC' Funktion beliebig viele Werte als Resultat zurückliefern. Eine Beschreibung der in SaC' verwendeten Typen wird in Abschnitt 2.2.3 gegeben. Da SaC' eine funktionale Sprache ohne Seiteneffekte ist, muss eine Funktion jedoch mindestens einen Rückgabewert liefern. Für den aus der Sprache C bekannten Rückgabetyp `void` gibt es keine funktionale Entsprechung. Daher ist die `return` Anweisung in SaC' nicht optional. Jeder Funktionsrumpf muss eine solche Anweisung am Ende enthalten. An anderen Positionen innerhalb des Rumpfes sind hingegen keine `return` Anweisungen erlaubt.

$$
\begin{array}{lcl}
\mathit{Program} & \Rightarrow & \mathit{Definitions}\ \mathit{main} \\
\mathit{Definitions} & \Rightarrow & [\ \mathit{Fundef}\ ]^{*} \\
\mathit{Fundef} & \Rightarrow & \mathit{Type}[\ \texttt{,}\ \mathit{Type}\ ]^{*}\ \mathit{Id}\ \texttt{(}\ [\ \mathit{Arg}[\ \texttt{,}\ \mathit{Arg}\ ]^{*}\ ]\ \texttt{)}\ \mathit{Block} \\
\mathit{main} & \Rightarrow & \texttt{int main ( )}\ \mathit{Block} \\
\mathit{Arg} & \Rightarrow & \mathit{Type}\ \mathit{Id} \\
\mathit{Block} & \Rightarrow & \texttt{\{}\ [\ \mathit{Assignment}\ ]^{*}\ \mathit{Return}\ \texttt{\}} \\
\mathit{Assignment} & \Rightarrow & \mathit{Id}\ \texttt{=}\ \mathit{Expr}\ \texttt{;} \\
\mathit{Expr} & \Rightarrow & \mathit{Const} \\
 & | & \mathit{Id} \\
 & | & \mathit{Id}\ \texttt{(}\ [\ \mathit{Expr}[\ \texttt{,}\ \mathit{Expr}\ ]^{*}\ ]\ \texttt{)} \\
 & | & \mathit{Expr}\ \mathit{PrimOp}\ \mathit{Expr} \\
\mathit{PrimOp} & \Rightarrow & \texttt{+} \\
 & | & \texttt{-} \\
 & | & \texttt{*} \\
 & | & \texttt{/} \\
\mathit{Return} & \Rightarrow & \texttt{return (}\ [\ \mathit{Expr}[\ \texttt{,}\ \mathit{Expr}\ ]^{*}\ ]\ \texttt{)}
\end{array}
$$

Abbildung 2.1.: Syntax der Sprache SaC' (in BNF)

```
1 int add( int a, int b)
  {
3   result = a + b;
    return( result);
5 }

7 int main()
  {
9   a = 2;
    b = 40;
11  sum = add( a, b);
    return( sum);
13 }
```

Abbildung 2.2.: Beispielprogramm in der Sprache SaC'

Eine besondere Stellung nimmt die Funktion `main` ein. Sie bestimmt den Startpunkt der Programmausführung. Analog zur Sprache C ist die Signatur der Funktion vorgegeben: sie erwartet keine Argumente und liefert einen ganzzahligen Wert als Resultat der Programmausführung zurück. Jedes SaC' Programm muss genau eine Definition einer Funktion `main` mit der gegebenen Signatur beinhalten.

Ausdrücke in SaC' umfassen neben Funktionsaufrufen grundlegende arithmetische Funktionen wie `+`, `-`, `*` und `/`. Diese sind für die Betrachtungen im Rahmen dieser Arbeit von untergeordneter Bedeutung, auf eine formale Beschreibung wird an dieser Stelle verzichtet. Allerdings finden sie in den gegebenen Beispielen Verwendung, um diese aussagekräftiger gestalten zu können.

Abbildung 2.2 zeigt ein Beispielprogramm in SaC'. Das dargestellte Programm besteht aus zwei Funktionsdefinitionen. In Zeile 1 ff. wird die Funktion `add` definiert. Die Funktion erwartet zwei ganzzahlige Werte als Argument und liefert die ganzzahlige Summe als Ergebnis. Als zweite Funktion wird `main` in Zeile 7 ff. definiert. In Zeile 11 innerhalb der Funktion `main` wird die Funktion `add` mit den Argumenten `2` und `40` aufgerufen. Als Ergebnis liefert die Funktion somit die Zahl `42`.

Der hier vorgestellte Sprachkern enthält noch keine Konstrukte zur Modularisierung von Programmen. Diese werden im Laufe der Arbeit ergänzt.

2.2.2. Semantik

Im Folgenden wird eine formale Definition der Semantik der Sprache SaC' vorgestellt. Zu diesem Zweck wird zuerst ein angewandter λ-Kalkül $\mathcal{F}un$ als Standard-Semantik eingeführt. Darauf aufbauend wird dann ein Transformationsschema von der Sprache SaC' in die Sprache $\mathcal{F}un$ vorgestellt. Die Syntax von $\mathcal{F}un$ ist in Abbildung 2.3 dargestellt.

$\mathcal{F}un$ umfasst die bekannten Elemente eines angewandten λ-Kalküls wie Bezeichner,

$$
\begin{array}{lcl}
\mathit{Program} & \Rightarrow & \mathit{Expr} \\
\\
\mathit{Expr} & \Rightarrow & \mathit{Const} \\
 & | & \mathit{Id} \\
 & | & \mathit{Lambda} \\
 & | & \mathit{Let} \\
 & | & \mathit{Letrec} \\
 & | & \mathit{Ap} \\
 & | & \mathit{PrimFun} \\
\\
\mathit{Lambda} & \Rightarrow & \lambda\ \mathit{Id}\ .\ \mathit{Expr} \\
\\
\mathit{Let} & \Rightarrow & \texttt{let}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr}\ \texttt{in}\ \mathit{Expr} \\
\\
\mathit{Letrec} & \Rightarrow & \texttt{letrec}\ [\ \mathit{Id}\ \texttt{=}\ \mathit{Expr}\ ]^{+}\ \texttt{in}\ \mathit{Expr} \\
\\
\mathit{Ap} & \Rightarrow & \texttt{(}\ \mathit{Expr}\ \mathit{Expr}\ \texttt{)} \\
\\
\mathit{PrimFun} & \Rightarrow & \texttt{+} \\
 & | & \texttt{-} \\
 & | & \texttt{*} \\
 & | & \texttt{/}
\end{array}
$$

Abbildung 2.3.: Syntax der Sprache $\mathcal{F}un$ (in BNF)

die Abstraktion (vgl. *Lambda* in Abbildung 2.3) und die Applikation (vgl. *Ap* in Abbildung 2.3). Des weiteren umfasst $\mathcal{F}un$ als syntaktischen Zucker mit dem `let` Konstrukt (vgl. *Let* in Abbildung 2.3) eine abkürzende Schreibweise für Applikationen von Abstraktionen. Rekursive Bindungen können in $\mathcal{F}un$ durch das `letrec` Konstrukt (vgl. *Letrec* in Abbildung 2.3) spezifiziert werden. Die Semantik ergibt sich durch die Standard-Definition der δ- und β-Reduktion [Bar81, HS86].

Neben den vorgestellten Grundelementen eines λ-Kalküls verfügt $\mathcal{F}un$ über primitive Operationen. Diese umfassen die arithmetischen Grundfunktionen `+`, `-`, `*` und `/`. Im Laufe der Arbeit wird die Menge der primitiven Operationen noch weiter ergänzt. Für alle primitiven Operationen wird dabei von einer strikten Semantik ausgegangen. Des weiteren wird als Reduktionsstrategie eine *applicative-order* Evaluation vorausgesetzt.

Abbildung 2.4 beschreibt das Transformationsschema von SaC' nach $\mathcal{F}un$. Da die Typinformationen in SaC' lediglich im Kontext der Überladung von Funktionen verwendet werden, ist eine Berücksichtigung für das hier vorgestellte Transformationsschema ohne Funktionsüberladung nicht notwendig. Sie werden daher bei der Transformation von SaC' nach $\mathcal{F}un$ verworfen.

$\mathbb{C}[\![$ *Fundefs* $]\!] \rightsquigarrow$ *(Program)*
```
letrec
    C[[ Fundefs ]]
in main
```

$\mathbb{C}[\![\ \tau_1, \ldots, \tau_n$ *Id* `(` τ_{n+1} *Arg*$_1$`,` $\ldots$ `,` τ_{n+m} *Arg*$_m$ `) {` *Body* `}` $]\!] \rightsquigarrow$ *(Fundef)*
Id `=` λ *Arg*$_1$ `.` $\cdots$ λ *Arg*$_m$ `.` $\mathbb{C}[\![$ *Body* $]\!]$

$\mathbb{C}[\![$ *Id* `=` *Expr* `;` *Assignments Return* $]\!] \rightsquigarrow$ *(Body)*
```
let
    Id = C[[ Expr ]]
in C[[ Assignments Return ]]
```

$\mathbb{C}[\![$ `return (` *Expr*$_1$, ..., *Expr*$_n$ `);` $]\!] \rightsquigarrow$ *(Return)*
$\mathbb{C}[\![$ *Expr*$_1$ $]\!] \cdots \mathbb{C}[\![$ *Expr*$_n$ $]\!]$

$\mathbb{C}[\![$ *Id* `(` *Expr*$_1$ `, ... ,` *Expr*$_n$ `)` $]\!] \rightsquigarrow$ *(Expr*$_1$*)*
`(` *Id* $\mathbb{C}[\![$ *Expr*$_1$ $]\!] \cdots \mathbb{C}[\![$ *Expr*$_n$ $]\!]$ `)`

$\mathbb{C}[\![$ *Expr*$_a$ *PrimOp Expr*$_b$ $]\!] \rightsquigarrow$ *(Expr*$_2$*)*
`(` `(` *PrimOp Expr*$_a$ `)` *Expr*$_b$ `)`

$\mathbb{C}[\![$ *Const* $]\!] \rightsquigarrow$ *Const* *(Expr*$_3$*)*

$\mathbb{C}[\![$ *Id* $]\!] \rightsquigarrow$ *Id* *(Expr*$_4$*)*

Abbildung 2.4.: Transformationsschema der Sprache SAC' in die Sprache $\mathcal{F}un$

Die Regel *Program* gibt die Transformationsvorschrift für Programme der Sprache SAC' an. Die innerhalb eines Programms enthaltenen Funktionsdefinitionen werden durch die Regel *Fundef* transformiert. Der resultierende Term der Sprache $\mathcal{F}un$ wird auf oberster Ebene durch ein `letrec` Konstrukt umgeben, welches den globalen Bindungsbereich des SAC' Programms abbildet. Als Startterm wird der Bezeichner `main` angegeben. Da in jedem SAC' Programm eine Funktion `main` definiert sein muss, ist garantiert, dass `main` durch das `letrec` Konstrukt gebunden ist.

Funktionsdefinitionen werden durch die Regel *Fundef* mittels des `letrec` Konstrukts auf oberster Ebene in eine globale Bindung transformiert. Weiter werden die freien Vorkommen der Argumente im Funktionsrumpf mittels λ-Abstraktionen nach außen gebunden und der Funktionsrumpf transformiert. Die Regel *Body* beschreibt diese Transformation. Jede Zuweisung innerhalb des Rumpfes wird in ein `let` Konstrukt transformiert. Dazu wird der Ausdruck der Sprache SAC' auf der rechten Seite der Zuweisung in einen Term der Sprache $\mathcal{F}un$ überführt und dieser dann an den Bezeichner auf der linken Seite der Zuweisung gebunden. Als Zielterm

```
letrec
   add =  λa. λb.  let
                      result = a + b
                   in result
   main =   let
               a = 2
            in  let
                   b = 40
                in  let
                       sum = (add a b)
                    in sum
in main
```

Abbildung 2.5.: Transformation des Beispielprogramms aus Abbildung 2.2

wird der Resultatsterm der Transformation des übrigen Rumpfes gewählt.

Eine solche Zuweisungskette wird immer durch eine `return` Anweisung terminiert. Diese wird mittels der Regel *Return* transformiert. Dabei wird jeder Ausdruck innerhalb der `return` Anweisung in einen Term der Sprache $\mathcal{F}un$ transformiert.

Der Regelsatz *Expr* beschreibt die Transformation eines Ausdrucks der Sprache SaC' in einen Term der Sprache $\mathcal{F}un$. Konstante Werte und Bezeichner werden direkt übernommen, da sie sich in beiden Sprachen entsprechen. Funktionsapplikationen in SaC' werden durch eine Applikation des Funktionsbezeichners auf die transformierten Argumente ersetzt. Der Funktionsbezeichner wurde durch die Regel *Fundef* vorher global gebunden. Anwendungen primitiver Operationen der Sprache SaC' werden analog zu Funktionsapplikationen transformiert.

Abbildung 2.5 zeigt das Resultat der Transformation des SaC' Beispielprogramms aus Abbildung 2.2 in die Sprache $\mathcal{F}un$.

2.2.3. Typen

Im Folgenden werden die in der Sprache SaC' verwendeten Typen vorgestellt. Dabei wird zwischen Datentypen und Funktionstypen unterschieden[1].

Datentypen

Wie bereits in Abschnitt 2.1 erwähnt, unterstützt die Sprache SaC Arrays als Objekte erster Klasse. Dies manifestiert sich auch in der Menge der in SaC' vorhandenen Typen: grundsätzlich ist jeder Datentyp in SaC' ein Array-Typ. Im Falle eines Skalars hat dieser Array-Typ die Dimensionalität 0.

Um sowohl den Typ der Elemente eines Arrays, als auch die Ausdehnung kodieren zu können, besteht jeder Datentyp in SaC' aus zwei Komponenten, dem Element-Typ und der Ausdehnung (*shape*) des Arrays.

[1] SaC unterstützt zusätzlich benutzerdefinierte Typen. Diese sind für das hier vorgestellte Modulsystem von untergeordneter Bedeutung und gehören daher nicht zum Sprachumfang von SaC'.

Als Basistypen für Elemente eines Arrays stehen alle aus C bekannten primitiven Typen (`int`, `double`, etc.) zur Verfügung. Die Ausdehnung eines Datentyps wird im sogenannten Shape-Vektor kodiert. Dabei werden 4 Klassen von Shape-Vektoren für Typen unterschieden:

AUD *Array unbekannter Dimensionalität* (*array of unknown dimension*) bezeichnet Arrays beliebiger, nicht näher bekannter Dimensionalität. Dies schließt auch Skalare als Arrays der Dimensionalität 0 ein.

AUDGZ *Array unbekannter Dimensionalität größer Null* (*array of unknown dimension greater zero*) bezeichnet Arrays beliebiger, nicht näher bekannter Dimensionalität mit Ausnahme von Skalaren (Arrays der Dimensionalität 0).

AKD *Array bekannter Dimensionalität* (*array of known dimension*) bezeichnet Arrays bekannter Dimensionalität. Die exakte Ausdehnung des Arrays ist jedoch unbekannt.

AKS *Array bekannter Ausdehnung* (*array of known shape*) bezeichnet Arrays, deren Ausdehnung exakt bekannt ist.

Allgemein wird der Shape-Vektor eines Typs als eine durch Kommata separierte Liste der Ausdehnung des Arrays in seiner jeweiligen Dimension dargestellt. Um anzudeuten, dass es sich um einen Vektor handelt, wird diese mit eckigen Klammern (`[ ]`) umgeben. Die Dimensionen werden dabei beginnend bei der ersten Dimension im Shape-Vektor des Typs aufgelistet. Für skalare Typen wird der leere Vektor `[]` verwendet. Abkürzend kann in diesem Fall der Shape-Vektor auch entfallen.

Um den Typ eines Arrays mit unbekannter Ausdehnung angeben zu können, wird das `.` (Punkt) Symbol als Platzhalter vereinbart. Ein dreidimensionales Array hat als Shape-Vektor seines Typs somit den Vektor mit drei Platzhaltern `[.,.,.]`. Für Arrays unbekannter Dimension wird der Platzhalter `*` (Stern) für **AUD** bzw. `+` (Plus) für **AUDGZ** verwendet. Abbildung 2.6 zeigt die Syntax von Typen in SaC' als Ergänzung der in Abbildung 2.1 vorgestellten Syntax.

Des weiteren ergibt sich eine natürliche Subtypbeziehung zwischen den einzelnen Klassen der Shape-Vektoren. Abbildung 2.7 gibt einen Überblick. Die Klasse **AUD** umfasst alle Arrays eines gegebenen Element-Typs. Somit ist jeder Array-Typ mit Element-Typ α ein Subtyp des **AUD** Typs α`[*]`.

Eine Ebene tiefer in der Hierarchie liegt die Klasse der **AUDGZ** Typen. Diese umfasst alle Arrays mit Ausnahme der Arrays der Dimensionalität 0. Somit sind alle Array-Typen mit Element-Typ α und Dimensionalität größer als 0 Subtypen des **AUDGZ** Typs α`[+]`. Auf der nächsten Ebene liegt die Klasse der **AKD** Typen. Jeder **AKD** Typ umfasst alle Arrays einer gegebenen Dimensionalität. Damit bildet er den Supertyp aller entsprechenden, in der untersten Ebene der Subtyphierarchie liegenden **AKS** Typen.

Funktionstypen

Funktionstypen werden in SaC' implizit durch die Funktionssignaturen angegeben. Dennoch werden im Rahmen dieser Arbeit explizite Funktionstypen verwendet. Dabei

$$
\begin{array}{lcl}
\mathit{Type} & \Rightarrow & \mathit{Basetype}\ \mathit{Shape} \\
\\
\mathit{Basetype} & \Rightarrow & \texttt{bool} \\
 & | & \texttt{char} \\
 & | & \texttt{int} \\
 & | & \texttt{double} \\
 & | & \texttt{float} \\
\\
\mathit{Shape} & \Rightarrow & \mathit{AUD} \\
 & | & \mathit{AKD} \\
 & | & \mathit{AKS} \\
\\
\mathit{AUD} & \Rightarrow & \texttt{[*]} \\
 & | & \texttt{[+]} \\
\\
\mathit{AKD} & \Rightarrow & \texttt{[.}\,[\,\texttt{, .}\,]^{+}\,\texttt{]} \\
\\
\mathit{AKS} & \Rightarrow & \texttt{[}\,[\,\mathit{Num}[\,\texttt{,}\,\mathit{Num}\,]^{+}\,]\,\texttt{]}
\end{array}
$$

Abbildung 2.6.: Syntax der Typen in SaC' (in BNF)

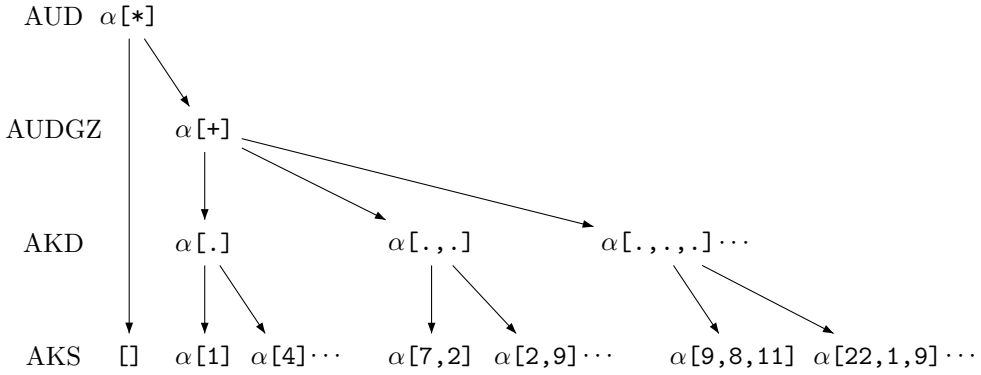


Abbildung 2.7.: Subtyphierarchie in SaC'

wird eine curryfizierte Darstellung benutzt. Da Funktionen in SaC' potenziell mehrere Rückgabewerte besitzen, werden diese zu einem Produkttypen zusammengefasst.

Der Funktionstyp der Funktion `int, float flip( float a, int b)` wird somit durch `float` → `int` → `int` × `float` dargestellt.

2.2.4. Funktionsüberladung

Wie einleitend erwähnt, unterstützt SaC das Überladen von Funktionen anhand der Funktionsargumente. Dieser Abschnitt erläutert die Grundlagen der Funktionsüberladung in SaC'.

Mit *Funktionsüberladung* bezeichnet man die Möglichkeit, einen Funktionsbezeichner simultan in mehreren Funktionsdefinitionen zu benutzen. Dies erlaubt es, Funktionen mit ähnlicher Funktionalität, unabhängig vom Typ der Argumente, den gleichen Bezeichner zuzuordnen. Im Folgenden werden die einzelnen Funktionsdefinitionen als *Instanzen* der überladenen Funktion bezeichnet. Die Gesamtheit der einzelnen Instanzen wird als *überladene Funktion* oder auch kurz *Funktion* bezeichnet.

Beispiele von überladenen Funktionen finden sich bereits in der Sprache C: hier sind die arithmetischen Grundfunktionen überladen. Unabhängig vom Typ der Argumente bezeichnet `+` immer die Addition[2]. Allerdings ist das Überladen von Funktionen in C auf eingebaute Funktionen beschränkt. Der Programmierer kann keine benutzerdefinierten Funktionen überladen oder benutzerdefinierte Instanzen zu überladenen Funktionen hinzufügen.

In SaC' kann potenziell jede Funktion überladen werden. Die einzelnen Instanzen werden hierbei durch Funktionsdefinitionen mit identischem Funktionsbezeichner angegeben. Eine gesonderte Hervorhebung der Instanzen einer überladenen Funktion, wie sie sich bei den in Haskell verwendeten Typklassen [HHPW96] findet, ist dabei nicht notwendig. Abbildung 2.8 zeigt die Spezifikation einer überladenen Funktion am Beispiel der Funktion `add`. Diese umfasst zwei Instanzen für den Elementtyp `int`. In Zeile 1 ff. wird eine Instanz für Arrays unbekannter Dimensionalität definiert. Die genaue Berechnungsvorschrift wurde dabei ausgelassen. Die zweite Instanz für Skalare Werte ist in Zeile 8 ff. definiert. Beide Instanzen bilden zusammen die überladene Funktion `add`. Zur Laufzeit des Programmes wird dann anhand der Typen der Argumente die passende Instanz gewählt. Dieser Vorgang wird auch als *Dispatch* bezeichnet.

[2] Dies gilt nur, solange es sich bei den Argumenten um Werte handelt, die als Zahl interpretiert werden können. In C ist dies für alle Basistypen der Fall.

```
1 int[*] add( int[*] A, int[*] B)
  {
3   result = ...

5   return( result);
  }
7
  int add( int a, int b)
9 {
    result = a + b;
11
    return( result);
13 }
```

Abbildung 2.8.: Beispiel für Funktionsüberladung in SaC'.

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \ldots \\
 & | & \mathit{LetrecOvld} \\
 & | & \mathit{Ovld} \\
 & & \\
\mathit{LetrecOvld} & \Rightarrow & \texttt{letrec\_ovld}\ \lceil\ \mathit{Id}\ \texttt{=}\ \mathit{Expr}\ \rceil^{+}\ \texttt{in}\ \mathit{Expr} \\
 & & \\
\mathit{Ovld} & \Rightarrow & \texttt{ovld}\ \texttt{(}\ \lceil\ \mathit{Id}\lceil\ \texttt{,}\ \mathit{Id}\ \rceil^{*}\ \rceil\ \texttt{)}
\end{array}
$$

Abbildung 2.9.: Erweiterte Syntax von $\mathcal{F}un$

$$
\begin{array}{l}
\mathbb{C}[\![\ \mathit{Fundefs}\ ]\!] \rightsquigarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{(Program)} \\
\quad \texttt{letrec\_ovld} \\
\qquad \mathbb{C}[\![\ \mathit{Fundefs}\ ]\!] \\
\quad \texttt{in main} \\
 \\
\mathbb{C}[\![\ \tau_1\texttt{,}\ldots\texttt{,}\tau_n\ \mathit{Id}\ \texttt{(}\ \tau_{n+1}\ \mathit{Arg_1}\texttt{,}\ \ldots\texttt{,}\ \tau_{n+m}\ \mathit{Arg_m}\ \texttt{)}\ \texttt{\{}\ \mathit{Body}\ \texttt{\}}\ ]\!] \rightsquigarrow \qquad \textit{(Fundef)} \\
\quad \mathit{Id}_{\phi(\tau_{n+1},\ldots,\tau_{n+m})}\ \texttt{=}\ \lambda \mathit{Arg_1}.\ \cdots\ \lambda \mathit{Arg_m}.\ \mathbb{C}[\![\ \mathit{Body}\ ]\!]
\end{array}
$$

Abbildung 2.10.: Erweiterung des Transformationsschemas für Funktionsüberladung

Wie an dem gegebenen Beispiel deutlich wird, ist es insbesondere möglich, verschiedene Instanzen einer Funktion für verschiedene Ausdehnungen und Dimensionalitäten der Argumente zu spezifizieren. Für eine Funktionsanwendung wird jeweils die Instanz für den Dispatch gewählt, welche für den kleinst möglichen Supertyp der entsprechenden Funktionsargumente (vgl. Abbildung 2.7) definiert ist. Für eine Anwendung der überladenen Funktion `add` aus Abbildung 2.8 auf ein Argument vom Typ `int[.]` wird somit die Instanz für `int[*]` gewählt. Bei der Definition der einzelnen Instanzen muss sichergestellt werden, dass diese Wahl der für den Dispatch zu verwendenden Instanz für alle Argumenttypen eindeutig ist. Eine Beschreibung der daraus resultierenden Bedingungen und der formalen Semantik findet sich in [Sch03].

Der Dispatch-Vorgang wird in SaC' durch sogenannte *Wrapper-Funktionen* übernommen. Alle Funktionsaufrufe überladener Funktionen werden statt zu einer Instanz der Funktion zur Wrapper-Funktion geleitet. Innerhalb dieser wird dann anhand der Anzahl und des Typs der Argumente der Aufruf an die eigentliche Instanz weitergeleitet. Eine genaue Beschreibung des Dispatch-Mechanismus in SaC wird in [Kre03] vorgestellt.

Um diesen Mechanismus auch in der Sprache $\mathcal{F}un$ abbilden zu können, wird diese um das `letrec_ovld` und `ovld` Konstrukt erweitert (vgl. Abbildung 2.9). Des weiteren müssen die Regeln *Program* und *Fundef* des Transformationsschemas aus Abbildung 2.4 wie folgt abgewandelt werden.

Die Regel *Program* erzeugt statt des `letrec` Konstruktes nun ein `letrec_ovld`, um die Funktionsbezeichner auf oberster Ebene zu binden. Die Semantik des `letrec_ovld` Konstruktes wird im Folgenden noch genauer beschrieben.

$$\mathcal{OVLD}\left\{\begin{array}{l}\texttt{letrec\_ovld}\\ \quad Fun^1_{\phi_1} = Expr^1_1\\ \quad\vdots\\ \quad Fun^1_{\phi_{j_1}} = Expr^1_{j_1}\\ \quad\vdots\\ \quad Fun^m_{\phi_1} = Expr^m_1\\ \quad\vdots\\ \quad Fun^m_{\phi_{j_m}} = Expr^m_{j_m}\\ \texttt{in}\ Expr\end{array}\right\} = \begin{array}{l}\texttt{letrec}\\ \quad \mathcal{OVLD}\,\{\{Fun_i = Expr_i | Fun_i \in \alpha_1\}\}\\ \quad\vdots\\ \quad \mathcal{OVLD}\,\{\{Fun_i = Expr_i | Fun_i \in \alpha_m\}\}\\ \texttt{in}\ Expr\end{array}$$

wobei $\{\alpha_1, \ldots, \alpha_m\}$ die Menge der Äquivalenzklassen der Funktionsbezeichner $\{Fun^1_{\phi_1}, \ldots, Fun^m_{\phi_{jm}}\}$ bezüglich der Gleichheit der Bezeichnernamen ist.

$$\mathcal{OVLD}\,\{\{Fun_{\phi_1} = Expr_1, \ldots, Fun_{\phi_n} = Expr_n\}\}$$

$$= \left\{\begin{array}{l}Fun_{\phi_1}\ \texttt{=}\ Expr_1\\ \vdots\\ Fun_{\phi_n}\ \texttt{=}\ Expr_n\\ Fun\ \texttt{=}\ \texttt{ovld(}\ Fun_{\phi_1}\texttt{,}\ \ldots\texttt{,}\ Fun_{\phi_n}\texttt{)}\end{array}\right.$$

Abbildung 2.11.: Transformationsschema für das `letrec_ovld` Konstrukt

Die Funktionsbezeichner selbst erhalten durch die Regel *Fundef* einen eindeutigen Index, welcher durch die Funktion ϕ aus der Anzahl und dem Typ der Argumente der Funktion abgeleitet wird. Dies dient einerseits dazu, sicherzustellen, dass alle Funktionsbezeichner innerhalb des `letrec` Konstrukts eindeutig sind. Weiterhin stehen somit die für den Dispatch notwendigen Typinformationen für den späteren Gebrauch zur Verfügung. Die genaue Definition der Funktion ϕ ist dabei von untergeordneter Bedeutung, so dass sie hier nicht angegeben wird. Abbildung 2.10 zeigt die neuen Regeln.

Das `letrec_ovld` Konstrukt kann, wie Abbildung 2.11 zeigt, auf das bekannte `letrec` abgebildet werden. Der Dispatch erfolgt dabei in $\mathcal{F}un$ durch die Funktion höherer Ordnung `ovld`. Diese ist über die verfügbaren Instanzen der jeweiligen überladenen Funktion (angegeben in runden Klammern) parametrisiert. Ein Vorkommen der `ovld` Funktion innerhalb einer Applikation wertet sich abhängig vom Typ der Argumente zur bezüglich der vorgestellten Dispatch-Regeln passenden Instanz aus. Sie bildet somit das Analogon zur Wrapper-Funktion in SaC'.

Um das `letrec_ovld` Konstrukt in das bekannte `letrec` Konstrukt zu transformieren, wird für jede überladene Funktion zusätzlich zu den Definitionen der einzelnen Instanzen eine Definition für den Funktionsbezeichner eingefügt. Diese enthält auf der rechten Seite die beschriebene `ovld` Funktion. Die einzelnen Instanzen werden

dabei über die Äquivalenz der Funktionsbezeichner unter Vernachlässigung des Indizes identifiziert. Da innerhalb der Definitionen der einzelnen Instanzen weiterhin der Funktionsbezeichner ohne Index für Funktionsanwendungen verwendet wird, ist somit sichergestellt, dass alle Funktionsanwendungen mittels der `ovld` Funktion der jeweiligen Instanz zugeordnet werden. Das Dispatch-Verfahren in $\mathcal{F}un$ entspricht damit dem in SaC'.

Mit den vorgestellten Erweiterungen kann somit SaC' vollständig in die Sprache $\mathcal{F}un$ abgebildet werden.

3. Herausforderungen

In diesem Kapitel werden die Anforderungen an ein Modulsystem, die sich aus den in Kapitel 2 vorgestellten Eigenschaften der Sprache SaC ergeben, näher untersucht. Die ersten beiden Abschnitte betrachten das Zusammenspiel zwischen getrennten Namensräumen und dem Überladen von Funktionen. In den darauf folgenden Abschnitten wird auf Herausforderungen, die aus den Code-Optimierungen des Compilationssystems resultieren, näher eingegangen.

Abschnitt 3.5 liefert eine Zusammenfassung und beschreibt die im Rahmen dieser Arbeit zu lösenden Aufgaben.

3.1. Namensräume vs. Überladung

Sowohl separate Namensräume, als auch Funktionsüberladung sind feste Bestandteile vieler moderner Programmiersprachen. Dieser Abschnitt gibt einen kurzen Überblick über beide Verfahren und erläutert ihre Vorteile aus Sicht der Softwareentwicklung. Im Anschluss werden auftretende Probleme am Beispiel diskutiert.

Unter separaten Namensräumen versteht man die Möglichkeit, Bezeichner in Gruppen zusammenzufassen. Jede dieser Gruppen bildet einen eigenen Namensraum, d.h. Bezeichner müssen innerhalb dieser Gruppe eindeutig sein, können aber in jeder Gruppe unterschiedlich definiert sein. Die einzelnen Namensräume sind somit voneinander separiert.

Um Bezeichner dennoch eindeutig adressieren zu können, wird der Namensraum dem Bezeichner vorangestellt. Als Trennzeichen zwischen Namensraum und Bezeichner dient hierbei im Rahmen dieser Arbeit das : (Doppelpunkt) Symbol. Ein solches <*Namensraum*> : <*Bezeichner*> Tupel wird auch *qualifizierter Bezeichner* genannt.

Im Vergleich zu vom Programmierer vergebenen Namenspräfixen bieten Namensräume den Vorteil, dass sie Teil der Programmiersprache sind und als solcher auch mittels geeigneter Sprachkonstrukte manipuliert werden können. So erlaubt die Angabe eines aktuellen oder lokalen Namensraums, den Namensraumpräfix für Bezeichner dieses Namensraums redundant zu machen. Kombiniert mit der Möglichkeit, Bezeichner aus mehreren Namensräumen im lokalen Namensraum sichtbar zu machen, ergibt sich eine deutlich verbesserte Lesbarkeit des Quelltextes.

Insbesondere modulares Programmdesign profitiert von separaten Namensräumen. Indem jedem Modul ein eigener Namensraum zugewiesen wird, kann effektiv verhindert werden, dass es beim Zusammenfügen der einzelnen Module zu Namenskollisionen kommt. Dies ist von besonderer Bedeutung, wenn die Module getrennt entwickelt wurden, wie es oft bei größeren Projekten oder Bibliotheken von Fremdanbietern der Fall ist.

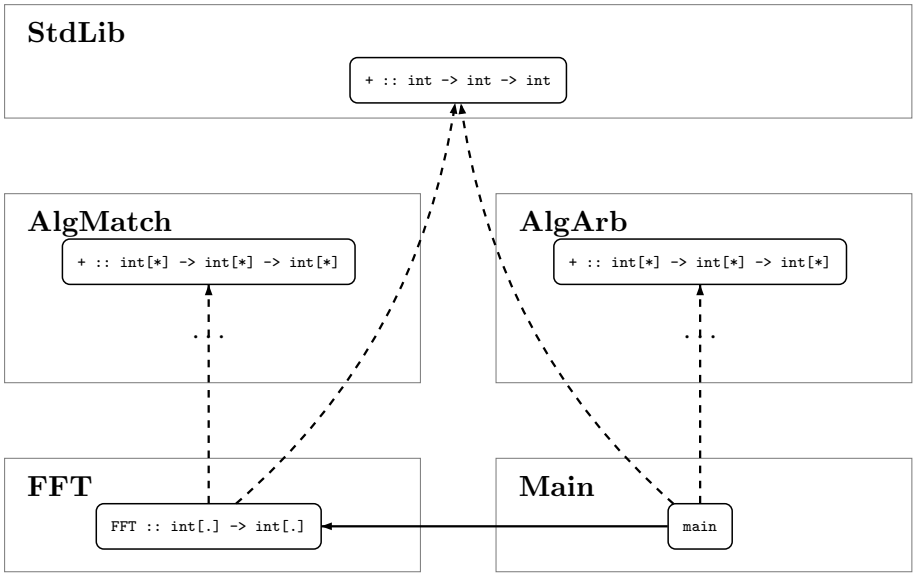


Abbildung 3.1.: Graphische Darstellung überladener Funktionen über Namensraumgrenzen

Zusammenfassend kann gesagt werden, dass separate Namensräume sowohl die Lesbarkeit des Quelltextes erhöhen, als auch die Wiederverwendbarkeit steigern. Sie erlauben die Trennung von Funktionalitäten innerhalb eines Programmes durch verteilen zusammenhängender Funktionseinheiten auf verschiedene Namensräume.

Auch bei der in Abschnitt 2.2.4 bereits vorgestellten Überladung von Funktionen steht die verbesserte Lesbarkeit des Quelltextes im Vordergrund. Insbesondere bei arithmetischen Grundfunktionen wie `+`, `-`, `*`, `/`, etc. ist schnell ersichtlich, dass einheitliche Funktionsbezeichner unabhängig vom Typ der Argumente die Lesbarkeit des Quelltextes erhöhen. Durch Überladung ist es möglich, die Funktionsbezeichner auf ihre eigentliche Bedeutung, eine Beschreibung der Semantik einer Funktion, zu reduzieren. Bedeutungsfremde Suffixe, wie z.B. Argumenttypen oder in SaC' die Ausdehnung eines Arrays, können entfallen. Dieser Vorteil setzt sich auch auf komplexere Funktionen jenseits der arithmetischen Grundfunktionen fort.

Betrachtet man die Ideen der separaten Namensräume und Funktionsüberladung näher, wird ein Widerspruch offenbar. Die grundlegende Idee der separaten Namensräume ist, verschiedene Definitionen mit dem gleichen Bezeichner zu trennen, während das Überladen von Funktionen gegensätzlich operiert. Hier werden verschiedene Definitionen mit dem gleichen Bezeichner zusammengefasst.

Obwohl beide Verfahren gegensätzlich operieren, ist es sinnvoll, beide innerhalb einer Programmiersprache zu unterstützen. In Abbildung 3.1 ist als Beispiel die Verwendung mehrerer Array-Algebren in einem Programm graphisch dargestellt. Die rechteckigen Blöcke symbolisieren die verschiedenen Namensräume. Der Name des jeweiligen Namensraumes ist links oben angegeben.

Im oberen Drittel der Abbildung ist der Namensraum **StdLib** abgebildet. In diesem sind die arithmetischen Grundfunktionen für den Datentyp **Integer** definiert. Die einzelnen Funktionsinstanzen sind dabei durch abgerundete Rechtecke dargestellt. Der jeweilige Name und Funktionstyp ist innerhalb des Rechtecks angegeben.

Im mittleren Drittel befinden sich zwei Namensräume. Auf der linken Seite ist der Namensraum **AlgMatch** abgebildet, welcher unter Verwendung der Funktionen aus der Standard-Bibliothek eine Array-Algebra auf Arrays vom Typ **Integer** definiert. Auf der rechten Seite ist der Namensraum **AlgArb** dargestellt. Dieser definiert ebenfalls eine Algebra auf Arrays von selbigem Typ.

Der hier vorgestellte Fall mehrerer Algebren innerhalb eines Programms tritt häufig bei komplexen numerischen Anwendungen auf, deren einzelne Module von verschiedenen Programmierern erstellt wurden. Er kann aber auch durch algorithmische Anforderungen entstehen. Ein anschaulicher Unterschied verschiedener Array-Algebren ist z.B. die Addition zweier nicht gleichförmiger Arrays. Eine Möglichkeit besteht darin, diese Art der Addition auszuschließen und einen Fehler zu erzeugen. Dies ist durch die Algebra im Namensraum **AlgMatch** angedeutet.

Es ist ebenso denkbar, ein Ergebnis zu errechnen, indem man eine gemeinsame Hoch- bzw. Herab-Projektion der Argumente verwendet. Ein hierfür häufig verwendetes Beispiel ist die Addition eines Skalars auf einen Vektor. In diesem Fall wird der Skalar zu einem Vektor hoch projiziert und dann die eigentliche Addition berechnet. Dieser Fall ist durch die im Namensraum **AlgArb** definierte Algebra angedeutet.

Zwei weitere Namensräume befinden sich im unteren Drittel der Abbildung 3.1. Der Namensraum **FFT** auf der linken Seite definiert eine gleichnamige Funktion, unter Verwendung der Array-Algebra aus dem Namensraum **AlgMatch**. Das eigentliche Hauptprogramm befindet sich im Namensraum **Main** auf der rechten Seite. Dieser definiert eine Funktion `main`, welche das Hauptprogramm symbolisiert. Innerhalb dieser Funktion wird sowohl die Funktion **FFT**, als auch die Array-Algebra aus dem Namensraum **AlgArb** benutzt.

Die gestrichelten Pfeile zeigen die verschiedenen Funktionsinstanzen, die in einem Aufruf einer überladenen Funktion berücksichtigt werden. Diese werden im Folgenden auch als *Dispatch-Möglichkeiten* bezeichnet. Funktionsaufrufe, welche durch den Programmierer statisch angegeben wurden, sind durch durchgezogene Pfeile angedeutet.

Aus Gründen der Übersichtlichkeit wurde jeweils nur die Addition aus den jeweiligen Algebren dargestellt. Die anderen Funktionen ergeben sich analog.

Im Beispiel aus Abbildung 3.1 finden sowohl getrennte Namensräume, als auch überladene Funktionen Verwendung. Sowohl im Namensraum **AlgMatch**, als auch im Namensraum **AlgArb** werden neue Instanzen der überladenen Funktion `+` aus der Standard-Bibliothek hinzugefügt. Allerdings sind die Instanzen beider Namensräume auf dem gleichen Typ definiert, so dass sie nicht gleichzeitig in einem Namensraum sichtbar sein können.

Im Folgenden werden zwei mögliche Präferenzregeln vorgestellt. Die *lokale Überladung* präferiert die Separation der Namensräume gegenüber der Überladung von Funktionen. Im Gegensatz dazu präferiert die *globale Überladung* die Funktionsüberladung gegenüber den separaten Namensräumen.

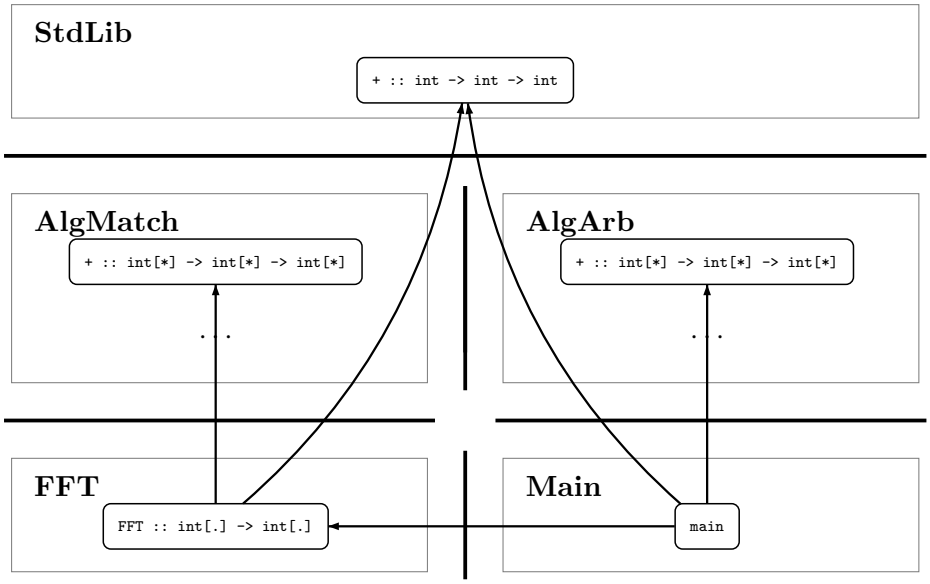


Abbildung 3.2.: Graphische Darstellung lokaler Überladung

Lokale Überladung

Mit *lokaler Überladung* wird die Idee bezeichnet, nur die Instanzen einer Funktion für eine überladene Funktion zu berücksichtigen, welche im aktuellen Namensraum definiert wurden. Aufrufe überladener Funktionen können dadurch den aktuellen Namensraum nicht verlassen. Namensraumübergreifende Überladung wird somit zugunsten einer strikten Trennung der Namensräume verhindert.

In Abbildung 3.2 ist das Beispiel aus Abbildung 3.1 unter der Annahme lokaler Überladung dargestellt. Dabei symbolisieren die dicken Linien zwischen den Namensräumen die Separierung durch Namensraumgrenzen.

Wie aus der Abbildung deutlich wird, findet in dem dargestellten Szenario keine Funktionsüberladung statt. Da alle Instanzen jeweils in einem separaten Namensraum liegen, stehen sie für die Funktionsüberladung nicht zur Verfügung.

Statt dessen wurden die Funktionsaufrufe überladener Funktionen durch statisch codierte Funktionsaufrufe ersetzt. In der Abbildung ist dies durch Ersetzen der Dispatch-Möglichkeiten durch Funktionsaufrufe dargestellt.

Für die Lesbarkeit des Quelltextes hat diese subtile Änderung drastische Auswirkungen. Für jeden Aufruf einer der arithmetischen Grundfunktionen muss nun je nach Argument-Typ der passende qualifizierte Bezeichner angegeben werden. Für eine Addition im Namensraum **FFT** ist somit für Integer Werte die Funktion `StdLib:+` zu verwenden, während eine Addition von Integer Arrays die Funktion `AlgMatch:+` erfordert.

Damit sind die Vorteile der Funktionsüberladung zugunsten separater Namensräume verloren. Das in Abbildung 3.1 skizzierte Szenario lässt sich mittels lokaler

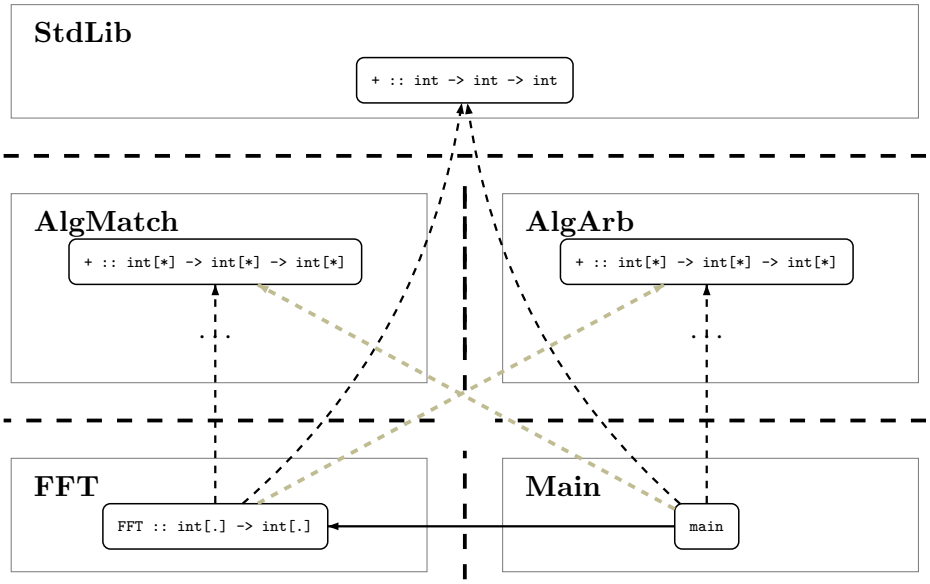


Abbildung 3.3.: Graphische Darstellung globaler Überladung

Überladung nicht hinreichend modellieren.

Globale Überladung

Im Gegensatz zur lokalen Überladung werden bei der *globalen Überladung* alle im globalen Kontext des Programms vorhandenen Instanzen einer Funktion für eine überladene Funktion berücksichtigt. Die Separation der Namensräume wird dadurch zugunsten namensraumübergreifender Funktionsüberladung abgeschwächt. Weiterhin werden alle überladenen Funktionen global, d.h. in jedem Namensraum sichtbar.

Die Möglichkeit der globalen Überladung ist in Abbildung 3.3 dargestellt. Die Namensraumgrenzen wurden hier gestrichelt dargestellt, um anzudeuten, dass sie für die Überladung von Funktionen durchlässig sind.

Im Vergleich zum gewünschten Szenario aus Abbildung 3.1 fallen zwei neue Dispatch-Möglichkeiten auf. Diese sind in grau hervorgehoben. Die Funktion `+` im Namensraum **FFT** beinhaltet nun zusätzlich die Instanz aus dem Namensraum **AlgArb**. Des weiteren ist im Namensraum **Main** die Instanz aus dem Namensraum **AlgMatch** zur überladenen Funktion `+` hinzugekommen.

Somit beinhalten nun beide Namensräume jeweils zwei Definitionen der Funktion `+` auf Argumenten vom Typ `int[*]`. Dies führt zu einem Konflikt: es ist nicht entscheidbar, welche der beiden Instanzen für passende Argumente zu wählen ist.

Das in Abbildung 3.1 skizzierte Szenario lässt sich damit auch durch globale Überladung nicht modellieren.

Im Kontext der Sprache SaC führt globale Überladung sogar zum Verlust separater Namensräume. Da in SaC potenziell jede Funktion überladen werden kann, ist jede Funktion insbesondere eine Instanz einer überladenen Funktion. Daraus folgt, dass

jede Funktion global sichtbar sein muss[1].

Zusammenfassend fällt auf, dass mittels beider Verfahren – lokaler und globaler Überladung – die vorgestellte Situation nicht abgebildet werden kann. Ein Modulsystem mit Unterstützung für separate Namensräume und Funktionsüberladung benötigt mächtigere und granularere Möglichkeiten, um zwischen strikter Trennung der Namensräume auf der einen Seite und Überladung von Funktionen auf der anderen Seite zu vermitteln.

3.2. Überladung und Rekursion über Namensraumgrenzen

Wie der vorige Abschnitt gezeigt hat, ist die Menge der Instanzen einer überladenen Funktion, die für einen Funktionsaufruf berücksichtigt werden muss, vom Namensraum der aufrufenden Funktion abhängig. Es können sich jedoch durch Rekursion weitere Abhängigkeiten ergeben. Um dies näher zu untersuchen betrachtet der folgende Abschnitt den Problemraum der Rekursion über Namensraumgrenzen unter dem besonderen Aspekt der in SaC verfügbaren Funktionsüberladung.

Wie in Kapitel 2 beschrieben, erlaubt SaC Funktionsüberladungen anhand der Ausdehnung und Dimensionalität der Argumente. Zusammen mit der vorgestellten Subtyphierarchie und der Garantie des best möglichen Dispatches erlaubt dies, rekursive Funktionen über die Dimensionalität und Ausdehnung eines Arrays mittels Funktionsüberladung zu spezifizieren.

Abbildung 3.4 zeigt das Beispiel der schnellen Fourier Transformation (FFT) als SaC Pseudocode. Die hier gezeigte Implementierung der schnellen Fourier Transformation entspricht dem Danielson-Lanczos Algorithmus [PFTV96]. Die in Zeile 1 ff. definierte generische Instanz für Vektoren komplexer Zahlen zerlegt den als Argument übergebenen Vektor `vect` in zwei Vektoren. `even( vect)` liefert alle Elemente des Vektors `vect` mit geradem Index, während `odd( vect)` jene mit ungeradem Index liefert. Die Berechnung wird dann rekursiv auf diesen Teilvektoren fortgesetzt.

Die Rekursion terminiert durch die in Zeile 12 ff. definierte Instanz der Funktion `FFT` für Vektoren der Länge 2. Durch die Dekomposition des Argumentvektors halbiert sich dessen Länge mit jedem rekursiven Aufruf. Somit wird nach endlich vielen Schritten statt der rekursiven Instanz für Vektoren beliebiger Länge[2], die nicht rekursive Instanz ausgeführt. Die Abbruchbedingung ist dabei im Typ der Argumente kodiert. Eine vollständige Beschreibung des FFT Algorithmus in SaC gibt [GS03].

Im angegebenen Beispiel sind beide Instanzen der Funktion `FFT` im gleichen Namensraum definiert. Somit stehen unabhängig von der gewählten Überladungsstrategie beide Instanzen für die überladene Funktion zur Verfügung. Die Situation ändert sich jedoch, wenn beide Instanzen auf verschiedene Namensräume verteilt sind.

[1] Auch eine überladene Funktion mit nur einer Instanz muss global sichtbar sein. Ansonsten würde das Hinzufügen einer Instanz die Sichtbarkeit der vorhandenen Instanz ändern – ein ungewünschter Seiteneffekt.

[2] Obwohl die Instanz für Vektoren beliebiger Länge deklariert ist, ist der FFT Algorithmus nach Danielson-Lanczos nur für Eingabevektoren der Länge 2^n definiert.

```
1 complex[.] FFT( complex[.] vect, complex[.] rofu)
  {
3   fft_even = FFT( even( vect), even( rofu));
    fft_odd = FFT( odd( vect), odd( rofu));
5
    left = fft_even + fft_odd * rofu;
7   right = fft_even - fft_odd * rofu;

9   return( left ++ right);
  }
11
  complex[2] FFT( complex[2] vect, complex[2] rofu)
13 {
    return( [ vect[0] + vect[1], vect[0] - vect[1]]);
15 }
```

Abbildung 3.4.: SAC Pseudocode der schnellen Fourier Transformation

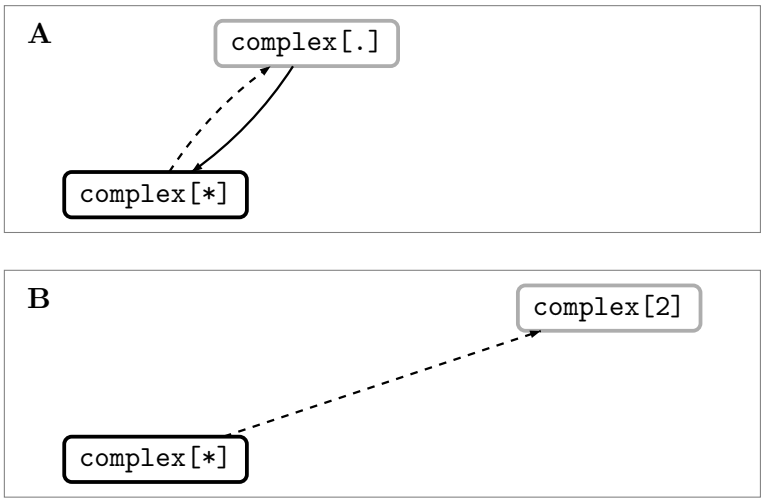


Abbildung 3.5.: Verteilte Definition der Funktion `FFT` (lokale Überladung)

Im vorigen Abschnitt wurde bereits gezeigt, dass globale Überladung für Sprachen, die massiven Gebrauch von Funktionsüberladung machen, nicht praktikabel ist. Auch mittels lokaler Überladung lässt sich der gegebene Algorithmus nicht auf zwei Namensräume verteilen. Abbildung 3.5 zeigt eine graphische Darstellung des Algorithmus unter der Annahme lokaler Überladung.

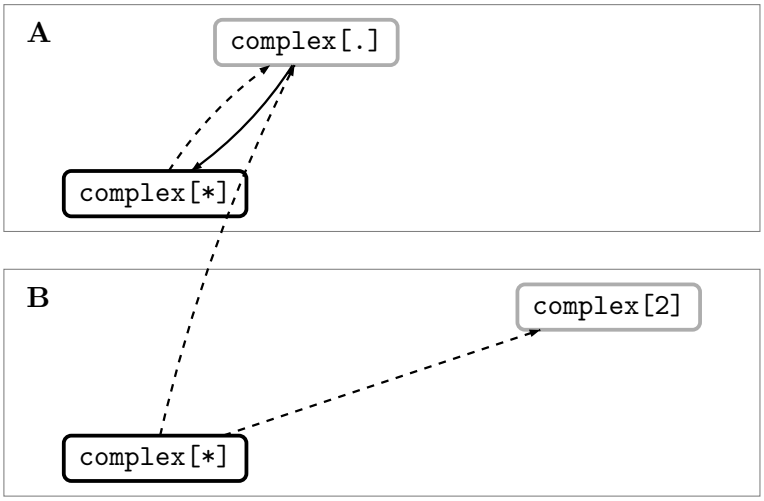


Abbildung 3.6.: Erweiterter Dispatch der Funktion `B:FFT`

Die zwei Namensräume sind durch die rechteckigen Blöcke dargestellt. Wie in vorangehenden Beispielen ist dabei der Name des jeweiligen Namensraumes links oben in den Blöcken angegeben.

Die beiden Instanzen der Funktion `FFT` sind auf diese Namensräume verteilt. In der Abbildung sind die Instanzen durch abgerundete graue Kästen dargestellt. Innerhalb des Kastens ist der jeweilige Argumenttyp angegeben. Im Namensraum **A** befindet sich die allgemeine Instanz für Vektoren beliebiger Länge. Die terminierende Instanz für Vektoren der Länge 2 ist im Namensraum **B** dargestellt.

Die jeweiligen überladenen Funktionen sind durch schwarze abgerundete Kästen dargestellt. Im Namensraum **A** befindet sich die überladene Funktion `A:FFT`, welche nur die allgemeine Instanz umfasst. Dies ist durch den gestrichelten Pfeil, welcher die Dispatch-Möglichkeiten darstellt, angedeutet. Der durchgezogene Pfeil von der Instanz zur überladenen Funktion veranschaulicht den rekursiven Charakter dieser Instanz. Sie ruft zur weiteren Berechnung der Fourier Transformation erneut die überladene Funktion auf. Die überladene Funktion `B:FFT` hingegen umfasst nur die terminierende Instanz für Vektoren der Länge 2.

Beide in Abbildung 3.5 dargestellten Funktionen sind keine vollständige Implementierung des FFT Algorithmus. Da für jede Dekomposition erneut die allgemeine Instanz aufgerufen wird, terminiert die Funktion `A:FFT` nicht. Dem gegenüber ist die Funktion `B:FFT` nur für Vektoren der Länge 2 definiert.

Eine nahe liegende Lösung wäre, die Instanz aus Namensraum **A** in die Dispatch-Möglichkeiten der überladenen Funktion `B:FFT` aufzunehmen. Dieses Szenario ist in Abbildung 3.6 dargestellt. Hier teilen sich beide überladenen Funktionen die allgemeine Instanz aus dem Namensraum **A**.

Somit ist die Funktion `B:FFT` nun auch für Vektoren beliebiger Länge definiert.

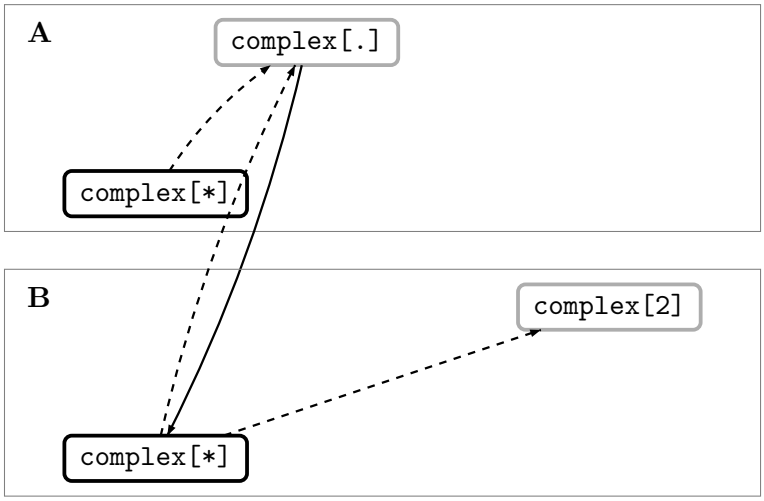


Abbildung 3.7.: Angepasster rekursiver Aufruf

Im Falle eines Vektors der Länge größer 2 wird die Instanz aus dem Namensraum **A** gewählt. Für Vektoren der Länge 2 kommt die Instanz aus dem Namensraum **B** zum Einsatz.

Eine genauere Betrachtung offenbart jedoch, dass die Funktion `B:FFT` für Vektoren der Länge größer 2 nicht terminiert. Obwohl die terminierende Instanz beim ursprünglichen Aufruf der Funktion mit in Betracht gezogen wird, wird für rekursive Aufrufe weiterhin die Funktion `A:FFT` verwendet. Somit wird ab dem ersten Rekursionsschritt die Instanz aus Namensraum **B** nicht mehr berücksichtigt.

Offensichtlich muss also die allgemeine Instanz aus dem Namensraum **A** die Funktion `B:FFT` verwenden, um eine korrekte Terminierung sicher zu stellen. Dies ist in Abbildung 3.7 dargestellt. Der rekursive Aufruf der Instanz des Namensraums **A** verwendet hier die überladene Funktion `B:FFT`, dargestellt durch den gestrichelten Pfeil. Somit werden nun für rekursive Aufrufe in Folge eines Aufrufs der Funktion `B:FFT` immer beide Instanzen berücksichtigt.

Dadurch kommt es jedoch zu einem unerwünschten Nebeneffekt für die Funktion `A:FFT`. Obwohl in **A** nur die allgemeine Instanz definiert ist, wird für rekursive Aufrufe auch die Instanz aus dem Namensraum **B** berücksichtigt. Die Funktion `A:FFT` terminiert somit für Vektoren, deren Länge größer als 2 ist. Für Vektoren der Länge 2 ist eine Termination jedoch nicht sichergestellt.

Allgemein bedeutet dies, dass durch Hinzufügen eines neuen Namensraums die Semantik einer in einem anderen Namensraum definierten Funktion verändert werden kann. Dies steht in direktem Widerspruch zum Ziel der separaten Namensräume. Diese wurden eingeführt, um eben solche Abhängigkeiten und Einflüsse zu verhindern. Insbesondere bei großen Softwareprojekten können derartige impliziten Abhängigkeiten zu unerwartetem Verhalten der Gesamtapplikation führen, ohne dass die einzelnen

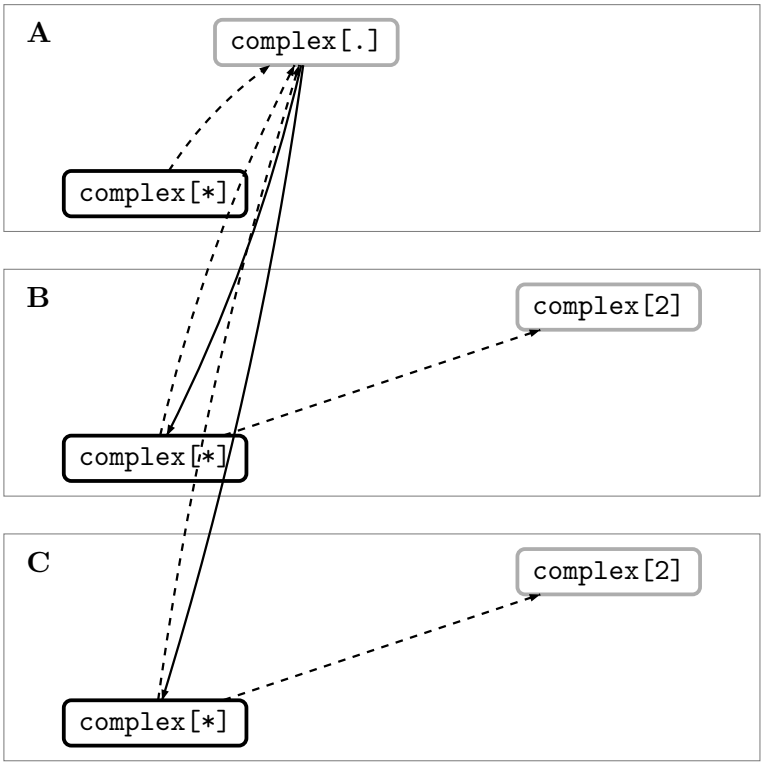


Abbildung 3.8.: Beispiel der Fourier Transformation mit 3 Namensräumen

Module als fehlerhaft erkannt werden können.

Desweiteren skaliert die vorgestellte Lösung schlecht. Fügt man einen dritten Namensraum **C** hinzu, der ebenfalls eine terminierende Instanz der Funktion `FFT` definiert, kommt es zu einem Konflikt. Abbildung 3.8 veranschaulicht diese Situation.

Um die jeweils gültigen Instanzen zu benutzen, muss der rekursive Aufruf in der allgemeinen Instanz aus Namensraum **A** den Aufrufkontext berücksichtigen. Wurde die Berechnung der Fourier Transformation durch einen Aufruf der Funktion `B:FFT` initiiert, so müssen auch die Instanzen dieser überladenen Funktion Verwendung finden. Startet die Berechnung hingegen mit einem Aufruf der Funktion `C:FFT`, so müssen im rekursiven Aufruf deren Instanzen verwendet werden. Dies lässt sich mittels eines statischen Funktionsaufrufs nicht erreichen. In Abbildung 3.8 sind daher zwei Aufrufe dargestellt.

Um Rekursion durch Funktionsüberladung im Rahmen eines Namensraumkonzeptes zu berücksichtigen, müssen also Methoden entwickelt werden, die eine Berücksichtigung des Aufrufkontextes ermöglichen. Des weiteren gilt es implizite Abhängigkeiten, wie sie durch die vorgestellte Lösung erzeugt werden, zu verhindern.

3.3. Optimierungen und Separate Compilation

Neben Eigenschaften der Sprache SaC stellt auch das compilierende System besondere Anforderungen an ein Modulsystem. Besonders hervorzuheben sind hierbei die Optimierungen des SaC Compilers `sac2c`. Wie bereits in Kapitel 2 erwähnt, wurde bei der Entwicklung der Sprache SaC ein besonderes Augenmerk auf Laufzeiteffizienz gelegt. Diese wird durch eine Vielzahl von Optimierungen erreicht. Dieser Abschnitt stellt das Funktionsinlining als eine der Optimierungen mit besonderen Anforderungen an das Modulsystem vor.

Funktionsinlining

Unter *Funktionsinlining* versteht man das Ersetzen von Funktionsaufrufen durch die im Rumpf der Funktionsdefinition angegebene Berechnungsvorschrift. Abbildung 3.9 zeigt ein einfaches Beispielprogramm in SaC.

In Zeile 1 ff. wird eine Funktion `Sqr` definiert, welche das Quadrat eines ganzzahligen Wertes berechnet. Die in Zeile 6 ff. definierte Funktion `main` berechnet unter Verwendung der Funktion `Sqr` das Quadrat der Zahl 2.

Im vorgestellten Beispiel sind alle Argumente konstant. Somit könnte durch partielle Auswertung das Resultat der Funktion `main` leicht berechnet werden. Allerdings ist die Auswertung konstanter Ausdrücke – wie die meisten Optimierungen in SaC – auf den Kontext des Funktionsrumpfes beschränkt, um unerwünschte Seiteneffekte zu vermeiden. Im Kontext der Funktion `Sqr` ist das Argument `a` jedoch nicht bekannt. Obiges Beispiel könnte folglich nicht weiter vereinfacht werden.

Durch Funktionsinlining lässt sich diese Beschränkung umgehen. Abbildung 3.10 zeigt das Ergebnis für das Beispielprogramm. Der Aufruf der Funktion `Sqr` in Zeile 8 wurde durch die Berechnungsvorschrift `a * a` für $a = 2$ ersetzt. Da somit die vollständige Berechnung im Rumpf der Funktion `main` stattfindet, sind weitere Optimierungen anwendbar. Insbesondere kann nun durch partielle Auswertung das Resultat der Funktion statisch berechnet werden.

Auch unabhängig von weiteren Optimierungen verbessert das Funktionsinlining die Laufzeit eines Programms. Sie eliminiert den durch Funktionsaufrufe erzeugten

```
1 int Sqr( int a)
  {
3   return( a * a);
  }
5
  int main()
7 {
    return( Sqr( 2));
9 }
```

Abbildung 3.9.: Code Beispiel für Funktionsinlining

```
1 int Sqr( int a)
  {
3    return( a * a);
  }
5
  int main()
7 {
    return( 2 * 2);
9 }
```

Abbildung 3.10.: Code Beispiel nach dem Funktionsinlining

Overhead. Dies geht allerdings zu Lasten der Größe des erzeugten Programmes. Eine genauere Diskussion des Funktionsinlining findet sich in [Sch77].

Separate Compilation

Die Möglichkeit der separaten Compilation ist in den meisten Compilern moderner Programmiersprachen vorgesehen. In objektorientierten Sprachen wie Java [GJS00] oder C# [HWG03] werden dabei üblicherweise einzelne Klassen getrennt übersetzt. Compiler für funktionale Sprachen wie Clean [PvE01], ML [MTH90, AM94] und Haskell [Jon03] bieten separate Compilation auf Modulebene. Sogar für relativ hardwarenahe Sprachen wie C sind rudimentäre Möglichkeiten, einzelne Code Bestandteile separat zu compilieren, vorhanden.

Gerade für große Projekte ist die separate Übersetzung einzelner Module unerlässlich. Sie erlaubt es, Übersetzungszeiten bei lokalen Änderungen des Quelltextes kurz zu halten und bietet damit einen Produktivitätsgewinn bei der Entwicklung und Wartung von Anwendungen und Bibliotheken.

Auch für die Distribution von Funktionsbibliotheken ist die separate Compilation von großer Bedeutung. Oftmals ist es aus lizenzrechtlichen Gründen oder zum Schutz des geistigen Eigentums nicht möglich, die Bibliotheken im Quelltext auszuliefern. Hier bieten separat compilierte Bibliotheken einen Ausweg. Aus dem Compilat ist eine Rekonstruktion des Quelltextes nur schwer möglich und in vielen Staaten rechtlich verboten. Somit bietet die separate Compilation einen ausreichenden Schutz.

In dieser Eigenschaft liegt die Herausforderung für ein Modulsystem mit separater Compilation im Kontext von Optimierungen wie dem hier vorgestellten Funktionsinlining. Die separate Compilation ist *definitions-zentrisch*, d.h. sie erfolgt an der Stelle der Funktionsdefinition. Die Module werden unabhängig von ihrer Anwendung übersetzt. Dies ist insbesondere bei Bibliotheken, die als Compilat vertrieben werden, der Fall.

Im Gegensatz dazu sind Optimierungen wie das Funktionsinlining *anwendungszentrisch*, d.h. sie erfolgen an der Stelle der Funktionsanwendung. In obigem Beispiel wird das Funktionsinlining erst durch die Anwendung der Funktion `Sqr` innerhalb der Funktion `main` angestoßen. Entgegen dem hier gegebenen Beispiel müssen die beiden

Funktionen jedoch nicht im gleichen Modul liegen.

Ist die Funktionsdefinition durch separate Compilation lediglich in übersetzter Form verfügbar, kann die Berechnungsvorschrift der Funktion nur noch schwer ermittelt werden. Insbesondere weitere Optimierungen auf hochsprachlichem Niveau, wie die partielle Auswertung in obigem Beispiel, sind nur noch schwer umsetzbar. Aufgrund dessen ist es für viele Optimierungen unerlässlich, den Quelltext eines Moduls verfügbar zu haben.

Speziell das Funktionsinlining führt jedoch durch die Verwendung des Quelltextes bereits compilierter Module zu einer mehrfachen Compilation des gleichen Quelltextes. Dies widerspricht der Idee der separaten Compilation und erhöht die benötigte Übersetzungszeit eines Programmes. Dieser Effekt kann abgemildert werden, indem statt des reinen Quelltextes ebenfalls die Ergebnisse der Codeanalyse aus der Compilation des Moduls gespeichert werden. Ein großer Teil der Übersetzung kann dann bei weiterer Verwendung des Quelltextes vermieden werden.

Für ein Modulsystem ist es also wünschenswert, sowohl separate Compilation zu unterstützen, als auch den Quelltext verfügbar zu halten. Dieser muss jedoch in einer Form gespeichert sein, die einen ausreichenden Schutz des geistigen Eigentums sicher stellt und eine ausschließliche Distribution der übersetzten Bibliothek erlaubt.

3.4. Funktionsspezialisierung

Insbesondere für Sprachen mit hohem Abstraktionsniveau ist die Funktionsspezialisierung ein wichtiges Verfahren zur Verbesserung der Laufzeiteffizienz. Unter *Funktionsspezialisierung* versteht man die automatische Generierung von auf Typ-Ebene partiell ausgewerteten Funktionsinstanzen aus einer allgemeineren Funktionsdefinition. Eine derart erzeugte Instanz wird auch als *Spezialisierung* einer Funktion bezeichnet.

Abbildung 3.11 veranschaulicht die Funktionsspezialisierung am Beispiel des in Abschnitt 3.2 vorgestellten FFT Algorithmus. Zusätzlich zu den beiden im Quelltext spezifizierten Instanzen für Vektoren beliebiger Länge und Vektoren der Länge 2 wurden zwei neue Spezialisierungen erzeugt. Eine ist für Vektoren der Länge 8, die

```
1 /* specified instance */
  complex[.] FFT( complex[.] vect, complex[.] rofu)
3
  /* specialized instances */
5 complex[8] FFT( complex[8] vect, complex[8] rofu)
  complex[4] FFT( complex[4] vect, complex[4] rofu)
7
  /* specified instance */
9 complex[2] FFT( complex[2] vect, complex[.] rofu)
```

Abbildung 3.11.: Spezialisierung am Beispiel der Funktion `FFT`

andere für solche der Länge 4 definiert. Beide sind aus der Instanz für Vektoren beliebiger Länge hervorgegangen und somit zu dieser innerhalb ihres Definitionsbereiches semantisch äquivalent. Die angegebenen generierten Instanzen sind nur zwei Beispiele der möglichen Spezialisierungen.

Allgemein sind spezialisierte Instanzen eine Herab-Projektion einer generischeren, vom Programmierer spezifizierten Funktionsdefinition. Bedingt durch die spezifischeren Typinformationen einer Spezialisierung liefern die hochsprachlichen Optimierungen für diese oft bessere Resultate als für die generische Instanz. Daraus resultiert eine verbesserte Laufzeiteffizienz der gesamten Anwendung.

Analog zu den im vorigen Abschnitt beschriebenen Optimierungen ist für die Funktionsspezialisierung die Verfügbarkeit des Quelltextes Voraussetzung. Während der Übersetzung einer Funktionsdefinition kann nicht entschieden werden, welche Spezialisierungen im restlichen Programm benötigt werden. Da es, bedingt durch die nicht endliche Subtyphierarchie in SaC, unendlich viele Möglichkeiten der Spezialisierung gibt, ist eine Erzeugung aller Spezialisierungen auf Vorrat ebenfalls nicht möglich.

Die Spezialisierung einer Funktion wird somit immer erst durch eine Funktionsanwendung ausgelöst. Liegt die Funktionsanwendung in einem anderen Modul als die Funktionsdefinition, treten die gleichen Probleme wie bei den im vorigen Abschnitt beschriebenen Optimierungen auf. Im Gegensatz zu den durch partielle Auswertung, etc. erzeugten Funktionsinstanzen sind die durch Spezialisierung erzeugten Instanzen jedoch innerhalb ihres Definitionsbereiches semantisch zur Ursprungsinstanz äquivalent. Sie sind somit nicht vom Kontext der Funktionsanwendung abhängig, d.h. sie sind global anwendbar.

Um die mit der Funktionsspezialisierung einher gehenden Vergrößerung des Quelltextes zu minimieren, ist es daher wünschenswert, eine spezifische Spezialisierung einer Funktionsinstanz nur einmal vorzunehmen. Dies bedeutet jedoch, dass die Menge der in einem Modul verfügbaren Instanzen einer Funktion nachträglich, bestenfalls ohne erneute Compilation des Moduls, erweitert werden können muss.

3.5. Zusammenfassung

In den vorangehenden Abschnitten wurden verschiedene Eigenschaften der Sprache SaC und des compilierenden Systems auf ihre Implikationen für ein Modulsystem untersucht.

Um sowohl separierte Namensräume, als auch Funktionsüberladung über Namensraumgrenzen hinweg zu ermöglichen, müssen geeignete Sprachelemente für die Sprache SaC entwickelt werden. Es wurde dargelegt, dass die beiden offensichtlichen Lösungen, die lokale bzw. globale Überladung, hierfür nicht ausreichen. Jenseits allgemeiner Präferenzregeln muss dem Programmierer eine granulare Steuerungsmöglichkeit gegeben werden, die es erlaubt, die Menge der sichtbaren Instanzen einer Funktion in einem Namensraum anzugeben.

Eine genaue Betrachtung der Rekursion über Namensraumgrenzen im Zusammenhang mit Funktionsüberladung hat gezeigt, das die Menge der für einen Funktionsauf-

ruf in Betracht zu ziehenden Instanzen einer Funktion kontextabhängig ist. Sie hängt nicht nur vom Namensraum der aufrufenden Funktion ab, sondern kann auch vom Namensraum der Funktion abhängen, welche die Rekursion initiiert hat. Dafür sind einerseits geeignete Sprachkonstrukte zu entwickeln, die erlauben, einen solchen Kontext anzugeben, ohne dabei einen der Vorteile der Überladung – die verbesserte Lesbarkeit des Quelltextes – zu opfern. Des weiteren ist der Überladungs-Mechanismus um einen Aufrufkontext zu erweitern.

Für die separate Compilation haben sich durch die in SaC zum Einsatz kommenden Optimierungen zwei weitere Anforderungen ergeben.

Modulübergreifende Optimierungen bei separater Compilation machen die Verfügbarkeit des Quelltextes unabdingbar. Dieser sollte, um die Vorteile der separaten Compilation einzelner Module bezüglich der benötigten Übersetzungszeit zu erhalten, bereits ermittelte Analyseergebnisse der Compilation beinhalten. Ferner ist der Quelltext derart zu speichern, dass der Schutz geistigen Eigentums bei einer binären Distribution der Module sicher gestellt ist.

Zudem wäre es für die Funktionsspezialisierung wünschenswert, bereits compilierte Module um weitere Instanzen erweitern zu können, ohne das gesamte Modul neu übersetzen zu müssen.

4. Vorhandene Ansätze

Sowohl separate Namensräume als auch die Überladung von Funktionen haben sich für moderne Programmiersprachen bereits etabliert. Im Rahmen dieses Kapitels werden die in anderen Sprachen gewählten Ansätze zur Kombination der Funktionsüberladung mit separaten Namensräumen diskutiert. Als Beispiele werden dabei die ebenfalls funktionalen Sprachen Haskell und Clean gewählt, da sie ein mit dem von SaC vergleichbares Konzept für die Überladung von Funktionen bieten.

In einem weiteren Abschnitt wird das Konzept der sogenannten *Mixin Module* als Beispiel für Rekursion über Modulgrenzen vorgestellt und die Anwendbarkeit im Kontext der Sprache SaC untersucht.

4.1. Überladung und Namensräume

Im Gegensatz zu SaC ist in Haskell und Clean nicht jede Funktion potenziell überladbar. Statt dessen verwenden beide Sprachen das Konzept der Typklassen [WB89] um Funktionsüberladung in die jeweilige Sprache zu integrieren.

Mit *Typklasse* bezeichnet man dabei ein Tupel bestehend aus einer Menge von Typen und einer Menge von darauf definierten Operationen. Für jeden Typ innerhalb einer Typklasse müssen Instanzen für alle Operationen definiert sein. Eine Typklasse ist somit eine Sammlung von überladenen Funktionen. Eine Instanz einer Typklasse definiert diese Funktionen für einen gegebenen Typ.

Des weiteren können Typklassen auch als Argumenttyp von Funktionsdefinitionen verwendet werden. Eine solche Funktion ist damit auf allen Typen dieser Typklasse definiert.

Obwohl sowohl Haskell als auch Clean Funktionsüberladung durch Typklassen unterstützen, verwenden sie gänzlich verschiedene Ansätze für separate Namensräume. Diese werden im Folgenden näher untersucht.

Namensräume in Haskell

Haskell kombiniert das Konzept der Typklassen mit einem Modulsystem, welches separate Namensräume unterstützt. Dabei bildet jedes Modul seinen eigenen Namensraum. Mittels der `import` Anweisung können einzelne Funktionen, Instanzen oder Typen auch in anderen Namensräumen sichtbar gemacht werden. Eine formale Beschreibung des Modulsystems von Haskell gibt [DJH02].

Die separaten Namensräume beziehen sich in Haskell auch auf Instanzen von Typklassen[1]. Innerhalb eines Namensraumes sind nur solche Instanzen sichtbar, wel-

[1] Dem Autor ist keine Beschreibung des Modulsystems von Haskell bezüglich der Funktionsüberladung bekannt. Alle hier gemachten Aussagen beziehen sich deshalb auf Er-

che lokal definiert wurden oder durch eine `import` Anweisung explizit als sichtbar deklariert wurden. Die Anwendung überladener Funktionen auf Argumente, für welche keine Instanz definiert wurde oder sichtbar ist, führt zu einem Übersetzungs- bzw Laufzeitfehler (im Falle eines interpretierenden Systems). Allerdings ist hierbei immer die Menge der Instanzen, die im Namensraum des aufrufenden Kontextes sichtbar ist, maßgeblich. Wird eine Funktion aus einem anderen Modul aufgerufen, in dem zusätzliche Instanzen einer überladenen Funktion definiert sind, so werden diese auch innerhalb der aufgerufenen Funktion für den Dispatch berücksichtigt. Ein Beispiel hierfür ist in Anhang A.1.1 angegeben. Dadurch ist es möglich, Instanzen einer überladenen Funktion in einen anderen Namensraum zu tragen, obwohl diese dort nicht als sichtbar deklariert sind. Insbesondere im Falle überlappender Instanzen kann so die Semantik einer Funktion nachträglich beeinflusst werden.

Weiter ist es nicht möglich, im Kontext eines Programmes mehrere Definitionen einer überladenen Funktion zu spezifizieren. Auch wenn in jedem Namensraum nur jeweils eine Instanz einer Typklasse für einen gegebenen Typ definiert bzw. als sichtbar deklariert ist, kommt es zu einem Übersetzungsfehler bzw. unerwartetem Laufzeitverhalten.

Der Haskell-Interpreter `HUGS` bricht mit dem Hinweis ab, dass mehrere Instanzen für den gleichen Typ definiert sind. Beim Haskell-Compiler `ghc` wird jeweils nur eine Instanz berücksichtigt, unabhängig davon welche in dem gegebenen Namensraum sichtbar ist. Beispiele, um das Verhalten zu verifizieren sind in Anhang A.1.2 angegeben.

Aus den Ergebnissen mit `HUGS` und `ghc` kann gefolgert werden, dass intern das Prinzip der globalen Überladung Verwendung findet. Die Sichtbarkeit der einzelnen Instanzen kann zwar angegeben werden und wird auch überprüft, letztendlich sind dennoch immer alle Instanzen einer Typklasse global sichtbar.

Wie in Abschnitt 3.1 gezeigt wurde, reicht das Konzept der globalen Überladung jedoch nicht aus, um die im Rahmen dieser Arbeit aufgestellten Anforderungen zu erfüllen. Das Beispiel aus Abschnitt 3.1 lässt sich in Haskell nicht umsetzen.

Namensräume in Clean

Die Sprache Clean unterstützt keine separaten Namensräume im hier vorgestellten Sinne. Statt dessen werden sogenannte Scopes verwendet, um verschiedene Teile eines Programms voneinander zu trennen.

Unter *Scope* wird dabei ein eindeutig definierter Sichtbarkeitsbereich verstanden. Dieser wird durch eine Schnittstellendefinition für jedes Modul angegeben. Nur innerhalb dieser Schnittstellendefinition angegebene Funktionen, Instanzen und Typen sind auch nach außen sichtbar. Ein solcher Sichtbarkeitsbereich kann sich auch über mehrere Module erstrecken. Allerdings ist es nicht möglich, eine Funktion aus einem anderen Scope zu referenzieren. Die verschiedenen Sichtbarkeitsbereiche sind strikt voneinander getrennt.

fahrungen, die mittels zweier Implementierungen gemacht wurden: `ghc` in der Version 6.2.2 (*http://www.haskell.org/ghc/*) und `HUGS` in der Version vom November 2003 (*http://www.haskell.org/hugs/*).

Die Definition eines Scope erfolgt auf Funktions-, Typ- und Instanzebene. Allerdings muss sie gewisse Abhängigkeiten erfüllen. So müssen innerhalb einer Schnittstellendefinition alle von enthaltenen Funktionen verwendeten Typklassen ebenfalls enthalten sein. Somit ist immer sicher gestellt, dass alle von einer Funktionen verwendeten Typklassen auch im Scope der Funktion sichtbar sind. Details zu Scopes in Clean gibt [PvE01].

Clean verwendet somit das Prinzip der lokalen Überladung bezogen auf die definierten Sichtbarkeitsbereiche. Da diese sich jedoch über mehrere Module erstrecken können, fallen die in Abschnitt 3.1 vorgestellten Einschränkungen dieser Herangehensweise nicht so stark ins Gewicht wie bei einer namensraumbasierten Sprache. Da getrennte Namensräume jedoch eine der an das zu spezifizierende Modulsystem gestellten Anforderungen sind, schließt sich eine Verwendung dieser Lösung aus.

4.2. Rekursion über Modulgrenzen

Das vorgestellte Problem der auf Funktionsüberladung basierenden Rekursion über Modulgrenzen kann in Haskell und Clean nicht nachgebildet werden. Beide Sprachen verbieten sich überlappende Instanzen überladener Funktionen [PJM97]. Da die Instanzen somit in jedem Fall disjunkt sind, bleibt eine einmal ausgewählte Instanz auch durch Hinzufügen neuer Instanzen gültig. Dadurch ist eine, für das gegebene Szenario notwendige, nachträgliche Erweiterung der für den Dispatch zu berücksichtigen Instanzen nicht möglich.

Ein allgemeiner Ansatz zur Rekursion über Modulgrenzen wurde in Form der Mixin Module [DS96, HL02] für die Sprache ML vorgeschlagen. Obwohl die Sprache ML keine Funktionsüberladung bietet, kann das in Abschnitt 3.2 vorgestellte Beispiel unter Verwendung von Mixin Modulen in ML modelliert werden. Dazu wird das Konzept der Mixin Module im Folgenden näher vorgestellt.

Unter *Mixin Modulen* versteht man unvollständige Module, die zu einer vollständigen Modulspezifikation kombiniert werden. Abbildung 4.1 zeigt ein Beispiel zweier Mixin Module in einer an ML angelehnten Syntax. Das Modul `Even` in Zeile 1-4

```
1 mixin Even = mix
    ? val odd : int -> bool
3   let even = λx. x = 0 or (odd x-1)
  end
5
  mixin Odd = mix
7   ? val even : int -> bool
    let odd = λx. x > 0 and (even x-1)
9 end
```

Abbildung 4.1.: Beispiel zweier Mixin Module

```
1 mixin Nat = mix
    let even = λx. x = 0 or (odd x-1)
3   let odd = λx. x > 0 and (even x-1)
  end
```

Abbildung 4.2.: Kombination beider Mixin Module

deklariert zwei Funktionen, `odd` und `even`. Dabei ist für die Funktion `odd` nur eine Typsignatur angegeben (angedeutet durch den `? val` Ausdruck). Die Definition der Funktion `even` in Zeile 3 enthält dennoch einen Aufruf der Funktion `odd`.

Im zweiten Modul `Odd` in Zeile 6-9 werden ebenfalls die beiden Funktionen deklariert. Hier ist jedoch für die Funktion `even` nur eine Typsignatur angegeben. Die Definition der Funktion `odd` in Zeile 8 enthält auch hier einen Aufruf der nicht definierten Funktion `even`.

Beide Module sind nur partiell definiert. Erst durch Kombination beider Module ergibt sich eine vollständige Definition der Funktionen `even` und `odd`. Dennoch können beide statisch auf korrekte Typen überprüft werden. Durch sogenanntes *late-binding* ist es sogar möglich, beide Mixin Module separat voneinander zu compilieren. Dazu werden für nicht definierte Funktionen Platzhalter eingesetzt, welche erst beim finalen Zusammenfügen der gesamten Anwendung durch die eigentliche Funktion ersetzt werden.

Abbildung 4.2 zeigt die durch `mixin Nat = Even + Odd` erzeugte Kombination beider Module. Die beide Funktionen `even` und `odd` rufen sich nun gegenseitig rekursiv auf.

Auf diese Weise erlauben Mixin Module die Spezifikation von rekursiven Funktionen auf verschiedene Module zu verteilen. Diese Art der Mixin Module ist für die Modellierung des in Abschnitt 3.2 vorgestellten Problems der Rekursion über Namensraumgrenzen im Kontext von Funktionsüberladung jedoch nicht ausreichend. Bei der Definition einer Instanz einer überladenen Funktion sind im Allgemeinen die später durch weitere Module hinzugefügten Instanzen nicht bekannt. Diese müssen bei den bisher vorgestellten Mixin Modulen jedoch immer bereits angegeben werden, damit ein entsprechender Platzhalter vorgesehen werden kann. Aufgrund der nicht endlichen Subtyphierarchie in SaC ist es auch nicht möglich, alle potenziellen Instanzen anzugeben bzw. für diese Platzhalter zu generieren. Des weiteren müssen Mixin Module in dieser Form immer vervollständigt werden, bevor die tatsächliche Funktion verwendet werden kann. Überladene Funktionen hingegen müssen nicht auf allen Subtypen definiert sein. Die bisher vorgestellten Mixin Module eignen sich somit nicht, die in SaC vorhandene Funktionsüberladung in ML zu simulieren.

Eine Möglichkeit Funktionsüberladung in ML nachzubilden besteht in der Verwendung von Pattern-Matching. Abbildung 4.3 gibt ein Beispiel. Zunächst wird in Zeile 1 ein Datentyp definiert, welcher den Basistypen `int` und `real` jeweils einen gleichnamigen Konstruktor zuordnet. Dadurch ist es möglich, in der Funktion `add` in Zeile 3 jeweils eine passende Definition für verschiedene Konstruktor-Muster anzugeben. Zur

```
datatype val = INT of int
             | REAL of real
fun add (INT x) (INT y) = ...
      | (REAL x) (REAL x) = ...
```

Abbildung 4.3.: Funktion mit Pattern-Matching

```
mixin IntAdd = mix
  datatype val = INT of int
  fun add (INT x) (INT y) = ...
        | x y = inner(x, y)
end

mixin RealAdd = mix
  datatype val = REAL of real
  fun add (REAL x) (REAL y) = ...
end
```

Abbildung 4.4.: Verteilte Funktion mit Pattern-Matching

Laufzeit kann dann anhand des übergebenen Konstruktors die passende Definition ausgewählt werden bzw. ein Laufzeitfehler generiert werden, falls keine solche Definition existiert. Zur Auswahl der passenden Definition sind mehrere Kriterien möglich. Im Allgemeinen wird das sogenannte *first-match* Kriterium verwendet, d.h. das erste passende Muster in der Reihenfolge der Definitionen wird gewählt.

Mixin Module erlauben es, sowohl Datentypdefinitionen wie die in Zeile 1, als auch auf Pattern-Matching basierende Funktionsdefinitionen auf mehrere Mixin Module zu verteilen. Abbildung 4.4 zeigt eine solche verteilte Definition. In Zeile 1 ff. wird das Mixin `IntAdd` definiert, welches nur den für Werte des Typs `int` relevanten Teil aus Abbildung 4.3 enthält. Die Definitionen für Werte des Typs `real` sind im Mixin in Zeile 7 ff. zusammengefasst. Im Vergleich zu Abbildung 4.3 wurde das allgemeine Muster in Zeile 4 hinzugefügt. Dieses passt auf alle möglichen Argumente und ist durch das Sprachkonstrukt `inner` definiert.

Durch eine `inner` Anweisung gibt der Programmierer die Stelle innerhalb des Pattern-Matching vor, an der Erweiterungen durch andere Mixins eingefügt werden sollen. Die Kombination beider Mixins durch `mixin Add = IntAdd + RealAdd` erzeugt somit den in Abbildung 4.3 angegebenen ursprünglichen Code.

Dabei ist die Reihenfolge der Kombination von Bedeutung. Einerseits enthält das Mixin `RealAdd` kein `inner` Konstrukt, d.h. eine Kombination der Form `mixin Add = RealAdd + IntAdd` ist nicht möglich. Dies lässt sich jedoch durch Hinzufügen eines weiteren `inner` Konstruktes im Mixin `RealAdd` beheben.

Des weiteren werden die einzelnen Muster in der Reihenfolge ihrer Definition ab-

gearbeitet, d.h. die Muster des zuerst genannten Mixins werden vor den Mustern des zweit genannten ausgewertet. Im vorgestellten Beispiel ist dies von untergeordneter Bedeutung, da sich die einzelnen Muster nicht überdecken[2]. Im Falle überladener Funktionen im Kontext von Subtypen kann es jedoch zu solchen Überdeckungen kommen. Hier reicht ein Anfügen neuer Fälle an einer Stelle nicht aus. Um den Ansatz der Mixin Module auf die Funktionsüberladung in SaC zu übertragen, müssten demnach neue Muster an geeigneter Stelle eingefügt werden oder statt einer first-match eine *best-match* Strategie verwendet werden.

Allgemein erfordern Mixin Module das Eingreifen des Programmierers und einen vorausschauenden Entwurf von Modulen. Ein Mixin ist nur an den vom Programmierer durch partielle Funktionsdeklarationen oder das `inner` Konstrukt vorgegebenen Stellen erweiterbar. Das Überladen von Funktionen hingegen ist weit gehend automatisiert, so dass sich der Programmierer auf die Spezifikation des eigentlichen Algorithmus konzentrieren kann. Um die Idee der Mixins auf die Funktionsüberladung zu übertragen, wäre es daher nötig, die Definition der Erweiterungspunkte zu automatisieren.

[2]Hiervon ausgenommen ist das allgemeine Muster der `inner` Definition. Da dieses durch die Kombination entfällt, ist es für die Reihenfolge der Auswertung im kombinierten Mixin nicht maßgeblich.

5. Konzeptuelle Lösung

Im Rahmen dieses Kapitels wird eine Lösung für die im Kapitel 3 gestellten Anforderungen an ein Modulsystem vorgestellt. Dazu werden im ersten Abschnitt die grundlegenden Eigenschaften des Modulsystems beschrieben. In weiteren Abschnitten wird die Sprache SaC' dann inkrementell um Sprachelemente des Modulsystems erweitert. Abschließend wird ein kurzer Überblick über die dadurch erreichte Funktionalität gegeben.

5.1. Grundlagen des Modulsystems

Primäre Aufgabe des im Folgenden vorgestellten Modulsystems ist es, in anderen Namensräumen definierte Funktionen und Instanzen auch über dessen Grenzen hinaus verfügbar zu machen. Im Kontext von Modulsystemen werden diese zusammenfassend auch als *Symbol* bezeichnet. Ein solches Symbol umfasst dabei in SaC' jeweils alle Instanzen einer überladenen Funktion.

In SaC gibt es neben Funktionen noch weitere Symbolklassen, beispielsweise die benutzerdefinierte Typen. Da die für SaC' gewählte Teilmenge von SaC diese jedoch nicht enthält, werden sie im Rahmen dieser Arbeit nicht weiter behandelt. Dennoch werden die vorgestellten syntaktischen Erweiterung auf Symbol-Ebene definiert, um eine Erweiterung um weitere Symbolklassen ohne Anpassung der Syntax zu ermöglichen.

Als weiteres Ziel des Modulsystems wurde in Kapitel 3 die separate Compilation einzelner Einheiten eines Programms motiviert. In SaC' erfolgt dies analog zu anderen funktionalen Programmiersprachen wie Haskell und Clean auf Modulebene. Abbildung 5.1 zeigt die Erweiterung der Syntax von SaC' (siehe Abbildung 2.1) um Module.

Im Unterschied zu Programmen beginnen Module in SaC' mit den Schlüsselwort `module` gefolgt von einem Modulbezeichner und dem `;` (Semikolon) Symbol. Analog zu Programmen folgen dann beliebige Typ- und Funktionsdefinitionen. Da Module nicht eigenständig ausgeführt werden können, enthalten sie keine Startfunktion `main`. Diese bleibt Programmen vorbehalten.

Neben einer Einheit zu separaten Compilation definieren Module auch jeweils einen

$$Module \quad \Rightarrow \quad \texttt{module}\ Id\ \texttt{;}\ Definitions$$

Abbildung 5.1.: Um Module erweiterte Syntax von SaC'

eigenen Namensraum. Der Namensraumbezeichner entspricht dem Modulbezeichner. Somit können die Begriffe Namensraum und Modul in SaC' synonym verwendet werden. Das Hauptprogramm wird dabei mit dem Namensraum `Main` assoziiert.

In den folgenden Abschnitten werden neue Sprachelemente vorgestellt, welche aufbauend auf dieses allgemeine Modulkonzept die in Kapitel 3 gegebenen Anforderungen erfüllen.

5.2. Kontrollierte Überladung über Namensraumgrenzen

Wie in Abschnitt 3.1 dargestellt wurde, ist es für die Überladung von Funktionen über Namensraumgrenzen hinweg nicht ausreichend, eine globale Präferenzregel wie die globale bzw. lokale Überladung zu etablieren. Vielmehr sind granularere Kontrollmöglichkeiten nötig, um für jeden Namensraum die Menge der sichtbaren Instanzen angeben zu können. Die zentrale Idee ist hierbei, zwei duale Import-Mechanismen für die Schnittstellendefinition zwischen Modulen einzuführen, um für die Funktionsüberladung durchlässige bzw. bezüglich der Funktionsüberladung abgeschlossene Namensraumgrenzen zu unterscheiden.

Wird für eine Schnittstellendefinition die durchlässige, hier mit *Import* bezeichnete, Variante verwendet, werden die Instanzen in beiden dadurch verbundenen Namensräume sichtbar. Sie werden zu einer, in beiden Namensräumen identischen überladenen Funktion kombiniert. Die so entstehende überladene Funktion entspricht der durch eine entsprechende lokale Definition aller Instanzen erzeugten Funktion. Dadurch ist es möglich, in anderen Namensräumen definierte überladene Funktionen zu erweitern.

Soll eine Funktion dagegen nur für Funktionsaufrufe verwendet werden ohne weitere Instanzen hinzuzufügen, wird die zweite Version der Schnittstellendefinition verwendet. Diese wird im Folgenden mit *Use* bezeichnet. Auch hier wird die entsprechende Funktion im benutzenden Namensraum sichtbar. Allerdings können keine weiteren Instanzen hinzugefügt werden. Die entsprechende Namensraumgrenze bleibt bezüglich der Funktionsüberladung abgeschottet.

Wird für eine Funktion weder eine Import-, noch eine Use-Schnittstellendefinition angegeben, bleibt die Sichtbarkeit der Funktion und der entsprechenden Instanzen auf den definierenden Namensraum beschränkt. Dies ermöglicht die Trennung einzelner Instanzen bzw. überladener Funktionen.

Die vorgestellte Variante der Use-Schnittstellendefinition erlaubt jedoch nicht die Verwendung von Funktionen mit identischem Bezeichner aus verschiedenen Namensräumen. In diesem Falle würden beide Bezeichner im lokalen Namensraum sichtbar, ohne jedoch zu einer gemeinsamen überladenen Funktion kombiniert zu werden. Dies führt somit zu einem Namenskonflikt. Um dies zu umgehen, kann für Funktionsaufrufe alternativ der in Abschnitt 3.1 vorgestellte qualifizierte Bezeichner der Funktion verwendet werden. In diesem Falle ist keine explizite Use-Schnittstellendefinition notwendig. Da somit die entsprechenden Bezeichner auch nicht im aktuellen Namensraum sichtbar werden, kommt es nicht zu einem Namenskonflikt.

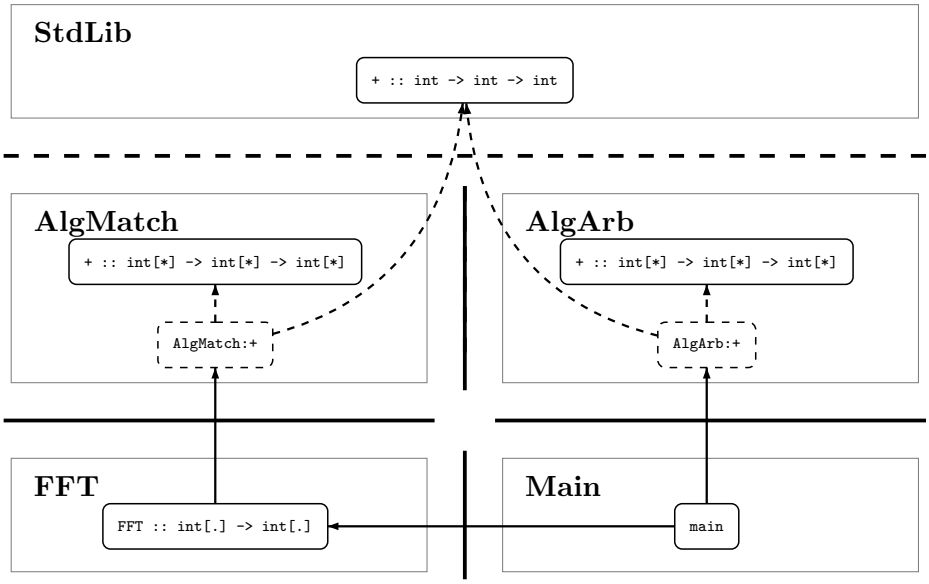


Abbildung 5.2.: Beispiel Überladung über Modulgrenzen

Unter Verwendung der verschiedenen Schnittstellendefinition kann nun das Beispiel aus Abschnitt 3.1 dargestellt werden. Abbildung 5.2 zeigt dieses Beispiel ergänzt um die jeweiligen gewünschten Namensraumgrenzen. Ist eine Namensraumgrenze durchlässig für die Überladung von Funktionen (Import), ist sie als gestrichelte Linie dargestellt. Dies ist der Fall zwischen den Namensräumen **StdLib** und **AlgMatch** bzw. **AlgArb**, da beide Algebren jeweils die in der Standard-Bibliothek definierten Instanzen für `+` um zusätzliche Instanzen für Arrays erweitern. Die dadurch entstehende lokale überladene Funktion ist durch die gestrichelten Kästen angedeutet. Die beiden überladenen Funktion `AlgMatch:+` und `AlgArb:+` umfassen jeweils die im entsprechenden Namensraum gültigen Instanzen, in der Grafik durch die abgehenden Dispatch-Pfeile angedeutet.

Namensraumgrenzen, die für die Überladung von Funktionen nicht durchlässig sind (Use), sind als durchgezogene Linien dargestellt. Dies ist für die Namensraumgrenzen zwischen **AlgMatch** und **FFT**, sowie zwischen **AlgArb** und **Main** der Fall. Sowohl die Funktion `FFT:FFT`, als auch `Main:main` verwenden die jeweiligen Instanzen aus dem entsprechenden Namensraum einer der Algebren. Sie fügen jedoch keine weiteren Instanzen hinzu. Somit müssen die jeweiligen Namensraumgrenzen nicht für die Funktionsüberladung durchlässig sein.

Weiterhin sind die Namensräume **AlgMatch** und **AlgArb** bezüglich der Funktionsüberladung strikt separiert. Dies ist zwingend erforderlich, da ansonsten in den jeweiligen Namensräumen mehrere Instanzen für die gleichen Argumenttypen definiert wären. Durch die Trennung ist sicher gestellt, dass in jedem Namensraum nur eine Instanz sichtbar ist und der Dispatch somit eindeutig ist.

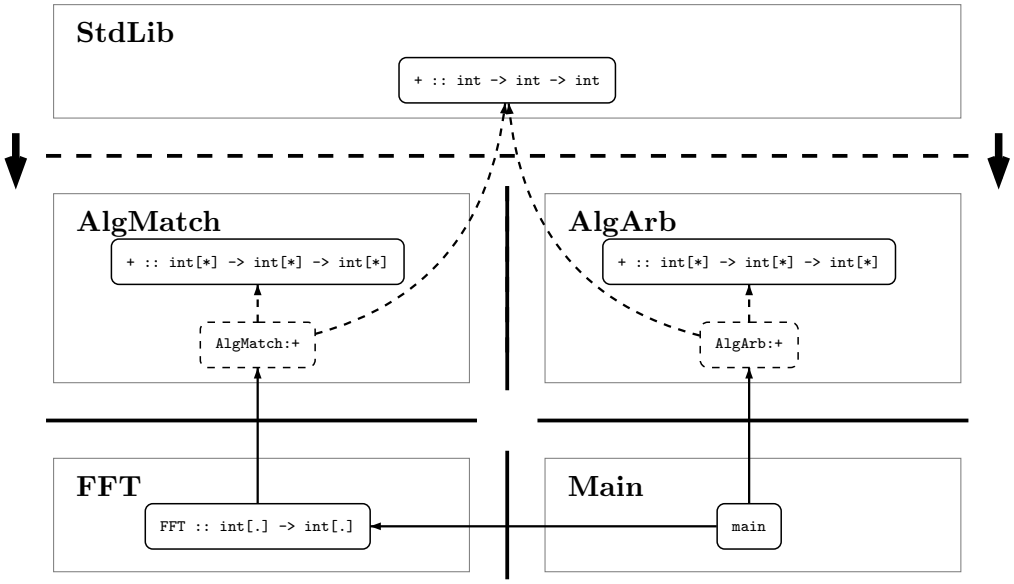


Abbildung 5.3.: Verfeinertes Beispiel für Überladung über Modulgrenzen

Auch die Namensräume **FFT** und **Main** sind durch eine strikte Namensraumgrenze getrennt. Da beide Namensräume jedoch keinerlei überladene Funktionen definieren, ist diese strikte Trennung nicht zwingend erforderlich.

Bei genauerer Betrachtung der Abbildung 5.2 fällt auf, dass die beiden Namensräume **AlgMatch** und **AlgArb** trotz der angegebenen Namensraumgrenzen über die gemeinsame Verbindung zum Namensraum **StdLib** verbunden sind. Da die jeweiligen Namensraumgrenzen durchlässig für die Überladung von Funktionen sind, sind alle definierten Instanzen für `+` in allen drei Namensräumen sichtbar. Gewünscht ist jedoch nur die weitere Überladung der im Namensraum `StdLib` definierten Instanzen innerhalb der jeweiligen Algebren. Der Austausch der Instanzen zwischen den einzelnen Namensräumen darf daher nicht frei erfolgen, sondern nur in einer vorgegebenen Richtung. Abbildung 5.3 zeigt dieses Szenario. Die Namensraumgrenze zwischen den Namensräumen **StdLib** und **AlgMatch** bzw. **StdLib** und **AlgArb** ist hier gerichtet, angedeutet durch die kleinen Pfeile. Aus dem Namensräumen **AlgMatch** und **AlgArb** sind die im Namensraum **StdLib** definierten Instanzen sichtbar. Sie können die entsprechende Namensraumgrenze in Richtung der Pfeile passieren. Der Namensraum **StdLib** hingegen bleibt für andere Instanzen abgeschottet, da in diesem Fall die Instanzen entgegen der angegebenen Richtung exportiert würden.

Wie das Beispiel zeigt, müssen die entsprechenden Schnittstellendefinitionen somit eine Möglichkeit bieten, die Richtung der Funktionsüberladung anzugeben. Dies kann erreicht werden, indem für jeden Namensraum sowohl eine Schnittstelle für den Export, als auch für den Import definiert wird. Die Export-Schnittstelle gibt dabei an, welche Instanzen potenziell in andere Namensräume exportiert werden dürfen. Dem

gegenüber steht die Import-Schnittstelle, welche die Instanzen angibt, die letztendlich in den entsprechenden Namensraum importiert werden sollen.

Damit erfüllt die in Abbildung 5.2 dargestellte Situation die in Kapitel 3 gestellten Anforderungen bezüglich der Überladung von Funktionen über Namensraumgrenzen. In den folgenden Abschnitten wird nun eine Erweiterung der Sprache SaC' vorgestellt, welche es dem Programmierer erlaubt, durchlässige und abgeschottete Namensraumgrenzen bezüglich der Funktionsüberladung zwischen einzelnen Modulen zu spezifizieren.

5.2.1. Erweiterte Syntax von SaC'

Um die zwei verschiedenen Arten von Schnittstellendefinitionen abbilden zu können, sind auch zwei verschiedene Sprachkonstrukte notwendig. Ein Hauptaugenmerk liegt dabei auf der einfachen Anwendbarkeit für den Programmierer und der guten Lesbarkeit des resultierenden Quellcodes. Abbildung 5.4 zeigt die entsprechenden Ergänzungen zur bisher vorgestellten Syntax von SaC'. Neben Funktionsdefinitionen beinhalten Modul- und Programmspezifikationen nun zusätzlich eine Schnittstellendefinition. Diese beschreibt einerseits, welche Symbole aus dem Namensraum des jeweiligen Moduls exportiert, d.h. für andere Namensräume sichtbar gemacht werden. Für den Namensraum **Main** von Programmen ist dies nicht vorgesehen, da diese auch nicht von anderen Modulen verwendet werden können. Andererseits umfasst die Schnittstellendefinition auch die Symbole aus anderen Namensräumen, welche im Namensraum der Schnittstellendefinition genutzt werden sollen. Um zwischen durchlässigen und abgeschotteten Namensraumgrenzen unterscheiden zu können, sind die

$$
\begin{array}{lcl}
Module & \Rightarrow & \texttt{module}\ Id\ \texttt{;}\ Interface\ Definitions \\
Program & \Rightarrow & [\ Import\ ]^{*}\ Definitions\ main \\
Interface & \Rightarrow & Export[\ Import\ ]^{*} \\
Export & \Rightarrow & [\ \texttt{provide}\ \texttt{\{}\ Id[\ \texttt{,}\ Id\ ]^{*}\ \texttt{\}}\ \texttt{;}\ ] \\
 & & \quad [\ \texttt{export}\ \texttt{\{}\ Id[\ \texttt{,}\ Id\ ]^{*}\ \texttt{\}}\ \texttt{;}\ ] \\
Import & \Rightarrow & \texttt{use}\ Id\ \texttt{:}\ \texttt{\{}\ Id[\ \texttt{,}\ Id\ ]^{+}\ \texttt{\}}\ \texttt{;} \\
 & | & \texttt{import}\ Id\ \texttt{:}\ \texttt{\{}\ Id[\ \texttt{,}\ Id\ ]^{+}\ \texttt{\}}\ \texttt{;} \\
Expr & \Rightarrow & Const \\
 & | & Id \\
 & | & [\ Id\ \texttt{:}\ ]Id\ \texttt{(}\ [\ Expr[\ \texttt{,}\ Expr\ ]^{*}\ ]\ \texttt{)} \\
 & | & Expr\ PrimOp\ Expr
\end{array}
$$

Abbildung 5.4.: Syntax der Schnittstellenbeschreibung in SaC' (in BNF)

jeweiligen Sprachkonstrukte doppelt ausgelegt.

Des weiteren werden angewandte Vorkommen von Funktionsbezeichnern um einen Namensraumbezeichner ergänzt. Durch Angabe des qualifizierten Bezeichners können so Symbole aus anderen Namensräumen referenziert werden. Im Folgenden wird eine genauere Beschreibung der einzelnen Erweiterungen gegeben.

Mittels `provide` und `export` wird die Schnittstelle eines Moduls bezüglich der potenziellen Sichtbarkeit lokal definierter Symbole in anderen Namensräumen festgelegt. Durch das `provide` Konstrukts werden die Symbole angegeben, welche auch in anderen Namensräumen verwendet werden dürfen. Allerdings kann eine Funktion, welche mittels `provide` für andere Namensräume sichtbar gemacht wurde, nicht weiter überladen werden. Sie wird nur mit genau der Menge an Instanzen, wie sie im aktuellen Namensraum definiert wurde, zur weiteren Verwendung zur Verfügung gestellt.

Dem gegenüber steht das `export` Konstrukt. Symbole, welche mittels `export` zur Verfügung gestellt werden, können in anderen Namensräumen wie lokal definierte Symbole verwendet werden. Für Funktionsdefinitionen schließt dies das Hinzufügen weiterer Instanzen ein.

Die lokale Sichtbarkeit in anderen Namensräumen definierter Symbole wird durch die `use` und `import` Konstrukte gesteuert. Das `use` Konstrukt ist dabei die Entsprechung des `provide` Konstrukts. Eine `use` Anweisung besteht aus dem Namensraumbezeichner des Namensraums, aus dem die Symbole verwendet werden sollen und einer Liste der zu verwendenden Symbole. Diese können dann, mit den beim `provide` Konstrukt beschriebenen Einschränkungen, wie lokal definierte Symbole verwendet werden.

Statt ein Symbol mittels `use` verfügbar zu machen, kann es auch direkt durch einen qualifizierten Bezeichner referenziert werden. Dies bietet sich insbesondere dann an, wenn im aktuellen Namensraum bereits ein Symbol gleichen Namens definiert ist und es durch die Verwendung von `use` zu einem Namenskonflikt kommen würde.

Als Gegenstück zum `export` Konstrukt dient das `import` Konstrukt. Eine `import` Anweisung besteht dabei analog zum `use` Konstrukt aus einem Namensraumbezeichner und einer Liste von Symbolen. Diese können im importierenden Namensraum wie lokal definierte Symbole verwendet werden. Dies schließt das Hinzufügen weiterer Instanzen zu einer überladenen Funktion ein. Insbesondere hervorzuheben ist die Möglichkeit, durch `import` Anweisungen Instanzen aus verschiedenen Namensräumen zu einer überladenen Funktion zu kombinieren.

Um die Spezifikation von Schnittstellendefinitionen einfacher zu gestalten und die Lesbarkeit zu verbessern, werden innerhalb dieser nicht die einzelnen Instanzen einer Funktion angegeben, sondern lediglich der Funktionsbezeichner. Dies erlaubt es einerseits, alle Instanzen einer überladenen Funktion mit nur einer Anweisung zur Verfügung zu stellen bzw. zu benutzen. Dadurch bleiben Schnittstellendefinitionen auch bei großer Anzahl von Instanzen kompakt und somit lesbarer. Andererseits wird so sicher gestellt, dass immer die tatsächliche Definition einer überladenen Funktion benutzt wird und nicht einzelne Instanzen ausgelassen werden. Ansonsten müsste der Anwender eines Moduls den genauen Aufbau der einzelnen Funktionen und die entsprechenden definierten Instanzen kennen. Insbesondere hätte eine Änderung der

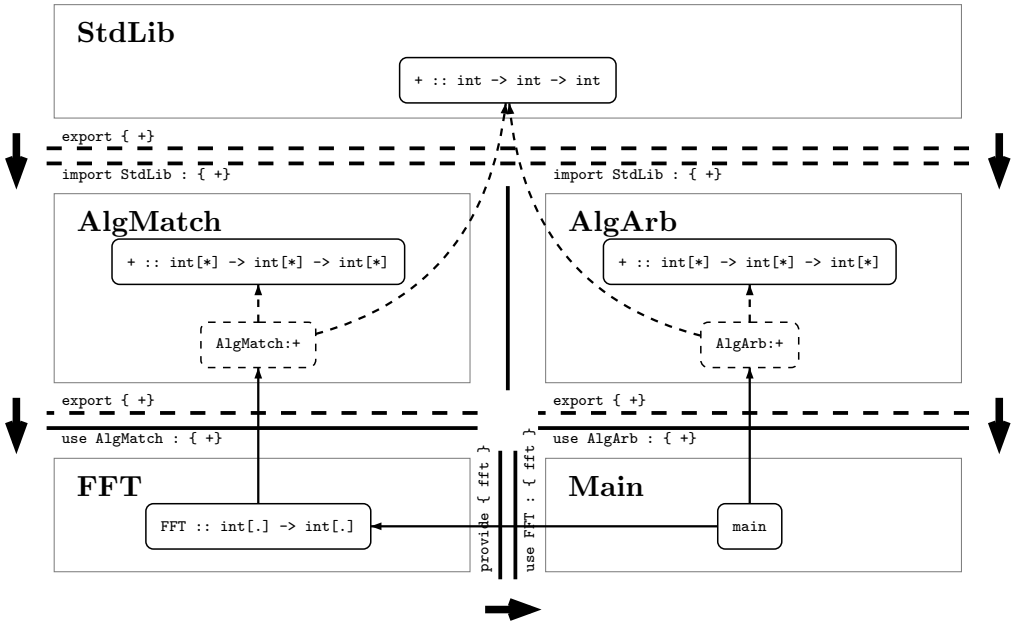


Abbildung 5.5.: Beispiel für `provide`, `use`, `export` und `import`

modulinternen Überladungsstruktur auch eine Änderung der Schnittstelle des Moduls zur Folge, ein im Kontext von modularem Programmdesign unerwünschter Nebeneffekt.

Abbildung 5.5 zeigt die Verwendung der neu eingeführten Konstrukte am bereits eingangs gezeigten Beispiel. Die durchlässige Namensraumgrenze zwischen den Modulen **StdLib** und **AlgMatch** bzw. **StdLib** und **AlgArb** ist dabei durch eine `export` Anweisung auf der Seite des Moduls **StdLib** und eine `import` Anweisung auf Seiten der Module **AlgMatch** und **AlgArb** realisiert.

Beide Algebra Module stellen die definierten Instanzen mittels einer `export` Anweisung zur Erweiterung um neue Instanzen zur Verfügung. Dies erlaubt eine spätere Ergänzung der Funktion `+` z.B. auf Arrays anderer Basistypen. Die Module **FFT** und **Main** machen von dieser Möglichkeit jedoch keinen Gebrauch und benutzen daher eine `use` Anweisung, um die jeweilige `+` Funktion lokal sichtbar zu machen. Somit verwendet die Funktion `FFT:FFT` die Funktion `AlgMatch:+` und damit die im Namensraum **AlgMatch** sichtbaren Instanzen der überladenen Funktion. Analog verhält es sich für die Funktion `Main:main`. Um auch die Funktion `FFT:FFT` verwenden zu können, wurde diese zusätzlich mittels einer `use` Anweisung im Namensraum **Main** sichtbar gemacht. Da die Funktion im Modul **FFT** nur durch eine `provide` Anweisung veröffentlicht wurde, ist eine Ergänzung um weitere Instanzen und die Verwendung des `import` Konstrukts im Modul **Main** ausgeschlossen.

5.2.2. Erweiterte Semantik von SAC'

Die vorgestellte erweiterte Syntax der Sprache SAC' erfüllt somit die in Abschnitt 3.1 gestellten Anforderungen. Im folgenden Abschnitt wird nun die formale Definition der Semantik ebenfalls um die für das Modulsystem notwendigen Konstrukte erweitert. Dazu wird in einem ersten Schritt die Sprache $\mathcal{F}un$ um Module ergänzt. In weiteren Schritten werden dann die einzelnen Sprachkonstrukte nach $\mathcal{F}un$ abgebildet.

Module können im λ-Kalkül durch benannte Records abgebildet werden [Rei97]. Unter einem *benannten Record* wird eine Menge von Bezeichner-Wert Paaren verstanden. Die Bezeichner dienen dabei der eindeutigen Referenzierung des entsprechenden Wertes. Der Zugriff auf diesen wird durch eine entsprechende Selektionsoperation ermöglicht.

Ein solcher Record bildet die Schnittstelle eines Moduls ab. Für jede exportierte Funktion ist innerhalb des Records ein entsprechender Eintrag vorhanden. Als Bezeichner für diesen Eintrag wird der jeweilige Funktionsbezeichner gewählt. Die Funktionsdefinition bildet den Wert des Eintrages. Der Record kapselt somit analog zu einem Modul die Funktionsdefinitionen und erlaubt einen selektiven Zugriff auf diese.

Da in SAC' zwei verschiedene Methoden der Schnittstellendefinition vorhanden sind, werden für $\mathcal{F}un$ ebenfalls zwei Records zur Darstellung eines Moduls benötigt. Jeweils ein Record für die Use-Schnittstelle und ein Record für die Import-Schnittstelle. Ein Modul wird in $\mathcal{F}un$ somit durch ein Tupel bestehend aus zwei Records dargestellt. Im Folgenden werden die dazu nötigen Erweiterung der Sprache $\mathcal{F}un$ vorgestellt.

Ein benannter Record wird in $\mathcal{F}un$ als Liste von *Bezeichner* `=` *Ausdruck* Paaren, umgeben von `[]` (eckigen Klammern), angegeben. Um Record-Bezeichner von Bezeichnern des λ-Kalküls zu unterscheiden, werden diese in `''` (einfache Anführungszeichen) gesetzt. Ein Tupel wird als Liste seiner Elemente, umgeben mit `<>` (spitzen Klammern), angegeben. Um einzelne Elemente eines Records bzw. Tupels selektieren zu können, wird eine Selektionsoperation der Form *Expr* `.` *'Id'* bzw. *Expr* `.` `#` *Num* eingeführt. Das `#` (Raute) Symbol dient dabei der Unterscheidung zwischen Selektionsindizes und den entsprechenden Konstanten des λ-Kalküls. Abbildung 5.6 zeigt die um Records und Tupel erweiterte Syntax von $\mathcal{F}un$ (vgl. Abbildung 2.3).

Eine Selektion der Form *Expr* `.` *'Id'*, wobei *Expr* zu einem Record ausgewertet werden kann, wird zum Element mit dem Bezeichner *Id* dieses Records ausgewertet. Die Selektion auf Tupeln *Expr* `.` `#` *Num*, wobei *Expr* zu einem Tupel evaluiert werden kann, wird zum Element des Tupels an der Position *Num* ausgewertet. Die Elemente werden dabei von links mit 1 beginnend gezählt.

Beide Formen der Selektion werden *lazy* ausgewertet, d.h. die Auswertung der Selektion wird solange wie möglich verzögert. Des weiteren werden Records und Tupel als solche nicht weiter ausgewertet, d.h. die Berechnung wird nicht innerhalb dieser Strukturen fortgesetzt. Dies verhindert die unnötige und möglicherweise nicht terminierende Berechnung nicht verwendeter Teile eines Records bzw. Tupels.

Unter Verwendung der vorgestellten Erweiterungen der Sprache $\mathcal{F}un$ können nun

```
Expr    ⇒ Const
        | Id
        | Lambda
        | Let
        | Letrec
        | Ap
        | PrimFun
        | Record
        | Select
        | Tuple

Record  ⇒ [ [ ' Id ' = Expr[ , ' Id ' = Expr ]* ] ]

Select  ⇒ Expr . ' Id '
        | Expr . # Num

Tuple   ⇒ < Expr , Expr >
```

Abbildung 5.6.: Erweiterte Syntax der Sprache $\mathcal{F}un$ (in BNF)

SaC'-Module in die eingangs beschriebene Tupel-Darstellung übersetzt werden. Abbildung 5.7 zeigt die entsprechenden Transformationsregeln. Module werden analog zum in Abbildung 2.4 vorgestellten Transformationsschema für Programme in ein `letrec` Konstrukt transformiert, welches die einzelnen Funktionsdefinitionen an den jeweiligen Bezeichner bindet. Als Zielterm des `letrec` Konstrukts wird ein Tupel aus zwei benannten Records verwendet. Der erste Record enthält alle mittels `provide` und `export` zur Verfügung gestellten Funktionen. Er umfasst somit alle Funktionen, welche mittels `use` in anderen Modulen verwendet werden können. Der zweite Record enthält nur die mittels `export` zur Verfügung gestellten Funktionen. Alle mittels `import` in anderen Modulen verwendbare Funktionen sind somit in diesem Record enthalten.

Insgesamt stellt somit dieses Tupel alle von außen referenzierbaren Funktionen eines Moduls zur Verfügung. Die Regeln *Provide* und *Export* zeigen eine formale Beschreibung der beiden Records.

Der Zugriff auf derart zur Verfügung gestellte Funktionen ist durch die beiden Regeln *Use* und *Import* abgebildet. Mittels `use` in den Namensraum eines Moduls eingebundene Funktionen werden durch die entsprechende Regel in lokale Definitionen im `letrec` Konstrukt des benutzenden Moduls transformiert. Dazu wird an den Bezeichner der Funktion die passende Definition aus dem definierenden Modul gebunden. Die Selektion `( `*Mod*` . #1 ) . '`*Id*`'` wählt diese aus dem ersten benannten Record des Modul-Tupels aus. Da dieser alle zur Verfügung gestellten Funktionsdefinitionen eines Moduls enthält, ist die entsprechende Definition damit unter ihrem

$\mathbb{C}[\![$ *Import Use Fundefs* $]\!] \rightsquigarrow$ *(Program)*

```
letrec
    C[[ Use ]]
    OVLD{ C[[ Import ]]
          C[[ Fundefs ]] }
in < [ 'main' = main], [] >
```

$\mathbb{C}[\![$ *Provide Export Use Import Fundefs* $]\!] \rightsquigarrow$ *(Module)*

```
letrec
    C[[ Use ]]
    OVLD{ C[[ Import ]]
          C[[ Fundefs ]] }
in < [ C[[ Provide ]], C[[ Export ]]], [C[[ Export ]] ]>
```

$\mathbb{C}[\![$ `use` *Mod* `:` `{` Id_1 `,` $\ldots$ `,` Id_n `}` `;` $]\!] \rightsquigarrow$ *(Use)*

Id_1 = `(` Mod_μ `.` `#1` `)` `.` 'Id_1'

$\vdots$

Id_n = `(` Mod_μ `.` `#1` `)` `.` 'Id_n'

$\mathbb{C}[\![$ `import` *Mod* `:` `{` Id_1 `,` $\ldots$ `,` Id_n `}` `;` $]\!] \rightsquigarrow$ *(Import)*

Id_1 = `(` Mod_μ `.` `#2` `)` `.` 'Id_1'

$\vdots$

Id_n = `(` Mod_μ `.` `#2` `)` `.` 'Id_n'

$\mathbb{C}[\![$ `provide` `{` Id_1 `,` $\ldots$ `,` Id_n `}` `;` $]\!] \rightsquigarrow$ *(Provide)*

'Id_1' = Id_1 , $\ldots$, 'Id_n' = Id_n

$\mathbb{C}[\![$ `export` `{` Id_1 `,` $\ldots$ `,` Id_n `}` `;` $]\!] \rightsquigarrow$ *(Export)*

'Id_1' = Id_1 , $\ldots$, 'Id_n' = Id_n

$\mathbb{C}[\![$ *Mod* `:` *Id* `(` $Expr_1$ `,` $\ldots$ `,` $Expr_n$ `)` $]\!] \rightsquigarrow$ *($Expr_5$)*

`(` `(` `(` Mod_μ `.` `#1` `)` `.` 'Id' `)` $\mathbb{C}[\![$ $Expr_1$ $]\!] \cdots \mathbb{C}[\![$ $Expr_n$ $]\!]$ `)`

Abbildung 5.7.: Erweitertes Transformationsschema von SAC' nach $\mathcal{F}un$

Bezeichner lokal verfügbar.

Analog wird für das `import` Konstrukt verfahren. Hier wird jedoch der zweite benannte Record verwendet, da dieser nur die mittels `export` zur Verfügung gestellten Funktionen enthält. Somit ist sicher gestellt, dass nur durch `export` exportierte Funktionen für eine `import` Anweisung verwendet werden können.

Die Überladung von Funktionen wird unter Verwendung der in Abschnitt 2.2.4 vorgestellten $\mathcal{OVLD}$ Transformation realisiert. Für die Funktionsüberladung werden allerdings nur lokal definierte Funktionen und solche, die mittels des `import` Kon-

```
module AlgArb;

import StdLib : { +};
export { +};

int[*] +( int[*] A, int[*] B)
{
  result = ...

  return( result);
}
```

Abbildung 5.8.: Vereinfachter SaC'-Code für das Modul `AlgArb`

strukts in den lokalen Namensraum importiert wurden, berücksichtigt. Funktionen, die durch eine `use` Anweisung in den Namensraum importiert wurden, können somit nicht weiter überladen werden. Die Regel *Module* gibt einen Überblick.

Durch die erneute Überladung importierter Funktionen kann es zu geschachtelten Vorkommen des `ovld` Konstrukts kommen. Diese können jedoch durch Bilden der transitiven Hülle in ein nicht geschachteltes `ovld` Konstrukt transformiert werden.

Für Programmdefinitionen wird analog zu den Moduldefinitionen verfahren. Da diese jedoch keine Exportanweisungen enthalten, ist das entsprechende Tupel fest vorgegeben. Es entspricht einer mittels `provide { main }` spezifizierten Schnittstellendefinition. Ein Programm stellt somit nur die spezielle Startfunktion `main` zur Verfügung. Die Regel *Program* zeigt eine formale Beschreibung.

Abschließend wurde noch die Regel $Expr_5$ ergänzt. Diese transformiert eine Anwendung einer mittels eines qualifizierten Bezeichners referenzierten Funktion in eine Funktionsanwendung in $\mathcal{F}un$. Dazu wird statt des Funktionsbezeichners eine Selektion der entsprechenden Funktionsdefinition aus dem durch den qualifizierten Bezeichner angegebenen Modul erzeugt. Da alle in der Schnittstellendefinition eines Moduls angegebenen Funktionen durch einen solchen qualifizierten Bezeichner referenziert werden können, findet der erste benannte Record des Modul-Tupels Verwendung.

Um die Bezeichner von Modulen von anderen in SaC' verwendeten Bezeichnern zu unterscheiden, werden sie in $\mathcal{F}un$ um den Index μ ergänzt. Somit ist sichergestellt, dass es in $\mathcal{F}un$ nicht zu Namenskonflikten zwischen Funktions- und Modulnamen kommen kann.

Abbildung 5.8 zeigt das Modul `AlgArb` als SaC'-Code. Die Berechnung der Addition in Zeile 8 wurde dabei ausgelassen. Das Ergebnis der Transformation in die Sprache $\mathcal{F}un$ ist in Abbildung 5.9 dargestellt. Die in Zeile 3 mittels einer `import` Anweisung aus dem Modul `StdLib` importierte Instanz der Funktion `+` wird in der $\mathcal{F}un$-Darstellung lokal gebunden. Auf der linken Seite steht dabei der Funktionsbezeichner, der entsprechende Index wird durch die $\mathcal{OVLD}$-Transformation erzeugt. Durch die Selektion auf der rechten Seite wird die Definition aus dem Modul-Tupel

```
letrec
   +φ( int[.] -> int[.] -> int[.]) = ( StdLibμ . #2) . '+'
   +φ( int[*] -> int[*] -> int[*]) =   let
                                         result = ...
                                      in result
   + = ovld( +φ( int[.] -> int[.] -> int[.]), +φ( int[*] -> int[*] -> int[*]))
in < [ '+' = +], [ '+' = +] >
```

Abbildung 5.9.: Ergebnis der Transformation des Beispiels aus Abbildung 5.8

```
letrec
   Mod1μ =  < [ ... ], [ ... ] >
   ⋮
   Modnμ =  < [ ... ], [ ... ] >
   Mainμ =  < [ 'main' = ... ], [] >
in ( Mainμ . #1) . 'main'
```

Abbildung 5.10.: Modulkontext in der Sprache $\mathcal{F}un$

des Moduls `StdLib` extrahiert. Da es sich um eine `import` Anweisung handelt, wird der zweite Record innerhalb des Tupels verwendet. Die entsprechende Instanz ist somit lokal verfügbar.

Ebenso wird eine Bindung für die lokale Instanz der Funktion `+` erzeugt. Auf der rechten Seite findet sich hier die Transformation des Funktionsrumpfes. Der Index des Funktionsbezeichners wird analog zur importierten Instanz durch die $\mathcal{OVLD}$-Transformation erzeugt. Diese erzeugt weiter die Bindung des Bezeichners `+` an das `ovld` Konstrukt, welches beide Instanzen, sowohl die importierte, als auch die lokal definierte, umfasst. Die Instanzen werden somit wie gewünscht zu einer gemeinsamen überladenen Funktion zusammengefasst.

Da diese überladene Funktion in Zeile 4 des SaC'-Codes wieder mittels `export` exportiert wird, werden entsprechende Einträge in beiden Records des Modul-Tupels erzeugt. Sie können somit sowohl mittels `import`, als auch mittels `use` in anderen Modulen verwendet werden.

In der abgebildeten $\mathcal{F}un$-Darstellung des Moduls `AlgArb` ist der Bezeichner $\texttt{StdLib}_\mu$ für das Modul `StdLib` ungebunden. Der entsprechende Kontext der vorhanden Module fehlt. In SaC' bildet das Dateisystem des Betriebssystems den Kontext, im Rahmen dessen Module gesucht werden. Alle dort abgelegten Module[1] sind innerhalb von Schnittstellendefinitionen verfügbar. Um diesen Modulkontext nach $\mathcal{F}un$ abbilden zu können, wird Fun auf oberster Ebene um eine weitere `letrec` Bindung erweitert. Dieser allgemeine Kontext ist in Abbildung 5.10 dargestellt.

[1] Im Allgemeinen wird das Auffinden von Modulen durch spezielle Verzeichnisse, innerhalb derer sich die Module befinden müssen, gelöst. Aber auch das Durchsuchen des gesamten Dateisystems bzw. vorgegebener Suchpfade ist möglich. Die genauen Details sind für die Betrachtung im Rahmen dieser Arbeit jedoch nicht relevant.

Für jedes Modul wird auf oberster Ebene das zugehörige Tupel an den Modulnamen gebunden. Dieser wird analog zum vorgestellten Transformationsschema um den Index μ ergänzt. Ebenso wird mit dem eigentlichen Programm verfahren. Dieses wird an den Bezeichner `Main`$_\mu$ gebunden. Da für das Hauptprogramm immer die Funktion `main` definiert und mittels `provide` zur Verfügung gestellt wird, kann davon ausgegangen werden, dass diese im entsprechenden Record vorhanden ist. Somit wird als Startterm im äußeren `letrec` diese Komponente mittels `( Main`$_\mu$ `. #1) . main` selektiert. Die innere Selektion wählt dabei innerhalb des Modul-Tupels den Record für mittels `provide` zur Verfügung gestellte Funktionen. Die äußere Selektion wählt dann die Definition für die Funktion `main`.

Mit dem vorgestellten Transformationsschema und den Erweiterungen der Sprache $\mathcal{F}un$ kann SaC' somit inklusive Modulen nach $\mathcal{F}un$ abgebildet werden.

5.2.3. Das Diamant-Import-Problem

Durch die im vorigen Abschnitt vorgestellte Möglichkeit der Funktionsüberladung über Modulgrenzen hinweg kann es zu sich gegenseitig überdeckenden Funktionsinstanzen innerhalb eines `ovld` Konstruktes kommen. Im Allgemeinen führt dies zu einem Übersetzungsfehler, da durch überdeckende Instanzen der Dispatch nicht mehr eindeutig ist.

Einen Sonderfall stellt das sogenannte *Diamant-Import-Problem* dar. Abbildung 5.11 zeigt eine solche Situation. Im oberen Drittel ist das Modul **A** dargestellt. Dieses definiert eine Instanz der Funktion `+` für Werte vom Typ `int` und stellt diese mittels `export { +}` zur Verfügung. Im mittleren Drittel sind zwei weitere Module dargestellt. Modul **B1** importiert die Instanz aus dem Modul **A** und ergänzt die überladene Funktion mit einer Instanz für Werte vom Typ `float`. Die so entstandene überladene

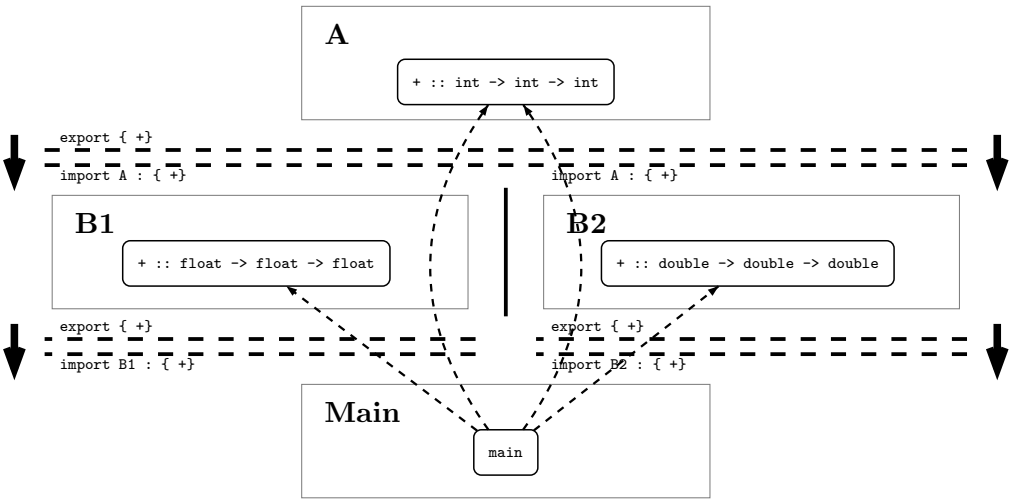


Abbildung 5.11.: Beispiel für das Diamant-Import-Problem

Funktion mit zwei Instanzen wird dann durch die Anweisung `export { +}` exportiert. Analog verfährt das Modul **B2** mit einer Instanz für Werte vom Typ `double`. Im unteren Drittel befindet sich das Modul **Main**, welches seinerseits die überladene Funktion `+` aus beiden Modulen **B1** und **B2** importiert.

Im Namensraum **Main** sind somit vier verschiedene Instanzen sichtbar, angedeutet durch die gestrichelten Pfeile. Sowohl die zwei Instanzen für `int` bzw. `float` Werte aus dem Modul **B1**, als auch die zwei Instanzen für `int` und `double` Werte aus dem Modul **B2** müssen für den Dispatch berücksichtigt werden. Allerdings überdecken sich die beiden Instanzen für Werte vom Typ `int`, so dass der Dispatch im Modul **Main** nicht eindeutig ist. Das Programm kann nicht übersetzt werden.

Bei genauerer Betrachtung der Situation in Abbildung 5.11 wird jedoch klar, dass beide Instanzen für Werte vom Typ `int` den gleichen Ursprung haben. Beide resultieren aus der Definition im Modul **A**. Berücksichtigt man die *Herkunft* einer Instanz, d.h. den Namensraum in dem sie definiert wurde, wird der Dispatch wieder eindeutig. Offensichtlich handelt es sich bei beiden Instanzen für Werte vom Typ `int` um die gleiche Instanz, so dass beide Instanzen im Namensraum **Main** zu einer zusammengefasst werden können.

Folglich muss das `ovld` Konstrukt derart gestaltet werden, dass es die Herkunft einer Instanz berücksichtigt und sich überdeckende Instanzen falls möglich zu einer Instanz zusammenfasst[2].

5.3. Rekursion über Namensraumgrenzen

Im vorigen Abschnitt wurde die Sprache SaC' um Konstrukte erweitert, mittels derer Namensräume und Module abgebildet werden können. Dabei wurden die in Abschnitt 3.1 motivierten Anforderungen berücksichtigt. Insbesondere ist die Menge der Instanzen einer überladenen Funktion in SaC' mit den vorgestellten Erweiterungen vom jeweiligen Namensraum abhängig. Wie in Abschnitt 3.2 gezeigt wurde, reicht dies für rekursive überladene Funktionen nicht aus. Im Folgenden wird daher der Modul-Mechanismus von SaC' noch um den geforderten Aufrufkontext erweitert.

Abbildung 5.12 zeigt das um einen Aufrufkontext erweiterte Szenario aus Abbildung 3.5. Der rekursive Aufruf der überladenen Funktion `FFT` durch die im Namensraum **A** definierte Instanz für Vektoren beliebiger Länge wird nun durch eine Funktionstabelle umgeleitet. Diese ist im Namensraum **A** durch einen eckigen Kasten dargestellt. Innerhalb des Rechtecks ist die aktuelle *Sicht* auf die Funktion `FFT`, d.h. die für den rekursiven Aufruf zu beachtenden Instanzen, angegeben. Im oben dargestellten Fall handelt es sich um die im Namensraum **A** sichtbaren Instanzen.

Die dargestellte Konfiguration wird immer dann verwendet, wenn die überladene Funktion `A:FFT` aufgerufen wird. In der Abbildung ist dies durch den von außen kommenden Aufrufpfeil angedeutet. Für Aufrufe der Funktion `A:FFT` werden durch die Funktionstabelle rekursive Aufrufe zur im Namensraum `A` definierten überladenen

[2] Die genaue Implementierung des `ovld` Konstrukts ist stark sprachabhängig. Eine Möglichkeit wird in Kapitel 6 vorgestellt.

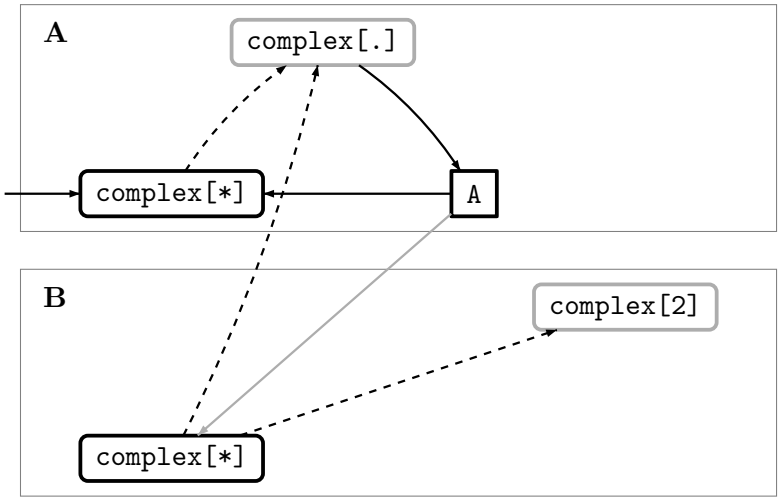


Abbildung 5.12.: Sicht der Funktion `FFT` im Modul **A**

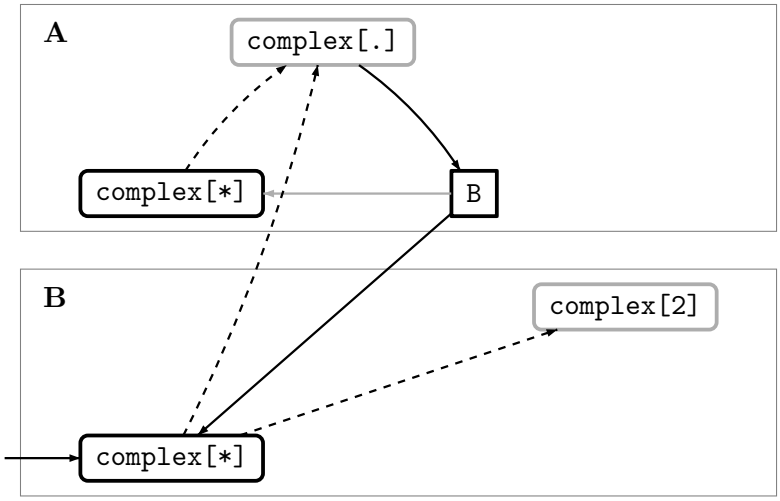


Abbildung 5.13.: Sicht der Funktion `FFT` im Modul **B**

Funktion `A:FFT` zurückgeleitet. Somit ist sicher gestellt, dass für rekursive Aufrufe nur die im Namensraum **A** sichtbaren Instanzen verwendet werden.

Abbildung 5.13 zeigt die Konfiguration für Aufrufe der Funktion `B:FFT`, in der Abbildung durch den auf `B:FFT` verweisenden Aufrufpfeil angedeutet. Die Funktionstabelle im Namensraum **A** ist nun auf die Konfiguration für die Sicht auf die Funktion `FFT` aus dem Namensraum **B** geändert. Der rekursive Aufruf durch die Instanz aus dem Namensraum **A** wird zur überladenen Funktion im Namensraum **B** weitergeleitet. Dadurch werden für einen Aufruf der Funktion `B:FFT` für die Rekursion immer

alle im Namensraum **B** sichtbaren Instanzen für den Dispatch berücksichtigt.

Der Wechsel der aktuellen Sicht innerhalb eines Namensraumes wird durch Funktionsaufrufe über Namensraumgrenzen hinweg eingeleitet. Bei jedem Namensraumwechsel werden dabei die entsprechenden Module für die Sicht des Zielnamensraums konfiguriert. In den beiden gezeigten Beispielen ist der Namensraumwechsel durch den jeweiligen Aufrufpfeil angedeutet. In Abbildung 5.12 überschreitet dieser die Grenze des Namensraumes **A**. Somit ist hier für die Funktionstabelle die Sicht für den Namensraum **A** zu wählen. Analog verhält es sich für den Aufruf der Funktion `FFT` im Namensraum **B**. Da hier die Grenze des Namensraums **B** überschritten wird, ist die Funktionstabelle für die entsprechende Sicht anzupassen.

Die vorgestellte Lösung skaliert auch für mehr als zwei Namensräume gut. Für jeden weiteren Namensraum ist lediglich eine neue Konfiguration der Funktionstabelle anzugeben. Die Sicht ist bei Namensraumwechseln jeweils für alle Module anzupassen, welche Instanzen der aufgerufenen überladenen Funktionen definieren. Des weiteren muss die ursprüngliche Sicht nach der Rückkehr aus einem entsprechenden Funktionsaufruf restauriert werden. Dadurch ist sichergestellt, dass für weitere Funktionsaufrufe wieder die Sicht des aktuellen Namensraums verwendet wird.

Das Konzept der Sichten auf überladene Funktionen ist für den Programmierer vollkommen transparent. Die jeweilige zu verwendende Sicht ist aus dem Kontext der Funktionsaufrufe ersichtlich. Ebenso kann die Menge der anzupassenden Funktionstabellen aus der Schnittstellendefinition der einzelnen Module ermittelt werden. Eine weitere Ergänzung der Syntax der Sprache SaC' ist damit nicht nötig.

Da potenziell nur mittels der `export` Anweisung zur Verfügung gestellte Funktionen um weitere Instanzen ergänzt werden können, sind auch nur diese für die jeweiligen Funktionstabellen eines Moduls zu berücksichtigen. Aufrufe mittels `provide` zur Verfügung gestellter Funktionen können weiter statisch einer Menge von Instanzen zugeordnet werden.

Durch Einführen von Sichten auf überladene Funktionen lässt sich somit das in Abschnitt 3.2 vorgestellte Problem der Rekursion durch Funktionsüberladung über Namensraumgrenzen hinweg lösen.

5.3.1. Erweiterte Semantik von SaC'

Im Folgenden wird aufbauend auf die bereits vorgestellte formale Semantik der Sprache SaC' eine formale Definition der Semantik des im vorigen Abschnitts beschriebenen Konzepts der Sichten vorgestellt.

Um die innerhalb eines Moduls für rekursive Aufrufe verwendete Sicht nachträglich anpassen zu können, werden die Module in $\mathcal{F}un$ entsprechend parametrisiert. Für jede mittels `export` exportierte Funktion werden dazu die zugehörigen Vorkommen in Funktionsanwendungen durch freie Variablen ersetzt. Diese können dann auf oberster Ebene des Moduls durch entsprechende Abstraktionen gebunden werden. Um eine spezifische Sicht zu erzeugen genügt es somit, die innerhalb der Sicht gültigen Definitionen der exportierten Funktionen dem parametrisierten Modul zuzuführen. Soll die lokale Definition eines Moduls verwendet werden, kann der ursprüngliche

$\mathbb{C}[\![$ *Provide Export Use Import Fundefs* $]\!] \rightsquigarrow$ *(Module)*

$\lambda\, Fun_1^*\,.\ \cdots\ \lambda\, Fun_k^*\,.$

```
letrec
```

$\quad \mathbb{C}[\![$ *Use* $]\!]$

$\quad \mathcal{OVLD} \left\{ \begin{array}{c} \mathbb{C}[\![\textit{ Import }]\!] \\ \mathbb{C}[\![\textit{ Fundefs }]\!] \end{array} \right\} [\mathtt{Fun}_1 \Leftarrow \mathtt{Fun}_1^*] \ \cdots\ [\mathtt{Fun}_k \Leftarrow \mathtt{Fun}_k^*]$

```
in < [ ℂ⟦ Provide ⟧, ℂ⟦ Export ⟧], [ ℂ⟦ Export ⟧]>
```

$\mathbb{C}[\![$ `use` *Mod* `:` `{` *Id*$_1$ `,` $\ldots$ `,` *Id*$_n$ `}` `;` $]\!] \rightsquigarrow$ *(Use)*

Id$_1$ = `(` Mod_μ^{local} `.` `#1` `)` `.` `'`*Id*$_1$`'`

$\vdots$

Id$_n$ = `(`Mod_μ^{local} `.` `#1` `)` `.` `'`*Id*$_n$`'`

$\mathbb{C}[\![$ `import` *Mod* `:` `{` *Id*$_1$ `,` $\ldots$ `,` *Id*$_n$ `}` `;` $]\!] \rightsquigarrow$ *(Import)*

Mod_μ^{view} = `(` $Mod_\mu\ \mathcal{F}(Fun_1) \cdots \mathcal{F}(Fun_k)$ `)`

Id$_1$ = `(` Mod_μ^{view} `.` `#2` `)` `.` `'`*Id*$_1$`'`

$\vdots$

Id$_n$ = `(` Mod_μ^{view} `.` `#2` `)` `.` `'`*Id*$_n$`'`

$\mathbb{C}[\![$ *Mod* `:` *Id* `(` *Expr*$_1$ `,` $\ldots$ `,` *Expr*$_n$ `)` $]\!] \rightsquigarrow$ *(Expr*$_5$*)*

`(` `(` `(` Mod_μ^{local} `.` `#1` `)` `.` `'`*Id*`'` `)` $\mathbb{C}[\![$ *Expr*$_1$ $]\!] \cdots \mathbb{C}[\![$ *Expr*$_n$ $]\!]$ `)`

wobei *Fun*$_1$, ..., *Fun*$_k$ die Bezeichner der vom jeweiligen Modul mittels `export` zur Verfügung gestellten Funktionen bezeichnet und $\mathcal{F}$ wie folgt definiert ist:

$$\mathcal{F}(Fun_i) = \begin{cases} Fun_i, & \text{falls Fun}_\text{i} \in \{Id_1\ \texttt{,}\ \ldots\ \texttt{,}\ Id_n\}; \\ \texttt{(}\ \texttt{(}\ Mod_\mu^{view}\ \texttt{.}\ \texttt{\#2)}\ \texttt{.}\ \texttt{'}Fun_i\texttt{'),} & \text{sonst.} \end{cases}$$

Abbildung 5.14.: Transformationsschema für SAC' inklusive Sichten

Zustand des Moduls durch eine Selbstanwendung wiederhergestellt werden.

Da im Allgemeinen nicht immer alle exportierten Funktionen auch wieder importiert werden, sind auch Mischformen möglich. Wird nur ein Teil der exportierten Funktionen importiert, werden nur die importierten Funktionen für die Sicht des importierenden Moduls konfiguriert. Alle anderen Parameter des Moduls werden durch die zugehörige lokale Definition des Moduls erfüllt.

Eine formale Beschreibung der Parametrisierung wird in Abbildung 5.14 vorgestellt. Sie zeigt die notwendigen Anpassungen der Transformationsregeln von SAC' in die Sprache $\mathcal{F}un$.

Die Erzeugung der Parametrisierung der Module ist in der Regel *Module* dargestellt. Innerhalb des Moduls werden in allen lokalen Funktionsdefinitionen (dies schließt mittels `import` aus anderen Modulen importierte Funktionen ein) die Funktionsbezeich-

```
letrec
   Mod_μ^1 = < [ ... ], [ ... ] >
   Mod_μ^1^local = ( Mod_μ^1 ( ( Mod_μ^1^local . #2) . 'Fun_1^1') ··· ( ( Mod_μ^1^local . #2) . 'Fun_k1^1 '))
   ⋮
   Mod_μ^n = < [ ... ], [ ... ] >
   Mod_μ^n^local = ( Mod_μ^n ( ( Mod_μ^n^local . #2) . 'Fun_1^n') ··· ( ( Mod_μ^n^local . #2) . 'Fun_kn^n '))
   Main_μ = < [ 'main' = ... ], [] >
in (Main_μ . #1) . 'main'
```

wobei Fun_1^i, ..., $Fun_{k_i}^i$ die vom Modul Mod^i mittels einer `export` Anweisung zur Verfügung gestellten Funktionen bezeichnet.

Abbildung 5.15.: Erweiterter Modulkontext in der Sprache $\mathcal{F}un$

ner Fun_i durch neue Bezeichner Fun_i^* ersetzt. In Abbildung 5.14 ist dies durch die Substitution $[Fun_i \Leftarrow Fun_i^*]$ dargestellt. Die so neu eingeführten Bezeichner werden dann auf oberster Ebene durch λ-Abstraktionen der Form $\lambda\, Fun_i^*\, . \cdots$ gebunden.

Da Module in $\mathcal{F}un$ somit parametrisiert sind, müssen die entsprechenden Parameter für Importe bzw. Funktionsanwendungen zuerst durch entsprechende Argumente besetzt werden. Dies ist in den restlichen Regeln aus Abbildung 5.14 dargestellt. Wird eine Funktion mittels einer `import` Anweisung in das aktuelle Modul importiert, muss eine für die lokale Sicht passende Konfiguration des exportierenden Moduls erzeugt werden. Die Konfiguration des exportierenden Moduls ist in der Regel *Import* enthalten. Die Sicht Mod_μ^{view} wird aus dem Modul Mod_μ erzeugt, indem die exportierten Funktionen mittels der lokalen Entsprechungen parametrisiert werden. Handelt es sich bei dem Parameter um eine importierte Funktion, wird dieser mittels der lokalen Definition der entsprechenden Funktion erfüllt. Somit wird innerhalb der erzeugten Sicht für Aufrufe der importierten Funktion die Definition des importierenden Moduls gewählt. Bei exportierten, aber nicht importierten Funktionen wird als Argument die Definition im exportierenden Modul gewählt. Aufrufe dieser Funktionen verwenden somit die im exportierenden Modul gültige Definition. Diese Unterscheidung ist durch die Funktion $\mathcal{F}$ abgebildet. Die so erzeugte Sicht wird dann für die weitere Überladung und Verwendung importierter Funktionen im importierenden Modul genutzt.

Wird eine solche Funktion vom importierenden Modul wieder exportiert, werden die entsprechenden Funktionsanwendungen durch die Regel *Module* erneut als Parameter des Moduls abstrahiert. Somit ist sichergestellt, dass die Rekursion durch Funktionsüberladung auch über mehrere Namensraumgrenzen erfolgen kann.

Bei einer mittels einer `use` Anweisung im aktuellen Modul sichtbar gemachten Funktion ist statt der Sicht des importierenden Moduls die Sicht des exportierenden Moduls zu verwenden. Diese wird global durch den Modulkontext der Sprache $\mathcal{F}un$ definiert. Der erweiterte Modulkontext ist in Abbildung 5.15 dargestellt. Die lokale Sicht $Mod_\mu^{i\,local}$ eines Moduls Mod^i wird analog zu den bereits beschriebenen angepassten Sichten erzeugt. Da keine Funktion importiert wird, werden alle Parameter durch die lokale Definition der entsprechenden Funktion im exportierenden Modul erfüllt.

Dadurch wird das ursprüngliche Modul wiederhergestellt. Die Regeln *Use* und $Expr_5$ wurden entsprechend angepasst. Statt des Modul-Records Mod_μ verwenden diese nun den jeweiligen Record Mod_μ^{local} für die lokale Sicht des Moduls.

Zusammen mit den in Abbildung 5.7 vorgestellten Regeln ergibt sich ein vollständiges Transformationsschema für SaC' inklusive Sichten.

5.3.2. Erweitertes Diamant-Import-Problem

Die in Abschnitt 5.2.3 vorgeschlagene Lösung des Diamant-Import-Problems, bei durch Importe entstandenen, sich überdeckenden Instanzen, die auf einer gemeinsamen Definition beruhen, die Herkunft einer Instanz bei der Überladung zu berücksichtigen, reicht im Kontext der im vorigen Abschnitt vorgestellten Sichten nicht aus.

Abbildung 5.16 zeigt ein Beispiel, welches sich an das in Abbildung 5.11 vorgestellte

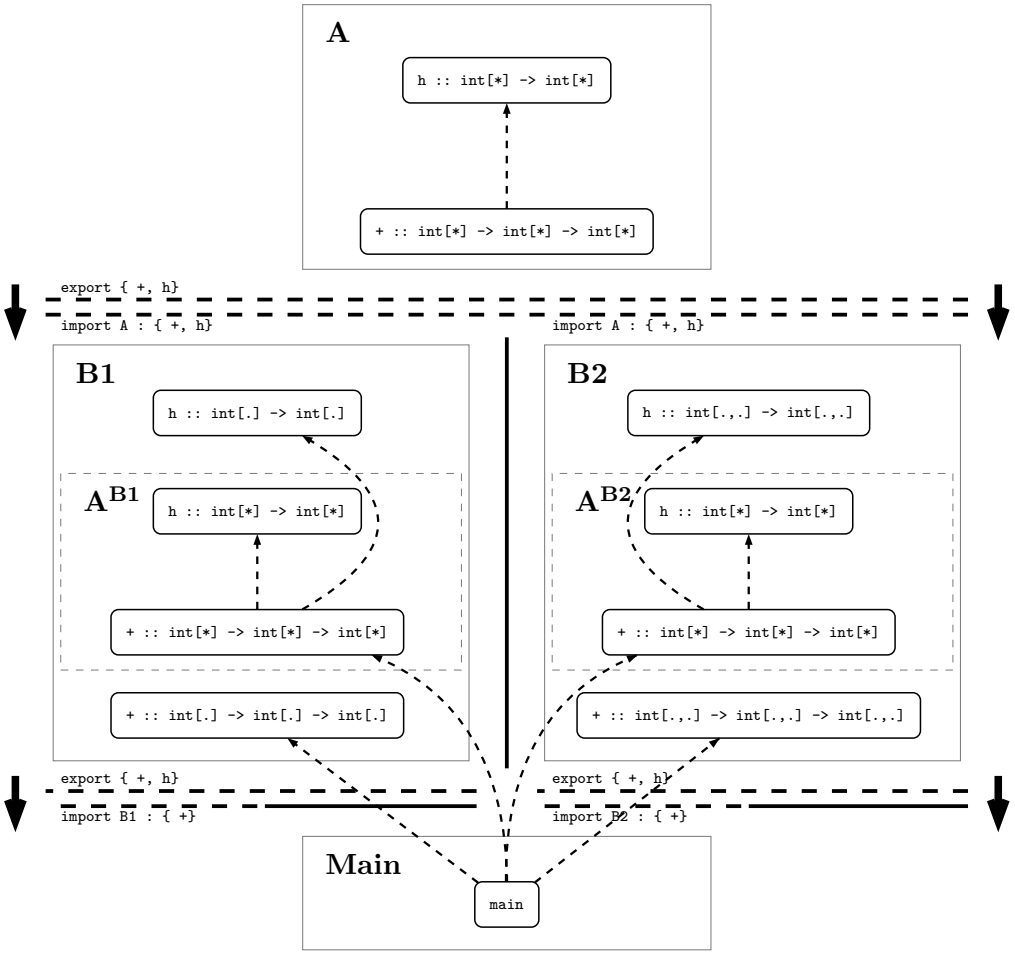


Abbildung 5.16.: Beispiel für das Diamant-Import-Problem im Kontext von Sichten

Beispiel anlehnt. Das im oberen Drittel abgebildete Modul **A** definiert eine Funktion `+` auf Arrays von Integer-Werten mit beliebiger Dimensionalität. Intern verwendet diese Funktion die Hilfsfunktion `h`, welche ebenfalls für Arrays von Integer-Werten mit beliebiger Dimensionalität definiert ist. Beide Funktionen werden mittels der Anweisung `export { +, h}` für andere Module zur Verfügung gestellt.

Im mittleren Drittel befinden sich analog zur Abbildung 5.11 die beiden Module **B1** und **B2**. Diese importieren jeweils beide vom Modul **A** zur Verfügung gestellten Funktionen und ergänzen beide um jeweils eine neue Instanz. Im Modul **B1** werden die Funktionen um Instanzen für Integer-Vektoren ergänzt. Die so entstandene Sicht auf die aus Modul **A** importierten Funktionen ist innerhalb des gestrichelten Blocks dargestellt. Die Bezeichnung $\mathbf{A}^{\mathbf{B1}}$ deutet an, dass es sich um die Sicht von Modul **B1** auf die Instanzen aus dem Modul **A** handelt. Insbesondere ist die für den Aufruf der Funktion `h` aus der im Modul **A** definierten Instanz der Funktion `+` hinzugekommene Dispatch-Möglichkeit durch einen gestrichelten Pfeil eingezeichnet. Ruft die Instanz der Funktion `+` die Funktion `h` auf, werden innerhalb des Moduls **B1** beide sichtbaren Instanzen für den Dispatch verwendet.

Im Modul **B2** werden zwei neue Instanzen für Integer-Matrizen ergänzt. Auch hier ist die Sicht des Moduls **B2** auf die Instanzen aus dem Modul **A** eingezeichnet. Analog zum Modul **B1** werden für Aufrufe der Funktion `h` beide im Namensraum **B2** sichtbaren Instanzen verwendet.

Im unteren Drittel befindet sich das Modul **Main**, welches das Hauptprogramm in Form der Funktion `main` enthält. Es importiert mittels der beiden Anweisungen `import B1 : { +}` und `import B2 : { +}` die jeweiligen Instanzen der Funktion `+` aus den Modulen **B1** und **B2**. Somit sind im Namensraum `Main` analog zum Beispiel aus Abschnitt 5.2.3 vier verschiedene Instanzen der Funktion `+` sichtbar, jeweils zwei aus dem Namensräumen **B1** und **B2**. Insbesondere überdecken sich auch im hier gezeigten Beispiel die beiden Instanzen für Integer-Arrays beliebiger Dimensionalität. Da beide jedoch auf die Definition im Modul **A** zurückgehen, würden durch die bisherige Lösung unter Beachtung der gemeinsamen Herkunft beide Instanzen zu einer zusammengefasst, d.h. eine der beiden Instanzen würde verworfen.

Bei genauerer Betrachtung der Situation in Abbildung 5.16 fällt jedoch auf, dass beide Instanzen trotz gleicher Herkunft nicht identisch sind. Im Modul **Main** werden zwar die jeweiligen Instanzen der Funktion `+` aus dem Namensräumen **B1** und **B2** importiert und somit zu einer gemeinsamen überladenen Funktion zusammengefasst, die Funktion `h` bleibt jedoch in ihrer jeweils lokalen Version erhalten. Diese wurde nicht in den Namensraum **Main** importiert. Sie ist in diesem nicht einmal sichtbar. Somit verwenden die beiden Instanzen trotz gleicher Herkunft verschiedene Versionen der überladenen Funktion `h`. Die Instanz aus dem Namensraum **B1** enthält einen Aufruf der Funktion `B1:h`, während die Instanz aus dem Namensraum **B2** einen Aufruf der Funktion `B2:h` verwendet.

Vergleicht man nun die beiden Funktionen `B1:h` und `B2:h`, fällt auf, dass diese aus unterschiedlichen Mengen von Instanzen bestehen. Für `B1:h` werden die Instanz aus dem Namensraum **A** und die lokale Instanz des Namensraumes **B1** für den Dispatch verwendet. Die Funktion `B2:h` verwendet hingegen statt der Instanz aus dem Na-

mensraum **B1** jene aus dem Namensraum **B2**. Es ist somit nicht sichergestellt, dass beide Funktionen semantisch äquivalent sind. Damit können die beiden Instanzen der Funktion `+` nicht zu einer gemeinsamen zusammengefasst werden, da sie durch den Aufruf der jeweiligen Funktion `h` ebenfalls nicht semantische äquivalent sein müssen.

Die Betrachtung der Herkunft einer Instanz reicht in diesem Falle also nicht aus. Wie das Beispiel zeigt, ist die Äquivalenz zweier Instanzen auch von den aufgerufenen Funktionen abhängig. Nur wenn deren Instanzen ebenfalls äquivalent sind, können zwei Instanzen zu einer gemeinsamen Instanz zusammengefasst werden. Letztendlich muss somit der komplette Aufrufgraph zweier Instanzen verglichen werden, um ihre Äquivalenz sicherstellen zu können.

Allerdings sind hierbei nur Aufrufe lokaler, d.h. im Namensraum in dem die Instanz definiert wurde liegender, Funktionen zu berücksichtigen. Aufrufe von Funktionen aus anderen Namensräumen können durch spätere Erweiterung der exportierten Funktion nicht mehr geändert werden. Dies wird durch die strikte Separierung der Namensräume verhindert. Somit sind diese Funktionen unabhängig von der aktuellen Sicht auf die aufrufende Instanz äquivalent.

Die Einschränkung auf lokale Funktionen reduziert die Menge der zu vergleichenden Instanzen erheblich, da im Allgemeinen ein Modul nur eine relativ kleine Menge von Instanzen enthält. In jedem Fall ist die Menge der zu vergleichenden Instanzen jedoch endlich und damit auch der zu vergleichende Aufrufgraph. Das Diamant-Import-Problem kann mit der vorgestellten Lösung somit auch im Kontext von Sichten gelöst werden.

5.4. Verfügbarkeit des Quelltextes

Wie in Abschnitt 3.3 gezeigt wurde, sind neben den Eigenschaften der Sprache SaC', hier insbesondere der Funktionsüberladung, auch Eigenschaften des compilierenden Systems bei der Entwicklung eines neuen Modulsystems zu berücksichtigen. Als besondere Anforderung wird in Abschnitt 3.3 die Verfügbarkeit des Quelltextes bereits übersetzter Module identifiziert. Dieser wird für verschiedene Optimierungen im Rahmen der Programmübersetzung benötigt. Wie sich gezeigt hat, reicht das Beilegen und Verwenden des ursprünglichen Quelltextes jedoch nicht aus. Einerseits würde dies die rein binäre Distribution von Modulen verhindern, andererseits widerspricht die Verwendung des Quelltextes dem Prinzip der separaten Compilation. Die Verwendung des bereits übersetzten Quelltextes für die Optimierungen schränkt diese wiederum stark ein. Somit gilt es, einen sowohl für die separate Compilation, als auch für die Code-Optimierung tragbaren Kompromiss zu finden.

Um die Mächtigkeit der Optimierungen zu erhalten, muss die zur Verfügung gestellte Darstellung des Moduls ein ausreichend hohes Abstraktionsniveau bieten. Idealerweise sollte dies auf dem hochsprachlichen Niveau der Sprache SaC' liegen. Die verwendete Darstellung muss somit alle Elemente der Sprache SaC' abbilden können.

Dem gegenüber steht der Anspruch, den Quelltext eines Moduls nur einmal zu übersetzen. Da für die Optimierungen eine hochsprachliche Darstellung nötig ist, ist

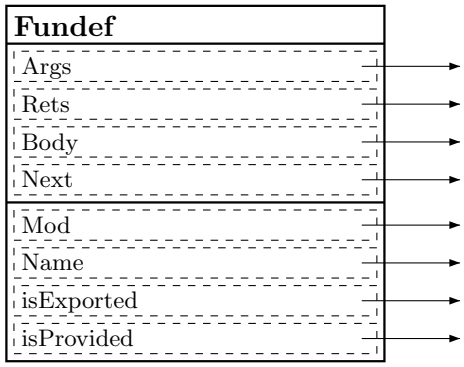


Abbildung 5.17.: Vereinfachte schematische Darstellung des `Fundef` Knotens des abstrakten Syntaxbaumes

dieses Ziel nicht zu erreichen, ohne auf die Optimierungen zu verzichten. Allerdings finden Teile der Übersetzung selbst auf einem hochsprachlichem Niveau statt. Als Beispiel ist hier die Quelltextvereinfachung und Typinferenz zu nennen. Um diese Übersetzungsschritte nicht erneut vollführen zu müssen, bietet es sich an, ihre Ergebnisse in die von den Optimierungen verwendete Darstellung mit aufzunehmen. Insbesondere die aufwändigen Analysen der Typinferenz müssten somit nur einmal erfolgen.

In der bisher vorgestellten Syntax der Sprache SaC' lassen sich diese Analyseergebnisse nicht darstellen. Um den bereits vorcompilierten Code eines Programms als Quelltext darzustellen, wäre somit eine Erweiterung der Syntax von SaC' nötig. Da der zu erzeugende Zwischencode jedoch nur während der Übersetzung benötigt wird und automatisch aus dem Quelltext eines Moduls erzeugt wird, d.h. insbesondere nicht von einem Programmierer zu erstellen ist, ist die Lesbarkeit hier nicht von entscheidender Rolle. Gerade im Hinblick auf den bereits in Abschnitt 3.3 erwähnten Schutz des geistigen Eigentums bei binär ausgelieferten Modulen ist eine gute Lesbarkeit sogar nicht wünschenswert. Somit bietet es sich an, statt eines vom Menschen lesbaren Quelltextformats, eine kompaktere, nur maschinenlesbare Repräsentation des Zwischencodes zu wählen.

Intern wird der Quelltext eines Programms bzw. Moduls durch einen *abstrakten Syntaxbaum*, im Folgenden mit AST abgekürzt, dargestellt. Vereinfacht dargestellt wird jedem Element der Sprache ein Knoten in diesem Baum zugeordnet. Die Wurzel bildet dabei der Programm- bzw. Modulknoten, welcher als Söhne die einzelnen Funktionsdefinitionen enthält. Ein solcher Definitionsknoten wiederum enthält weitere Söhne, welche die Definition der entsprechenden Funktion bilden. In Abbildung 5.17 ist ein vereinfachter Ausschnitt des abstrakten Syntaxbaumes graphisch dargestellt. Es handelt sich hierbei um den `Fundef` Knoten, welcher die Wurzel einer Funktionsdefinition bildet. Die gestrichelten Kästchen symbolisieren die einzelnen Söhne des Knotens. Eine genauere Beschreibung der Bedeutung dieser Söhne wird in Abschnitt 6.2.1 gegeben.

Durch die verschiedenen Analysephasen während der Übersetzung werden an den

aus dem Quelltext erstellten Baum weitere Söhne angefügt. Diese bilden die Ergebnisse der Analysen ab. Um das Ergebnis der Übersetzung nach den Analysen auf hochsprachlicher Ebene darzustellen, genügt es somit den entsprechenden abstrakten Syntaxbaum zu sichern. Wird während der Optimierungen der Quelltext einer Funktionsdefinition benötigt, kann der entsprechende Teil des gesicherten AST rekonstruiert werden und für die Optimierungen verwendet werden. Da er bereits die Ergebnisse der Analysen auf hochsprachlicher Ebene enthält, können diese Analysen für die so erhaltene interne Darstellung des Quelltextes entfallen. Dieser Ansatz bietet somit einen Kompromiss zwischen der nötigen erneuten Verarbeitung des Quelltextes und der gewünschten weitgehend separaten Übersetzung einzelner Module.

Durch die vollständige Sicherung des AST wird jedoch eine starke Abhängigkeit sowohl des Modulsystems, als auch der erzeugten Module von der internen Repräsentation des Quelltextes eingeführt. Eine Änderung oder Ergänzung der internen Darstellung führt somit direkt zu Inkompatibilitäten zwischen erzeugten Modulen, als auch zu einem nicht unerheblichen Anpassungsbedarf seitens des Modulsystems.

Auf Anwenderseite stellt die Inkompatibilität mit verschiedenen Compilerversionen erzeugter Module ein beherrschbares Problem dar. Da die Releasezyklen für Compiler im Allgemeinen eher konservativ sind, ist nicht damit zu rechnen, dass die Module häufig neu erzeugt werden müssen. Desweiteren besteht im Allgemeinen auch kein Migrationszwang, d.h. die erneute Compilation lässt sich durch beibehalten der verwendeten Compilerversion verhindern. Es ist lediglich sicher zu stellen, das inkompatible Module zu einer Fehlermeldung führen, um unerwartetes Laufzeitverhalten zu vermeiden.

Für die Entwicklung des Compilers stellt die starke Abhängigkeit des Modulsystems von der internen Darstellung des Quelltextes jedoch einen bedeutenden Nachteil dar. Durch den zu erwartenden hohen Aufwand für die Anpassung des Modulsystems an eine modifizierte interne Darstellung ist eine signifikante Verlangsamung des Entwicklungsprozesses zu erwarten. Daher ist für die Implementierung die Entkopplung des Modulsystems von der genauen Ausgestaltung des AST wünschenswert.

5.4.1. Entkopplung des Modulsystems

Um die gewünschte Entkopplung des Modulsystems zu ermöglichen, wird im Folgenden eine Abstraktionsschicht zwischen dem AST und den einzelnen Subsystemen des Compilers vorgeschlagen. Diese basiert auf einer Meta-Beschreibung der Struktur des AST und einer darauf aufbauenden Compiler Infrastruktur, welche sich automatisch an Veränderung der Beschreibung anpasst. Abbildung 5.18 zeigt den schematischen Aufbau der zu schaffenden Infrastruktur. Die einzelnen Schichten sind dabei durch rechteckige Blöcke angedeutet. Die gestrichelten Blöcke symbolisieren Teile der Compiler Infrastruktur. Diese sollen in einer späteren Implementierung automatisch aus einer formalen Beschreibung des Syntaxbaumes abgeleitet werden.

Um die gewünschte Entkopplung zu erreichen, müssen für alle Zugriffe auf den AST geeignete Schnittstellen definiert werden. Dies entspricht der in der Abbildung angegebenen Zugriffs-Schicht. Dabei sind die Schnittstellen derart zu entwerfen, dass eine

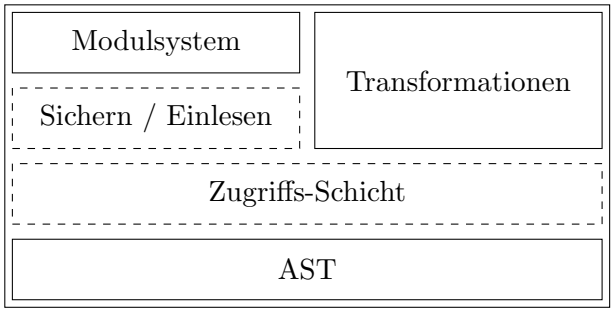


Abbildung 5.18.: Schematische Darstellung der Abstraktionsschicht

Ergänzung weiterer Knoten im AST nur in einer lokalen Veränderung der Schnittstelle resultiert. Diese Veränderung sollte sich möglichst nur auf die Teile des Compilers erstrecken, welche die im ergänzten Knoten enthaltenen Daten auch tatsächlich verarbeiten. Dadurch ist sichergestellt, dass Erweiterungen einzelner Teile des Compilers nur minimale Anpassungen im restlichen Compiler erfordern.

Das Modulsystem als solches verarbeitet kaum im AST enthaltene Daten. Es muss jedoch in der Lage sein, den kompletten AST zu sichern und die erstellte Sicherung wieder einzulesen. In der Abbildung ist dies durch die Sichern/Einlesen-Schicht angedeutet. Um das Modulsystem von Änderungen der Struktur des AST zu entkoppeln, muss diese Komponente ebenfalls durch die Compiler Infrastruktur zur Verfügung gestellt werden. Die entsprechende Schnittstelle zum Sichern und Einlesen des AST ist so zu gestalten, dass eine Veränderung der Struktur des AST sie nicht beeinflusst.

Durch die so erreichte Trennung zwischen der Darstellung des AST und den die enthaltenen Daten verarbeitenden Teilen des Compilers ist sicher gestellt, dass die starke Abhängigkeit des Modulsystems vom AST sich nicht nachteilig auswirkt. Die Umsetzung des hier vorgestellten Ansatzes wird in Kapitel 6 genauer beschrieben.

5.5. Funktionsspezialisierung und Adaptive Module

In Abschnitt 3.4 wurden die durch die Funktionsspezialisierung an das Modulsystem gestellten Anforderungen beschrieben. Für das Problem der Verfügbarkeit des Quelltextes wurde bereits im vorigen Abschnitt eine Lösung entwickelt. Um die Spezialisierung von Funktionen auch über Modulgrenzen hinweg zu ermöglichen, muss jedoch auch die nachträgliche Ergänzung der in einem Modul sichtbaren Instanzen ermöglicht werden. Im Folgenden werden zwei mögliche Lösungen des Problems vorgestellt und ihre Vor- und Nachteile gegeneinander abgewogen.

Lokale Spezialisierung

Die *lokale Spezialisierung* verwendet das in Abschnitt 5.2 vorgestellte Konzept der Sichten, um die Menge der Instanzen einer Funktion nachträglich zu verändern. Wird aufgrund eines Funktionsaufrufs in einem Modul eine Funktion aus einem anderen Modul spezialisiert, so werden die neuen Instanzen lokal im Namensraum des aufru-

fenden Moduls hinzugefügt. Dies kann, analog zur Ergänzung von Instanzen durch eine `import` Anweisung, durch Erzeugen einer neuen Sicht erfolgen. Die lokale Spezialisierung stellt somit keine weiteren Anforderungen an das Modulsystem. Die bereits vorhandenen Verfahren können weiter genutzt werden. Dies minimiert den Aufwand für die Implementierung dieser Lösung.

Da durch das Hinzufügen von spezialisierten Instanzen die Semantik der überladenen Funktion nicht geändert wird, widerspricht die Erzeugung einer Sicht ohne `import` Anweisung auch nicht der strikten Separierung der Namensräume. Die erzeugte Sicht ist semantisch zum Ursprungsnamensraum äquivalent, lediglich die Laufzeiteffizienz wird durch die hinzugefügten Instanzen verbessert.

Allerdings führt die lokale Spezialisierung zu einer Vergrößerung des erzeugten Codes, da die jeweiligen Spezialisierungen in jedem Namensraum, in dem die entsprechende Funktion aufgerufen wird, erneut erzeugt werden müssen. Dem kann nur durch die Erzeugung gewisser häufig benötigter Spezialisierungen auf Vorrat entgegnet werden. Die Entscheidung, welche Spezialisierungen hierfür zu erzeugen sind, ist nicht ohne weitere Analysen der zu erwartenden Funktionsanwendungen möglich. Entsprechende Verfahren müssten demnach noch entwickelt werden.

Globale Spezialisierung

Eine weitere Lösung ist die *globale Spezialisierung*. Dabei werden die durch Spezialisierung generierten Instanzen der Menge der im definierenden Modul sichtbaren Instanzen hinzugefügt. Dies erfordert jedoch Anpassungen des Modulsystems. Insbesondere muss die Möglichkeit geschaffen werden, eine Instanz einem bereits übersetzten Modul hinzuzufügen. Eine erneute Übersetzung des Moduls kommt aus Gründen der gewünschten separaten Compilation nicht in Frage.

Um die hinzugefügten Instanzen tatsächlich global sichtbar zu machen, müssen auch alle auf dem Modul basierenden Sichten dahingehend angepasst werden, dass sie die neu hinzugekommenen Instanzen berücksichtigen. Der implementatorische Aufwand ist dementsprechend hoch.

Des weiteren stellt die globale Spezialisierung besondere Anforderungen an das dem Modulsystem zu Grunde liegende Bibliotheksformat. Es muss das nachträgliche Hinzufügen bzw. Ersetzen von Funktionen ermöglichen. Dies ist bei den Bibliotheksformaten der gängigen Betriebssysteme nicht vorgesehen. Somit muss für die Umsetzung der globalen Spezialisierung ein entsprechendes proprietäres Bibliotheksformat entwickelt werden.

Eine konsequente Umsetzung der globalen Spezialisierung führt zur Idee der *adaptiven Module*. Durch die Ergänzung spezialisierter Instanzen passen sich die Module zunehmend dem bearbeiteten Problemraum an. Dies wird am Beispiel der in Abschnitt 3.2 vorgestellten schnellen Fourier Transformation deutlich. Diese wurde allgemein für Vektoren beliebiger Länge spezifiziert. Wird das entsprechende Modul nun häufig für die Bearbeitung von Vektoren einer spezifischen Länge verwendet, werden zunehmend die benötigten Instanzen ergänzt. Die Laufzeiteffizienz steigt dementsprechend für die Berechnung auf solchen Vektoren mit zunehmender Verwendung an. Das Modul adaptiert somit an den gegebenen Problemraum der Berechnung der FFT auf

Vektoren dieser Länge. Da die Spezialisierung global operiert, erfolgt diese Anpassung auch über die Grenzen einzelner Programme hinweg.

Zusammenfassung

Zusammenfassend kann gesagt werden, dass beide Lösungen die Spezialisierung über Modulgrenzen erlauben. Für die lokale Spezialisierung spricht der geringe implementatorische Aufwand, der allerdings durch eine Vergrößerung des erzeugten Codes erkauft wird. Die globale Spezialisierung vermindert die Vergrößerung des Codes deutlich. Allerdings ist der implementatorische Aufwand bedeutend größer, da für diese Lösung ein proprietäres Bibliotheksformat entwickelt werden muss. Ein besonderer Vorteil der globalen Spezialisierung liegt im Konzept der adaptiven Module.

Langfristig ist somit die globale Spezialisierung der lokalen Spezialisierung vorzuziehen. Zur Implementierung der in Kapitel 3 geforderten Eigenschaften genügt jedoch auch eine Umsetzung der lokalen Spezialisierung.

5.6. Fazit

Die hier vorgestellten Lösungen erfüllen die in Kapitel 3 an das Modulsystem gestellten Anforderungen. Es werden Verfahren für die granulare Steuerung der Sichtbarkeit von Funktionsinstanzen innerhalb verschiedener Namensräume beschrieben. Die dafür nötige Erweiterung der Syntax der Sprache SaC besteht im wesentlichen aus Sprachkonstrukten um die Schnittstelle zwischen einzelnen Namensräumen zu definieren. Sie bieten eine für den Programmierer leicht verständliche Semantik, welche durch eine formale Beschreibung untermauert wurde. In einem gesonderten Abschnitt wird das Diamant-Import-Problem behandelt und eine Lösung für den vorgestellten Ansatz entwickelt. Die in Kapitel 3 vorgestellten Szenarien können mit den vorgestellten Erweiterungen vollständig in der Sprache SaC abgebildet werden.

Außerdem wird ein Verfahren zur Sicherung des Quelltextes vorgestellt. Dabei wird insbesondere auf eine leichte Erweiterbarkeit der Darstellung in Hinsicht auf die weitere Entwicklung des Compilers `sac2c` geachtet. Die eingeführte Lösung passt sich weitgehend automatisch an Erweiterungen der Sprache bzw. des Compilers an.

Abschließend werden zwei Lösungen für das Problem der Funktionsspezialisierung über Namensraumgrenzen hinweg vorgestellt und ihre Eigenschaften gegeneinander abgewogen.

6. Implementierung

Die im Rahmen dieser Arbeit vorgestellten Verfahren für die Überladung von Funktionen über Namensraumgrenzen und die Optimierung über Modulgrenzen sind als Erweiterung des durch das SAC-Projekt (*http://www.sac-home.org*) entwickelten Compilers `sac2c` implementiert. Dieses Kapitel widmet sich diesen Erweiterungen. Der hier vorgestellte Teil der Implementierung bezieht sich dabei auf die in der Sprache SAC' vorhandenen Sprachelemente. Das im Rahmen der Diplomarbeit implementierte Modulsystem umfasst zusätzlich die in SAC' fehlenden Bestandteile von SAC, so zum Beispiel benutzerdefinierte Typen und externe Funktionen. Die hierbei verwendeten Konzepte wurden bereits in [Gre96] vorgestellt, so dass auf eine weitere Beschreibung im Rahmen dieser Arbeit verzichtet wird.

Im folgenden Abschnitt wird zuerst ein Überblick über den Aufbau des Compilers gegeben. In darauf folgenden Abschnitten werden dann die einzelnen Erweiterungen beschrieben. Dies sind im Detail eine neue Compiler-Infrastruktur um die Sicherung des abstrakten Syntaxbaumes zu ermöglichen und die Implementierung der vier hinzugefügten Sprachelemente des Modulsystems. Abschließend wird ein Fazit der Implementierung gezogen.

6.1. Das compilierende System

Der Compiler `sac2c` übersetzt Programme der Quellsprache SAC in die Zielsprache C. Dabei findet ein *multi-pass* Übersetzungsverfahren Verwendung. In einer ersten Phase wird der Quelltext des SAC-Programms bzw. Moduls eingelesen und in eine interne Darstellung, den abstrakten Syntaxbaum transformiert. Dieser wird dann in den folgenden Phasen analysiert und in weitere Zwischendarstellungen transformiert. Die abschließende Phase generiert aus dem abstrakten Syntaxbaum ANSI konformen C-Quelltext [Bri03], welcher mittels eines entsprechenden C-Compilers in ein ausführbares Programm bzw. in eine Bibliothek übersetzt werden kann.

Abbildung 6.1 zeigt eine schematische Darstellung der einzelnen Phasen[1]. Innerhalb der ersten Phase (*Scanner/Parser*) wird aus dem Quelltext der abstrakte Syntaxbaum aufgebaut. In diesem Stadium der Compilation handelt es sich dabei noch um eine 1:1 Abbildung des SAC-Quelltextes. Die darauf folgende *LaC2Fun* (*Loops and Conditionals to Functions*) Phase konvertiert diese in eine rein funktionale Darstellung, in welcher Schleifen durch tail-end-rekursive Funktionen ersetzt und Konditionale in spezielle Konditionalfunktionen transformiert sind [MW01]. Unter Verwendung dieser

[1] Die Darstellung dient lediglich der Illustration und erhebt nicht den Anspruch auf Vollständigkeit.

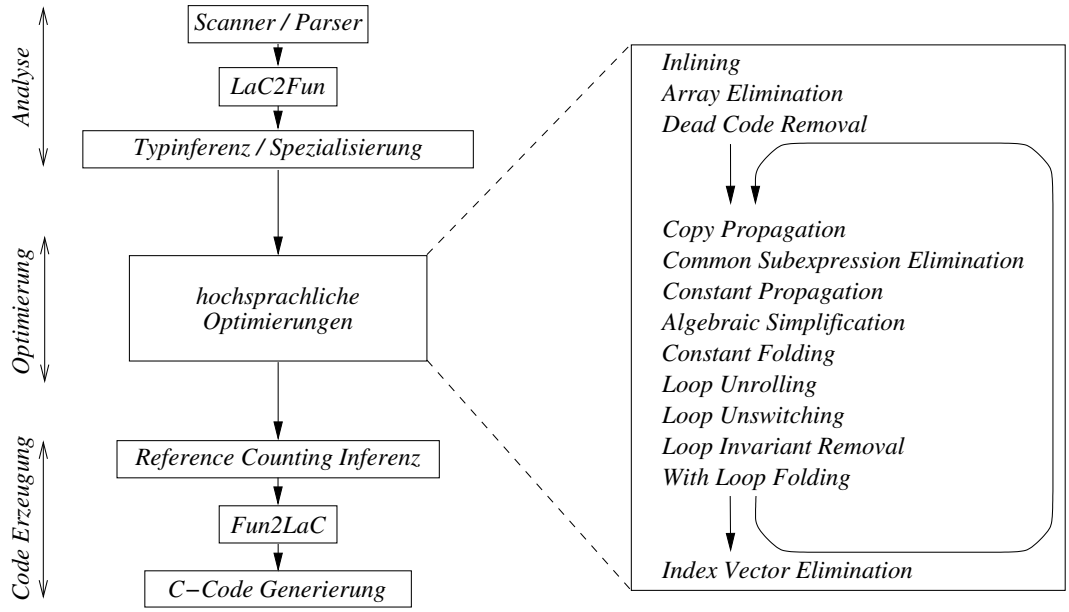


Abbildung 6.1.: Schematische Darstellung der Übersetzungsphasen

Darstellung findet dann in einer weiteren Phase die Typinferenz und Funktionsspezialisierung statt. Diese Phase beinhaltet des weiteren die Erzeugung der für den Dispatch verwendeten Wrapper-Funktionen [Kre03]. Zusammenfassend können diese Phasen als Quelltextanalyse bezeichnet werden.

Nach der Quelltextanalyse folgen die hochsprachlichen Optimierungen. In der rechten Hälfte der Abbildung 6.1 sind einige dieser Optimierungen aufgezählt. Unter anderem beinhalten sie das im Kapitel 3 vorgestellte Funktionsinlining. Die Optimierungen sind in eine Fixpunkt-Iteration (in Abbildung 6.1 durch die Schleife dargestellt) bezüglich der Programmtransformation gekapselt.

Im Anschluss an die Optimierungen findet die *Reference Counting* Inferenz statt [GT04]. Abschließend wird in weiteren Phasen die rein funktionale Darstellung wieder in eine Darstellung mit Schleifen und Konditionalen transformiert und aus dieser C-Code generiert.

Der Compiler `sac2c` verwendet als Dateiformat für die erzeugten Bibliotheken die vom jeweiligen Betriebssystem unterstützte Form von C-Bibliotheken. Da C als Zielsprache verwendet wird, können diese somit durch die entsprechenden Entwicklungswerkzeuge des verwendeten C-Compilers verarbeitet werden. Ein Großteil der Aufgaben eines Modulsystems, insbesondere das Auflösen von Abhängigkeiten und das Linken des ausführbaren Programmes, können somit auf die Ebene der Sprache C verlagert werden. Das hier vorgestellte Modulsystem operiert daher nur auf dem hochsprachlichen Niveau von SaC.

6.2. Sichern des abstrakten Syntaxbaumes

Um die Ergebnisse der Code-Analyse sichern zu können, findet die Sicherung in einer gesonderten Compiler-Phase nach den Analysen statt. Obwohl auch ein Sichern nach der Phase der Optimierungen möglich wäre, wird dieser Ansatz verworfen. Das Sichern der Ergebnisse dieser Phase birgt keinen Vorteil bezüglich der separaten Compilation. Da die Ergebnisse der Optimierungen stark vom umgebenden Kontext abhängig sind, kann eine erneute Optimierung des Quelltextes auch durch Sichern der Ergebnisse vorheriger Optimierungen nicht verhindert werden. Des weiteren bringen viele der Optimierungen eine Vergrößerung des zu sichernden Syntaxbaumes mit sich. Ein Beispiel hierfür ist das bereits in Kapitel 3 vorgestellte Funktionsinlining, welches Teile des Syntaxbaumes vervielfältigt. Eine Sicherung nach dieser Optimierung würde zu Redundanzen innerhalb der gesicherten Darstellung führen und somit den Platzbedarf der Module unnötig erhöhen.

Wie in Kapitel 5 beschrieben, ist für die Sicherung des abstrakten Syntaxbaumes eine Compiler-Infrastruktur nötig, welche den Zugriff und das Sichern bzw. Einlesen des Syntaxbaumes vom restlichen Compiler entkoppelt. Dazu ist in einem ersten Schritt eine formale Beschreibung des AST zu entwickeln. Als Beschreibungssprache wird hierfür XML [DuC98] gewählt.

Für die Wahl von XML spricht vor allem die weit gehende Standardisierung und Etablierung in vielen Bereichen der Informationsverarbeitung. In Folge dessen sind viele der für die Compiler-Infrastruktur benötigten Entwicklungswerkzeuge bereits vorhanden und für eine Vielzahl von Plattformen verfügbar. Insbesondere existiert mit XSLT [Kay04] eine flexibel einsetzbare Transformationssprache, um aus XML-Dokumenten den für die Compiler-Infrastruktur benötigten C-Quellcode zu generieren[2]. Auch die Möglichkeit, mittels XML-Schema [vdV02] eine formale Beschreibung der Syntax eines XML-Dokuments angeben zu können, hat sich im Laufe der Implementierung als vorteilhaft erwiesen. Gerade bei einer komplexen und zentralen Struktur wie dem abstrakten Syntaxbaum eines Compilers ist es von besonderer Wichtigkeit, Fehler in der Spezifikation leicht aufspüren zu können.

Ein weiterer Vorteil von XML ist die Möglichkeit, die spezifizierte Beschreibung des Syntaxbaumes ohne großen implementatorischen Aufwand visualisieren zu können. Viele heute verfügbaren Webbrowser[3] unterstützen bereits nativ die Anzeige von XML-Dokumenten. Ein Beispiel für die im Rahmen dieser Arbeit entwickelte Visualisierung der Spezifikation des Syntaxbaumes findet sich in Anhang B.3.

Im Anschluss an die Implementierung des Modulsystems hat sich ein weiterer Vorteil von XML als Beschreibungssprache gezeigt. XML wurde mit dem Ziel entwickelt, modulare und erweiterbare Beschreibungen von Daten spezifizieren zu können. Insbesondere der Aspekt der Erweiterbarkeit hat sich in der Praxis als hilfreich erwiesen. Obwohl die XML-Beschreibung des Syntaxbaumes mittlerweile auch an anderen Stel-

[2] Im Rahmen dieser Arbeit wird der frei erhältliche Sablotron XSLT Processor (*http://www.gingerall.com/charlie/ga/xml/p_sab.xml*) verwendet.

[3] Als Beispiele seien hier die Produkte `Firefox` der Mozilla Foundation (*http://www.mozilla.org*) und `Internet Explorer` von Microsoft (*http://www.microsoft.com*) genannt.

len im Compiler Verwendung findet, sind durch die damit einhergehende Erweiterung der Beschreibung keine Anpassungen an der Compiler-Infrastruktur nötig.

In den folgenden Abschnitten werden die einzelnen Komponenten der Implementierung der einzelnen Schichten der Infrastruktur beschrieben. Dabei kann jeweils nur ein grober Überblick über die verwendeten Verfahren gegeben werden. Die vollständige Implementierung ist auf Anfrage beim SAC-Projekt (*http://www.sac-home.org*) erhältlich.

6.2.1. Beschreibung des abstrakten Syntaxbaumes

Zentraler Bestandteil der erstellten Compiler-Infrastruktur ist die formale Beschreibung des abstrakten Syntaxbaumes. Die zugehörige XML-Schema Definition ist im Anhang B.1 angegeben.

Die eigentlichen Sprachelemente der Sprache SAC werden im abstrakten Syntaxbaum durch Knoten abgebildet. In der XML-Beschreibung entspricht einem Knoten das `node` Element. Abbildung 6.2 zeigt als Beispiel eine vereinfachte Version des Knotens für eine Funktionsdefinition. Diese entspricht der in Abbildung 5.17 angegebenen graphischen Darstellung. Die vollständige Definition des `Fundef` Knotens ist im Angang B.2 angegeben.

Jedem Knoten wird ein eindeutiger Bezeichner zugeordnet, welcher durch das Attribut `name` in der Beschreibung angegeben wird. Im angegebenen Beispiel ist dies der Bezeichner `Fundef`. Die jeweiligen Bezeichner werden innerhalb der XML-Beschreibung verwendet, um Referenzen auf einen bestimmten Knotentyp abzubilden.

Um eine Baumstruktur abbilden zu können, hat jeder Knoten eine Menge von möglichen Söhnen. Diese sind im Unterelement `sons` kodiert. Jeder Sohn ist hier mittels des `son` Elements angegeben. Im Falle einer Funktionsdefinition sind dies die Argumente (`Args`), die Rückgabewerte (`Rets`), der Funktionsrumpf (`Body`) und die nächste Funktionsdefinition (`Next`). Der jeweilige Bezeichner ist wieder durch das Attribut `name` kodiert. Das Attribut `mandatory` legt fest, ob ein Sohn zwingend erforderlich ist. Im Falle einer Funktionsdefinition ist dies für die Rückgabewerte und den Funktionsrumpf der Fall. Im Gegensatz dazu sind Argumente, ebenso wie weitere Funktionsdefinitionen, optional. Für jeden Sohn wird zusätzlich die Menge der erlaubten Zielknoten durch das Element `targets` angegeben. Diese Informationen werden bei der späteren Erzeugung des Syntaxbaumes verwendet, um die Konsistenz sicherzustellen.

Neben Söhnen hat jeder Knoten eine Menge von Attributen, in der XML-Beschreibung durch das `attributes` Element dargestellt. Diese kodieren Eigenschaften eines Knotens. Im Falle des im Beispiel angegebenen `Fundef` Knotens sind dies der Name der Funktion (`name`), sowie der Namensraumbezeichner (`Mod`). Diese sind durch ein entsprechendes `attribute` Element spezifiziert. Im Gegensatz zu Söhnen gibt es bei Attributen keinen Zielknoten, sondern eine Typangabe. Dadurch können Attribute im abstrakten Syntaxbaum auf beliebige Datenstrukturen zeigen. Im gegebenen Beispiel sind dies Strukturen vom Typ `String`, welche dem C Datentyp `char*` entsprechen. Die erlaubten Typangaben sind ebenfalls innerhalb der XML-Beschreibung kodiert.

```
<node name="Fundef">
  <sons>
    <son name="Args" mandatory="no">
      <target>
        <node name="Arg" />
      </target>
    </son>
    <son name="Rets" mandatory="yes">
      <target>
        <node name="Ret" />
      </target>
    </son>
    <son name="Body" mandatory="yes">
      <target>
        <node name="Block" />
      </target>
    </son>
    <son name="Next" mandatory="no">
      <target>
        <node name="Fundef" />
      </target>
    </son>
  </sons>
  <attributes>
    <attribute name="Name" mandatory="yes">
      <type name="String" />
      <phases>
        <all />
      </phases>
    </attribute>
    <attribute name="Mod" mandatory="yes">
      <type name="String" />
      <phases>
        <all />
      </phases>
    </attribute>
  </attributes>
  <flags>
    <flag name="IsExported" />
    <flag name="IsProvided" />
  </flags>
</node>
```

Abbildung 6.2.: Darstellung des `Fundef` Knotens in XML.

Auf die Details wird an dieser Stelle verzichtet. Eine weitere in der Beschreibung eines Attributs erfasste Information ist die Gültigkeit während der einzelnen Übersetzungsphasen. Diese ist im `phases` Element kodiert. Im Falle der hier angegebenen Attribute gibt es keine Einschränkung der Gültigkeit, angegeben durch das `all` Element.

Als dritte Komponente enthält eine Knotenbeschreibung das `flags` Element. Dieses beinhaltet boolsche Variablen des Knotens. Im Falle des `Fundef` Knotens sind dies

die beiden Werte `IsExported` und `IsProvided`, welche die zur Funktion gehörende Schnittstellendefinition im Syntaxbaum abbilden.

Eine besondere Klasse von Attributen bilden die Attribute vom Typ `Link`. Diese kodieren Querverweise innerhalb des Syntaxbaumes. Eine Anwendung von Querverweisen sind Funktionsaufrufe. Die entsprechenden Knoten des abstrakten Syntaxbaumes beinhalten einen Verweis auf die dem Aufruf zugeordnete Funktionsdefinition. Diese Information werden nicht als Sohn kodiert, da sie ansonsten die Baumstruktur auf den Söhnen zerstören würde. Bezieht man die Attribute mit in die Betrachtung ein, handelt es sich beim abstrakten Syntaxbaum durch die Querverweise um einen gerichteten Graphen.

6.2.2. Implementierung der Zugriffs-Schicht

Aus der im vorigen Abschnitt vorgestellten XML-Beschreibung des abstrakten Syntaxbaumes kann nun in einem weiteren Schritt die Zugriffs-Schicht generiert werden.

Um den restlichen Compiler von der Darstellung des abstrakten Syntaxbaum zu entkoppeln, wird dieser durch einen abstrakten Datentyp `node*` in C abgebildet. Dadurch, dass die genaue Implementierung dieses Datentyps nicht bekannt ist, werden direkte Zugriffe auf den Syntaxbaum verhindert. Alle Operationen, wie das Erzeugen und Löschen von Knoten, aber auch der Zugriff auf Söhne und Attribute eines Knotens, sind durch entsprechende Funktionen bzw. Makros gekapselt.

Diese Kapselung wird ermöglicht, indem aus der XML-Beschreibung mittels XSLT-Transformationen der entsprechende C-Code generiert wird. Dieser definiert für jeden Knotentyp eine Konstruktor- und Destruktorfunktion, sowie Zugriffsmakros für die entsprechenden Söhne und Attribute. Dadurch können unabhängig von der internen Darstellung Knoten erzeugt und wieder gelöscht werden. Da der Zugriff auf Söhne und Attribute ebenfalls gekapselt ist, muss bestehender Code bei Ergänzungen der XML-Beschreibung nicht geändert werden. Lediglich die Bestandteile des Compilers, welche auf die neuen Söhne und Attribute zugreifen, sind anzupassen.

Um die Implementierung des Compilers auch von der Menge der Knoten und Söhne unabhängig zu machen, wird weiter ein allgemeines Traversiersystem installiert. Ist eine Phase des Compilers nur auf Teilen der Knoten definiert, werden alle anderen Knoten durch den Traversiermechanismus ausgeblendet. Die Traversierung wird transparent über nicht benötigte Knoten fortgesetzt. Ebenso ist es möglich, die Traversierung nur für einen Teil der Söhne oder für alle Söhne eines Knotens fortzusetzen.

Mit der beschriebenen Abstraktionsschicht existiert ein flexibles Instrumentarium, um eine zu großen Teilen von der Struktur und Darstellung des abstrakten Syntaxbaumes unabhängige Implementierung des Compilers zu ermöglichen.

6.2.3. Sichern des Syntaxbaumes

Eine der Hauptaufgaben der XML-basierten Compiler-Infrastruktur ist das Sichern des abstrakten Syntaxbaumes. In diesem Abschnitt wird das dazu verwendete Verfahren vorgestellt.

Wie in vorigen Abschnitten bereits erwähnt wurde, handelt es sich bei der internen Struktur des abstrakten Syntaxbaumes um einen gerichteten Graphen. Die einzelnen Knoten ohne Berücksichtigung der `Link` Attribute bilden dabei einen Spannbaum innerhalb dieses Graphen. Daher wird der Syntaxbaum in zwei Schritten gespeichert. In einem ersten Schritt wird der Spannbaum bestehend aus Knoten und Attributen ohne das `Link` Attribut serialisiert und gesichert. Unter *Serialisierung* wird dabei die Transformation des Baumes in eine flache, serielle Struktur verstanden. Die `Link` Attribute werden dann gesondert in einem zweiten Schritt gespeichert.

Als serialisierte Darstellung des Syntaxbaumes kommt eine Konstruktor-Darstellung zum Einsatz. Dazu wird eine allgemeine, n-stellige Konstruktorfunktion verwendet. Als erstes Argument erwartet diese den Typ des jeweiligen Knotens, gefolgt von allen Söhnen und Attributen in Reihenfolge ihrer Definition in der XML-Beschreibung. Hiervon ausgenommen sind lediglich die speziellen `Link` Attribute. Der Typ eines Knotens kann eindeutig durch den in der XML-Beschreibung durch das `name` Attribut angegebenen Bezeichner beschrieben werden. Die Söhne werden ebenfalls durch ihre jeweilige Konstruktor-Darstellung angegeben. Für jeden Attributtyp muss eine geeignete Konstruktorfunktion spezifiziert werden. Diese hängt von der tatsächlichen Ausgestaltung des Attributs ab und kann nicht aus der XML-Beschreibung erzeugt werden. Folglich muss diese vom Entwickler selbst spezifiziert werden.

Aufgrund der vorgestellten Transformation lässt sich der abstrakte Syntaxbaum ohne `Link` Attribute in eine Sequenz von Konstruktor-Funktionen abbilden. Die noch fehlenden `Link` Attribute werden nachträglich hinzugefügt. Da die serialisierte Darstellung bereits alle Knoten des Baumes enthält, reicht es die noch fehlenden Kanten zu speichern. Dazu wird jedem Knoten ein eindeutiger Index zugewiesen. Eine Möglichkeit hierzu besteht darin, alle Knoten in Reihenfolge ihres Vorkommens in der Konstruktor-Darstellung zu nummerieren. Unter Verwendung der Indizes können dann die `Link` Attribute als Zahlentripel dargestellt werden. Die erste Komponente des Tripels gibt dabei an, zu welchem Knoten das entsprechende `Link` Attribut gehört. Da ein Knoten mehrere solcher Attribute haben kann, wird in der zweiten Komponente ein Index gespeichert, welches `Link` Attribut abgebildet wird. Der Index kann leicht durch Nummerieren der Attribute in der Reihenfolge ihrer Definition in der XML-Beschreibung generiert werden. Die dritte Komponente gibt den Ziel-Knoten des Attributs an.

Somit kann der gesamte Graph in eine serielle Darstellung transformiert werden und damit gesichert werden.

Da bei der späteren Verwendung der so erzeugten Darstellung nicht der gesamte Graph erzeugt werden soll, sondern nur einzelne Funktionsdefinitionen, wird der Syntaxbaum in einzelnen Blöcken gespeichert. Für jede Funktion werden zwei Teile erzeugt, die Funktionsdeklaration (die Signatur der Funktion) und der Funktionsrumpf. Dadurch ist es später möglich, gezielt einzelne Teile des AST zu rekonstruieren. Insbesondere besteht die Möglichkeit, den Rumpf einer Funktion nur nachzuladen, wenn dieser auch tatsächlich für Optimierungen benötigt wird.

Da der abstrakte Syntaxbaum eine Graph-Struktur hat, kann es jedoch vorkommen, dass zwei dieser Blöcke durch mehr als eine Kante verbunden sind. Insbesondere ist

dies bei Funktionsanwendungen der Fall. Diese zeigen aus einer Funktionsdefinition auf eine andere. Da jede dieser Funktionen jedoch einzeln gesichert wird, muss eine Möglichkeit geschaffen werden, externe Kanten auszudrücken. Dies wurde in Form der `ExtLink` Attribute getan. Bei der Serialisierung des Syntaxbaumes werden diese durch besondere Konstruktor-Funktionen abgebildet. Als Argument erhält diese den qualifizierten Bezeichner der referenzierten Funktion. Bei der späteren Wiederherstellung des Baumes können so auch diese externen Kanten rekonstruiert werden.

Wie bereits erwähnt wurde, verwendet SaC als Dateiformat für Module das von C bereit gestellte Format für C-Bibliotheken. Um das Aufsplitten eines SaC-Moduls in mehrere Dateien zu verhindern und das auf vielen Plattformen verfügbare Format der C-Bibliotheken weiterhin verwenden zu können, muss ein Weg gefunden werden, die durch obige Transformation erzeugte Konstruktor-Darstellung in den C-Bibliotheken zu speichern.

Da es sich bei der erzeugten Darstellung um eine Konstruktor-Darstellung handelt, kann diese leicht nach C abgebildet werden. Dazu wurden die benötigten Konstruktor-Funktionen in C implementiert. Für die Knoten wurde ein XSLT-Script entwickelt, welches die entsprechenden Konstruktor-Funktionen aus der XML-Darstellung erzeugt. Lediglich die Konstruktor-Funktionen für Attribute sind statisch implementiert. Durch die vorgestellte Transformationen entstehen somit reguläre Ausdrücke der Sprache C, welche als Wert die vollständige Rekonstruktion des kodierten Teils des AST liefern. Ebenso können die externen Kanten durch Ausdrücke der Sprache C dargestellt werden. Dazu ist eine Funktion `addLink` implementiert, welche als Argumente den durch die Rekonstruktion entstandenen Spannbaum (oder einen diesen enthaltenden Graphen) und das der externen Kante entsprechende Zahlentripel als Argumente erwartet. Als Rückgabewert liefert sie den um die entsprechende Kante ergänzten Graphen zurück.

Die so entstandene Darstellung wird in eine Funktionsdefinition in der Sprache C gekapselt. Diese werden im Folgenden auch als *Generatorfunktion* bezeichnet. Da es sich um reguläre C-Funktionen handelt, können diese mittels der üblichen C-Entwicklungswerkzeuge in dynamische Bibliotheken übersetzt werden. Die so erzeugten Bibliotheken können dann von jedem C-Programm verwendet werden, insbesondere auch von durch den SaC-Compiler `sac2c` erzeugten Programmen und dem SaC-Compiler selbst.

Das vorgestellte Verfahren zur Sicherung des abstrakten Syntaxbaumes passt sich weitgehend automatisch an Veränderungen der XML-Beschreibung an und ermöglicht die Integration des Quelltextes in das bereits vom Compiler `sac2c` verwendete Dateiformat. Auf diese Weise erfüllt es die in Abschnitt 5.4.1 gestellten Anforderungen.

6.2.4. Einlesen des Syntaxbaumes

Im vorigen Abschnitt wurde das Verfahren zur Sicherung des abstrakten Syntaxbaumes durch in C implementierte Generatorfunktionen vorgestellt. Durch den gewählten Ansatz zum Speichern des abstrakten Syntaxbaumes ist ein späteres Einlesen

durch den Compiler `sac2c` ohne weitere Transformationen möglich. Da es sich um direkt ausführbare Funktionen handelt, ist ein aufwendiges Scannen und Parsen der verwendeten Darstellung nicht nötig. Statt dessen genügt es, die entsprechende Generatorfunktion aufzurufen, um den darin kodierten Teil des abstrakten Syntaxbaumes zu rekonstruieren.

Um die entsprechende Generatorfunktion zur Rekonstruktion des AST aufrufen zu können, müssen die erzeugten dynamischen C-Bibliotheken während der Übersetzung eines SaC-Programms bzw. Moduls durch den Compiler geladen werden. Dazu wird die im IEEE 1003.1 Standard [The02] definierte `dlopen` Schnittstelle verwendet. Diese wird von den gängigen Betriebssystemen[4] aus dem UNIX Umfeld implementiert. Sie ermöglicht das nachträgliche Laden dynamischer Bibliotheken zur Laufzeit und das Aufrufen darin enthaltener Funktionen. Somit kann während der Übersetzung eines Programms bzw. Moduls auf den benötigten Quelltext zugegriffen werden.

Durch den granularen Aufbau der erzeugten Generatorfunktionen ist es möglich, nur die benötigten Teile der Programm- bzw. Moduldefinition zu rekonstruieren. Für verwendete Funktionen kann so gezielt die Funktionsdeklaration nachgeladen werden. Wird für spätere Optimierungen der Funktion ebenfalls die Funktionsdefinition benötigt, kann diese nachträglich ergänzt werden. Dadurch werden immer nur die aktuell benötigten Teile des abstrakten Syntaxbaumes im Speicher gehalten. Insbesondere für große Projekte wird so der benötigte Arbeitsspeicher reduziert. Sie bleiben trotz der nur eingeschränkt separaten Compilation übersetzbar.

Auch für das Auflösen von Abhängigkeiten zwischen einzelnen Funktionen bietet die gewählte Repräsentation des abstrakten Syntaxbaumes Vorteile. Wie im vorigen Abschnitt beschrieben, werden externe Verweise (in der XML-Darstellung durch `ExtLink` Attribute abgebildet) durch eine spezielle Konstruktorfunktion abgebildet. Dies betrifft insbesondere auch Funktionsaufrufe. Diese Konstruktorfunktion ist derart implementiert, dass sie den entsprechenden Knoten im vorhandenen abstrakten Syntaxbaum sucht oder, falls dieser nicht vorhanden ist, die Rekonstruktion des entsprechenden Teils des AST triggert. Somit werden bei der Rekonstruktion von Teilen des AST auch die benötigten Abhängigkeiten rekonstruiert. Dieses Verfahren berücksichtigt insbesondere auch Abhängigkeiten über Modulgrenzen hinweg. Für Funktionsaufrufe bedeutet dies, dass die zugehörige Funktionsdeklaration, falls sie noch nicht im abstrakten Syntaxbaum vorhanden ist, unabhängig vom definierenden Modul rekonstruiert wird und somit im aktuellen Kontext zur Verfügung steht. Eine gesonderte Analyse der Abhängigkeiten eines Moduls bzw. einer Funktion kann aufgrund dessen entfallen.

[4] Die Implementierung wurde auf Sun Solaris (*http://www.sun.com/software/solaris/*), GNU Linux (*http://www.gnu.org/*) und FreeBSD (*http://www.freebsd.org/*) getestet. Für MacOS X (*http://www.apple.com/macosx/*) ist eine Umsetzung der `dlopen` Schnittstelle frei verfügbar (*http://www.opendarwin.org/projects/dlcompat/*).

6.3. Implementierung der Erweiterung um Module

In den folgenden Abschnitten wird die Implementierung der vier Sprachkonstrukte zur Spezifikation von Modulen beschrieben. Dabei findet die im vorigen Abschnitt beschriebene Compiler-Infrastruktur Verwendung.

6.3.1. `provide` und `export` Anweisungen

Mittels der beiden Anweisungen `provide` und `export` wird in SaC die Export-Schnittstelle eines Moduls angegeben. Nur auf durch eines der beiden Konstrukte explizit veröffentlichte Symbole eines Moduls kann in anderen Modulen zugegriffen werden. Dabei ist zusätzlich zu unterscheiden, ob ein Symbol mittels `provide` oder `export` veröffentlicht wurde. Im ersten Fall kann auf das entsprechende Symbol nur durch eine `use` Anweisung, oder durch Angabe eines qualifizierten Bezeichners zugegriffen werden. Wird das Symbol hingegen durch `export` veröffentlicht, so kann darüber hinaus auch eine `import` Anweisung verwendet werden, um das entsprechende Symbol im aktuellen Namensraum sichtbar zu machen. Um eine Kontrolle der entsprechenden Zugriffsrechte zu ermöglichen, muss diese Information im übersetzten Modul kodiert werden. Da ein Modul durch die im vorigen Abschnitt vorgestellte Sicherung des abstrakten Syntaxbaumes diesen vollständig enthält, genügt es, die Sichtbarkeitsinformationen im Syntaxbaum zu annotieren. Eine gesonderte Sicherung im entsprechenden Modul ist nicht erforderlich.

6.3.2. `use` Anweisungen und qualifizierte Bezeichner

In Kapitel 5 wurde die `use` Anweisung eingeführt. Sie ermöglicht den Zugriff auf in anderen Modulen mittels `provide` oder `export` veröffentlichte Symbole. Eine weitere Möglichkeit, auf entsprechende Symbole zuzugreifen, besteht durch die Angabe qualifizierter Bezeichner. Um die Implementierung zu vereinfachen werden daher `use` Anweisungen auf die Verwendung qualifizierter Bezeichner zurückgeführt. Dazu werden alle Vorkommen von in `use` Anweisungen enthaltenen Bezeichnern im gesamten Quelltext durch entsprechende qualifizierte Bezeichner ersetzt. Da für die angegebenen Symbole keine lokalen Definitionen existieren dürfen, ist die Annotation der Namensräume für unqualifizierte Bezeichner eindeutig. Kommt der Bezeichner in einer der `use` Anweisungen als Symbol vor, wird der in der Anweisung enthaltene Namensraum annotiert. Für alle anderen Bezeichner wird der lokale Namensraum annotiert. Nach dem Auflösen der `use` Anweisungen sind somit alle Bezeichner voll qualifiziert.

Für jedes durch einen qualifizierten Bezeichner angegebenes Symbol wird dann in einem zweiten Schritt die entsprechende Deklaration aus dem zugehörigen Modul zum abstrakten Syntaxbaum hinzugefügt. Da in SaC der Namensraumbezeichner mit dem Namen des zugehörigen Moduls übereinstimmt, kann die zu ladende Bibliothek direkt aus dem qualifizierten Bezeichner abgeleitet werden. Enthält das entsprechende Modul keine passende Deklaration, oder ist für die passende Deklaration weder eine Veröffentlichung per `provide`, noch per `export` Anweisung annotiert, wird die Über-

setzung mit einem entsprechenden Übersetzungsfehler abgebrochen. Somit ist sicher gestellt, dass nur auf veröffentlichte Symbole eines Moduls zugegriffen werden kann. Das so entstandene vervollständigte SaC-Programm bzw. -Modul kann nun übersetzt werden.

Das Zusammenfügen der separat übersetzen Bestandteile des Programms findet auf Ebene der Sprache C statt. Dazu werden die qualifizierten Bezeichner in den durch den `sac2c` Compiler erzeugten C-Quelltext übernommen. Die entsprechenden Symbole sind somit in den zugehörigen bereits übersetzten Modulen als C-Funktion verfügbar. Da es sich bei SaC-Modulen um reguläre C-Bibliotheken handelt, kann das Zusammenfügen somit dem zur C-Entwicklungsumgebung gehörenden Linker überlassen werden. Die Analyse der Abhängigkeiten eines Programms bzw. Moduls zu Symbolen aus anderen Modulen ist im Rahmen des `sac2c` Compilers nicht nötig.

6.3.3. Erzeugen der Sichten für eine `import` Anweisung

In Kapitel 5 wurden Sichten auf überladene Funktionen vorgestellt. Diese ermöglichen eine granulare Spezifikation der innerhalb eines Namensraums zur Verfügung stehenden Instanzen einer überladenen Funktion. In der konzeptuellen Beschreibung wurden dabei Funktionstabellen verwendet, um für rekursive Aufrufe die jeweils aktuell gültige Menge an Instanzen zu berücksichtigen.

Insbesondere im Kontext einer auf Laufzeiteffizienz optimierenden Sprache wie SaC ist die Lösung mit Funktionstabellen jedoch nicht optimal. In SaC wird für Aufrufe überladener Funktionen wenn möglich ein statischer Aufruf der entsprechenden Instanz verwendet, um den durch einen Dispatch zur Laufzeit entstehenden Overhead einzusparen. Dies ist unter Verwendung von Funktionstabellen nicht möglich. Während der Übersetzung eines Moduls ist durch die Indirektion durch Funktionstabellen die tatsächliche Sicht für einen Funktionsaufruf nicht bekannt. Die Menge der für einen Funktionsaufruf zu berücksichtigenden Instanzen kann somit statisch nicht ermittelt werden. Aufrufe überladener Funktionen können dadurch nur noch durch einen dynamischer Dispatch zur Laufzeit abgebildet werden. Dies führt zu einer schlechteren Laufzeiteffizienz.

Dieser negative Einfluss wird noch dadurch verstärkt, dass Funktionstabellen eine weitere Indirektionsstufe einführen. Für jeden Funktionsaufruf muss zusätzlich zur zu verwendenden Instanz zuerst die Menge der verfügbaren Instanzen ermittelt werden. Insbesondere das Anpassen der aktuellen Sicht bei einem Aufruf einer Funktion aus einem anderen Namensraum erhöht hierbei die Kosten eines Funktionsaufrufs.

Um trotz Einführung der Sichten die hohe Laufzeiteffizienz von SaC-Programmen zu erhalten, wurde ein anderes Verfahren zur Implementierung der Sichten gewählt. Statt einer Indirektion über Funktionstabellen werden die Sichten im jeweiligen Namensraum statisch erzeugt. Die aus anderen Namensräumen importierten Instanzen werden im aktuellen Namensraum erneut definiert und somit eine lokale, an die aktuelle Sicht angepasste Version erzeugt. Da es sich dadurch bei allen Instanzen einer überladenen Funktion immer um lokale, im aktuellen Namensraum definierte Instanzen handelt, ist die Menge der für einen Funktionsaufruf zu verwendenden Instanzen

während der Übersetzung eines Programms bzw. Moduls statisch bekannt. Das semistatische Dispatch-Verfahren [Kre03] des SAC-Compilers kann somit weiter verwendet werden.

Die jeweiligen lokalen Versionen einer Funktionsinstanz unterscheiden sich dabei von der ursprünglich definierten Instanz nur durch die Menge der für Funktionsaufrufe berücksichtigten Instanzen. Innerhalb einer importierten Instanz müssen alle Funktionsaufrufe ebenfalls importierter Funktionen auf die lokalen Definitionen umgeleitet werden. Um dies zu ermöglichen, wird erneut auf die vorgestellte Compiler-Infrastruktur zurückgegriffen.

Da für jede Instanz die Definition in der Sicherung des abstrakten Syntaxbaumes verfügbar ist, kann diese im lokalen Kontext des aktuellen Moduls rekonstruiert werden. Um die einzelnen Sichten einer Instanz zu unterscheiden, wird der qualifizierte Bezeichner um einen Bezeichner für die zugehörige Sicht ergänzt. Dieser enthält sowohl das definierende Modul, als auch die vollständige Historie der zwischen dem definierenden Modul und dem aktuellen Modul liegenden Module. Dadurch ist der erweiterte Bezeichner für jede Instanz und Sicht eindeutig. Er wird nur intern durch den Compiler verwendet und kann vom Programmierer nicht angegeben werden. Somit ist auch sicher gestellt, dass spezifische Sichten nicht direkt angesprochen werden können, sondern nur vom Compiler selbst verwendet werden. Durch die Erweiterung der Bezeichner kann die zu verwendende Sicht direkt im Funktionsaufruf kodiert werden. Eine Erweiterung des Compilationsschemas für SAC-Programme in die Zielsprache C ist nicht nötig.

Die Ersetzung der Funktionsaufrufe durch die entsprechenden lokalen Definition erfolgt direkt während der Rekonstruktion des Syntaxbaumes. Dazu wird die Konstruktorfunktion für externe Verweise um eine Ersetzungstabelle erweitert. Diese beinhaltet die Bezeichner der jeweiligen durch eine `import` Anweisung zum aktuellen Namensraum hinzugefügten Instanzen. Wird ein Verweis auf eine importierte Funktion rekonstruiert, wird statt des kodierten Aufrufziels die lokale Definition verwendet. Somit wird automatisch eine angepasste Version der importierten Funktionen erzeugt.

Um nur die durch einen Import veränderten Teile einer Sicht erzeugen zu müssen, wird während der Analyse-Phase die transitive Hülle der von einer Funktion aufgerufenen Funktionen im Syntaxbaum annotiert. Dadurch ist es während der Rekonstruktion des Syntaxbaumes möglich, für Aufrufe von Funktionen, die keine der durch die `import` Anweisung angegebenen Funktionen verwenden, die bereits in einem anderen Modul vorhandene Definition zu verwenden. Die Menge der zu erzeugenden Funktionsdefinitionen kann dadurch stark vermindert werden.

Das hier vorgestellte Verfahren entspricht einer Umsetzung der in Abschnitt 5.3.1 vorgestellten Darstellung von Sichten durch die Sprache $\mathcal{F}un$. Ein Nachteil der expliziten Erzeugung der Sichten ist die durch das Kopieren von Funktionsdefinitionen entstehende Vergrößerung sowohl des abstrakten Syntaxbaumes, als auch der erzeugten Programme. Dies entspricht dem klassischen Kompromiss zwischen hoher Laufzeiteffizienz und kompaktem Code. Da für SAC-Programme die Laufzeiteffizienz von übergeordneter Bedeutung ist, ist die durch die vorgestellte Implementierung entstehende Vergrößerung des erzeugten Codes jedoch vernachlässigbar. Der Effekt

wird zusätzlich durch die vorgestellte Reduzierung der erzeugten Funktionen auf die tatsächlich veränderten Funktionen abgemildert.

6.3.4. Lösen des Diamant-Import-Problems

In Kapitel 5 wurde das Diamant-Import-Problem im Kontext von Sichten vorgestellt. Es tritt immer dann auf, wenn eine Instanz einer überladenen Funktion durch verschiedene Importpfade in einem Modul mehrfach sichtbar wird. In solchen Situationen ist es trotz überdeckender Instanzen wünschenswert, das entsprechende Modul dennoch übersetzen zu können. Insbesondere durch das Konzept der Sichten können solche Instanzen trotz gleicher Herkunft eine unterschiedliche Semantik haben. Als Lösung wurde vorgeschlagen, durch einen Diamant-Import entstandene sich überdeckende Instanzen zuzulassen, falls eine äquivalente Semantik sicher gestellt werden kann. Dies ist, wie in Abschnitt 5.3.2 beschrieben, immer dann der Fall, wenn zwei Instanzen auf eine gemeinsame Definition zurückzuführen sind und die von den Instanzen verwendeten Funktionen ebenfalls äquivalent sind. Um die Äquivalenz zweier Instanzen zu überprüfen, muss somit der gesamte von den jeweiligen Instanzen ausgehende Aufrufgraph verglichen werden. Da jeder im Aufrufgraph enthaltene Knoten nur einmal überprüft werden muss, können durch Rekursion entstehende Kreise innerhalb des Graphen unberücksichtigt bleiben.

Wie im vorigen Abschnitt beschrieben wurde, ist die Sicht einer Instanz im jeweiligen Bezeichner kodiert. Insbesondere enthält dieser die Herkunft einer Instanz. Diese kann somit für zwei Instanzen leicht verglichen werden.

Des weiteren muss die Äquivalenz der in den jeweiligen Instanzen verwendeten Funktionen sicher gestellt werden. Wird in beiden Instanzen der gleiche Bezeichner für die verwendete Funktion benutzt, verwenden beide auch die gleiche Definition der Funktion. In diesem Falle ist die Äquivalenz dadurch gegeben.

Verwenden beide Instanzen verschiedene Sichten einer Funktion, muss der Vergleich für diese fortgesetzt werden. Dazu werden die entsprechenden Instanzen der beiden Funktionen nach dem vorgestellten Schema auf Äquivalenz überprüft. Der Vergleich kann abgebrochen werden, sobald eine nicht äquivalente Instanz gefunden wurde.

Da die Menge der verwendeten Funktionen endlich ist und nur der kreisfreie Aufrufgraph betrachtet wird, terminiert der Vergleich. Dadurch, dass nur die Funktionen einer Sicht generiert werden, welche sich in der Menge der verwendeten Instanzen unterscheiden, ist des weiteren der tatsächlich zu vergleichende Graph im Allgemeinen stark begrenzt.

Durch das Abbilden der vollständigen Historie der Importe im Bezeichner der jeweiligen Funktionen kann im Falle zweier überdeckender, nicht äquivalenter Instanzen der genaue Importpfad rekonstruiert werden und eine aussagekräftige Fehlermeldung generiert werden. Somit steht dem Programmierer ein Werkzeug zur Verfügung, um durch fehlerhafte Importschnittstellen entstandene unzulässige Diamant-Importe zu lösen.

6.4. Erzeugen der Funktionsspezialisierungen

Für die Implementierung der Funktionsspezialisierung wird das in Abschnitt 5.5 vorgestellte Prinzip der lokalen Spezialisierung verwendet. Dazu wird die im vorigen Abschnitt vorgestellte Implementierung der Sichten genutzt. Wurde die zu spezialisierende Funktion mittels einer `import` Anweisung in den aktuellen Namensraum importiert oder lokal definiert, werden durch Spezialisierung entstandene Instanzen analog zu vom Programmierer angegebenen Instanzen der aktuellen Sicht hinzugefügt. Im Falle einer durch eine `use` Anweisung bzw. durch einen qualifizierten Bezeichner verwendeten Funktion wird für die durch Spezialisierung entstandenen Instanzen eine Sicht erzeugt. Diese wird analog zu der durch einen Import der Funktion mittels einer `import` Anweisung erzeugten Sicht generiert. Um durch Spezialisierung entstandene Sichten von durch einen Import entstandene Sichten zu trennen, werden diese durch einen besonderen Namen gekennzeichnet. Dadurch ist sichergestellt, dass es zwischen mittels einer `import` Anweisung bzw. durch Spezialisierung erzeugten Sichten nicht zu Namenskonflikten kommt.

6.5. Fazit

Die in Kapitel 3 an das Modulsystem gestellten Anforderungen werden durch die hier vorgestellte Implementierung erfüllt. Zentraler Bestandteil ist dabei das auf einer XML-Beschreibung basierende Compiler-Framework. Es findet sowohl für die Sicherung des Quelltextes für spätere Optimierungen, als auch für die Erzeugung der für die Überladung über Namensraumgrenzen benötigten Sichten Verwendung.

Primäres Ziel der Implementierung ist dabei die Bewahrung der Laufzeiteffizienz des durch den Compiler erzeugten Codes. Dies wird durch die vorgestellte Implementierung der Sichten erreicht. Sie wirkt sich lediglich auf die Größe des erzeugten Codes negativ aus.

Für die Funktionsspezialisierung wird eine Umsetzung der in Kapitel 5 beschriebenen lokalen Spezialisierung vorgestellt. Für zukünftige Erweiterungen wäre eine Implementierung der ebenfalls beschriebenen globalen Spezialisierung jedoch wünschenswert.

7. Zusammenfassung und Ausblick

Die vorliegende Arbeit stellt einen neuen Ansatz zur Kombination von separaten Namensräumen mit der Überladung von Funktionen vor. Wie sich gezeigt hat, reichen in diesem Kontext die vorhandenen Ansätze der globalen bzw. lokalen Überladung nicht aus. Beide führen zu einer Einschränkung der Separation der Namensräume bzw. der Möglichkeiten zur Funktionsüberladung. Daher werden zusätzlich zu den bekannten `export` und `import` Anweisungen mit `provide` und `use` zwei weitere Sprachelemente zur Beschreibung der Schnittstelle zwischen Modulen eingeführt. Mittels dieser ist es möglich, sowohl eine Überladung von Funktionen über Namensraumgrenzen hinweg zuzulassen (globale Überladung), als auch einen Namensraum bezüglich der Funktionsüberladung abzuschließen (lokale Überladung). Somit steht dem Programmierer ein mächtiges Werkzeug zur Verfügung, um die Sichtbarkeit einzelner Instanzen innerhalb eines Namensraumes zu spezifizieren.

Auch in Bezug auf sich überlappende Instanzen überladener Funktionen und die daraus resultierende Abhängigkeit rekursiver Funktionen vom jeweiligen Aufrufkontext bietet das vorgestellte Modulsystem eine Lösung. Es zeigt mit dem Konzept der Sichten eine Möglichkeit, die Menge der für einen rekursiven Aufruf zu verwendenden Instanzen automatisch aus der entsprechenden Schnittstellendefinition und dem Aufrufkontext abzuleiten. Die vorgestellte Lösung kommt dabei ohne die Komplexität parametrisierter Module aus. Insbesondere für große Projekte mit einer Vielzahl von Modulen bleibt der gewählte Ansatz für den Programmierer überschaubar. Die aufwendige Analyse der Abhängigkeiten zwischen einzelnen Modulen und die Zusammenstellung der sichtbaren Instanzen wird vom zugrunde liegenden Modulsystem übernommen.

Die Semantik der eingeführten Sprachkonstrukte zur Schnittstellendefinition wird auf Basis eines angewandten λ-Kalküls formal beschrieben. Der vorgestellte Ansatz kann somit leicht auf andere Sprachen übertragen werden. In der vorliegenden Arbeit wird eine auf dieser Beschreibung basierende Implementierung für die Sprache SaC vorgestellt. Da es sich bei SaC um eine auf Laufzeiteffizienz ausgelegte Sprache handelt, sind hierbei insbesondere Aspekte der Optimierung über Modulgrenzen zu berücksichtigt. Dem wird in der vorliegenden Arbeit durch das vorgestellte XML-basierte Compiler-Framework Rechnung getragen. Es ermöglicht die Sicherung des Quelltextes innerhalb der erzeugten Module, so dass dieser für spätere modulübergreifende Optimierungen zur Verfügung steht. Das Modulsystem verhält sich dadurch vollständig transparent gegenüber den hochsprachlichen Optimierungen und minimiert den Einfluss der Modularisierung auf die Laufzeiteffizienz des erzeugten Codes.

Ein weiterer zu beachtender Aspekt im Kontext der Laufzeiteffizienz ist die Funktionsspezialisierung. Hierfür werden zwei Verfahren – die lokale und globale Spe-

zialisierung – vorgestellt und mögliche Implementierungen skizziert. Für die lokale Spezialisierung wird weiter die Umsetzung als Erweiterung des Compilers `sac2c` vorgestellt.

Unter dem Aspekt der Compilerentwicklung bietet das vorgestellte XML-basierte Compiler-Framework eine große Menge an weiteren Entwicklungsmöglichkeiten. Neben dem im Rahmen dieser Arbeit entwickelten Modulsystem sind bereits weitere Teile des Compilers auf eine abstrakte Beschreibung umgestellt. Dazu zählt zum Beispiel die Erfassung und Verwaltung der einzelnen Compilerphasen. Neben den Compilerphasen selbst sind dafür für jeden Knoten die Gültigkeit seiner Söhne bzw. Attribute während der einzelnen Phasen in die Spezifikation aufgenommen worden. Mittels der so kodierten Informationen ist es möglich, nach jeder Compilerphase den abstrakten Syntaxbaum auf seine Konformität bezüglich der XML-Beschreibung zu überprüfen. Insbesondere die Überprüfung, ob eine im abstrakten Syntaxbaum vorhandene Vater-Sohn Beziehung der formalen Spezifikation entspricht, hat sich als wertvolle Hilfe für die Fehlerbeseitigung bewährt. Sie deckt Fehler innerhalb der durch die Optimierungen durchgeführten Transformationen auf.

Auch die vollständige Entkopplung der internen Darstellung des Syntaxbaumes von den darauf operierenden Bestandteilen des Compilers hat sich als sehr vorteilhaft erwiesen. So ist, aufbauend auf dem in dieser Arbeit entwickelten System, eine alternative Darstellung des Syntaxbaumes entwickelt worden, welche sich insbesondere zur Fehleranalyse eignet. Des weiteren konnten die nach dieser Arbeit entstandenen Erweiterungen des Compilers leichter in die bestehende Codebasis eingepflegt werden.

7.1. Ausblick

Im Rahmen dieser Arbeit wird eine Implementierung der Sichten durch statische Erzeugung der entsprechenden Instanzen vorgestellt. Diese bietet sich insbesondere wegen des guten Laufzeitverhaltens an. Nachteilig wirken sich jedoch die dadurch eingeführten Abhängigkeiten zwischen einzelnen Modulen aus. Wird die Implementierung einer Instanz im definierenden Modul verändert, müssen auch alle Module, welche die veränderte Instanz importieren, erneut übersetzt werden. Insbesondere während der Entwicklung einer Anwendung wirkt sich dies negativ auf die Übersetzungszeit aus. Auch ist in dieser Phase die Laufzeiteffizienz nicht von entscheidender Bedeutung. Somit wäre es wünschenswert, die ebenfalls vorgestellte Implementierung der Sichten durch Funktionstabellen als alternatives Übersetzungsmodell anzubieten. Dadurch könnte während der Entwicklung einer Anwendung die benötigte Übersetzungszeit zu Lasten der Laufzeiteffizienz verbessert werden. Da beide Verfahren semantisch äquivalent sind, kann nach abgeschlossener Entwicklung eine auf Laufzeit optimierte Version erzeugt werden. Für den Compiler `sac2c` ließe sich dies durch entsprechende, vom Anwender spezifizierbare Optimierungsstufen umsetzen.

Ebenso wäre eine Implementierung der vorgestellten globalen Spezialisierung für den Compiler `sac2c` wünschenswert. Insbesondere das Konzept der adaptiven Module bietet Möglichkeiten, die Laufzeiteffizienz der erzeugten Programme weiter zu

steigern. Durch die wiederholte Verwendung eines adaptiven Moduls passt sich dieses mit zunehmender Anzahl der vorhandenen Instanzen an den vom Anwender bearbeiteten Problemraum an. Das Modul erlernt somit iterativ die benötigten Instanzen. Dies entschärft das Problem, die benötigten Spezialisierungen aus der nicht endlichen Menge von möglichen Spezialisierungen zu wählen. Verbunden mit einer Analyse des Laufzeitverhaltens der erstellten Anwendungen (sogenanntes *Profiling*) können so genau die benötigten Spezialisierungen ermittelt werden. Das so gewonnene Wissen bleibt dabei direkt im Modul gespeichert.

Durch die Kombination von globaler Überladung mit den genannten Laufzeitanalysen erhält man die Möglichkeit der *dynamischen Spezialisierung*. Dabei werden fehlende Instanzen einer Funktion zur Laufzeit der Anwendung erkannt und nachträglich erzeugt. Da das vorgestellte Modulsystem den Quelltext der zugehörigen Funktionsdefinition mit im Modul speichert, ist dies auch ohne direkten Zugriff auf den Quelltext des Programms möglich. Durch die verwendeten C-Bibliotheken und das dynamische Linken sind für die Implementierung keine bedeutenden Anpassungen am Modulsystem zu erwarten.

A. Codebeispiele

A.1. Überladung über Namensraumgrenzen in Haskell

A.1.1. Beispiel für das Einschmuggeln von Instanzen

Modul `Fun`

```
module Fun( Fun( oracle)) where

class Fun a where
  oracle :: a -> String
```

Modul `IntFun`

```
module IntFun( Fun( oracle), definedOnInt) where

import Fun( Fun( oracle))

instance Fun Int where
  oracle x = "Integer"

definedOnInt x = oracle x
```

Modul `FloatFun`

```
module FloatFun( Fun( oracle), definedOnFloat) where

import Fun( Fun( oracle))

instance Fun Float where
  oracle x = "Float"

definedOnFloat x = oracle x
```

Modul `Main`

```
module Main where

import IntFun( definedOnInt)
import FloatFun( definedOnFloat)
```

```
6 toInt :: Int -> Int
  toInt x = x
8
  toFloat :: Float -> Float
10 toFloat x = x

12 main = putStr "definedOnInt␣1.5␣is␣" >>
          putStr (definedOnInt (toFloat 1.5)) >>
14        putStr "␣definedOnFloat␣1␣is␣" >>
          putStr (definedOnFloat( toInt 1))
```

Erwartete Ausgabe
Bei strikter Separation der Namensräume müsste obiges Beispiel einen Übersetzungsfehler erzeugen. Die Funktion `definedOnInt` aus dem Modul `IntFun` ist nur für Werte vom Typ `Int` definiert, da im definierenden Modul keine weiteren Instanzen der Typklasse `Fun` sichtbar sind. Analog verhält es sich für die Funktion `definedOnFloat` aus dem Modul `FloatFun`. Diese ist nur für Werte vom Typ `Float` definiert. Im Modul `Main` werden beide Funktionen importiert, die Typklasse `Fun` hingegen ist hier nicht sichtbar. Es wird also zu keiner Zeit eine kombinierte Typklasse mit beiden Instanzen spezifiziert. Daher sind die beiden Aufrufe der importierten Funktionen nicht Typkorrekt.

Ausgabe `HUGS`
```
definedOnInt 1.5 is Float definedOnFloat 1 is Integer
```

Ausgabe `ghc`
```
definedOnInt 1.5 is Float definedOnFloat 1 is Integer
```

A.1.2. Beispiel für überdeckende Instanzen

Modul `StdLib`
```
1 module StdLib ( Alg( oracle)) where

3 class Alg a where
    oracle :: a -> String
5
  instance Alg Int where
7  oracle a = "StdLib"
```

Modul `AlgMatch`
```
1 module AlgMatch ( Alg( oracle)) where
```

```
3 import StdLib( Alg( oracle) )

5 instance Alg Float where
    oracle x = "AlgMatch"
```

Modul `AlgArb`

```
  module AlgArb ( Alg( oracle)) where
2
  import StdLib ( Alg( oracle))
4
  instance Alg Float where
6   oracle x = "AlgArb"
```

Modul `FFT`

```
  module FFT ( fft) where
2
  import AlgMatch ( Alg( oracle))
4
  fft x = oracle x
```

Modul `Main`

```
1 module Main where

3 import FFT( fft)
  import AlgArb( Alg( oracle))
5
  toFloat :: Float -> Float
7 toFloat x = x

9 main = putStr (fft (toFloat 4.2)) >>
         putStr "␣and␣" >>
11       putStr (oracle (toFloat 4.2))
```

Erwartete Ausgabe

Im Modul `FFT` ist nur die im Modul `AlgMatch` definierte Instanz für Werte vom Typ `Float` sichtbar. Ein Aufruf der Funktion `oracle` innerhalb der Funktion `fft` müsste somit den String `"AlgMatch"` liefern. Dem gegenüber ist im Modul `Main` die Instanz aus dem Modul `AlgArb` sichtbar. Hier müsste ein Aufruf der Funktion `oracle` den String `"AlgArb"` liefern. Insbesondere sind in keinem Modul beide Instanzen für Werte vom Typ `Float` gleichzeitig sichtbar. Damit ergibt sich folgende erwartete Ausgabe:

```
AlgMatch and AlgArb
```

Ausgabe HUGS

```
ERROR "AlgArb.hs":6 - Overlapping instances for class "Alg"
*** This instance   : Alg Float
*** Overlaps with   : Alg Float
*** Common instance : Alg Float
```

Ausgabe ghc

```
AlgMatch and AlgMatch
```

B. XML-Darstellung des abstrakten Syntaxbaumes

B.1. Verwendetes XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="all">
    <xsd:complexType />
  </xsd:element>
  <xsd:element name="attribute">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="description" minOccurs="0"
        maxOccurs="1" />
        <xsd:element ref="type" minOccurs="1"
                     maxOccurs="1" />
        <xsd:element ref="phases" minOccurs="1"
                     maxOccurs="1" />
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NMTOKEN"
                     use="required" />
      <xsd:attribute name="default" type="xsd:string" />
      <xsd:attribute name="mandatory" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="no" />
            <xsd:enumeration value="yes" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="attributes">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="attribute" minOccurs="0"
                     maxOccurs="unbounded" />
```

```
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="attributetypes">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="type" minOccurs="0"
                     maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="name" type="xsd:NMTOKEN"
                           use="required" />
            <xsd:attribute name="ctype" type="xsd:string"
            use="required" />
            <xsd:attribute name="copy" use="required">
              <xsd:simpleType>
                <xsd:restriction base="xsd:NMTOKEN">
                  <xsd:enumeration value="function" />
                  <xsd:enumeration value="hash" />
                  <xsd:enumeration value="literal" />
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="init" type="xsd:string"
            use="required" />
            <xsd:attribute name="size"
            type="xsd:positiveInteger" />
            <xsd:attribute name="persist">
              <xsd:simpleType>
                <xsd:restriction base="xsd:NMTOKEN">
                  <xsd:enumeration value="yes" />
                  <xsd:enumeration value="no" />
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="definition">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="attributetypes" minOccurs="1"
        maxOccurs="1" />
```

```
        <xsd:element ref="nodesets" minOccurs="1"
79                   maxOccurs="1" />
        <xsd:element ref="syntaxtree" minOccurs="1"
81                   maxOccurs="1" />
      </xsd:sequence>
83  </xsd:complexType>
</xsd:element>
85 <xsd:element name="description">
  <xsd:complexType mixed="true" />
87 </xsd:element>
<xsd:element name="node">
89  <xsd:complexType>
    <xsd:sequence>
91    <xsd:element ref="description" minOccurs="0"
                   maxOccurs="1" />
93    <xsd:element ref="sons" minOccurs="1"
                   maxOccurs="1" />
95    <xsd:element ref="attributes" minOccurs="1"
                   maxOccurs="1" />
97    <xsd:element ref="flags" minOccurs="1"
                   maxOccurs="1" />
99  </xsd:sequence>
    <xsd:attribute name="name" type="xsd:NMTOKEN"
101                use="required" />
  </xsd:complexType>
103 </xsd:element>
<xsd:element name="nodeset">
105  <xsd:complexType>
    <xsd:sequence>
107   <xsd:element ref="target" minOccurs="1"
                   maxOccurs="1" />
109 </xsd:sequence>
    <xsd:attribute name="name" type="xsd:NMTOKEN"
111                use="required" />
  </xsd:complexType>
113 </xsd:element>
<xsd:element name="nodesets">
115  <xsd:complexType>
    <xsd:sequence>
117   <xsd:element ref="nodeset" minOccurs="1"
                   maxOccurs="unbounded" />
119 </xsd:sequence>
  </xsd:complexType>
121 </xsd:element>
```

```
  <xsd:element name="phase">
123   <xsd:complexType>
        <xsd:attribute name="name" type="xsd:NMTOKEN"
125                    use="required" />
      </xsd:complexType>
127 </xsd:element>
    <xsd:element name="phases">
129   <xsd:complexType>
        <xsd:choice>
131       <xsd:element ref="all" minOccurs="1"
                       maxOccurs="1" />
133       <xsd:sequence minOccurs="1" maxOccurs="unbounded">
            <xsd:choice>
135           <xsd:element ref="phase" minOccurs="0"
                           maxOccurs="1" />
137           <xsd:element ref="range" minOccurs="0"
                           maxOccurs="1" />
139         </xsd:choice>
          </xsd:sequence>
141       <xsd:element ref="unknown" minOccurs="1"
                       maxOccurs="1" />
143     </xsd:choice>
      </xsd:complexType>
145 </xsd:element>
    <xsd:element name="range">
147   <xsd:complexType>
        <xsd:attribute name="to" type="xsd:NMTOKEN"
149                    use="required" />
        <xsd:attribute name="from" type="xsd:NMTOKEN"
151                    use="required" />
      </xsd:complexType>
153 </xsd:element>
    <xsd:element name="set">
155   <xsd:complexType>
        <xsd:attribute name="name" use="required">
157       <xsd:simpleType>
            <xsd:restriction base="xsd:NMTOKEN">
159           <xsd:enumeration value="Expr" />
              <xsd:enumeration value="Instr" />
161         </xsd:restriction>
          </xsd:simpleType>
163     </xsd:attribute>
      </xsd:complexType>
165 </xsd:element>
```

```
  <xsd:element name="son">
167   <xsd:complexType>
      <xsd:sequence>
169     <xsd:element ref="description" minOccurs="0"
                     maxOccurs="1" />
171     <xsd:element ref="target" minOccurs="1"
                     maxOccurs="1" />
173   </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NMTOKEN"
175                  use="required" />
      <xsd:attribute name="default" type="xsd:string" />
177   <xsd:attribute name="mandatory" use="required">
        <xsd:simpleType>
179       <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="no" />
181         <xsd:enumeration value="yes" />
          </xsd:restriction>
183     </xsd:simpleType>
      </xsd:attribute>
185   </xsd:complexType>
  </xsd:element>
187 <xsd:element name="sons">
    <xsd:complexType>
189   <xsd:sequence>
        <xsd:element ref="son" minOccurs="0"
191                  maxOccurs="unbounded" />
      </xsd:sequence>
193   </xsd:complexType>
  </xsd:element>
195 <xsd:element name="syntaxtree">
    <xsd:complexType>
197   <xsd:sequence>
        <xsd:element ref="node" minOccurs="1"
199                  maxOccurs="unbounded" />
      </xsd:sequence>
201   <xsd:attribute name="version" type="xsd:double"
                     use="required" />
203   </xsd:complexType>
  </xsd:element>
205 <xsd:element name="target">
    <xsd:complexType>
207   <xsd:choice>
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
209       <xsd:element name="node" minOccurs="0"
```

```
                        maxOccurs="unbounded">
211             <xsd:complexType>
                  <xsd:attribute name="name"
213                              type="xsd:NMTOKEN"
                                 use="required" />
215             </xsd:complexType>
              </xsd:element>
217           <xsd:element name="set" minOccurs="0"
                        maxOccurs="unbounded">
219             <xsd:complexType>
                  <xsd:attribute name="name"
221                              type="xsd:NMTOKEN"
                                 use="required" />
223             </xsd:complexType>
              </xsd:element>
225         </xsd:choice>
            <xsd:element ref="unknown" minOccurs="1"
227              maxOccurs="1" />
        </xsd:choice>
229   </xsd:complexType>
  </xsd:element>
231 <xsd:element name="type">
    <xsd:complexType>
233   <xsd:sequence>
        <xsd:element ref="target" minOccurs="0"
235                  maxOccurs="1" />
      </xsd:sequence>
237   <xsd:attribute name="name" type="xsd:NMTOKEN"
                     use="required" />
239 </xsd:complexType>
  </xsd:element>
241 <xsd:element name="unknown">
    <xsd:complexType />
243 </xsd:element>
  <xsd:element name="flags">
245   <xsd:complexType>
      <xsd:sequence>
247     <xsd:element ref="flag" minOccurs="0"
                     maxOccurs="unbounded" />
249   </xsd:sequence>
    </xsd:complexType>
251 </xsd:element>
  <xsd:element name="flag">
253   <xsd:complexType>
```

```
      <xsd:attribute name="name" type="xsd:NMTOKEN"
255                  use="required" />
      <xsd:attribute name="default">
257     <xsd:simpleType>
          <xsd:restriction base="xsd:NMTOKEN">
259         <xsd:enumeration value="TRUE" />
            <xsd:enumeration value="FALSE" />
261       </xsd:restriction>
        </xsd:simpleType>
263   </xsd:attribute>
       <xsd:element ref="description" minOccurs="0"
265                 maxOccurs="1" />
    </xsd:complexType>
267 </xsd:element>
</xsd:schema>
```

B.2. Ausschnitt aus der XML Beschreibung

```
<node name="Fundef">
2 <sons>
    <son name="Rets" mandatory="no">
4     <target>
        <node name="Ret" />
6     </target>
    </son>
8   <son name="Args" mandatory="no">
      <target>
10      <node name="Arg" />
      </target>
12  </son>
    <son name="Body" mandatory="yes">
14    <target>
        <node name="Block" />
16    </target>
    </son>
18  <son name="Next" mandatory="no">
      <target>
20      <node name="Fundef" />
      </target>
22  </son>
    <son name="Objects" default="NULL" mandatory="no" >
24    <target>
        <node name="Linklist" />
26    </target>
```

```
    </son>
  </sons>
  <attributes>
    <attribute name="Name" mandatory="yes">
      <type name="String" />
      <phases>
        <all />
      </phases>
    </attribute>
    <attribute name="Mod" mandatory="yes">
      <type name="SharedString" />
      <phases>
        <all />
      </phases>
    </attribute>
    <attribute name="SymbolName" default="NULL"
               mandatory="no">
      <type name="String" />
      <phases>
        <range from="export" to="final" />
      </phases>
    </attribute>
    <attribute name="LinkName" default="NULL"
               mandatory="no">
      <type name="String" />
      <phases>
        <range from="respragma" to="final" />
      </phases>
    </attribute>
    <attribute name="WrapperType" default="NULL"
               mandatory="yes">
      <type name="NewType" />
      <phases>
        <all />
      </phases>
    </attribute>
    <attribute name="FunNo" default="0" mandatory="yes">
      <type name="Integer" />
      <phases>
        <phase name="annotatefuncalls" />
      </phases>
    </attribute>
    <attribute name="Pragma" default="NULL"
               mandatory="no">
```

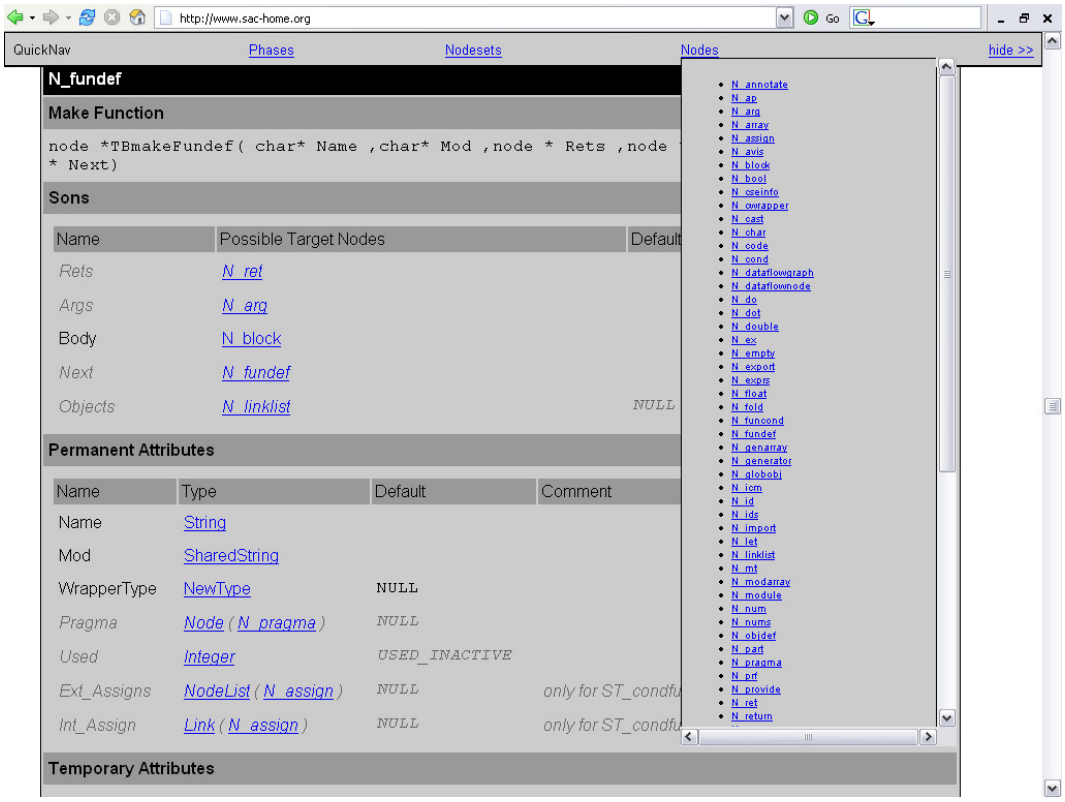
```
      <type name="Node" >
72      <target>
          <node name="Pragma" />
74      </target>
      </type>
76    <phases>
        <all />
78    </phases>
    </attribute>
80  <attribute name="Types" mandatory="no" >
      <type name="OldType" />
82    <phases>
        <range from="typecheck" to="final" />
84    </phases>
    </attribute>
86  <attribute name="Used" default="USED_INACTIVE"
            mandatory="no">
88    <type name="Integer" />
      <phases>
90      <all />
      </phases>
92  </attribute>
    <attribute name="Ext_Assigns" default="NULL"
94          mandatory="no">
      <description>only for ST_condfun, ST_dofun und
96                ST_whilefun</description>
      <type name="NodeList">
98      <target>
          <node name="Assign" />
100     </target>
      </type>
102   <phases>
        <all />
104   </phases>
    </attribute>
106 <attribute name="Int_Assign" default="NULL"
            mandatory="no">
108   <description>only for ST_condfun, ST_dofun und
                ST_whilefun</description>
110   <type name="Link">
        <target>
112       <node name="Assign" />
        </target>
114   </type>
```

```
      <phases>
116     <all />
      </phases>
118   </attribute>
      <attribute name="Specs" mandatory="yes">
120     <type name="Integer" />
      <phases>
122     <phase name="new_typecheck" />
      </phases>
124   </attribute>
      <attribute name="Return" mandatory="yes">
126     <type name="DownLink">
        <target>
128       <node name="Return" />
        </target>
130     </type>
      <phases>
132     <range from="typecheck" to="compile" />
      </phases>
134   </attribute>
      <attribute name="Impl" mandatory="no">
136     <type name="ExtLink">
        <target>
138       <node name="Fundef" />
        </target>
140     </type>
      <phases>
142     <phase name="typecheck" />
      </phases>
144   </attribute>
      <attribute name="Icm" mandatory="yes">
146     <type name="Node">
        <target>
148       <node name="Icm" />
        </target>
150     </type>
      <phases>
152     <range from="compile" to="print" />
      </phases>
154   </attribute>
      <attribute name="VarNo" mandatory="yes">
156     <type name="Integer" />
      <phases>
158     <range from="optimize" to="final" />
```

```
      </phases>
160   </attribute>
      <attribute name="InlRec" mandatory="yes">
162     <type name="Integer" />
        <phases>
164       <phase name="inline" />
        </phases>
166   </attribute>
      <attribute name="TCStat" mandatory="yes">
168     <type name="Integer" />
        <phases>
170       <phase name="typecheck" />
        </phases>
172   </attribute>
      <attribute name="DFM_Base" mandatory="yes">
174     <type name="DFMaskBase" />
        <phases>
176       <range from="rc" to="final" />
          <range from="fun2lac" to="lac2fun" />
178     </phases>
      </attribute>
180   <attribute name="ArgTab" mandatory="yes">
        <type name="ArgTab" />
182     <phases>
          <range from="precompile" to="compile" />
184     </phases>
      </attribute>
186   <attribute name="LiftedFrom" mandatory="no">
        <description>only for ST_spmdfun</description>
188     <type name="Link">
          <target>
190         <node name="Fundef" />
          </target>
192     </type>
        <phases>
194       <range from="liftsmpd" to="compile" />
        </phases>
196   </attribute>
      <attribute name="Worker" mandatory="no">
198     <description>only for ST_spmdfun</description>
        <type name="Link">
200       <target>
            <unknown />
202       </target>
```

```
      </type>
      <phases>
        <unknown />
      </phases>
    </attribute>
    <attribute name="Companion" mandatory="no">
      <description>only for ST_spmdfun</description>
      <type name="Link">
        <target>
          <node name="Fundef" />
        </target>
      </type>
      <phases>
        <range from="rfin" to="final" />
        <range from="mtfin" to="final" />
      </phases>
    </attribute>
    <attribute name="ExecMode" mandatory="no">
      <type name="MTExecMode"/>
      <phases>
        <range from="BuildMultithread" to="CreateCells"/>
      </phases>
    </attribute>
  </attributes>
  <flags>
    <flag name="IsExported" />
    <flag name="IsProvided" />
    <flag name="IsLocal" default="TRUE" />
    <flag name="IsNeeded" />
    <flag name="IsFoldfun" />
    <flag name="IsCondfun" />
    <flag name="IsDoFun" />
    <flag name="IsExtern" />
    <flag name="IsInline" />
    <flag name="AllowsInfix" />
    <flag name="HasDotArgs" />
    <flag name="HasDotRets" />
    <flag name="IsSpmdfun" />
    <flag name="IsWrapperfun" />
    <flag name="IsZombie" />
    <flag name="IsClassConversionFun" />
  </flags>
</node>
```

B.3. Visualisierung der XML Beschreibung



C. Literaturverzeichnis

[AM94] Appel, Andrew W. und David B. MacQueen: *Separate Compilation for Standard ML.* In: *SIGPLAN Conference on Programming Language Design and Implementation*, Seiten 13–23, 1994.

[Bar81] Barendregt, H.P.: *The Lambda Calculus, Its Syntax and Semantics*, Band 103 der Reihe *Studies in Logics and the Foundations of Mathmatics*. North-Holland, 1981.

[Bri03] British Standards Institute: *The C Standard : Incorporating Technical Corrigendum 1.* John Whiley & Sons, 2003.

[DDH72] Dahl, Ole-Johan, Edsger W. Dijkstra und C. A. R. Hoare: *Structured Programming.* Academic Press, 1972.

[DJH02] Diatchki, Iavor S., Mark P. Jones und Thomas Hallgren: *A formal specification of the Haskell 98 module system.* In: *Proceedings of the ACM SIGPLAN workshop on Haskell*, Seiten 17–28. ACM Press, 2002.

[DK75] DeRemer, Frank und Hans Kron: *Programming-in-the large versus programming-in-the-small.* In: *Proceedings of the international conference on Reliable software*, Seiten 114–121, 1975.

[DS96] Duggan, Dominic und Constantinos Sourelis: *Mixin modules.* In: *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, Seiten 262–273. ACM Press, 1996.

[DuC98] DuCharme, Bob: *XML: The Annotated Specification.* Prentice Hall PTR, 1998.

[GJS00] Gosling, J., B. Joy und G. Steele: *The Java Language Specification, 2nd edition.* Java Series. Addison-Wesley, 2000. ISBN 0-20131-008-2.

[Gre96] Grelck, C.: *Integration eines Modul- und Klassen-Konzeptes in die funktionale Programmiersprache* Sac – *Single Assignment C.* Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.

[GS00] Grelck, C. und S.-B. Scholz: *HPF vs. SAC – A Case Study.* In: Bode, A., T. Ludwig und R. Wismüller (Herausgeber): *Euro-Par*

2000 Parallel Processing, Band 1900 der Reihe *LNCS*, Seiten 620–624. Springer, 2000.

[GS03] Grelck, C. und S.-B. Scholz: *Towards an Efficient Functional Implementation of the NAS Benchmark FT.* In: Malyshkin, V. (Herausgeber): *Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia*, Band 2763 der Reihe *LNCS*, Seiten 230–235. Springer, 2003.

[GT04] Grelck, Clemens und Kai Trojahner: *Implicit Memory Management for* SaC. In: Grelck, Clemens und Frank Huch (Herausgeber): *Implementation and Application of Functional Languages*, Nummer 0408 in *Technical Report*, Seiten 335–348. Christian-Albrechts-Universität zu Kiel, 2004.

[HHPW96] Hall, Cordelia V., Kevin Hammond, Simon L. Peyton Jones und Philip L. Wadler: *Type classes in Haskell.* ACM Trans. Program. Lang. Syst., 18(2):109–138, 1996.

[HL02] Hirschowitz, Tom und Xavier Leroy: *Mixin modules in a Call-by-Value Setting.* In: Le Métayer, Daniel (Herausgeber): *European Symposium on Programming*, Band 2305 der Reihe *LNCS*, Seiten 6–20, 2002.

[HS86] Hindley, J.R. und J.P. Seldin: *Introduction to Combinators and Lambda Calculus*, Band 1 der Reihe *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[HWG03] Hejlsberg, Anders, Scott Wiltamuth und Peter Golde: *The C# Programming Language.* Addison-Wesley Professional, 2003.

[Jon94] Jones, M.P.: *Dictionary-free Overloading by Partial Evaluation.* In: *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* ACM-Press, 1994.

[Jon03] Jones, S.L. Peyton: *Haskell 98 Language and Libraries.* Cambridge University Press, Cambridge, UK, 2003.

[Kay04] Kay, Michael: *XSLT 2.0 Programmer's Reference.* Wrox Press, 3. Auflage, 2004.

[KR90] Kernighan, B.W. und D.M. Ritchie: *Programmieren in C.* PC professionell. Hanser, 1990. ISBN 3-446-15497-3.

[Kre03] Kreye, D.J.: *A Compiler Backend for Generic Programming with Arrays.* Doktorarbeit, Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität Kiel, 2003.

[MTH90] Milner, R., M. Tofte und R. Harper: *The Definition of Standard ML.* MIT Press, 1990. ISBN 0-262-63132-6.

[MTHM97] Milner, Robin, Mads Tofte, Robert Harper und David MacQueen: *The Definition of Standard ML - Revised*. MIT Press, 1997.

[MW01] Marcussen-Wulff, N.: *Über die Verwendung der Static Single Assignment Form zur hochsprachlichen Optimierung der funktionalen Programmiersprache SAC*. Diplomarbeit, Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität Kiel, 2001.

[Par72] Parnas, D. L.: *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 15(12):1053–1058, Dezember 1972.

[PFTV96] Press, William H., Brian P. Flannery, Saul A. Teukolsky und William T. Vetterling: *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, 2. Auflage, 1996.

[PJ93] Peterson, John und Mark Jones: *Implementing type classes*. In: *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, Seiten 227–236. ACM Press, 1993.

[PJM97] Peyton Jones, Simon, Mark Jones und Erik Meijer: *Type classes: exploring the design space*. In: *Procedings of the Haskell Workshop 1997*, 1997.

[PvE01] Plasmeijer, R. und M. van Eekelen: *Concurrent Clean 1.3.1 Language Report*. High Level Software Tools B.V. and University of Nijmegen, 2001.

[Rei97] Reinke, Claus: *Functions, Frames and Interactions – completing a λ-calculus-bases purely functional language with respect to programming-in-the-large and interactions with runtime environments*. Doktorarbeit, Christian-Albrechts-Universität Kiel, 1997.

[Sch77] Scheifler, Robert W.: *An analysis of inline substitution for a structured programming language*. Communications of the ACM, 20(9):647–654, 1977.

[Sch94] Scholz, S.-B.: *Single Assignment C – Functional Programming Using Imperative Style*. In: Glauert, John (Herausgeber): *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*. University of East Anglia, 1994.

[Sch98] Scholz, S.-B.: *With-loop-folding in* Sac*–Condensing Consecutive Array Operations*. In: Clack, C., K.Hammond und T. Davie (Herausgeber): *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, Band 1467 der Reihe *LNCS*, Seiten 72–92. Springer, 1998.

[Sch01] Scholz, S.-B.: *A Type System for Inferring Array Shapes* . In: Arts, T. und M. Mohnen (Herausgeber): *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL'01), Stockholm, Sweden*, Seiten 65–82. Ericsson Computer Science Laboratory, 2001.

[Sch03] Scholz, Sven-Bodo: *Generic Array Programming.* Habilitation, Universität Kiel, (submitted), 2003.

[The02] The Single Unix Specification: *The Authorized Guide to Version 3.* The Open Group, 2002.

[vdV02] Vlist, Eric van der: *XML Schema. The W3C's Object-Oriented Descriptions for XML.* O'Reilly, 2002.

[WB89] Wadler, P. und S. Blott: *How to Make ad-hoc Polymorphism Less ad hoc.* In: *POPL '89*, Seiten 60–76. ACM Press, 1989.

[Weh85] Wehnes, H.: *FORTRAN-77: Strukturierte Programmierung mit FORTRAN-77.* Carl Hanser Verlag, 1985.

[Wir85] Wirth, N.: *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer, 1985. ISBN 0-387-15078-1.

[Wir94] Wirth, Niklaus: *A Brief History of Modula and Lilith.* In: *Advances in Modular Languages – Proceedings of the Joint Modular Languages Conference.* Universitätsverlag Ulm, 1994.

Eidesstattliche Erklärung
Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 16. Februar 2005